# Program Analysis for Adaptivity Analysis

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Labeled While Language</td>
<td>2</td>
</tr>
<tr>
<td>1.1</td>
<td>Labeled Language</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Trace-based Operational Semantics for Labeled While Language</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Event and Trace</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Event</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Trace</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>Dependency and Adapativity</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Semantics-based Dependency Graph</td>
<td>8</td>
</tr>
<tr>
<td>3.2</td>
<td>May-Dependency</td>
<td>9</td>
</tr>
<tr>
<td>3.3</td>
<td>Trace-based Adapativity</td>
<td>11</td>
</tr>
<tr>
<td>3.4</td>
<td>The Walk Through Example</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>The Adaptivity Analysis Algorithm - AdaptFun</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>A guide to AdaptFun</td>
<td>13</td>
</tr>
<tr>
<td>4.2</td>
<td>Vertex and Query Annotation Estimations</td>
<td>14</td>
</tr>
<tr>
<td>4.3</td>
<td>Edge and Weight Estimation</td>
<td>14</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Abstract Transition Graph</td>
<td>14</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Edge Estimation</td>
<td>19</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Weight Estimation</td>
<td>22</td>
</tr>
<tr>
<td>4.4</td>
<td>Graph Construction</td>
<td>25</td>
</tr>
<tr>
<td>4.5</td>
<td>Adaptivity Upper Bound Computation</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>Examples and Experimental Results</td>
<td>32</td>
</tr>
<tr>
<td>5.1</td>
<td>Examples</td>
<td>32</td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation Results</td>
<td>35</td>
</tr>
<tr>
<td>5.3</td>
<td>More Discussions on The Evaluated Examples</td>
<td>37</td>
</tr>
<tr>
<td>5.3.1</td>
<td>More on The Two Rounds Adaptive Data Analysis</td>
<td>37</td>
</tr>
<tr>
<td>5.3.2</td>
<td>mRComplete</td>
<td>38</td>
</tr>
<tr>
<td>5.3.3</td>
<td>tRGD</td>
<td>38</td>
</tr>
<tr>
<td>Appendices</td>
<td></td>
<td>40</td>
</tr>
</tbody>
</table>
A Proofs of Lemmas for the Language Model 40
   A.1 Proof of Lemma 1.1 ................................................. 40
   A.2 Proof of Lemma 2.1 ................................................. 40
   A.3 Proof of Lemma 2.0.1 .............................................. 41
B Proof of Theorem 4.2 41
C Soundness of Edge Estimation 45
   C.1 Main Theorem ......................................................... 45
   C.2 Soundness of flowsTo w.r.t. the Event ............................. 46
   C.3 Inversion Lemmas and Helper Lemmas ................................ 48
D Soundness of The Weight Estimation 60
   D.1 Proof of Lemma 4.1 ................................................... 60
   D.2 Proof of Lemma 4.2 ................................................... 61
   D.3 Soundness of Weight Estimation, Theorem 4.1 ......................... 61
E Soundness of Adaptivity Computation Algorithm 64
F Conditional Completeness of Adaptivity Computation Algorithm 67
G The Detail Evaluation Table 69
H The Programs and Codes of The Evaluated Examples in Table2 69
   H.1 The Programs for Examples from line:6 - 15 in Table 2 ............... 69
   H.2 The Programs for Examples from line:16 - 20 in Table 2 ............. 73

1 Labeled While Language

1.1 Labeled Language

<table>
<thead>
<tr>
<th>Category</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic Operators</td>
<td>$\oplus_a := + - \cdot \div \max \min$</td>
</tr>
<tr>
<td>Boolean Operators</td>
<td>$\oplus_b := \vee \land$</td>
</tr>
<tr>
<td>Relational Operators</td>
<td>$\sim := &lt; \leq \geq$</td>
</tr>
<tr>
<td>Label</td>
<td>$l \in \mathbb{N} \cup {\text{in,ex}}$</td>
</tr>
<tr>
<td>Arithmetic Expression</td>
<td>$a := n \in \mathbb{N}^\infty \mid x \mid a \oplus_a a \mid \log a \mid \text{sign} a$</td>
</tr>
<tr>
<td>Boolean Expression</td>
<td>$b := \text{true} \mid \text{false} \mid \neg b \mid b \oplus b \mid a \sim a$</td>
</tr>
<tr>
<td>Expression</td>
<td>$e := v \mid a \mid b \mid [e,\ldots,e]$</td>
</tr>
<tr>
<td>Value</td>
<td>$v := n \mid \text{true} \mid \text{false} \mid [\ ] \mid [v,\ldots,v]$</td>
</tr>
<tr>
<td>Query Expression</td>
<td>$\psi := \alpha \mid a \mid \psi \oplus_a \psi \mid \chi[a]$</td>
</tr>
<tr>
<td>Query Value</td>
<td>$\alpha := n \mid \chi[n] \mid a \oplus_a a \mid n \oplus_a \chi[n] \mid \chi[n] \oplus_a n$</td>
</tr>
<tr>
<td>Labeled Command</td>
<td>$c := [x \leftarrow e]^t \mid [x \leftarrow \text{query}(\psi)]^t \mid \text{while} [b]^t \text{do} c \mid c \mid \text{if} [b]^t, c, c \mid [\text{skip}]^t$</td>
</tr>
<tr>
<td>Event</td>
<td>$\epsilon := (x, l, v, \bullet) \mid (x, l, v, a) \quad \text{Assignment Event}$</td>
</tr>
<tr>
<td></td>
<td>$\mid (b, l, v, \bullet) \quad \text{Testing Event}$</td>
</tr>
</tbody>
</table>
We use following notations to represent the set of corresponding terms:

\[\text{VAR} : \text{Set of Variables}\]
\[\text{VAL} : \text{Set of Values}\]
\[\text{QL}\text{VAL} : \text{Set of Query Values}\]
\[\text{C} : \text{Set of Commands}\]
\[\text{E} : \text{Set of Events}\]
\[\text{E}_{\text{asn}} : \text{Set of Assignment Events}\]
\[\text{E}_{\text{test}} : \text{Set of Testing Events}\]
\[\text{L} : \text{Set of Labels}\]
\[\text{D\text{L}} : \text{Set of Labeled Variables}\]
\[\text{T} : \text{Set of Traces}\]
\[\Omega\text{D} : \text{Domain of Query Results}\]

Environment \(\rho : \mathcal{T} \rightarrow \text{VAR} \rightarrow \text{VAL} \cup \{\bot\}\)

\[
\begin{align*}
\rho((x, l, v, \bullet)) &\triangleq v & \rho((y, l, v, \bullet)) &\triangleq \rho(r)x, y \neq x & \rho((b, l, v, \bullet)) &\triangleq \rho(r)x \\
\rho((x, b, \bullet)) &\triangleq \rho(r)x, y \neq x & \rho((b, b, \bullet)) &\triangleq \bot
\end{align*}
\]

1.2 **Trace-based Operational Semantics for Labeled While Language**

\[
\begin{array}{c|c|c|c}
\hline
\langle t, a \rangle \downarrow a v & \text{Trace} \times \text{Arithmetic Expr} \Rightarrow \text{Arithmetic Value} \\
\hline
\langle t, n \rangle \downarrow a n & \langle t, x \rangle \downarrow a v & \langle t, a_1 \rangle \downarrow a v_1 & \langle t, a_2 \rangle \downarrow a v_2 & v_1 \oplus a v_2 = v \\
\hline
\langle t, a \rangle \downarrow a v' & \text{log } v' = v & \langle t, a \rangle \downarrow a v' & \text{sign } v' = v \\
\hline
\langle t, log a \rangle \downarrow a v & \langle t, sign a \rangle \downarrow a v \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
\hline
\langle t, b \rangle \downarrow b v : \text{Trace} \times \text{Boolean Expr} \Rightarrow \text{Boolean Value} \\
\hline
\langle t, false \rangle \downarrow b false & \langle t, true \rangle \downarrow b true & \langle t, \neg b \rangle \downarrow b v \\
\hline
\langle t, b_1 \rangle \downarrow b v_1 & \langle t, b_2 \rangle \downarrow b v_2 & v_1 \oplus b v_2 = v \\
\hline
\langle t, a_1 \rangle \downarrow a v_1 & \langle t, a_2 \rangle \downarrow a v_2 & v_1 \sim v_2 = v \\
\hline
\langle t, a_1 \sim a_2 \rangle \downarrow b v \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
\hline
\langle t, e \rangle \downarrow e v : \text{Trace} \times \text{Expression} \Rightarrow \text{Value} \\
\hline
\langle t, a \rangle \downarrow a v & \langle t, b \rangle \downarrow b v & \langle t, e_1 \rangle \downarrow e v_1 \cdots \langle t, e_n \rangle \downarrow e v_n \\
\hline
\langle t, b \rangle \downarrow e v & \langle t, [e_1, \ldots, e_n] \rangle \downarrow e [v_1, \ldots, v_n] \\
\hline
\langle t, v \rangle \downarrow e v \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
\hline
\langle t, \psi \rangle \downarrow q a : \text{Trace} \times \text{Query Expr} \Rightarrow \text{Query Value} \\
\hline
\langle t, a \rangle \downarrow q n & \langle t, \psi_1 \rangle \downarrow q a_1 & \langle t, \psi_2 \rangle \downarrow q a_2 & \langle t, a \rangle \downarrow q n \\
\hline
\langle t, \psi_1 \oplus q \psi_2 \rangle \downarrow q a_1 \oplus q a_2 & \langle t, a \rangle \downarrow q n & \langle t, \chi[a] \rangle \downarrow q \chi [n] & \langle t, a \rangle \downarrow q a \\
\end{array}
\]
The labeled variables and assigned variables are sets of variables annotated by a label. We use \( L \) to represent the universe of all the labeled variables and \( AV_c \) to represent the set of assigned variables and labeled variables for a labeled command \( c \), defined in Definition 1 and 2.

\[ FV : e \rightarrow \mathbb{P}(VAR) \]

computes the set of free variables in an expression. To be precise, \( FV(a) \), \( FV(b) \) and \( FV(\psi) \) represent the set of free variables in arithmetic expression \( a \), boolean expression \( b \) and query expression \( \psi \) respectively. Labeled variables in \( c \) is the set of assigned variables and all the free variables showing up in \( c \) with a default label \( in \). The free variables showing up in \( c \), which aren’t defined before be used, are actually the input variables of this program.

**Definition 1 (Assigned Variables)**

\[
AV_c \triangleq \begin{cases} 
[\{x\}] & c = [x \leftarrow e] \\
[\{x\}] & c = [x \leftarrow \text{query(}\psi\text{)}] \\
AV_{c_1} \cup AV_{c_2} & c = e_1; e_2 \\
AV_{c_1} \cup AV_{c_2} & c = \text{if} ([b]; c_1; c_2) \\
AV_{c_1} \cup AV_{c_2} & c = \text{while} ([b]; c_1; c_2) \\
\end{cases}
\]
Definition 2 (labelled Variables $LV$).

$$LV_c \triangleq \begin{cases} \{x^i\} \cup FV(e)^{in} & c = [x \leftarrow e]^l \\ \{x^i\} \cup FV(\psi)^{in} & c = [x \leftarrow \text{query}(\psi)]^l \\ LV_{c_1} \cup LV_{c_2} & c = c_1 \cdot c_2 \\ LV_c \cup LV_{c_2} \cup FV(b)^{in} & c = \text{if}([b]^l, c_1, c_2) \\ LV_c \cup FV(b)^{in} & c = \text{while}([b]^l, c') \end{cases}$$

We also defined the set of query variables for a program $c$, it is the set of variables set to the result of a query in the program formally in Definition 3.

Definition 3 (Query Variables ($QV : \mathcal{C} \rightarrow \mathcal{P}(LV)$)). Given a program $c$, its query variables $QV(c)$ is the set of variables set to the result of a query in the program. It is defined as follows:

$$QV(c) \triangleq \begin{cases} \emptyset & c = [x \leftarrow e]^l \\ \{x^i\} & c = [x \leftarrow \text{query}(\psi)]^l \\ QV(c_1) \cup QV(c_2) & c = c_1 \cdot c_2 \\ QV(c_1) \cup QV(c_2) & c = \text{if}([b]^l, c_1, c_2) \\ QV(c') & c = \text{while}([b]^l, c') \end{cases}$$

It is easy to see that a program $c$’s query variables is a subset of its labeled variables, $QV(c) \subseteq LV(c)$. Every labeled variable in a program is unique, formally as follows with proof in Appendix A.1.

Lemma 1.1 (Uniqueness of the Labeled Variables). For every program $c \in \mathcal{C}$ and every two labeled variables such that $x^i, y^j \in LV(c)$, then $x^i \neq y^j$.

$$\forall c \in \mathcal{C}, x^i, y^j \in \mathcal{L} . x^i, y^j \in \mathcal{L} \iff x^i \neq y^j.$$
2 Event and Trace

2.1 Event

Event projection operators $\pi_i$ projects the $i$th element from an event:

$$\pi_i : \mathcal{E} \rightarrow \mathcal{VAR} \cup \mathcal{B}$$

where $\mathcal{VAR}$ is the set of free variables in the event.

Free Variables: $\mathcal{FV} : e \rightarrow \mathcal{P}(\mathcal{VAR})$, the set of free variables in an expression.

Definition 4 (Equivalence of Query Expression). Two query expressions $\psi_1$, $\psi_2$ are equivalent, denoted as $\psi_1 =_q \psi_2$, if and only if:

$$\forall \tau \in \mathcal{T}, \exists \alpha_1, \alpha_2 \in \mathcal{VAR} \cup \mathcal{L} \cdot \langle \tau, \psi_1 \rangle \models q \alpha_1 \land \langle \tau, \psi_2 \rangle \models q \alpha_2 \land (\forall D \in D \mathcal{B}, r \in D \cdot \exists \nu \in \mathcal{VAR} \cup \mathcal{L} \cdot \langle \tau, \alpha_1[\nu/\chi] \rangle \models q \nu \land \langle \tau, \alpha_2[r/f] \rangle \models q \nu)$$

where $\tau \in D$ is a record in the database domain $D$. As usual, we will denote by $\psi_1 \neq_q \psi_2$ the negation of the equivalence.

Definition 5 (Event Equivalence). Two events $\epsilon_1, \epsilon_2 \in \mathcal{E}$ are equivalent, denoted as $\epsilon_1 = \epsilon_2$ if and only if:

$$\pi_1(\epsilon_1) = \pi_1(\epsilon_2) \land \pi_2(\epsilon_1) = \pi_2(\epsilon_2) \land \pi_3(\epsilon_1) = \pi_3(\epsilon_2) \land \pi_4(\epsilon_1) = \pi_4(\epsilon_2)$$

As usual, we will denote by $\epsilon_1 \neq \epsilon_2$ the negation of the equivalence.

2.2 Trace

Definition 6 (Trace Concatenation, $\cdot : \mathcal{T} \rightarrow \mathcal{T} \rightarrow \mathcal{T}$). Given two traces $\tau_1, \tau_2 \in \mathcal{T}$, the trace concatenation operator $\cdot$ is defined as:

$$\tau_1 \cdot \tau_2 \triangleq \begin{cases} \tau_1 & \tau_2 = [l] \\ (\tau_1 \cdot \tau_2) :: e & \tau_2 = \tau_2 :: e \end{cases}$$

Definition 7. (An Event Belongs to A Trace) An event $e \in \mathcal{E}$ belongs to a trace $\tau$, i.e., $e \in \tau$ are defined as follows:

$$e \in \tau \triangleq \begin{cases} \text{true} & \tau = \tau' :: e' \land e = e' \\ \text{false} & \tau = \tau' :: e' \land e \neq e' \end{cases}$$

As usual, we denote by $e \notin \tau$ that the event $e$ doesn’t belong to the trace $\tau$.

We introduce a counting operator $\text{cnt} : \mathcal{T} \rightarrow \mathcal{N} \rightarrow \mathcal{N}$ whose behavior is defined as follows,

$$\text{cnt}(\tau :: (x, l, v, \bullet), l) \triangleq \text{cnt}(\tau, l) + 1$$

$$\text{cnt}(\tau :: (x, l, v, \alpha), l) \triangleq \text{cnt}(\tau, l) + 1$$

$$\text{cnt}(\tau :: (b, l', v, \bullet), l) \triangleq \text{cnt}(\tau, l, l' \neq l)$$

$$\text{cnt}(\sigma, l) \triangleq 0$$

We introduce an operator $i : \mathcal{T} \rightarrow \mathcal{VAR} \cup \mathcal{L} \cup \{\bot\}$, which takes a trace and a variable and returns the label of the latest assignment event which assigns value to that variable. Its behavior is defined as follows,

$$i(\tau :: (x, l, \_, \_)) x \triangleq l$$

$$i(\tau :: (y, l, \_, \_)) x \triangleq y \neq x$$

$$i(\tau :: (b, l, v, \bullet)) x \triangleq i(\tau) x$$
The operator $\mathbb{T}L : \mathcal{T} \to \mathcal{P}(\mathcal{L})$ gives the set of labels in every event belonging to a trace, whose behavior is defined as follows,

$$\mathbb{T}L(\tau :: (\_, l, \_)) \triangleq \{l\} \cup \mathbb{T}L(\tau) \quad \mathbb{T}L(\[]) \triangleq \{\}$$

If we observe the operational semantics rules, we can find that no rule will shrink the trace. So we have the Lemma 2.1 with proof in Appendix A.2 specifically the trace has the property that its length never decreases during the program execution.

**Lemma 2.1 (Trace Non-Decreasing).** For every program $c \in \mathcal{C}$ and traces $\tau, \tau' \in \mathcal{T}$, if $(c, \tau) \rightarrow^* (\text{skip}, \tau')$, then there exists a trace $\tau'' \in \mathcal{T}$ with $\tau'' \tau = \tau'$

$$\forall \tau, \tau' \in \mathcal{T}, c . \langle c, \tau \rangle \rightarrow^* \langle \text{skip}, \tau' \rangle \implies \exists \tau'' \in \mathcal{T} . \tau'' \tau = \tau'$$

Since the equivalence over two events is defined over the query value equivalence, when there is an event belonging to a trace, if this event is a query assignment event, it is possible that the event showing up in this trace has a different form of query value, but they are equivalent by Definition 4. So we have the following Corollary 2.0.1 with proof in Appendix A.3.

**Corollary 2.0.1.** For every event and a trace $\tau \in \mathcal{T}$, if $\epsilon \in \tau$, then there exist another event $\epsilon' \in \mathcal{E}$ and traces $\tau_1, \tau_2 \in \mathcal{T}$ such that $\tau_1 \epsilon \tau_2 = \tau$ with $\epsilon$ and $\epsilon'$ equivalent but may differ in their query value.

$$\forall \epsilon \in \mathcal{E}, \tau \in \mathcal{T} . \epsilon \in \tau \implies \exists \tau_1, \tau_2 \in \mathcal{T}, \epsilon' \in \mathcal{E} . (\epsilon \in \epsilon') \land \tau_1 \epsilon \tau_2 = \tau$$
3 Dependency and Adapativity

In this section, we formally present the definition of adaptivity of a program, which is the length of the ‘longest’ walk with the most queries involved in the semantics-based dependency graph of this program. We first present the construction of the semantics-based dependency graph before the introduction of the formal definition of adaptivity.

3.1 Semantics-based Dependency Graph

The semantics-based dependency graph is formally defined in Definition 8. For a program $c$, there are some notations used in the definition. The labeled variables of $c$, $\mathbb{L}V(c) \subseteq \mathbb{L}V$ contains all the variables in $c$’s assignment commands, with the command labels as superscripts. The set of query-associated variables (in query request assignments), $QV(c) \subseteq \mathbb{L}V(c)$ contains all labeled variables in $c$’s query requests. The set of initial traces of $c$, $\mathcal{T}_0(c) \subseteq \mathcal{T}$ contains all possible initial trace of $c$. Each initial trace, $\tau_0 \in \mathcal{T}_0(c)$ contains the initial values of all input variables of $c$. For instance, the initial trace of twoRounds(k) example contains the initial value of the input variable $k$.

Definition 8 (Semantics-based Dependency Graph). Given a program $c$, its semantics-based dependency graph $G_{\text{trace}}(c) = (V_{\text{trace}}(c), E_{\text{trace}}(c), W_{\text{trace}}(c), Q_{\text{trace}}(c))$ is defined as follows.

- **Vertices** $V_{\text{trace}}(c) := \{x^j \mid x^j \in \mathbb{L}V(c)\}$
- **Directed Edges** $E_{\text{trace}}(c) := \{(x^i, y^j) \mid x^i, y^j \in \mathbb{L}V(c) \land \text{DEP}_{\text{var}}(x^i, y^j, c)\}$
- **Weights** $W_{\text{trace}}(c) := \{(x^i, w) \mid w : \mathcal{T}_0(c) \rightarrow \mathbb{N} \land x^i \in \mathbb{L}V(c) \land \forall \tau_0 \in \mathcal{T}_0(c), \tau' \in \mathcal{T}, (c, \tau_0) \rightarrow \ast (\text{skip}, \tau_0, \tau') \land w(\tau_0) = \text{cnt}(\tau', l)\}$
- **Query Annotations** $Q_{\text{trace}}(c) := \{(x^i, n) \mid x^i \in \mathbb{L}V(c) \land n = 1 \iff x^i \in QV(c) \land n = 0 \iff x^i \notin QV(c)\}$

where $\ast : \mathcal{T} \rightarrow \mathcal{T}$ is the trace concatenation operator, which combines two traces, and $\text{cnt} : \mathcal{T} \rightarrow \mathbb{N}$ is the counting operator, which counts the occurrence of a labeled variable in the trace. All the definition details are in the appendix. A semantics-based dependency graph $G_{\text{trace}}(c) = (V_{\text{trace}}(c), E_{\text{trace}}(c), W_{\text{trace}}(c), Q_{\text{trace}}(c))$ is well-formed if and only if $\forall x^j \mid (x^i, w) \in W_{\text{trace}}(c) = V_{\text{trace}}(c)$.

There are four components in this graph.

1. The vertices $V_{\text{trace}}(c)$ of a program $c$ are all its labeled variables, $\mathbb{L}V(c)$ which are statically collected.
2. $Q_{\text{trace}}(c)$ contains the query annotation for every vertex $x^j \in V_{\text{trace}}(c)$. It indicates whether $x^j$ comes from a query request (1) or not (0) by checking if the labeled variable $x^j$ of the vertex is in $QV(c)$.
3. Edges in $E_{\text{trace}}(c)$ are built from the $\text{DEP}_{\text{var}}(x^i, y^j, c)$ relation between two labeled variables. This is the key definition in order to formalize the intuitive may-dependency relation between queries and the adaptivity. We present this formalization detail in Section 3.2 below.
4. The weight function in $W_{\text{trace}}(c)$ for each vertex, $w : \mathcal{T} \rightarrow \mathbb{N}$ maps from a starting trace $\tau_0 \in \mathcal{T}_0(c)$ to a natural number. For each vertex $x^i$, it tracks its visiting times (i.e., the evaluation times of the command with the label $l$) when the program $c$ is evaluated from the initial trace $\tau_0$ into $\text{skip}$, $(c, \tau_0) \rightarrow \ast (\text{skip}, \tau_0, \tau')$. The visiting times is computed by the counter operator $\text{cnt}(\tau', l)$ by counting the occurrence of the label $l$ in $\tau'$. As an instance, in the semantics-based dependency graph of twoRounds in Figure 3(b), the weight, $w_k$ of the vertex $x^3$ is a...
function of type $\text{twoRound}(\tau_0) \rightarrow \mathcal{N}$. Given input $\tau_0$, we execute the program under $\tau_0$ as $(\text{twoRound}(\tau_0), \tau_0) \rightarrow^{*} (\text{skip}, \tau_0 \cdots \tau')$. Then $\nu_k(\tau_0)$ outputs the occurrence time of the label 3 in $\tau'$.

The main novelty of the semantics-based dependency graph is the combination of the quantitative and dependency information. It can tell both the dependency between queries via the directed edge, and the times they depend on each other via the weight.

### 3.2 May-Dependency

This section formalizes the may-dependency relation between queries and introduces the variable may-dependency definition.

There are two possible situations that a query will be “influenced” by previous queries’ results, where either the query request is changed when the results of previous queries are changed (data dependency), or the query request is disappeared when the results of previous queries are changed (control dependency). In this sense, our formal dependency definition considers both the two cases as follows,

1. One query may depend on a previous query if and only if a change of the value returned to the previous query request may also change this query request.

2. One query may depend on a previous query if and only if a change of the value returned to the previous query request may also change the appearance of this query quest.

The first case captures the data dependency. For instance, in a simple program $c_1 = [x \leftarrow \text{query}(\chi[2]) ]^{1}; [y \leftarrow \text{query}(\chi[3] + x)]^{2}$, we think $\text{query}(\chi[3] + x)$ (variable $y^3$) may depend on the query $\text{query}(\chi[2])$ (variable $x^1$), because the equipped function of the former $\chi[3] + x$ may depend on the data stored in $x$ assigned with the result of $\text{query}(\chi[2])$. From our perspective, $\text{query}(\chi[1])$ is different from $\text{query}(\chi[2])$.

The second case captures the control dependency. For instance, in the program $c_2 = [x \leftarrow \text{query}(\chi[1]) ]^{1}; [y \leftarrow \text{query}(\chi[2])]^{3}, [\text{skip}]^{4}$, we think the query $\text{query}(\chi[2])$ ( or the labeled variable $y^3$) may depend on the query $\text{query}(\chi[1])$ (via the labeled variable $x^1$).

Since both of the two “influences” are passing through labeled variables, we choose to formally define the may-dependency relation over all labeled variables, and then recover the query requests from query-associated variables, $Q \forall (c)$. It relies on the formal observation of the “influence” via events in Definition[9] and the may-dependency between events in Definition[10].

**Definition 9 (Events Differ in Value (Diff)).** Two events $e_1, e_2 \in E$ differ in their value, or query value, denoted as $\text{Diff}(e_1, e_2)$, if and only if:

$$
\pi_1(e_1) = \pi_1(e_2) \land \pi_2(e_1) = \pi_2(e_2) \land (\pi_3(e_1) \neq \pi_3(e_2) \land \pi_4(e_1) = \pi_4(e_2) = \psi) \lor (\pi_4(e_1) \neq \psi \land \pi_4(e_2) \neq \psi \land \pi_4(e_1) \neq \pi_4(e_2))
$$

where $\psi_1 = q \psi_2$ denotes the semantics equivalence between query values, and $\pi_i$ projects the $i$-th element from the quadruple of an event.

$$
\pi_1(e_1) = \pi_1(e_2) \land \pi_2(e_1) = \pi_2(e_2) \text{ at Eq}[2](a) \text{ requires that } e_1 \text{ and } e_2 \text{ have the same variable name and label. This guarantees that } e_1 \text{ and } e_2 \text{ are generated from the same labeled command. In Eq}[2](b), \text{two kinds of comparisons between the third and fourth element are for the non-query assignment and query request separately. For events generated from the non-query assignments (via checking}
$$
\( \pi_4(e_1) = q \pi_4(e_2) = \bullet \), we only compare their assigned values through \( \pi_4(e_1) \neq \pi_3(e_2) \). But for these from query requests (via checking \( \pi_4(e_1) \neq \bullet \wedge \pi_4(e_2) \neq \bullet \)), we are comparing their query expressions by \( \pi_4(e_1) \neq q \pi_4(e_2) \) rather than the assigned value computed from the unknown database server. This matches the intuitive data dependency between queries, where one query is influenced by others as long as the query request is changed.

Below is the event may-dependency between events based on formally observing their differences via Diff.

**Definition 10 (Event May-Dependency).** An event \( e_2 \) is in the event may-dependency relation with an assignment event \( e_1 \in \mathcal{E}_{\text{ann}} \) in a program \( c \) with a hidden database \( D \) and a witness trace \( t \in \mathcal{T} \), \( \text{DEP}_q(e_1, e_2, [e_1], \cdots, [e_2], c, D) \) if and only if

\[
\exists r_0, t_1, t' \in \mathcal{T}, e'_1 \in \mathcal{E}_{\text{ann}}, c_1, c_2 \in \mathcal{E} . \text{Diff}(e_1, e'_1) \wedge (\exists e'_2 \in \mathcal{E} . \quad \begin{cases} (c, t_0) & \rightarrow^* (c_1, t_1, \cdots, [e_1]) \rightarrow^* (c_2, t_{1,2}, \cdots, [e_2]) \\ \wedge (c_1, t_1, \cdots, [e'_1]) & \rightarrow^* (c_2, t_{1,2}, \cdots, [e'_2]) \\ \wedge \text{Diff}(e_2, e'_2) \wedge \text{cnt}(t, \pi_2(e_2)) = \text{cnt}(t', \pi_2(e'_2)) \end{cases}) \quad (3a) \]

\[
\begin{align*}
\forall (c, t_0) & \rightarrow^* (c_1, t_1, \cdots, [e_1]) \rightarrow^* (c_2, t_{1,2}, \cdots, [e_2], b) \\
\wedge (c_1, t_1, \cdots, [e'_1]) & \rightarrow^* (c_2, t_{1,2}, \cdots, [e'_2], b) \\
\wedge \text{TL}(t_3) \cap \text{TL}(t'_3) & = \emptyset \wedge \text{cnt}(t', \pi_2(e_2)) = \text{cnt}(t, \pi_2(e_2)) \wedge e_2 \in t_3 \wedge e_2 \notin t'_3
\end{align*}
\quad (3b)
\]

where \( \mathbb{T}(t) \subseteq \mathcal{L} \) is the set of the labels in all the events from trace \( t \) and \( e_2 \in t_3 \) or \( e_2 \notin t_3 \) denotes that \( e_2 \) belongs to \( t_3 \) or not.

The first line in Eq. (3a) requires that \( e_1 \) comes from an assignment command and then modifies its assigned value via Diff\( (e_1, e'_1) \).

Then, the following two parts in Eq (3b) and (c) capture the intuitive value dependency and control dependency respectively. Both parts execute the program two times w.r.t. the different values in \( e_1 \) (as line:1 in Eq (3b) and line:2 in Eq (3c)) and \( e'_1 \) (as line:2 in Eq (3b) and line:3 in Eq (3c)), but observe the difference in the newly generated traces in different ways (via 3rd line in Eq (3b) and 4th line in Eq (3c)). This idea is similar to the dependency definition from [1].

In Eq (3b) line:2, if the newly generated trace, \( t' + [e'_2] \) still contains \( e_2 \) as \( e'_2 \), we check the difference on their value in line:3. If they only differ in their assigned values, i.e., Diff\( (e_2, e'_2) \) and they are in the same loop iteration (via cnt\( (t, \pi_2(e_2)) = \text{cnt}(t', \pi_2(e'_2)) \)), then we say there is a value may-dependency relation between \( e_1 \) and \( e_2 \).

The Eq (3c) captures the control dependency through observing the disappearance \( e_2 \) from newly generated traces, \( t' + [\neg e_b], t'_3 \) in the second execution (line:3). \( e_2 \in t_3 \wedge e_2 \notin t'_3 \) in Eq (3c) line:4 specifies this disappearance. cnt\( (t', \pi_2(e_2)) = \text{cnt}(t, \pi_2(e_2)) \) is used to make sure the two executions are in the same loop iteration as well. Different from Eq (3b) line:3, we use a testing event, \( e_b \) here because cnt\( (t, \pi_2(e_2)) = \text{cnt}(t', \pi_2(e'_2)) \) cannot guarantee the disappearance if there are nested loops. This is correct because the control dependency can only be passed through the guard of if or while command, and this guard must be evaluated into two different values (\( e_b \) and \( \neg e_b \)) in the two executions.

Then Considering all the assignment events newly generated during a program's executions, as long as there is one pair of events satisfying the event may-dependency, we say that the two labeled variables in the two assignment events satisfy the variable may-dependency relation below.
The length of \( \mathbf{k} \) walk. We show the definition of a finite walk as follows. This is formally defined below.

From the definition, a labeled assigned variables \( x_2^j \) may depend on another labeled assigned variable \( x_1^i \) in a program \( c \) under the hidden database \( D \), as long as there exist two assignment events \( e_1 \) (for \( x_1^i \)) and \( e_2 \) for \( x_2^j \) satisfy the event may-dependency relation under a witness trace \( \tau \).

### 3.3 Trace-based Adaptivity

Given a program \( c \)'s semantics-based dependency graph \( G_{\text{trace}}(c) \), we define adaptivity with respect to an initial trace \( \tau_0 \in \mathcal{T}_0(c) \) by the finite walk in the graph, which has the most query requests along the walk. We show the definition of a finite walk as follows.

**Definition 12** (Finite Walk \((k)\)). Given the semantics-based dependency graph \( G_{\text{trace}}(c) = (V_{\text{trace}}, E_{\text{trace}}, W_{\text{trace}}, Q_{\text{trace}}) \) of a program \( c \), a finite walk \( k \) in \( G_{\text{trace}}(c) \) is a function \( k \). Given an input initial trace \( \tau_0 \in \mathcal{T}_0(c) \), \( k(\tau_0) \) is a sequence of edges \( (e_1, e_2) \) for which there is a sequence of vertices \( (v_1, \ldots, v_n) \) such that:

- \( e_i = (v_i, v_{i+1}) \in E_{\text{trace}} \) for every \( 1 \leq i < n \).
- every \( v_i \in V_{\text{trace}} \) and \( (v_i, w_i) \in W_{\text{trace}} \). \( v_i \) appears in \( (v_1, \ldots, v_n) \) at most \( w(\tau_0) \) times.

The length of \( k(\tau_0) \) is the number of vertices in its vertices sequence, i.e., \( \text{len}(k)(\tau_0) = n \).

\( \mathcal{W}_{\mathcal{K}}(G_{\text{trace}}(c)) \) is the set of all the finite walks \( k \) in \( G_{\text{trace}}(c) \), and \( k_{v_1 \rightarrow v_2} \in \mathcal{W}_{\mathcal{K}}(G_{\text{trace}}(c)) \) denotes the walk from vertex \( v_1 \) to \( v_2 \).

Because the adaptivity are intuitively describing the dependency between queries, so we calculate a special “length”, the query length of a walk by counting only the vertices corresponding to queries. This is formally defined below.

**Definition 13** (Query Length of the Finite Walk \((\text{len}^3)\)). Given the semantics-based dependency graph \( G_{\text{trace}}(c) \) of a program \( c \), and a finite walk \( k \in \mathcal{W}_{\mathcal{K}}(G_{\text{trace}}(c)) \). The query length of \( k \), \( \text{len}^3(k) \) is a function \( \mathcal{T}_0(c) \rightarrow \mathbb{N} \), such that given an input initial trace \( \tau_0 \), \( \text{len}^3(k)(\tau_0) \) is the number of vertices which correspond to query variables in the vertex sequence, \( (v_1, \ldots, v_n) \) as follows,

\[
\text{len}^3(k)(\tau_0) = |\{ v | v \in (v_1, \ldots, v_n) \land Q(v) = 1 \}|.
\]

Then the definition of adaptivity is presented in Definition 14 below.

**Definition 14** (Adaptivity of a Program). Given a program \( c \), its adaptivity \( A(c) \) is function \( A(c) : \mathcal{T} \rightarrow \mathbb{N} \) such that for an initial trace \( \tau_0 \in \mathcal{T}_0(c) \),

\[
A(c)(\tau_0) = \max \{ \text{len}^3(k)(\tau_0) | k \in \mathcal{W}_{\mathcal{K}}(G_{\text{trace}}(c)) \}
\]

### 3.4 The Walk Through Example
Figure 2: (a) The program towRounds(k), an example with two rounds of adaptivity (b) The corresponding semantics-based dependency graph (c) The estimated dependency graph from AdaptFun.
4 The Adaptyvity Analysis Algorithm - AdaptFun

In this section, we present our static program analysis for computing an upper bound on the adaptivity of a given program $c$.

4.1 A guide to AdaptFun

In order to have a sound and accurate upper bound on the adaptivity of a program $c$, we design a program analysis framework named AdaptFun. This framework composes two algorithms as shown in the double-stroke box and the dashed box in Fig. 3. The first algorithm in the double-stroke box combines the quantitative and dependency analysis techniques. It produces an estimated data-dependency graph for a program. The second algorithm in the dashed box is a walk length estimation algorithm. It computes the upper bound on the program’s adaptivity over the estimated graph. Below is the outline of the AdaptFun.

1. **Graph Estimation** Because adaptivity is defined over a program’s quantitative dependency graph (in Definition 8), this algorithm first estimates this graph for the program statically in Section 4.4. It estimates the four components of this graph in two steps and then composes them into an estimated dependency graph in the last step. The steps are summarized as follows.

   (a) **Vertex and Query Annotation Estimation** Vertices and query annotations in this graph are the assigned variables with unique labels. These are extracted directly from the program as in Section 4.2.

   (b) **Edge and Weight Estimation**

   This step estimates the edge and weight for a quantitative dependency graph. It combines the control, data-flow analysis algorithm and the loop bound inference algorithm. There are three computation steps in this algorithm.

   **Abstract Control Flow Graph.** In order to perform the dependency analysis and quantitative analysis, this step first generates an abstract control flow graph for a program in Section 4.3.1.

   **Edges Estimation via Combined Flow Analysis.** The step is presented in Section 4.3.2. It performs over the abstract control flow graph, which combines both control flow and data flow analysis. It estimates the dependency relation between each pair of the labeled variables in a program by considering both the control flow and data flow. Then it uses the estimated dependency relation to approximate the edge between each pair of vertices.

   **Weights Estimation via Quantitative Analysis.** This step is presented in Section 4.3.3. It performs over the same abstract control flow graph and computes the upper bound on the maximal visiting times of each labeled variable for a program. It estimates the reachability bound for every vertex over the abstract control flow graph, and this reachability bound is
used to estimate the maximal visiting times of each labeled variable in a program and the weight of the corresponding vertex.

(c) **Graph Estimation.** In Section 4.4, we construct the final approximated graph, named *estimated dependency graph* by simply composing the four estimated ingredients. Overall, this *estimated dependency graph* has a similar topology structure as the *semantics-based dependency graph*. It has the same vertices and query annotations, but approximated edges and weights.

2. **Adaptivity Computation.** Likewise the adaptivity in Definition 14, the static estimation on the adaptivity also relies on finding a walk in the *estimated dependency graph*. We discuss some challenges in finding the ‘appropriate’ walk in the graph, and how our algorithm responds to these challenges as in Section 4.5.

### 4.2 Vertex and Query Annotation Estimations

**Vertex Estimation** The first component of the *estimated dependency graph* is the vertex set, which is identical to the *semantics-based dependency graph*. Every vertex is an assigned variable in the program, which comes from an assignment command or query request command with a unique label. These vertices are collected by statically scanning the program, like what we do for vertices of the *semantics-based dependency graph*, as follows.

\[
V_{\text{est}}(c) \triangleq \{ x_l \in LV \mid x_l \in LV(c) \}
\]

where \(A_{\text{in}}\) is the set of arithmetic expressions over \(\mathbb{N}\) and program’s input variables.

**Query Annotation Computation** The static scanning of the programs also tells us whether one vertex (assigned variable) is assigned by a query request. Identically to the *semantics-based dependency graph*, \(Q_{\text{est}}(c)\) is a set of pairs \(Q_{\text{est}}(c) \in \mathcal{P}(LV \times \{0,1\})\) mapping each \(x^l \in V_{\text{est}}(c)\) to either 0 or 1. 1 denotes \(x^l\) is a member of \(QV_c\), which is the set of program’s variables assigned with query requests, and 0 means \(x^l\) not in this set. It is defined formally below.

\[
Q_{\text{est}}(c) = \{ (x^l, n) \in LV \times \{0,1\} \mid x^l \in LV(c), n = 1 \iff x^l \in QV_c \land n = 0 \iff x^l \notin QV_c \}
\]

### 4.3 Edge and Weight Estimation

The edges and weight are estimated through a combined control, data flow, and loop bound analysis. Because these analyses are all performed on basis of the *Abstract Transition Graph* of the program, we first introduce how to generate this *abstract transition graph* in Section 4.3.1. Then Section 4.3.2 presents the edge estimation based on a combined control and data flow analysis algorithm, and Section 4.3.3 computes the weight through a loop bound analysis.

#### 4.3.1 Abstract Transition Graph

This section shows how to generate the abstract transition graph \(ab\mathcal{G}(c)\) of a program \(c\) through constructing its vertices and edges.

An *Abstract Transition Graph*, \(ab\mathcal{G}(c)\) for a program \(c\) is composed of a vertex set \(ab\mathcal{V}(c)\) and an edge set \(ab\mathcal{E}(c)\), \(ab\mathcal{G}(c) \triangleq (ab\mathcal{V}(c), ab\mathcal{E}(c))\). Every vertex \(I \in ab\mathcal{V}(c)\) is the label of a labeled command in \(c\), which is unique. We also call the
unique label as program point.
Each edge \((l \xrightarrow{dc} l') \in \text{absE}(c)\) is an abstract transition between two program points \(l, l'\). There is an edge from \(l\) to \(l'\) if and only if the command with label \(l'\) can execute right after the execution of the command with label \(l\). Each edge is annotated by a constraint \(dc \in \mathcal{DC}^\top\), which is generated from the command with label \(l\). This constraint describes the abstract execution of the command with \(l\).

Abstract Control Flow Graph Vertices Construction  Every vertex \(l \in \text{absV}(c)\) corresponds to a program point \(l\), which is a unique label of a command in this program. Concretely, the vertices of this graph is the set of \(c\)’s labels with the exit label \(\text{ex}\) formally as follows,

\[
\text{absV}(c) = \mathcal{L}(c) \cup \{\text{ex}\}
\]

Abstract Control Flow Graph Edge Construction  Each edge \((l \xrightarrow{dc} l') \in \text{absE}(c)\) is an abstract transition between two program points \(l, l'\). There is an edge from \(l\) to \(l'\) if and only if the command with label \(l'\) can execute right after the execution of the command with label \(l\). Each edge is annotated by a constraint \(dc \in \mathcal{DC}^\top\) generated from the command with label \(l\). This constraint describes the abstract execution of the command with \(l\). This step shows how to generate the abstract transition graph \(\text{absG}(c)\) of a program \(c\) through constructing its vertices and edges.

The vertices can be easily collected and the key point of the abstract transition graph for a program is constructing the edge set, \(\text{absE}(c)\) for a program \(c\). It relies on the control flow analysis and the program abstraction of each command. To make it easy to understand, it is an enriched control flow graph with an annotation on each edge. The edge set is constructed by a program abstraction method in three steps.

In the first step, Constraint Computation generates a constraint over the expression for every program’s labeled command, which is used as the annotation of an edge.

In the second step, Initial and Final State Computation generates two sets for each command. The initial state is a set that contains the program point where this command starts executing, and the final state is a set that contains the constraint of this command and the continuation program points after the execution of this command.

In the third step, Abstract Event Computation generates a set of edges for the program. Each edge is a pair of initial and final state.

Constraint Computation  In this step, we first show how to compute the constraints for expressions in a program \(c\), by a program abstraction method adopted from the algorithm in Section 6 in [6]. Given a program \(c\), every expression in an assignment command or in the guard of a if or while command is transformed into a constraint.

Notations / Formal Definitions:

- Operator: \(\text{absexpr} : \mathcal{A} \cup \mathcal{B} \rightarrow \mathcal{DC}(\mathcal{VAR} \cup \mathcal{SMBCST}) \cup \mathcal{B} \cup \{\top\}\)

- Constrains, \(\mathcal{DC}^\top\) is composed of the Difference Constraints \(\mathcal{DC}(\mathcal{VAR} \cup \mathcal{SMBCST})\), the Boolean Expressions \(\mathcal{B}\) and \(\top\).
  - The difference constraints \(\mathcal{DC}(\mathcal{VAR} \cup \mathcal{SMBCST})\) is the set of all the inequality of form \(x' \leq y + v\) or \(x' \leq v\) where \(x \in \mathcal{VAR}\), \(y \in \mathcal{VAR}\) and \(v \in \mathcal{SMBCST}\). The Symbolic Constant set \(\mathcal{SMBCST} = \mathbb{N} \cup \mathcal{VAR}_{\mathbb{R}} \cup \{\infty\} \cup Q_m\) is the set of natural numbers with \(\infty\), the input variables, and a symbol \(Q_m\) representing the abstract value of a query request. An inequality
\[ x' \leq y + v \] describes that the value of \( x \) in the current state is at most the value of \( y \) in the previous state plus the symbolic constant \( v \). An inequality \( x' \leq v \) describes that the value of \( x \) in the current state is at most the value \( v \). When a difference constrain shows up as an edge annotation, \( l \xrightarrow{x'\leq y + v} l' \), it denotes that the value of variable \( x \) after executing the command at \( l \) is at most the value of variable \( y \) plus \( v \) before the execution, and \( l \xrightarrow{x'\leq v} l' \) respectively denotes value of variable \( x \) after executing the command at \( l \) is at most the value of the symbolic constant \( v \) before the execution. For every expression in each of the label command, it is computed in three steps via program abstraction method adopted from the Section 6 in [6].

- The Boolean Expressions \( b \) from the set \( \mathcal{B} \). \( b \) on an edge \( l \xrightarrow{b} l' \) describes that after evaluating the guard with label \( l \), \( b \) holds and the command with label \( l \) will execute right after.
- The top constraint, \( \top \) denotes true. It is preserved for \( \text{skip} \) command, or commands that don’t involve any counter variable.

Computation Steps:

**Definition 15** (Constraint Computation). For a program \( c \), a boolean expression \( b \) in the guard of a if or while command or an expression \( e \) and a variable \( x \) in an assignment command \( x \leftarrow e \), the constraint \( \text{absexpr}(b,_) \) or \( \text{absexpr}(x - v, x) \) is computed as follows,

\[
\begin{align*}
\text{absexpr}(x - v, x) &= x' \leq x - v & x \in \mathcal{VAR}_{\text{guard}} \land v \in \mathbb{N} \\
\text{absexpr}(y + v, x) &= x' \leq y + v & x \in \mathcal{VAR}_{\text{guard}} \land v \in \mathbb{Z} \land y \in (\mathcal{VAR}_{\text{guard}} \cup \mathcal{SMBCST}) \\
\text{absexpr}(x, x) &= x' \leq y & x \in \mathcal{VAR}_{\text{guard}} \land v \in (\mathcal{VAR}_{\text{guard}} \cup \mathcal{SMBCST}) \\
\forall \mathcal{VAR}_{\text{guard}} = \mathcal{VAR}_{\text{guard}} \cup \{y\} & x \in \mathcal{VAR}_{\text{guard}} \land v \in \mathbb{Z} \land y \notin (\mathcal{VAR}_{\text{guard}} \cup \mathcal{SMBCST}) \\
\text{absexpr}(\psi, x) &= x' \leq Q_m & x \in \mathcal{VAR}_{\text{guard}} \land \psi \text{ is a query expression} \\
\text{absexpr}(e, x) &= x' \leq \infty & x \in \mathcal{VAR}_{\text{guard}} \land e \text{ doesn’t have any of the forms as above} \\
\text{absexpr}(\top, x) &= \mathcal{VAR}_{\text{guard}} \\
\text{absexpr}(b,_) &= b \\
\forall \mathcal{VAR}_{\text{guard}} = \mathcal{VAR}_{\text{guard}} \cup \text{FV}(b) & x \in \mathcal{VAR}_{\text{guard}} \land b \text{ is a boolean expression}
\end{align*}
\]

\( \forall \mathcal{VAR}_{\text{guard}} \) is the set of variables used in the guard expression of every while command in the program \( c \). In the case 4, if a variable \( x \), belonging to the set \( \forall \mathcal{VAR}_{\text{guard}} \) is updated by a variable \( y \), which isn’t in this set, we add \( y \) into the set \( \forall \mathcal{VAR}_{\text{guard}} \) and repeat above procedure until \( \forall \mathcal{VAR}_{\text{guard}} \) and \( \text{absexpr}(e, x) \) is stabilized.

Specifically we handle a normalized expression, \( x > 0 \) in guards of while loop headers, and the counter variable \( x \) only increase, decrease or reset by simple arithmetic expression (mainly multiplication, division, minus and plus (able to extend to max and min)). The counter variable \( x \) is generalized into norm when the boolean expression \( x > 0 \) in \( \text{while} \) doesn’t have the form \( x > 0 \). The way of normalizing the guards and computing the norms is adopted from the computation step 1 in Section 6.1 in paper [6].

**Definition 16** (Symbolic Expression (\( \mathcal{A}_S \))). \( \mathcal{A}_S \) is the set of all the symbolic expressions over \( \text{SMBCST} \).

The symbolic expression set is a subset of arithmetic expressions over \( \mathbb{N} \) with input variables, i.e., \( \mathcal{A}_S \subseteq \mathcal{A}_{in} \).
Abstract Initial and Final State Computation  This step computes two sets for each command. The initial state is a set that contains the program points before executing this command, which is computed by the standard initial state generation method from control flow analysis. The final state is a set that contains the constraint of this command and the program points after the execution of this command. This set is enriched from the standard control flow analysis.

Notations / Formal Definitions:

- The abstract initial state: \( \text{absinit}(c) \in L \)
- The abstract Final State: \( \text{absfinal}(c) \in P(L \times DC^T) \)

Computation Steps:

- The abstract initial state, \( \text{absinit}(c) \in P(L) \) for a command \( c \) is the set of the initial program points. Each point in this set is a unique program label corresponds to the command before executing this command.

Given a program \( c \), its abstract initial state, \( \text{absinit}(c) \) is computed as follows,

\[
\begin{align*}
\text{absinit}([x \leftarrow e]) &= \{l\} \\
\text{absinit}([x \leftarrow \text{query}(\psi)]) &= \{l\} \\
\text{absinit}([\text{skip}]) &= \{l\} \\
\text{absinit}([b] \text{ then } c_1 \text{ else } c_2) &= \{l\} \\
\text{absinit}([\text{while} \ b \ do \ c]) &= \{l\} \\
\text{absinit}(c_1; c_2) &= \text{absinit}(c_1)
\end{align*}
\]

- The abstract final state of the program \( c \), \( \text{absfinal}(c) \in P(L) \times DC^T \times L \) is a set of pairs, \((l, dc)\) with a program point (i.e., a label), \( l \) as the first component and a constraint, \( dc \) as the second component. The program point \( l \) corresponds to the labeled command after the execution of \( c \), and the constraint \( dc \) in this pair is computed by \( \text{absexpr} \) for the expression in \( c \).

Given a program \( c \), its final state, \( \text{absfinal}(c) \) is computed as follows,

\[
\begin{align*}
\text{absfinal}([x \leftarrow e]) &= \{(l, \text{absexpr}(e, x))\} \\
\text{absfinal}([x \leftarrow \text{query}(\psi)]) &= \{(l, x' \leq 0 + Q_m)\} \\
\text{absfinal}([\text{skip}]) &= \{(l, T)\} \\
\text{absfinal}([b] \text{ then } c_1 \text{ else } c_2) &= \text{absfinal}(c_1) \cup \text{absfinal}(c_2) \\
\text{absfinal}([\text{while} \ b \ do \ c]) &= \{(l, \text{absexpr}(b, T))\} \\
\text{absfinal}(c_1; c_2) &= \text{absfinal}(c_2)
\end{align*}
\]

Abstract Event Computation  Each abstract event is an edge between two vertices in the abstract transition graph. It is generated by computing the initial state and finial state interactively and recursively for a program \( c \).

Notations / Formal Definitions:

- Abstract Event: \( \delta \in L \times DC^T \times L \)
- Abstract Event Computation: \( \text{abstrace} \in C \rightarrow P(L \times DC^T \times L) \)

Its type is defined as follows,
Definition 17 (Abstract Event). Abstract Event: \( e \in \mathcal{L} \times \mathcal{D}^{\top} \times \mathcal{L} \) is a triple where the first and third components are labels, second component is a constraint from \( \mathcal{D}^{\top} \).

In an abstract event \( (l, dc, l') \) of a program \( c \), the first label \( l \in \mathcal{L} \) corresponds to an initial state of \( c \), and the second label \( l' \in \mathcal{L} \) with the constraint \( dc \in \mathcal{D}^{\top} \) correspond to an abstract final state of \( c \). The abstract initial state is a label from \( \mathcal{L} \). We abuse the notation \( \mathcal{P}(\hat{e}) \) for the power set of all abstract events.

Computation Steps:
The set of the abstract events \( \text{abstract}(c) \) for a program \( c \) is computed as follows in Definition[18]

Definition 18 (Abstract Event Computation). \( \text{abstract} \in \mathcal{C} \to \mathcal{P}(\mathcal{L} \times \mathcal{D}^{\top} \times \mathcal{L}) \)

We first append a skip command with the label \( \text{ex} \), i.e., \([\text{skip}]_{\text{ex}}\) at the end of the program \( c \), and construct the program \( c' = c; [\text{skip}]_{\text{ex}} \). Then, we compute the \( \text{abstract}(c) = \text{abstract}'(c') \) for \( c' \) as follows,

\[
\begin{align*}
\text{abstract}'(l \leftarrow e)[l'] &= \emptyset \\
\text{abstract}'(l \leftarrow \text{query}(\psi))[l'] &= \emptyset \\
\text{abstract}'([\text{skip}])[l'] &= \emptyset \\
\text{abstract}'([\text{if} \, [b] \, \text{then} \, c_l \, \text{else} \, c_f])[l'] &= \text{abstract}'(c_l) \cup \text{abstract}'(c_f) \\
&\quad \cup \langle (l, \text{absexpr}(b, T), \text{absinit}(c_l)), (l, \text{absexpr}(\neg b, T), \text{absinit}(c_f)) \rangle \\
\text{abstract}'([\text{while} \, [b] \, \text{do} \, c_{\mu}])[l'] &= \text{abstract}'(c_{\mu}) \cup \langle (l, \text{absexpr}(b, T), \text{absinit}(c_{\mu})) \rangle \\
&\quad \cup \langle (l', dc, l')(l', dc) \in \text{absexpr}(c_{\mu}) \rangle \\
\text{abstract}'(c_l, c_2) &= \text{abstract}'(c_1) \cup \text{abstract}'(c_2) \\
&\quad \cup \langle (l, dc, \text{absinit}(c_2)) \rangle \quad (l, dc) \in \text{absexpr}(c_1) \\
\end{align*}
\]

Notice \( \text{abstract}'(l \leftarrow e)[l'] \), \( \text{abstract}'(l \leftarrow \text{query}(\psi))[l'] \) and \( \text{abstract}'([\text{skip}])[l'] \) are all empty sets.

Theorem Guarantee: For every event \( e \) with label \( l \) in an execution trace \( \tau \) of program \( c \), there is an abstract event in program’s abstract execution trace of form \((l, \_, \_)\). This soundness is presented below with the proof in Appendix[17]

Lemma 4.1 (Soundness of the Abstract Events). For every program \( c \) and an execution trace \( \tau \in \mathcal{T} \) that is generated w.r.t. an initial trace \( \tau_0 \in \mathcal{E}_0(c) \), there is an abstract event \( \hat{e} = (l, \_, \_) \in \text{abstract}(c) \) for every event \( e \in \tau \) having the label \( l \), i.e., \( e = (l, \_, \_) \). \( \forall e \in \mathcal{C}, \tau_0 \in \mathcal{E}_0(c), \tau \in \mathcal{T}, \exists \hat{e} = (l, \_, \_) \in \text{absinit}(\tau_0) \land \epsilon \in \tau \)

For every program point \( l \) corresponding to an assignment command in a program \( c \), there is a unique abstract event in the program’s abstract events set \( \hat{e} \in \text{abstract}(c) \) of form \((l, \_, \_)\).

Lemma 4.2 (Uniqueness of the Abstract Events Computation). For every program \( c \) and an execution trace \( \tau \in \mathcal{T} \) that is generated w.r.t. an initial trace \( \tau_0 \in \mathcal{E}_0(c) \), there is a unique abstract event \( \hat{e} = (l, \_, \_) \in \text{abstract}(c) \) for every assignment event \( e \in \mathcal{E}_{\text{ass}} \) in the execution trace having the label \( l \), i.e., \( e = (l, \_, \_) \) and \( e \in \tau \).

\( \forall e \in \mathcal{C}, \tau_0 \in \mathcal{E}_0(c), \tau \in \mathcal{T}, e = (l, \_, \_) \in \mathcal{E}_{\text{ass}}, \langle c, \tau_0 \rangle \to^e \langle \text{skip}, \tau_0, \tau \rangle \land e \in \tau \)

This lemma is proved in Appendix[17]
The edge for the program’s abstract events set, abstract(c) as follows,

\[ \text{abstract}(c) = \{(l_1, dc, l_2)| (l_1, dc, l_2) \in \text{abstract}(c)\} \]

Abstract Transition Graph Construction With the vertices absV(c) and edges absE(c) ready, we construct the abstract transition graph, formally in Definition 19.

Definition 19 (Abstract Transition Graph). Given a program c, its abstract transition graph absG(c) = (absV(c), absE(c)) is computed as follows,

\[ \text{abstract}(c) = \{(l_1, dc, l_2)| (l_1, dc, l_2) \in \text{abstract}(c)\}, \]

\[ \text{abstract}(c) = L \cup \{\text{ex}\} \]

Example The edge (1 $\xrightarrow{j \leq k}$ 2) on the top tells us the command \( j \leftarrow k \) is executed with a continuation point 2 such that the guard \( j > 0 \) will be evaluated next. The annotation $j' \leq k$ is a difference constraint computed for the expression \( k \) from the assignment command \( j \leftarrow k \). It represents that the value of \( j \) is less than or equal to value of input variable \( k \) after the execution of \( a \leftarrow 0 \) and before executing the loop. The boolean constraint \( j \leq 0 \) on the edge 2 $\xrightarrow{j \leq 0}$ 6 represents the negation of the testing guard \( j > 0 \) of the while command with header at label 2.

4.3.2 Edge Estimation

Since the edges of the semantics-based graph of a program relies on the dependency relation, it contains both control flow and data flow. In this sense, We first develop a feasible data-flow relation to estimate the data dependency relation, which catches these two flows. Then we construct the edges for \( G_{est}(c) \) based on this feasible data-flow relation. This algorithm named Feasible Data-Flow Generation. It considers both the control flow and data flow and is a sound approximation of the edges in the semantics based dependency graph. The three steps in this algorithm is summarized as follows,

1. The Reaching Definition analysis computes a set of labeled variables, RD(l, c) for every label l in c over its abstract control flow graph, absG(c). The computation performs the standard reaching definition analysis and working-list algorithm over the abstract control flow graph, absG(c). RD(l, c) contains all the labeled variables which are reachable at program point l.
2. The Feasible Data-Flow computation combines the $\text{RD}(l, c)$, $\text{absG}(c)$ and data flow analysis. It computes the feasible data-flow relation, $\text{flowsTo}(x^l, y^l, c)$ for each pair of the $c$’s labeled variables, $x^l, y^l \in L \forall (c)$ in Definition 20. $\text{flowsTo}(x^l, y^l, c)$ is a sound approximation of the variable may-dependency relation, $\text{DEP}_{\text{var}}(x^l, y^l, c)$ for every $x^l, y^l \in L \forall (c)$. The formal proof is in the Appendix. We also discuss that the combined analysis gives more precise approximation on the data may-dependency than single analysis in Appendix.

3. Edge Estimation Using the $\text{flowsTo}(x^l, y^l, c)$ relation, we define the estimated directed edges as set of pairs of vertices $x^l, y^l \in V_{\text{est}}(c), E_{\text{est}}(c) \in P(L \forall \times L \forall)$. Every $\text{flowsTo}(x^l, y^l, c)$ relation indicates a directed edge from the $x^l$ to $y^l$ if and only if it is true.

The details are as follows.

Reaching definition analysis  This part performs the standard reaching definition analysis given a program $c$, on every label in $\text{absV}(c)$. This step generates set of all the reachable variables at location of label $l$ in the program $c$. The $\text{RD}(l, c)$ represent the analysis result, which is the set of reachable labeled variables in program $c$ at the location of label $l$. For every labelled variable $x^l$ in this set, the value assigned to that variable in the assignment command associated to that label is reachable at the point of executing the command of label $l$. It is computed in five steps as follows,

1. The block, is either the command of the form of assignment, skip, or a test of the form of $[b]^l$, denoted by $\text{blocks}(c)$ the set of all the blocks in program $c$, where $\text{blocks} : C \rightarrow P(C \cup [b]^l)$. A block is either the command of the form of assignment, skip, or test of the form of $[b]^l$.
   The operator $\text{blk} : C \rightarrow \text{blocks}$ gives all the blocks in program $c$.
   Set $? = \text{undefined}.

2. The operator $\text{kill} : \text{blocks} \rightarrow P(V \forall \mathcal{R} \times L \cup [?])$ produces the set of labelled variables of assignment destroyed by the block.

3. The operator $\text{gen} : \text{blocks} \rightarrow P(V \forall \mathcal{R} \times L \cup [?])$ generates the set of labelled variables generated by the block.

4. The operator $\text{in}(l), \text{out}(l) : L \rightarrow L \forall \cup [?]$ for every block in program $c$ is defined as follows,

   \[
   \begin{align*}
   \text{in}(l) & = \{x^l | x^l \in L \forall, l = \text{absinit}(c) \cup \{\text{out}(l')|l' \in \text{absE}(c) \land l \neq \text{absinit}(c)\} \\
   \text{out}(l) & = \text{gen}(B^l) \cup \{\text{in}(l) \setminus \text{kill}(B^l)\}
   \end{align*}
   \]

   computing $\text{in}(l)$ and $\text{out}(l)$ for every $B^l \in \text{blocks}(c)$, and repeating these two steps until the $\text{in}(l)$ and $\text{out}(l)$ are stabilized for every $B^l \in \text{blocks}(c)$ We use $\text{RD}(l, c)$ to represent denote the stabilized result of $\text{in}(l)$ at label $l$ in program $c$.

5. The stabilized $\text{in}(l)$ and $\text{out}(l)$ for program $c$, as well as $\text{RD}(l, c)$, is computed by the standard work-list algorithm with detail as below.

   (a) $\text{initialize} \ \text{in}[0]=\text{out}[0]=\emptyset; \ \text{in}[0] = \emptyset$

   (b) $\text{initialize} \ \text{a work queue} \ W$, contains all the blocks in $c$

   (c) while $|W| \neq 0$

   \[
   \begin{align*}
   \text{pop} \ l \ \text{in} \ W & \\
   \text{old} & = \text{out}[l]
   \end{align*}
   \]

20
Another edge

This part presents the computation of the feasible data-flow relation between each pair of labeled variables in a program $c$, formally in Definition 20.

**Definition 20** (Feasible Data-Flow). Given a program $c$ and two labeled variables $x^i, y^j$ in this program, $\text{flowsTo}(x^i, y^j, c)$ is

\[
\begin{align*}
\text{flowsTo}(x^i, y^j, [y \leftarrow e]) & \triangleq (x^i, y^j) \in ((x^i, y^j) | x \in \text{FV}(e) \land x^i \in \text{RD}(l, [y \leftarrow e]^l)) \\
\text{flowsTo}(x^i, y^j, [y \leftarrow \text{query}(\psi)]) & \triangleq (x^i, y^j) \in ((x^i, y^j) | x \in \text{FV}(\psi) \land x^i \in \text{RD}(l, [y \leftarrow \text{query}(\psi)]^l)) \\
\text{flowsTo}(x^i, y^j, \text{skip}) & \triangleq \emptyset \\
\text{flowsTo}(x^i, y^j, \text{if } (b^1, c_1, c_2)) & \triangleq \text{flowsTo}(x^i, y^j, c_1) \lor \text{flowsTo}(x^i, y^j, c_2) \\
& \lor ((x^i, y^j), x \in \text{FV}(b) \land x^i \in \text{RD}(l), \text{if } (b^1, c_1, c_2)) \land y^j \in \text{LV}(c_1) \\
& \lor ((x^i, y^j), x \in \text{FV}(b) \land x^i \in \text{RD}(l), \text{if } (b^1, c_1, c_2)) \land y^j \in \text{LV}(c_2) \\
\text{flowsTo}(x^i, y^j, \text{while } (b^1, c_w)) & \triangleq \text{flowsTo}(x^i, y^j, c_w) \\
& \lor ((x^i, y^j), x \in \text{FV}(b) \land x^i \in \text{RD}(l), \text{while } (b^1, c_w)) \land y^j \in \text{LV}(c_w) \\
\text{flowsTo}(x^i, y^j, c_1, c_2) & \triangleq \text{flowsTo}(x^i, y^j, c_1) \lor \text{flowsTo}(x^i, y^j, c_2)
\end{align*}
\]

We prove that the transitive closure of the feasible data-flow relation is a sound approximation of the variable May-Dependency relation over labeled variables for every program, in Appendix B.

**Improvement Analysis.** Combining the result of reaching definition, $\text{RD}(l, c)$ with the abstract control flow graph, $\text{absG}(c)$ with control flow analysis into the feasible data-flow generation improves the data-dependency relation approximation accuracy. For example, a program $\{x = 0; x = 2; y = x + 1\}^3$. The standard data flow analysis tells us that both the labeled variable $x^1$ and $x^2$ may flow to $y^3$, which will result in an unnecessary edge $(x^1, y^3)$. The result of reaching definition can help us eliminate this kind of edge by telling us, at line 3, only variable $x^2$ is reachable.

**Edge Estimation** The edge estimation is based on the $\text{flowsTo}(x^i, y^j, c)$ relation. For each pair of vertices $x^i, y^j$ in $V_{\text{est}}(c)$, there is a directed edge from $x^i$ to $y^j$ if and only if they have feasible flows-to relation, i.e., $\text{flowsTo}(x^i, y^j, c)$ is true. Using the $\text{flowsTo}(x^i, y^j, c)$ relation, we define the estimated directed edges as a set which contains all pairs of vertices $x^i, y^j$ in $V_{\text{est}}(c)$, $E_{\text{est}}(c) \in \mathcal{P}(\mathcal{L} \times \mathcal{L})$ satisfying this relation formally as follows,

\[
E_{\text{est}}(c) \triangleq \{(y^j, x^i) \mid y^j, x^i \in V_{\text{est}}(c) \land \exists n, z_1^n, \ldots, z_n^n \in \mathcal{L}(c) . \land n \geq 0 \land \text{flowsTo}(x^i, z_1^n, c) \land \cdots \land \text{flowsTo}(z_n^n, y^j, c)\}
\]

We prove that this estimated directed edge set $E_{\text{est}}(c)$ is a sound approximation of the edge set in $c$'s semantics-based dependency graph by Lemma B.2 in Appendix B.

**Example** As in the Figure 3(c), the edge $l^6 \rightarrow a^5$ is built by $\text{flowsTo}(l^6, a^5, c)$ relation because $a$ is used directly in the query expression $\chi[k] * a$ in the command $[l \leftarrow \text{query}(\chi[k] * a)]^6$, i.e., $a \in \text{FV}(\chi[k] * a)$. And we also have $a^5 \in \text{RD}(6, \text{twoRounds}(k))$ from the reaching definition analysis. Another edge $x^3 \rightarrow j^5$ in the same graph represents the control flow from $j^5$ to $x^3$, which is soundly caught by our $\text{flowsTo}$ relation.
4.3.3 Weight Estimation

This section presents the quantitative analysis algorithm, which performs over the same abstract control flow graph, \( \text{absG}(c) \) of a program \( c \) as well. As the \( \text{W}_{\text{trace}}(c) \) defined in Definition 8 the weight of every \( x^I \in \text{V}_{\text{est}}(c) \) is the execution times of the command with label \( l \). In this sense, to estimate weight of \( x^I \), this step first computes an upper bound, the reachability-bound \( [3] \) for every \( l \in \text{absV}(c) \) on the execution times of the command with label \( l \). Then, the reachability-bound is used to estimate the maximal visiting times of the labeled variable \( x^I \in \text{LV}(c) \) and the weight of the vertex \( x^I \in \text{V}_{\text{est}}(c) \). The two computation steps are summarized as follows,

1. **Reachability Bound Analysis** As the \( \text{W}_{\text{trace}}(c) \) defined in Definition 8, the weight of every \( x^I \in \text{V}_{\text{est}}(c) \) is the execution times of the command with label \( l \). In this sense, to estimate weight of \( x^I \), this step first computes an upper bound, the reachability-bound for every \( l \in \text{absV}(c) \) on the execution times of the command with label \( l \). Then, the reachability bound on \( l \in \text{absV}(c) \) is used to estimate the weight of the vertex \( x^I \in \text{V}_{\text{est}}(c) \).

2. **Weight Estimation** Because the vertex in program’s \( \text{absG}(c) \) shares the same unique label with the vertex in \( \text{V}_{\text{est}}(c) \), we use the reachability-bound on the vertex \( l \in \text{absV}(c) \) directly as the weight of the vertex \( x^I \) in \( \text{V}_{\text{est}}(c) \).

**Step-1: Reachability Bound Analysis** This symbolic reachability bound analysis performs over the same abstract control flow graph, \( \text{absG}(c) \) of a program \( c \). It first computes a reachability bound for every edge \( l \xrightarrow{dc} l' \in \text{absE}(c) \), which is a symbolic bound on the maximum execution times of the command with label \( l \) of \( c \). Then the reachability bound for edge \( l \xrightarrow{dc} l' \) is used as the bound on the maximum visiting times of the vertex \( l \in \text{absV}(c) \). It is a sound upper bound on the visiting times of every label \( l \in \text{absV}(c) \), named reachability-bound. The computation steps are summarized as follows,

1. It first collects three edge sets for each variable, in which the variable increases, decreases and reset respectively.

2. Then, for each edge \( l \xrightarrow{dc} l' \in \text{absE}(c) \), it assigns a variable \( x \) (or a symbolic constant \( c \in \text{SMBCST} \)) if \( x \) (or \( c \)) decreases in \( dc \), as this edge’s local bound.

3. It then computes the bound on the maximum value of the local bound for each edge, and the reachability-bound on the execution times of the corresponding edge recursively.

4. The last step uses reachability-bound \( w \) for edge \( l \xrightarrow{dc} l' \) as the bound on the maximum visiting times of the vertex \( l \in \text{absV}(c) \) and generates a set \( \text{absW}(c) \) contains a pair \((l, w)\) for every \( l \in \text{absV}(c) \).

The algorithm in this step is inspired from the Algorithm.2 in paper [5], the Algorithm.3 in paper [7], and the Definition.25 in Section 4 of paper [6].

- Algorithm.3 in paper [7] assigns a set of variables to each transition in which these variables decrease as the local bound and estimates the maximum value each variable in this set.
- Algorithm.2 in paper [5] assigns a variable to each edge on which this variable decrease as its ranking function and then estimates the maximum value for the ranking function.
- The Definition.25 in paper [6] assigns each transition with a variable that decreases in this transition, as the local bound and computes the bound similarly.
The computation steps are as follows,

1. **Variable Modifications** For each variable \( x \) in a program \( c \), this step computes three edge sets, \( \text{inc}(c, x) \), \( \text{dec}(c, x) \), and \( \text{re}(c, x) \) for \( x \). Every edge in a set corresponds to a transition in which \( x \) is increased, decreased or reset respectively.

\[
\begin{align*}
\text{inc} & : \mathit{VAR} \rightarrow \mathcal{P}(\mathit{VAR}) \rightarrow \mathcal{P}(\hat{e}) \text{ is the set of the edges where the variable increase,} \\
\text{inc}(c, x) & = \{ e | e = (l, l', x' \leq x + v) \wedge \hat{e} \in \text{abstrace}(c) \} \\
\text{dec} & : \mathit{VAR} \rightarrow \mathcal{P}(\mathit{VAR}) \rightarrow \mathcal{P}(\hat{e}) \text{ is the set of abstract events where the variable decrease,} \\
\text{dec}(c, x) & = \{ e | e = (l, l', x' \leq x - v) \wedge \hat{e} \in \text{abstrace}(c) \} \\
\text{re} & : \mathit{VAR} \rightarrow \mathcal{P}(\mathit{VAR}) \rightarrow \mathcal{P}(\hat{e}) \text{ is the set of the abstract events where the variable is reset,} \\
\text{re}(c, x) & = \{ e | e = (l, l', x' \leq y - v) \wedge x \neq y \wedge \hat{e} \in \text{abstrace}(c) \} \\
\text{rechain} & : \mathit{VAR} \rightarrow \mathcal{P}(\mathit{VAR}) \rightarrow \mathcal{P}(\hat{e}) \text{ is the set of the chain of abstract events where the variable is reset through the chain.} \\
\text{rechain}(c, x) & = \{ e | e = (l, l', x' \leq y - v) \wedge x \neq y \wedge \hat{e} \in \text{abstrace}(c) \}
\end{align*}
\]

In addition to collect the edge set that \( x \) is reset on every edge in this set, i.e., compute the \( \text{re}(c, x) \), we also compute a set, \( \text{rechain}(c, x) \) contains sequences of edges for \( x \) based on the Definition 20 in [6]. In each sequence, \( (e_0, \ldots, e_m) \in \text{rechain}(c, x) \) a variable \( x_i \) is reset by another variable \( x_{i+1} \) on edge \( e_i \) and \( x_{i+1} \) is reset on edge \( e_{i+1} \) recursively for every \( i = 0, \ldots, m-1 \). \( x \) is reset on the first edge \( e_0 \) of every sequence in \( \text{rechain}(c, x) \). Rephrase: Each edge \( e_i \) in a sequence \( (e_0, \ldots, e_m) \in \text{rechain}(c, x) \) resets a variable \( x_i \) by another variable \( x_{i+1} \) such that \( x_{i+1} \) is reset on edge \( e_{i+1} \) recursively. The first edge \( e_0 \) of each sequence resets the variable \( x \).

In the following steps, \( c \) is omitted in \( \text{inc}(x) \), \( \text{dec}(x) \) and \( \text{re}(x) \) for concise when the reference of a program \( c \) is clear in the context.

2. Assigning The Local Bound To An Edge For each edge in the transition graph \( \text{absG}(c) \) of a program \( c \), this step assigns the variable that decreases on this edge as the local bound of this edge. This step adopts the local bound computation method in Section 4 of [6] to assign the local bound to each edge, formally as follows.

**Definition 21** (Local Bound Generation). For every edge \( \hat{e} \) in the transition graph \( \text{absG}(c) \) of a program \( c \), its local bound, \( \text{locb}(\hat{e}, c) \) is the variable that decreases on this edge, computed as follows,

\[
\begin{align*}
\text{locb}(\hat{e}, c) & \triangleq 1 \text{ if } \hat{e} \notin \text{SCC}(&\text{absG}(c)) \\
& \quad \text{ or } \hat{e} \in \text{SCC}(\text{absG}(c)) \wedge \hat{e} \in \text{dec}(x) \wedge \hat{e} = (\bot, \bot, x' \leq x - v) \\
& \quad \text{ or } \hat{e} \in \text{SCC}(\text{absG}(c)) \wedge \hat{e} \notin \bigcup_{x \in \mathit{VAR}} \text{dec}(x) \wedge \hat{e} \notin \text{SCC}(\text{absG}(c) \setminus \text{dec}(x)).
\end{align*}
\]

\(\text{SCC}(\text{absG}(c))\) is the set of all the strong connected components of \( \text{absG}(c) \).

The first case is straightforward. For the label \( l \) which is not in any while loop, the labeled command with the label \( l \) will be evaluated at most once. The second and third cases are guaranteed by the Discussion on Soundness in Section 4 in [6]. We formalized the soundness and proof by Lemma [D.1] in Appendix [D].

3. **Local Bound Estimation** This step estimates the upper bound, \( \text{Vinvar}(x, c) \in A_{in} \) on the maximum value for each local bound \( x \in \mathit{VAR} \cup \mathit{SM} \cup \mathit{EST} \).

For a program \( c \), the local bound of an \( x \), \( \text{Vinvar}(\text{locb}(\hat{e}, c)) \in A_{in} \) is the bound on the maximum value of the local bound assigned to the edge \( \hat{e} \in \text{absE}(c) \), formally in Definition [22] and [23].

In order to estimate the maximum value of \( \text{locb}(\hat{e}, c) \) assigned to edge \( \hat{e} \in \text{absE}(c) \), the bound
on the iteration times of each corresponding edge, $\text{TB}(\hat{e}, c)$ is computed interactively in a path-insensitive manner.

$\text{Vinvar} : (VAR \cup SMBCST \times \mathcal{C}) \rightarrow A_{\text{in}}$

$\text{TB} : (c \times \mathcal{C}) \rightarrow A_{\text{in}}$

**Definition 22 (Local Bound Computation).** For a program $c$ and an edge $\hat{e} \in \text{absE}(c)$, the local bound, $\text{Vinvar}(\text{locb}(\hat{e}, c), c)$ for the local bound $\text{locb}(\hat{e}, c)$ of this edge is computed as follows,

\[
\text{Vinvar}(x, c) \triangleq x \quad x \in \text{SMBCST}
\]

\[
\text{Vinvar}(x, c) \triangleq \text{increase}(x, c) + \max(\{\text{Vinvar}(y, c) + \nu \mid (l, x' \leq y + \nu, l') \in \text{re}(x)) \quad x \notin \text{SMBCST}
\]

\[
\text{increase}(x, c) \triangleq \sum_{\hat{e} \in \text{inc}(x)} \text{TB}(\hat{e}, c) \times \nu \mid \hat{e} = (l, x' \leq x + \nu, l')
\]

**Definition 23 (Transition Bound).** For a program $c$ and an edge $\hat{e} \in \text{absE}(c)$, the path-insensitive transition bound, $\text{TB}(\hat{e}, c) \in A_{\text{in}}$ for this edge is computed as follows,

\[
\text{TB}(\hat{e}, c) \triangleq \text{Vinvar}(\text{locb}(\hat{e}, c), c) \quad \text{if} \quad \text{locb}(\hat{e}, c) \in \text{SMBCST}
\]

\[
\text{TB}(\hat{e}, c) \triangleq \text{increase}(x, c) + \sum_{\hat{e} \in \text{re}(x), x'} \text{TB}(\hat{e}, x'c) \times \{\text{Vinvar}(y, c) + \nu\} \quad \text{if} \quad \text{locb}(\hat{e}, c) = x \land x' \notin \text{SMBCST}
\]

Then we construct the set of reachability bound $w$ for every program point $l$, as $\text{absW}(c)$. For each pair $(l, w) \in \text{absW}(c)$, $w = \sum \{\text{TB}(\hat{e}, c) \mid \hat{e} = (l, \_ , \_ )\}$.

**Theorem Guarantee** For a program $c$ and an edge $\hat{e} \in \text{absE}(c)$, $\text{TB}(\hat{e})$ is a sound upper bound on the execution times of this transition by paper [9]. The soundness theorem is attached in Theorem [4.1] in Appendix [D].

**Theorem 4.1 (Soundness of the Transition Bound).** For each program $c$ and an edge $\hat{e} = (l, \_ , \_ ) \in \text{absE}(c)$, if $l$ is the label of an assignment command, then its path-insensitive transition bound $\text{TB}(\hat{e}, c)$ is a sound upper bound on the execution times of this assignment command in $c$.

\[
\forall c \in \mathcal{C}, l \in \mathcal{L}(c), \tau_0 \in \mathcal{I}_0(c), \tau \in \mathcal{T}, \nu \in \mathbb{N} \cdot \langle c, \tau_0 \rangle \rightarrow^* \langle \text{skip}, \tau_0 \ldots \tau \rangle \land (\text{TB}(\hat{e}, c), \tau_0) \downarrow_n v \land \text{cnt}(\tau, l) \leq v
\]

**Example** We perform the symbolic reachability bound analysis on the abstract control flow graph in Figure 4(b) and compute the result in Figure 4(c). We would like to generate the closure of every edge, which is an equality relation between variables. Solving this closure gives us the reachability bound for this edge. With all the bound for all the edges in the abstract control flow graph, we can calculate the weight for every vertex in this graph. For example, we show the closure generated for the edge $4 \xrightarrow{\frac{j \leq j - 1}{k}} 5$, $\text{TB}(4 \xrightarrow{\frac{j \leq j - 1}{k}} 5) = \text{Vinvar}(j)$. The invariant for variable $j$, $\text{Vinvar}(j)$ used here is $\text{Vinvar}(j) = k * \text{TB}(1 \xrightarrow{\frac{j \leq k}{k}} 2)$, which is generated by all the difference constraints involving $j$ in the graph. Notice the $k$ in $\text{Vinvar}(j)$ comes from considering both difference constraint $j' \leq k$ from edge $1 \xrightarrow{\frac{j \leq k}{k}} 2$ and $j' \leq j - 1$ from $4 \xrightarrow{\frac{j \leq j - 1}{k}} 5$, which intuitively reflects the while loop whose counter is set to $k$ at the beginning and decreases by 1 at each iteration. With all the closures for all the edges of the abstract control flow graph, we can solve them to obtain the reachability bound of every edge. We decide the weight for
With the four components \( V \) which we aim to estimate. Every vertex from \( V \), Vertex Weight Computation

Because the vertices in the two graph share the same unique label, \( LV \) flow graph \( absG \)

c sound approximation of the quantitative dependency graph for a program Dependency Graph. It has the same vertices but approximated edges and weights. This graph is a by soundness of each component formally in Appendix.

4.4 Graph Construction

It is formally defined in Definition 24 as follows.

**Definition 24 (Estimated Dependency Graph).** Given a program \( c \), with its abstract weighted control flow graph \( absG(c) = (absV, absE) \) and feasible data flow relation \( flowsTo(x^i, y^j, c) \) for every \( x^i, y^j \in LV(c) \), its estimated dependency graph is generated as follows,

\[
G_{est}(c) = (V_{est}(c), E_{est}(c), W_{est}(c), Q_{est}(c))
\]
Given a program \( c \), the estimated adaptivity upper bound for a program \( c \) is estimated as the length of the longest finite walk over \( W^K(G_{est}(c)) \) formally in Definition 27 and computed by Algorithm 1. \( W^K(G_{est}(c)) \) represents the set of all finite walks on \( G_{est}(c) \). Different from the finite walk on \( G_{trace}(c) \), the \( k \) does not rely on the initial trace. The occurrence time of every \( v_i \) in \( k \)’s vertices sequence is bound by an arithmetic expression \( w_i \) where \( (v_i, w_i) \in W_{est}(c) \) is \( v_i \)'s estimated weight. Then its query length \( \text{len}^q(\kappa) \) and the estimated adaptivity \( \lambda_{est}(c) \) are both arithmetic expression as well. They are formally defined as follows.

**Definition 25 (Finite Walk on estimated dependency graph (\( x \)).**

Given a program \( c \)’s estimated dependency graph \( G_{est}(c) = (V_{est}(c), E_{est}(c), W_{est}(c), Q_{est}(c)) \), a finite walk \( k \) in \( G_{trace}(c) \) is a sequence of edges \( (e_1\ldots e_{n-1}) \) for which there is a sequence of vertices \( (v_1, \ldots, v_n) \) such that:

- \( e_i = (v_i, v_{i+1}) \in E_{est}(c) \) for every \( 1 \leq i < n \).
- every vertex \( v_i \) in \( V_{est}(c) \), and \( (v_i, w_i) \in W_{est}(c) \), \( v_i \) appears in \( (v_1, \ldots, v_n) \) at most \( w_i \) times.

The length of \( k \) is the number of vertices in its vertex sequence, i.e., \( \text{len}(k) = a \).

We abuse the notation \( W^K(G_{est}(c)) \) represents the walks over the estimated dependency graph for \( c \). Different from the walks on a program \( c \)’s semantics based graph, \( k \in W^K(G_{trace}(c)) \), \( k \in W^K(G_{est}(c)) \) doesn’t rely on initial trace. The occurrence times of every \( v_i \) in \( k \)’s vertex sequence is bound by an arithmetic expression \( w_i \) where \( (v_i, w_i) \in V_{est}(c) \), \( v_i \)'s estimated weight. The length of a finite walk \( k \in W^K(G_{est}(c)) \) is an arithmetic expression as well, i.e., \( \text{len}(k) \in \mathcal{A}_{in} \).

Then the query length of a finite walk in \( G_{est}(c) \) is an arithmetic expression as well as follows.

**Definition 26 (Query Length of the Finite Walk on estimated dependency graph (\( \text{len}^q \))).**

Given a program \( c \)’s semantics-based dependency graph \( G_{est}(c) = (V_{est}(c), E_{est}(c), W_{est}(c), Q_{est}(c)) \), and a finite walk \( k \in W^K(G_{est}(c)) \). The query length of \( k \), \( \text{len}^q(k) \in \mathcal{A}_{in} \) is the number of vertices which correspond to query variables in the vertices sequence of this walk \( k \) \( (v_1, \ldots, v_n) \) as follows,

\[
\text{len}^q(k) = |\{ v \mid v \in (v_1, \ldots, v_n) \land v \in Q_{est}(c) \}|
\]

**Definition 27 (estimated Adaptivity).**

Given a program \( c \) and its estimated dependency graph \( G_{est}(c) \) the estimated adaptivity for \( c \) is

\[
\lambda_{est}(c) = \max \{ \text{len}^q(k) \mid k \in W^K(G_{est}(c)) \}
\]
whileSim(k) ≜ 
\[
\begin{align*}
[j &- k]_0; [x \leftarrow \text{query}(\chi[0])]_1; \\
\text{while } [j > 0]^2 \text{ do} & \quad \left[ [x \leftarrow \text{query}(\chi[x])]_1; [j \leftarrow j - 1]_4 \right]
\end{align*}
\]

Figure 5: (a) The simple k adaptivity rounds while loop example (b) The estimated dependency graph generated from AdaptFun.

Based on the soundness of the estimated dependency graph, our estimated adaptivity is a sound upper bound of its adaptivity in Definition 14.

**Theorem 4.2** (Soundness of AdaptFun). For every program c, its estimated adaptivity is a sound upper bound of its adaptivity.

\[ \forall \tau_0 \in T_0(c), v \in \mathbb{N}^\infty. \langle A_{\text{est}}(c), \tau_0 \rangle \not\leq v \Rightarrow A(c)(\tau_0) \leq c \]

The proof is in Appendix B. To compute \( A_{\text{est}}(c) \) accurately and soundly, we develop an adaptivity computation algorithm named AdaptSearch. It combines the depth first search and breath first search strategies and computes a sound upper bound on \( A_{\text{est}}(c) \). AdaptSearch also involves another algorithm AdaptSearchSCC in 2 recursively, which finds the longest walk for a strong connected component (SCC) (SCC is the maximal strongly connected subgraph) of \( G_{\text{est}}(c) \). Theorem 4.3 below formally describes the soundness of this algorithm with proof in Appendix E.

**Theorem 4.3** (Soundness of AdaptSearch). For every program c, we have

\[ \text{AdaptSearch}(G_{\text{est}}(c)) \geq A_{\text{est}}(c). \]

By Definition 27, the key point is to find the walks in the estimated dependency graph. We first discuss two challenges when we try to find the walks, and then show that how we solve them using our algorithms.

**Non-Termination Challenge**: One naive walk finding method is to simply traverse on this graph and decrease the weight of every node by one after every visiting. However, this simple traversing strategy leads to non-termination dilemma for most programs which we are interested in. Because the weight of each vertex in a program’s estimated dependency graph, which is an arithmetic expression containing input variables. In this sense, the simple traversing could never terminate when domain of the input variables isn’t finite. However, it is very common that the domain of program’s input variables is infinite such as natural number \( \mathbb{N} \), real number \( \mathbb{R} \), or etc. As the simple while loop example program in Figure 5 with \( k \) adaptivity rounds, the input variable \( k \) has domain \( \mathbb{N} \). If we traverse on the estimated dependency graph, and decrease the weight of \( x_3 \) (the weight \( k \) is symbolic) by one after every visit, we will never terminate because we only know \( k \in \mathbb{N} \).

To solve this non-termination challenge, we switch to another walk finding approach: finding the longest path in the estimated dependency graph via depth first search and then use this path as the estimated longest walk. Through a simple depth first search algorithm, we find the longest weighted path as the dotted arrow in Figure 5(c), \( x_3 : k \rightarrow x_1 : 1 \). Then, by summing up the weights on this path where the vertices have query annotations 1, depth first search algorithm gives the adaptivity bound \( k \). This is a tight bound for this simple \( k \) adaptivity rounds example program.

**Approximation Challenge**: However, this naive approximation via depth first searching over-approximates the adaptivity rounds largely in many cases. It computes \( \infty \) adaptivity upper bound for our twoRounds example program in Figure 2 which has only 2 adaptivity rounds. More specifically,
the depth first searching finds the longest weighted path, \( x^3, k \rightarrow a^5, k \rightarrow l^6, 1 \). Then, it computes the weighted length, \( 1 + k \). If we use this path to approximate the longest finite walk, and weight of each vertex as its visiting time, then we have a walk, \( x^3 \rightarrow \cdots \rightarrow x^3 \rightarrow a^5 \rightarrow \cdots \rightarrow a^5 \rightarrow l^6 \). However, this isn’t a qualified walk by our Definition 12. Because \( l^6 \) has weight 1, it can only be visited as most once. In this sense, \( x^3 \) is only able to be visited at most once as well, because the only way to re-visit \( x^3 \) is through \( l^6 \rightarrow a^5 \rightarrow x^3 \). Contradictory, \( x^3 \) is visited \( k \) times in this approximated walk. As a result, the weighted length of this path is \( 1 + k \), which over approximates this two rounds example program’s adaptivity rounds, which is supposed to be 2.

**Adaptivity Computation Algorithm**

To this end, we combine the depth first search and breath first search strategies in our longest walk estimation algorithm. Our algorithm reduces the task of computing the longest walk into the computation of local adaptivity and the composition of local adaptivity into global adaptivity. We exploit the structure of the estimated dependency graph \( G_{est}(c) \) for a program \( c \): 1). Partitioning the PDG of programs into its strongly connected components (SCCs) (SCCs are maximal strongly connected subgraphs). 2). Then, for each SCC, we compute an adaptivity bound \( 3 \). In the last, we compose these local bounds to an overall adaptivity bound. AdaptSearch\((c,G_{est}(c))\) algorithm in Algorithm 1 arranges the estimated dependency graph \( G_{est}(c) \) into SCCs \( SCC_1, \ldots, SCC_n \) and obtains the adaptivity local bound of each SCC from AdaptSearch\(_{scc}(c,SCC_i)\) algorithm in Algorithm 2. Then AdaptSearch shrinks the estimated dependency graph into a directed acyclic graph (DAG) by reducing each SCC into a vertex with the weight equal to its adaptivity local bound. In this way, it simply computes the length of the longest path over this DAG.

**Algorithm 1** Adaptivity Computation Algorithm (**AdaptSearch\((c,G_{est}(c))\)**)

**Require**: The program \( c \), Its estimated dependency graph: \( G_{est}(c) = (V,E,W,Q) \)

1: _init_

\( q \): empty queue.

adapt : the adaptivity of this graph initialize with 0.

2: _for_ every SCC: \( SCC_1, \ldots, SCC_n, 0 \leq n \leq |V| \).

3: adapt\(_{scc}[SCC_i] = \text{AdaptSearch}_{scc}(c,SCC_i) \);

4: _for_ every SCC:\( i \):

5: _q.append(SCC_i) ;

6: adapt\(_{tmp} = 0; \)

7: _while_ \( q \) isn’t empty:

8: \( s = q.pop(); \) \#{take the top SCC from head of queue}

9: adapt\(_{tmp0} = \text{adapt}_{\text{tmp}} ; \) \#{record the adaptivity of last level}

10: SCC\(_{max} ; \) \#{record the SCC with longest walk in this level}

11: _for_ every different SCC, \( s' \) connected by \( s \) by a directed edge from \( s \):

12: _if_ (adapt\(_{\text{tmp}} < \text{adapt}_{\text{tmp0}} + \text{adapt}_{\text{tmp}}[s']\)):

13: \( \text{adapt}_{\text{tmp}} = \text{adapt}_{\text{tmp0}} + \text{adapt}_{\text{tmp}}[s'] ; \)

14: SCC\(_{max} = s' ; \) \#{update the SCC with the longest walk in this level}

15: _q.append(SCC\(_{max} \)) ;

16: adapt = max(adapt, adapt\(_{\text{tmp}} \)) ;

17: _return_ adapt.
The Adaptness Computation Algorithm (AdaptSearch(c, G_{\text{est}}(c))) At Line:3, this algorithm first finds all the SCCs of G_{\text{est}}(c), SCC_1, \ldots, SCC_n where 0 \leq n \leq |V| by the standard Kosaraju’s algorithm, where each SCC_i = (V_i, E_i, W_i, Q_i). Then, it computes the adaptivity local bound on every SCC_i in line:4-5 by AdaptSearch_{\text{SCC}}(c, SCC_i). We guarantee the soundness of the adaptivity local bound on an SCC by Lemma [E.1] with formal proof in Appendix [E]. The G_{\text{est}}(c) is then shrunk into a directed acyclic graph where SCC_1, \ldots, SCC_n are all the vertices and the adaptivity local bounds are their weights. There is an edge s_i \rightarrow s_j in this shrank graph, as long as we can find an edge v_i \rightarrow v_j \in E_{\text{est}}(c) such that v_i \in V_i, v_j \in V_j, and i \neq j. Then, we use the standard breath first search strategy to find the longest weighted path on this DAG and return this length as the adaptivity upper bound.

We guarantee that the length of this longest weighted path is a sound computation of the adaptivity for program c and this longest weighted path is a sound computation of the finite walk having the longest query length on c’s estimated dependency graph in Theorem [E.1] in Appendix [E]. If a program c’s estimated dependency graph G_{\text{est}}(c) is a DAG, then we prove that the adaptivity upper bound by Algorithm [I] is tight formally in Theorem [E.1] in Appendix [E].

Adaptivity Computation Algorithm on An SCC (AdaptSearch_{\text{SCC}}(c, SCC_i)) This algorithm takes the program, and an SCC (a subgraph), SCC_i of a program’s estimated dependency graph G_{\text{est}}(c) as input and outputs the adaptivity local bound of SCC_i. For an SCC containing only one vertex without any edge, it returns the query annotation of this vertex as adaptivity. For SCC containing at least one edge, there are three steps in this algorithm: 1. It first collects all the paths in the input SCC 2. Then it calculates the adaptivity of every path by a novel adaptivity computation method. 3. The maximal adaptivity among over all paths is the adaptivity of this SCC in the end. Because the input graph is SCC, when the algorithm starts to traverse from a vertex, it finally goes back to the same vertex. In this sense, the paths collected in step 1 are all simple cycles with the same starting and ending vertex. The most interesting part is step 2. It recursively computes the adaptivity upper bound on the fly of paths collecting through a depth first search procedure dfs from line: 5-15. It designs a novel adaptivity computation method, which guarantees the visiting times of each vertex by its weight and addresses the Approximation Challenge. The guarantee is achieved by two special parameters flowcapacity and querynum and the updating operations in line:7 and line:10.

- flowcapacity is a list of arithmetic expression A_{\text{in}}. It tracks the minimum weight along the path during the searching procedure. For each vertex, it updates the minimum weight when the path reaches that vertex with \infty as the initial value.
- querynum is a list of integer initialized by query annotation Q_i(v) for every vertex. It tracks the total number of vertices with query annotation 1 along the path.
- The updating operation during the traversal (line: 7) and at the end of the traversal (line: 10) is flowcapacity[v] \times querynum[v]. Because querynum[v] is the # of the vertices with query annotation 1 and flowcapacity[v] is the minimum weight over this path, this number is the accurate query length of this path. It guarantees the visiting times of each vertex on the path reaching a vertex v is no more than the maximum visiting times it can be on a qualified walk by flowcapacity[v], and in the same time compute the query length instead of weighted length through querynum[v].

In this way, we resolve the Approximation Challenge without losing the soundness, formally in Appendix [E]. This step also guarantees the termination through a boolean list, visited in line:7 and line:13.
Algorithm 2 Adaptivity Computation Algorithm on An SCC (AdaptSearch\textsubscript{scc}(c,SCC\textsubscript{i}))

\textbf{Require:} The program $c$, An strong connected component of $G_{\text{est}}(c)$: $\text{SCC}_{i} = (V_{i}, E_{i}, W_{i}, Q_{i})$

1: \textbf{init} \\
\hspace{1em} $r_{\text{scc}}$: $\mathcal{A}_{1n}$, initialized 0, the Adaptivity of this SCC

2: \textbf{init} \\
\hspace{1em} visited : $\{0, 1\}$ List, \\
\hspace{1em} \#\{length $|V_{i}|$, initialize with 0 for every vertex, recording whether a vertex is visited.\} \\
\hspace{1em} $r$ : $\mathcal{A}_{1n}$ List, \\
\hspace{1em} \#\{length $|V_{i}|$, initialize with $Q(v)$ for every vertex, recording the adaptivity reaching each vertex.\} \\
\hspace{1em} flowcapacity: $\mathcal{A}_{1n}$ List, \\
\hspace{1em} \#\{length $|V_{i}|$, initialize with $\infty$ for every vertex, recording the minimum weight when the walk reaching that vertex, inside a cycle\} \\
\hspace{1em} querynum: INT List, \\
\hspace{1em} \#\{length $|V_{i}|$, initialize with $Q(v)$ for every vertex, recording the query numbers when the path reaching that vertex, inside a cycle\} \\

3: \textbf{if} $|V_{i}| = 1$ and $|E_{i}| = 0$: \\
4: \hspace{1em} \textbf{return} $Q(v)$ \\
5: \textbf{def} dfs(G, s, visited): \\
6: \hspace{1em} \textbf{for} every vertex $v$ connected by a directed edge from $s$: \\
7: \hspace{2em} \textbf{if} visited[$v$] = false: \\
8: \hspace{3em} flowcapacity[$v$] = min($W_{i}(v)$, flowcapacity[$s$]); \\
9: \hspace{3em} querynum[$v$] = querynum[$s$] + $Q_{i}(v)$; \\
10: \hspace{3em} $r[v]$ = max($r[v]$, flowcapacity[$v$] $\times$ querynum[$v$]); \\
11: \hspace{3em} visited[$v$] = 1; \\
12: \hspace{3em} dfs(G, v, visited); \\
13: \hspace{2em} \textbf{else}: \#\{There is a cycle finished\} \\
14: \hspace{3em} $r[v]$ = max($r[v], r[s] + \min(W_{i}(v), \text{flowcapacity}[s]) \times (\text{querynum}[s] + Q_{i}(v)))$; \\
\hspace{3em} \#\{update the length of the longest walk reaching this vertex on this cycle\} \\
15: \hspace{2em} \textbf{return} $r[c]$ \\
16: \textbf{for} every vertex $v$ in $V_{i}$: \\
17: \hspace{1em} initialize the visited, $r$, flowcapacity, querynum with the same value at line:2. \\
18: \hspace{1em} $r_{\text{scc}} = \max(r_{\text{scc}}, \text{dfs}(\text{SCC}_{i}, v, \text{visited}))$; \\
19: \textbf{return} $r_{\text{scc}}$
Algorithm Detail Steps The detail steps of dfs from line: 2-15 in Algorithm 2 is described as follows.
Line:2 initialize parameters:
1. flowcapacity is a list of arithmetic expressions with length \(|Q_i(c)|\) and the initial value \(\infty\) for every element. For every vertex, it records the minimum weight when the path traverses to this vertex.
2. querynum is a list of integer with length \(|V_i(c)|\) and the initial value \(Q_i(v)\) for every element. For every vertex, it records the total query numbers when the path traverses to this vertex.
3. The visited is initialized by 0 for every element and has length \(|V_{est}(c)|\) as well. It is used to guarantee the termination during recursion.
4. \(x\) is a list of \(A_{in}\) initialized with query annotation for each vertex. For each vertex, it maintains the longest query length when the recursion reaches it.

Line:7-12 updates the parameters and recursively traverses for every unvisited vertex heading out from \(v\). In each recursion, Line:8 maintains the minimum weight for the flowcapacity and Line:9 updates the number of query vertices querynum so far when the traversing reaches \(v\). Line:10 updates the longest query length \(x\) alone the path when the traverse arrives vertex \(v\) by \(flowcapacity[v] \times querynum[v]\). This computation guarantees: 1. The visiting times of each vertex on the walk reaching \(v\) is no more than the maximum visiting it can be on this walk; 2. Only the vertices have annotation 1 are counted in adaptivity. In this way, we accurately approximate a walk using this path and computes the query length of this walk safely. This addresses the Approximation Challenge and in the same time without losing the soundness.

At line: 14, if this vertex \(v\) is visited, i.e., the traverse of this path goes back to its starting point, we only update the longest query length \(x[v]\) for \(v\) in the same way as Line:11. However, we do not update querynum and flowcapacity in this case. This improves the accuracy and still guarantees the soundness. The soundness is formally proved in Lemma E.1 in Appendix E. We also discuss how these computations guarantee the soundness and improves the accuracy in the following example.

Example The example program in Figure 6 illustrates how these special operations computes accurate and sound adaptivity for the program. AdaptSearch first find the SCC contains vertices \(y_6^0\) and \(x_9^0\), \(SCC = (V,E)\) where \(V = \{y_6^0, x_9^0\}\) and \(E = \{(y_6^0, y_6^9), (x_9^0, x_9^9), (x_9^9, y_6^9), (y_6^9, x_9^9)\}\). Then AdaptSearch\_SCC(SCC, nestedW) takes this SCC as input. When start from vertex \(y_6^0\), it first finds the path \(y_6^0 \rightarrow y_6^9\). By updating parameters through Line:10 and 14, it computes the longest query length for this path as \(k\). As highlighted in Line:14, we do not update querynum and flowcapacity when we identify the simple cycle \(y_6^9 \rightarrow y_6^0\). This improves the accuracy and still guarantees the soundness. Because in the following recursions, we continuously search for walks heading out from \(y_6^0\), the flowcapacity of this simple cycle does not restrict the walks going out of this vertex that do not interleave with the cycle \(y_6^9 \rightarrow y_6^0\). However, if we keep updating the minimum weight, then we restrict the visiting times of vertices on a walk by using the minimum weight of vertices that do not on this walk. This leads to the unsoundness in computing adaptivity. Concretely, if we update the flowcapacity\([y_6^0]\) as \(k\) after visiting \(y_6^0\) the second time on this walk, and continuously visit \(x_9^9\), then the flowcapacity\([k]\) is updated as \(\min(k, k^2)\). So the visiting times of \(x_9^9\) is restricted by \(k\) on the walk \(y_6^9 \rightarrow y_6^0 \rightarrow x_9^9\). This restriction excludes the finite walk \(y_6^0 \rightarrow y_6^9 \rightarrow x_9^9 \rightarrow x_9^9\) where \(y_6^0\) and \(x_9^9\) visited by \(k^2\) times in the computation. However, the finite walk \(y_6^0 \rightarrow y_6^9 \rightarrow x_9^9 \rightarrow x_9^9\) where \(y_6^0\) is visited \(k\) times and \(x_9^9\) \(k^2\) times is a qualified walk, and exactly the longest walk we aim to find. So, by Non-updating the flowcapacity after visiting \(y_6^0\) again, we guarantee that the visiting times of vertices on every searched walk will not be restricted by weights not on this walk, i.e., the soundness. Line: 15 returns the adaptivity heading out from its input vertex. Line:16-18 applies dfs on every vertex of this SCC and computes the adaptivity of this SCC by taking the maximum value. The soundness is formally guaranteed in Lemma E.1 in Appendix E.
nestedW(k) ≜
[i ← i−1]; [x ← query(χ[I])];
while [i > 0]
[ j ← j−1]; [y ← query(χ(I) + j)];
while [j > 0]
[ x ← query(χ(ln(x) + y))];

Algorithm 3 Over-Approximated Adaptivity on SCC

Require: G = (V, E, W, Q) #An Strong Connected Symbolic Weighted Directed Graph
1: AdaptSearch_{scc-naive}(G):
2: init
   r_{scc}: the Adaptivity of this SCC
3: for every vertex v in V:
4:   r_{scc} += W(v) * Q(v)
5: return r[c]

Theorem 4.4 (Soundness of AdaptSearch). For every program c, given its estimated dependency graph G_{est},

\[ \text{AdaptSearch}(G_{est}) \geq A_{est}(G_{est}). \]

5 Examples and Experimental Results

We present four examples, illustrating AdaptFun. Then we show our implementation of AdaptFun and its experimental results on 18 examples including these four examples.

5.1 Examples

Example 5.1 (Multiple Rounds Algorithm). We look at an advanced adaptive data analysis algorithm - multipleRounds algorithm in Figure 7(a). This is a simplified version of the Monitor Augment from [4] with complete program in Appendix. It takes the user input k which decides the number of iterations. It starts from an initialized empty tracking list I, goes k rounds and at every round, tracking list I is updated by a query result of query(χ[I]). After r rounds, the algorithm returns the columns of the hidden database D not specified in the tracking list I. The functions updnscore(p, a), updcscore(p, a), update(I, ns, cs) simplify the computations of updating ns, cs and I.

Different from twoRounds(k) in Figure 2, the query request, \[ a ← \text{query}(I) \] in each loop iteration is not independent. query(I) in each iteration depends on the tracking list I from all the previous iterations, and I is updated by all the query results in the previous iterations as well. In this sense, all these k queries are adaptively chosen according to our discussion in overview. The program-based dependency graph is presented in Figure 7(b). We omitted its execution-based dependency graph G_{trace}(multipleRounds(k)) because they have the same graph topology and only differ in weights. For each vertex v in G_{est}(multipleRounds(k)) in Figure 7(b), we use w_v to denote its weight function in G_{trace}(multipleRounds(k)).
Figure 7: (a) The simplified multiple rounds example (b) The estimated dependency graph by $\text{AdaptFun}$

\[
\begin{align*}
\text{lR}(k, r) & \triangleq \\
[ a - 0 ]^6; [ c - 0 ]^1; [ j - k ]^2; & \\
\text{while} [ j > 0 ]^3 \text{ do } & \\
[ [ da - \text{query}(2 - \chi[1] - (\chi[0] \times a + c)) \times (\chi[0])) ]^4; & \\
[ dc - \text{query}(2 - \chi[1] - (\chi[0] \times a + c))]^7; & \\
[ a - a - r \ast da]^5; [ c - c - r \ast dc]^7; & \\
[ j - j - 1 ]^8; & 
\end{align*}
\]

As the adaptivity definition in our formal adaptivity model in Definition 12 there is a finite walk along the dashed arrows, $a^6 \rightarrow I^3 \rightarrow ns^7 \rightarrow \cdots \rightarrow ns^7$, where the vertices $a^6$, $I^3$ and $ns^7$ are visited $w_{a^6}(\tau_0)$, $w_{I^3}(\tau_0)$ and $w_{ns^7}(\tau_0)$ times respectively with input $\tau_0$. The vertex $a^6$ has query annotation 1, and it is visited $w_{a^6}(\tau_0)$ times. In this sense, the adaptivity of this program is $w_{a^6}(\tau_0)$ given input $\tau_0$, i.e., $A(\text{multipleRounds}(k))\tau_0 = w_{a^6}(\tau_0)$. Since $w_{a^6}(\tau_0)$ counts the execution times of command $[ a - \text{query}(\cdot) ]^6$; this count is at most the loop iteration numbers, i.e., $k$’s initial value, $\rho(\tau_0)k$ from the initial trace $\tau_0$. Next, we show that $\text{AdaptFun}$ provides a tight upper bound for this example by AdaptSearch(\text{multipleRounds}(k)). It first finds a path on $G_{\text{est}}(\text{multipleRounds}(k))$ $a^6 \ast I^3 \ast ns^7 \ast \cdots \ast ns^7$ with three weighted vertices. Then AdaptSearch approximates this path to a walk $a^6 \ast I^3 \ast ns^7 \ast \cdots \ast a^6 \ast I^3 \ast ns^7 \ast \cdots$, In this walk, $a^6, I^3, ns^7$ are all visited $k$ times respectively. So $A_{\text{est}}(\text{multipleRounds}(k)) = k$. We know for any initial trace $\tau_0$. $\tau_0 \not\in \rho(\tau_0)$ So we guarantee $A(\text{multipleRounds}(k))\tau_0 \leq \rho(\tau_0)$ for any $\tau_0$ and $k$ is a sound bound.

Example 5.2 (Linear Regression Algorithm with Gradient Decent Optimization). The linear regression algorithm with gradient decent Optimization works well in our $\text{AdaptFun}$ as well. It computes $A_{\text{est}}(\text{lR}(k, r)) = k$.

This linear regression algorithm aims to find a linear relationship, $y = a \times x + c$ between a dependent variable $y$ and an independent variable $x$, by approximating the model parameter $a$ and $c$. In order to have a good approximation on the model parameter $a$ and $c$, it sends query to a training data set adaptively in each iteration. This training data set contains two columns (can extend to higher dimensional data sets), the first column contains the observed values of the independent variable $x$ and the second column for the dependent variable $y$. $\text{lR}(k, r)$ is the program of this example written in our language model in Figure 12(a) with input variables $k$ and $r$. 

33
1R(k, r) starts from initializing the linear model parameters and the counter variable in commands 0, 1, 2, and then goes into the while iterations. In each iteration, it computes the differential value w.r.t. parameter a and c, through two query requests, query(−2 * (χ[1] − (χ[0] x a + c)) x (χ[0])) and query(−2 * (χ[1] − (χ[0] x a + c))) in commands of label 4 and 5. Then, it uses these two differential values stored in variables da and dc to update the model parameters a and c. In this sense, the two query requests in each iteration depends on queries in the previous iterations and the intuitive adaptivity rounds is k. Its program-based dependency graph, G_{est}(1R(k, r)) is shown in Figure 12(b), and G_{trace}(1R(k, r)) is omitted for the same reason as Example 5.1. In Figure 12(b), we omit the edges which are constructed by the transition of flowsTo relation for concise, but these edges exist in G_{trace}(1R(k, r)) because they can be constructed directly by DEP_{var} relation. Given an input τ₀, there are multiple walks having the same longest query length in G_{trace}(1R(k, r)), such as the walks \(c¹ \rightarrow dc⁵ \rightarrow c¹ \rightarrow \cdots \rightarrow dc³\) and \(a⁶ \rightarrow da⁴ \rightarrow a⁶ \rightarrow \cdots \rightarrow da⁴\) along the dotted arrows. The vertices \(c¹, dc⁵, a⁶, da⁴\) in the two walks are visited \(w_{c¹}, w_{dc⁵}, w_{a⁶}, w_{da⁴}\) respectively. Though the different weight functions count the execution times of the different corresponding command, the counts are expected to be the same number, i.e., the loop iterations. And the loop iterations are indeed the initial value of input k from initial trace τ₀. In this sense, \(A(1R(k, r))(τ₀) = w_{dc⁵}(τ₀) = w_{da⁴}(τ₀)\). Similar to Example 5.1, AdaptFun estimates the tight adaptivity bound, k for this example.

Example 5.3 (Multiple Rounds Odds Algorithm). The AdaptFun comes across an over-approximation due to its path-insensitive nature. It occurs when the control flow can be decided in a particular way in front of conditional branches, while the static analysis fails to witness. As in Figure 12(a), multiRoundsS(k) is an example program with 1 + k adaptivity rounds and two paths while loop. In each iteration, the query \([y \leftarrow \text{query}(χ[x])]^5\) and \([p \leftarrow \text{query}(χ[x])]^6\) are based on previous query results stored in x, which is similar to Example 5.1. The difference is that, only the query answer from \([y \leftarrow \text{query}(χ[x])]^5\) in the first branch is used in the query in command 7, query(χ[ln(y)]), and the first branch is only executed in even iterations \((i = 0, 2, \cdots)\). From the Semantics-based dependency graph in Figure 6(b), the weight \(w_{χ[x]}(τ₀)\) for the vertex \(χ³\) will count the precise evaluation times of \([y \leftarrow \text{query}(χ[x])]^5\), i.e., half of the iteration numbers. This number is expected to be half of the initial value of input k from τ₀. However, AdaptFun fails to realize that all the odd iterations only execute the first branch and only even iterations execute the second branch. So it considers both branches for every iteration when estimating the adaptivity. In this sense, the weight estimated for \(χ³\) and \(p⁶\) are both k as in Figure 9(c). As a result, AdaptFun computes \(χ³ \rightarrow x⁷ \rightarrow y⁵ \rightarrow \cdots \rightarrow x⁷\) as the longest walk in Figure 9(c), where each vertex is visited k times. In this sense, the estimated adaptivity is \(1 + 2 * k\), instead of \(1 + k\).

Example 5.4 (Over-Defined Adaptivity Example). The program’s adaptivity definition in our formal model, (in Definition 14) comes across an over-approximation on capturing the program’s intuitive adaptivity rounds. It is resulted from the difference between its weight calculation and the variable may-dependency definition. It occurs when the weight is computed over the traces different from the traces used in witnessing the variable may-dependency relation.

The program multiRoundsS(k) in Figure 10(a) demonstrates this over-approximation. It is a variant of the multiple rounds strategy with input k. In each iteration the query query(χ[y] + p) in command 7 is based on previous query results stored in p and y differ from Example 5.1 only the query answer from the one iteration, the \((k-2)^{th}\) one is used in query request \([p \leftarrow \text{query}(χ[y] + p)]^7\). Because the execution will reset p’s value by the constant 0 in all the other iterations at line 10 after this.

\(^1\)Again, we omit the edges which are constructed by the transition of flowsTo relation for concise, but these edges exist in G_{trace}(multiRoundsS(0(k)) because they can be constructed directly by DEP_{var} relation.
Though the AdaptFun We implemented the final output. python program and the python program provides the adaptivity upper bound and the query number as and outputs the program-based dependency graph and the abstract transition graph, feeds into the algorithm shown in Section 4.5 in Python. The OCaml program takes the labeled command as input and outputs the program-based dependency graph and the abstract transition graph, feeds into the python program and the python program provides the adaptivity upper bound and the query number as the final output.

5.2 Implementation Results

Figure 9: (a) The multiple rounds odd example (b) The semantics-based dependency graph (c) The estimated dependency graph from AdaptFun.

Figure 10: (a) The multi rounds single example (b) The semantics-based dependency graph.

query request. In this way, all the query answers stored in p is erased and isn’t used in the query request at next iteration, except the one at the (k-2)th iteration. In this sense, the intuitive adaptivity rounds for this example is 2. However, our adaptivity definition fails to realize that there is only dependency relation between p^7 and p^7 in one single iteration, but not in all the others. As shown in the semantics-based dependency graph in Figure 10(b), there is an edge from p^7 to itself representing the existence of Variable May-Dependency from p^7 on itself, and the visiting times of labeled variable p^7 is w(τ₀). w(τ₀) will count the execution times of command [ p ← query(χ(y) + p) ] during execution, which is expected to be equal to the loop iteration numbers, i.e., initial value of input k from the initial trace τ₀. As a result, the walk with the longest query length is p^7 → ... → p^7 → y^4 → z^1 with the vertex p^7 visited w_p(τ₀), as the dotted arrows. The adaptivity based on this walk is 2 + w_p(τ₀), instead of 2. Though the AdaptFun is able to give us 2 + k, as an accurate bound w.r.t this definition. x

5.2 Implementation Results

We implemented AdaptFun as a tool which takes a labeled command as input and outputs two upper bounds on the program adaptivity and the number of query requests respectively. This implementation consists of an abstract control flow graph generation, edge estimation (as presented in Section 4.3.2), and weight estimation (as presented in Section 4.3.3) in Ocaml, and the adaptivity computation algorithm shown in Section 4.5 in Python. The OCaml program takes the labeled command as input and outputs the program-based dependency graph and the abstract transition graph, feeds into the python program and the python program provides the adaptivity upper bound and the query number as the final output.
We evaluated this implementation on 23 example programs with the evaluation results shown in Table 2. In this table, the first column is the name of each program. For each program $c$, the second column is its intuitive adaptivity rounds, the third column is the output of the AdaptFun implementation, which consists of two expressions. The first one is the upper bound for adaptivity and the second one is the upper bound for the total number of query requests in the program. And the last column is the performance evaluation w.r.t. the program size.

The last column is the performance evaluation. The time contains three parts. The first part is the running time of the Ocaml code, which parses the program and generates the $G_{set}(c)$. The second and third parts are the running times of the reachability bound analysis algorithm and the adaptivity computation algorithm, AdaptSearch($c$).

The first 5 programs are adapted from real world data analysis algorithms. The first two programs twoRounds($k$), multiRounds($k$) are the same as Figure 2(a) and Figure 7(a). AdaptFun computes tight adaptivity bound for the first 3. For the forth program multiRoundsO($k$), AdaptFun outputs an over-approximated upper bound $1 + 2 \times k$ for the $A(c)$, which is consistent with our expectation as discussed in Example 5.3. The fifth program is the evaluation results for the example in Example 5.4 where AdaptFun outputs the tight bound for $A(c)$ but $A(c)$ is a loose definition of the program’s actual adaptivity rounds.

The programs from Tab. 2 line:6-17 all have small size but complex structures, to test the programs under different situations including data, control dependency, the multiple paths nested loop with related counters, etc. Both implementations compute the tight bound for examples in line:6-14 and over-approximate the adaptivities for 15th and 16th due to path-insensitivity. For the 17th one, implementation I gives tight bound bound while II gives loose bound, so we keep both implementations.

The last six programs are composed of some programs above in order to test the performance limitation when the input program is large. From the evaluation results, the performance bottleneck is the reachability bound analysis algorithm. By implementing the bound analysis algorithm in Section 4.3.3 (adapted from [6]), we are unable to evaluate the Jumbo in a reasonable time period. Alternatively, we implement another light reachability bound analysis algorithm and compute the adaptivity for jumboS, jumbo and big effectively.

Overall for these examples, our system gives both the accurate adaptivity definition and estimated adaptivity upper bound through our formalization and analysis framework AdaptFun. The complete programs are defined below from Example 5.5 to Example 11.14 in the Appendix [H].
Table 1: Experimental results of AdaptFun implementation

| Program C | adaptivity | AdaptFun | \( f(1|B) \) | \( f(1|B) \) | \( g \) | \( h \) | \( \text{query}(1|B) \) | \( \text{lines} \) | \( \text{performance} \) | \( \text{running time (ms)} \) |
|-----------|------------|----------|----------------|----------------|-------|-------|----------------|----------------|----------------|----------------|
| TwoRounds(k) | 2 | 2 | \( k \) | \( k \) | \( k \) | \( k \) | \( k \) | \( k \) | \( k \) | \( k \) |
| multiRounds(k) | 4 | 4 | \( k \) | \( k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) |
| mR1(d) | 1 | 1 | \( k \) | \( k \) | \( k \) | \( k \) | \( k \) | \( k \) | \( k \) | \( k \) |
| mR2(d) | 2 | 2 | \( 1 + k \) | \( 1 + 3k \) | \( 1 + 3k \) | \( 1 + 3k \) | \( 1 + 3k \) | \( 1 + 3k \) | \( 1 + 3k \) | \( 1 + 3k \) |
| mR3(d) | 3 | 3 | \( 1 \) | \( 1 \) | \( 1 \) | \( 1 \) | \( 1 \) | \( 1 \) | \( 1 \) | \( 1 \) |
| while(k) | 1 | 1 | \( k \) | \( k \) | \( 1 \) | \( 1 \) | \( 1 \) | \( 1 \) | \( 1 \) | \( 1 \) |
| while(k2) | 1 | 1 | \( k \) | \( k \) | \( 1 \) | \( 1 \) | \( 1 \) | \( 1 \) | \( 1 \) | \( 1 \) |
| while(k2) | 1 | 1 | \( k \) | \( k \) | \( 1 \) | \( 1 \) | \( 1 \) | \( 1 \) | \( 1 \) | \( 1 \) |
| while(k2) | 1 | 1 | \( k \) | \( k \) | \( 1 \) | \( 1 \) | \( 1 \) | \( 1 \) | \( 1 \) | \( 1 \) |
| mR4(d) | 4 | 4 | \( k \) | \( k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) |
| mR5(d) | 5 | 5 | \( k \) | \( k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) |
| mR6(d) | 6 | 6 | \( k \) | \( k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) |
| mR7(d) | 7 | 7 | \( k \) | \( k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) |
| mR8(d) | 8 | 8 | \( k \) | \( k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) |
| mR9(d) | 9 | 9 | \( k \) | \( k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) |
| mR10(d) | 10 | 10 | \( k \) | \( k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) | \( 3k \) |

5.3 More Discussions on The Evaluated Examples

5.3.1 More on The Two Rounds Adaptive Data Analysis

Example 5.5 (Complete Two Rounds Algorithm).

\[
\begin{align*}
[a - 1]^3; \\
[j - k]^3; \\
\text{while } [j > 0]^3 \text{ do} \\
\text{twoRounds(k)} \triangleq \\
\left( [x \leftarrow \text{query}(k - j) \cdot \chi(k)]^4; \\
[j - j - 1]^3; \\
[a - x :: a]^3; \right) \\
\left[ I \leftarrow (\text{sign}(\sum_{i \in [k]} \chi[i] \times \ln \frac{1 + a[i]}{1 - a[i]}) \right]^7 \\
\end{align*}
\]

Algorithm 4 A two-round analyst strategy for random data (The example in [2])

Require: Mechanism \( \mathcal{M} \) with a hidden data set \( D \in \{-1, +1\}^{n \times (k+1)} \subset \mathbb{D} \).

for \( j \in [k] \) do

\text{define } q_j(d) = d(j) \cdot d(k) \text{ where } d \in \{D(i) \mid i = 0, \ldots, n \} \subset \{-1, +1\}^{k+1}.

let \( a_j = \mathcal{M}(q_j) \).

\{In the line above, \( M \) computes approx. the exp. value of \( q_j \) over \( D \). So, \( a_j \in \{-1, +1\} \.)

\text{define } q_k(d) = d(k) \cdot \text{sign}(\sum_{i \in [k]} \chi(i) \cdot \ln \frac{1 + a[i]}{1 - a[i]}) \text{ where } x \in \{-1, +1\}^{k+1}.

\{In the line above, \( \text{sign}(y) = \begin{cases} +1 & \text{if } y \geq 0 \\ -1 & \text{otherwise} \end{cases} \)

let \( a_{k+1} = \mathcal{M}(q_{k+1}) \).

\{In the line above, \( M \) computes approx. the exp. value of \( q_{k+1} \) over \( X \). So, \( a_{k+1} \in \{-1, +1\} \.)

return \( a_{k+1} \).

Ensure: \( a_{k+1} \in \{-1, +1\} \)}
A multi-round analyst strategy for random database \cite{2}

Ay variable dimensional data sets), the first column contains the observed values of the independent variable and an independent variable \( x \). In order to have a good approximation on the model parameter \( a \) and \( c \), it sends query to a training data set adaptively in each iteration. This training data set contains two columns (can extend to higher dimensional data sets), the first column contains the observed values of the independent variable \( x \) and

\[
\text{multiRounds}(k,c,N) \triangleq [j \leftarrow N]^0;[cs \leftarrow 0]^1;[ns \leftarrow 0]^2;[I \leftarrow 0]^2;[w \leftarrow k]^4;
\]

while \( [j > 0]^3 \) do
\[
\left( [j \leftarrow j - 1] \right]^6;[cs \leftarrow 0 + cs]^7;[ns \leftarrow 0 + ns]^8);\]

while \( [w > 0]^9 \) do
\[
\left( [w \leftarrow w - 1] \right]^{10};[p \leftarrow c]^11;[q \leftarrow c]^12;[a \leftarrow \text{query}(x[I])]^13;\]

\[ [i \leftarrow N]^14; \text{while } [i > 0]^{15} \text{ do } \]
\[
\left( [i \leftarrow i - 1]^{16};[cs(i) \leftarrow cs(i) + (a - p) \cdot (q - p)]^{17};\]

\[ \text{if } ([I < i]^{18},[ns(i) \leftarrow ns(i) + (a - p) \cdot (q - p)]^{19},[ns \leftarrow ns(i)]^{20});\]

\[ [i2 \leftarrow N]^{21}; \text{while } [i2 > 0]^{22} \text{ do } \]
\[
\left( [i2 \leftarrow i2 - 1]^{23}; \text{if } ([ns(i2) > max(cs)]^{24},[I \leftarrow i + 1]^{25},[I \leftarrow I]^{26}) \right)\]

Figure 11: (a) The labeled program implementing the multiple round algorithm (b)The same program in the SSA version

5.3.2 mRComplete

**Algorithm 5** A multi-round analyst strategy for random database \cite{2}

**Example 5.6** (Complete Multiple Round Algorithm). \textbf{Require:} Mechanism \( M \) with a hidden state \( X \in [N]^n \)

Define control set \( C \) as an \( [0,1, \cdots, c - 1] \)

Initialize \( N\text{score}(i) = 0 \) for \( i \in [N] \), \( I = \emptyset \) and \( C\text{score}(C(i)) = 0 \) for \( i \in [c] \)

\[ \text{for } j \in [k] \text{ do } \]
\[ \text{let } p = \text{uniform}(0,1) \]
\[ \text{define } q(x) = \text{bernoulli}(p) \]
\[ \text{define } qc(x) = \text{bernoulli}(p) \]
\[ \text{let } a = M(q) \]
\[ \text{for } i \in [N] \text{ do } \]
\[ \text{Nscore}(i) = N\text{score}(i) + (a - p) \cdot (q(i) - p) \text{ if } i \notin I \]
\[ \text{for } i \in [c] \text{ do } \]
\[ C\text{score}(C(i)) = C\text{score}(C(i)) + (a - p) \cdot (qc(i) - p) \]
\[ \text{let } I = [i] \in [N] \land N\text{score}(i) > \max(C\text{score}) \]
\[ \text{let } D = D \setminus I \]
\[ \text{return } D. \]

5.3.3 1RGD

**Example 5.7** (Linear Regression Algorithm with Gradient Decent Optimization). The linear regression algorithm with gradient decent Optimization works well in our AdaptFun as well. It computes \( A_{\text{est}}(1R(k,r)) = k \).

This linear regression algorithm aims to find a linear relationship, \( y = ax + c \) between a dependent variable \( y \) and an independent variable \( x \), by approximating the model parameter \( a \) and \( c \). In order to have a good approximation on the model parameter \( a \) and \( c \), it sends query to a training data set adaptively in each iteration. This training data set contains two columns (can extend to higher dimensional data sets), the first column contains the observed values of the independent variable \( x \) and
The linear regression algorithm (a) The estimated dependency graph by AdaptFun

The linear regression algorithm

\[ lR(k, r) \triangleq \]
\[ [a \leftarrow 0^0; c \leftarrow 0^1; j \leftarrow k^2]; \]
while \[ j > 0^3 \] do
\[ \{ da \leftarrow \text{query}(−2 * (χ[k] − (χ[0] × a + c)) × (χ[0]))\}; \]
\[ \{ dc \leftarrow \text{query}(−2 * (χ[k] − (χ[0] × a + c)))\}; \]
\[ [a \leftarrow a - r * da]^6; [c \leftarrow c - r * dc]^7; \]
\[ [j \leftarrow j - 1]^8; \]

The estimated dependency graph by AdaptFun

Figure 12: (a) The linear regression algorithm (b) The estimated dependency graph by AdaptFun

1R(k, r) starts from initializing the linear model parameters and the counter variable in commands 0, 1, 2, and then goes into the while iterations. In each iteration, it computes the differential value w.r.t. parameter \( a \) and \( c \), through two query requests, \( \text{query}(−2 * (χ[k] − (χ[0] × a + c)) × (χ[0])) \) and \( \text{query}(−2 * (χ[k] − (χ[0] × a + c))) \) in commands of label 4 and 5. Then, it uses these two differential values stored in variables \( da \) and \( dc \) to update the model parameters \( a \) and \( c \). In this sense, the two query requests in each iteration depends on queries in the previous iterations and the intuitive adaptivity rounds is \( k \). Its program-based dependency graph, \( G_{\text{est}}(1R(k, r)) \) is shown in Figure 12(b), and \( G_{\text{trace}}(1R(k, r)) \) is omitted for the same reason as Example 5.1. In Figure 12(b), we omit the edges which are constructed by the transition of \( \text{flowsTo} \) relation for concise, but these edges exist in \( G_{\text{trace}}(1R(k, r)) \) because they can be constructed directly by \( \text{DEP}_{\text{var}} \) relation. Given an input \( τ_0 \), there are multiple walks having the same longest query length in \( G_{\text{trace}}(1R(k, r)) \), such as the walks \( c^7 \rightarrow dc^5 \rightarrow c^7 \rightarrow \cdots \rightarrow dc^5 \) and \( a^6 \rightarrow da^4 \rightarrow a^6 \rightarrow \cdots \rightarrow da^4 \) along the dotted arrows. The vertices \( c^7, dc^5, a^6, da^4 \) in the two walks are visited \( w_{c^7}, w_{dc^5}, w_{a^6}, w_{da^4} \) respectively. Though the different weight functions count the execution times of the different corresponding command, the counts are expected to be the same number, i.e., the loop iterations. And the loop iterations are indeed the initial value of input \( k \) from initial trace \( τ_0 \). In this sense, \( A(1R(k, r))(τ_0) = w_{dc^5}(τ_0) = w_{da^4}(τ_0) \). Similar to Example 5.1, AdaptFun estimates the tight adaptivity bound, \( k \) for this example.
Appendices

A Proofs of Lemmas for the Language Model

A.1 Proof of Lemma 1.1

Proof. This is proved directly by the consistency property of the command label.

A.2 Proof of Lemma 2.1

Proof. Taking arbitrary trace τ ∈ T, by induction on program c, we have the following cases:

**case:** \( c = [x - e] \)

By the evaluation rule asssn, we have \( ([x - a], e) \to (\text{skip}, \tau :: (x, l, v, \bullet)) \), for some \( v \in \mathbb{N} \).

Picking \( τ' = \tau :: (x, l, v, \bullet) \) and \( τ'' = (\tau :: (x, l, v, \bullet)) \), it is obvious that \( τ \cdot τ'' = τ' \). This case is proved.

**case:** \( c = [x - \text{query}(y)] \)

This case is proved in the same way as **case:** \( c = [x - e] \).

**case:** while \( [b] \) do \( c \)

By the first rule applied to \( c \), there are two cases:

**sub-case:** while-t

If the first rule applied to \( c \) is while-t, we have

\[ \langle \text{while } [b] \text{ do } c_w, \tau \rangle \to \langle c_w, \text{while } [b] \text{ do } c_w, \tau :: (b, \text{l}, \text{true}, \bullet) \rangle \]

Let \( τ'_w \in T \) be the trace satisfying following execution:

\[ \langle c_w, \tau :: (b, \text{l}, \text{true}, \bullet) \rangle \to \langle \text{skip}, τ'_w \rangle \]

By induction hypothesis on sub program \( c_w \) with starting trace \( \tau :: (b, \text{l}, \text{true}, \bullet) \) and ending trace \( τ'_w \), we know there exist \( τ'_w \in \tau \) such that \( τ'_w = \tau :: (b, \text{l}, \text{true}, \bullet) \).

Then we have the following execution continued from (1):

\[ \langle \text{while } [b] \text{ do } c_w, \tau \rangle \to \langle c_w, \text{while } [b] \text{ do } c_w, \tau :: (b, \text{l}, \text{true}, \bullet) \rangle \to \langle \text{while } [b] \text{ do } c_w, \tau :: (b, \text{l}, \text{true}, \bullet) \rangle \to \cdots \]

By repeating the execution (1) and (2) until the program is evaluated into skip, with trace \( τ'_w \) for \( i = 1, \ldots, n \in \mathbb{N} \) in each iteration, we know in the \( i \text{-}th \) iteration, there exists \( τ'_w \in \tau \) such that

\[ τ'_w = \tau^{(i-1)} :: (b, \text{l}, \text{true}, \bullet) + + + + + \]

Then we have the following execution:

\[ \langle \text{while } [b] \text{ do } c_w, \tau \rangle \to \langle c_w, \text{while } [b] \text{ do } c_w, \tau :: (b, \text{l}, \text{true}, \bullet) \rangle \to \langle \text{while } [b] \text{ do } c_w, \tau :: (b, \text{l}, \text{true}, \bullet) \rangle \to \cdots \]

**sub-case:** while-f

If the first rule applied to \( c \) is while-f, we have

\[ \langle \text{while } [b] \text{ do } c_w, \tau \rangle \to \langle \text{skip}, \tau :: (b, \text{l}, \text{false}, \bullet) \rangle \]

Picking \( \tau' = \tau :: (b, \text{l}, \text{false}, \bullet) \) and \( \tau'' = (\tau :: (b, \text{l}, \text{false}, \bullet)) \), we have

\[ τ \cdot + + + + + = τ' \]

This case is proved.

**case:** if \( [b], c_t, c_f \)

This case is proved in the same way as **case:** \( c = \text{while } [b] \) do \( c \).
**A.3 Proof of Lemma 2.0.1**

*Proof.* By unfolding the $aq \in aq \cdot t$, we have the following cases:

- **case:** $t = []$
  The hypothesis is false, this case is proved.

- **case:** $t = aq' :: t' \land aq' = aq aq$
  Let $t_1 = []$ and $t_2 = t'$, by unfolding the list concatenation operation, we have:
  \[ t_1 + +[aq'] + +t_2 = [] + +[aq'] + +t' = t \]
  Since $aq' = aq aq$ by the hypothesis, this case is proved.

- **case:** $t = aq' :: t' \land aq' \neq aq aq$
  By induction hypothesis on $aq \in aq t'$, we know:
  \[ \exists t_1', t_2', aq'' \cdot s.t., (aq = aq aq'') \land t_1' + +[aq''] + +t_2' = t' \]
  Let $t_1 = aq' :: t_1'$ and $t_2 = t_2'$, by unfolding the list concatenation operation, we have:
  \[ t_1 + +[aq''] + +t_2 = (aq' :: t_1') + +[aq''] + +t_2' = aq' :: t' = t \]
  Since $aq'' = aq aq$ by the hypothesis, this case is proved.

**B Proof of Theorem 4.2**

**Theorem B.1** *(Soundness of AdaptFun).* For every program $c$, its estimated adaptivity is a sound upper bound of its adaptivity.

\[ \forall \tau_0 \in T_0(c), v \in \mathbb{N}^\omega \cdot (A_{est}(c), \tau_0) \Downarrow v \implies A(c)(\tau_0) \leq c \]

*Proof Summary:*

1. prove the one-on-one mapping from $V_{est}$ to $V_{trace}$, in Lemma [B.1];
2. prove the total map from $E_{trace}$ to $E_{est}$, in Lemma [B.2];
3. prove that the weight of every vertex in $G_{trace}$ is bounded by the weight of the same vertex in $G_{est}$, in Lemma [B.3];
4. prove the one-on-one mapping from $Q_{est}$ to $Q_{trace}$, in Lemma [B.4];
5. show every walk in $WK(G_{trace})$ is bounded by a walk in $WK(G_{est})$ of the same len$^q$.
6. get the conclusion that $A(c)$ is bounded by the $A_{est}(c)$.

*Proof.* Given a program $c$, we construct its program-based graph $G_{est}(c) = (V_{est}, E_{est}, W_{est}, Q_{est})$ by Definition 24 and trace-based graph $G_{trace}(c) = (V_{trace}, E_{trace}, W_{trace}, Q_{trace})$ by Definition 8.
The parameter \((c)\) for the components in the two graphs are omitted for concise.

According to the Definition 27 and Definition 14, it is sufficient to show:

\[
\forall \tau \in T . \ (\max\{1\text{en}^3(k) \mid k \in \mathcal{W}X(G_{\text{est}}(c))\}, \tau) \not\subseteq n \iff n \geq \max\{1\text{en}^3(k) \mid k \in \mathcal{W}X(G_{\text{trace}}(c))\}
\]

Then it is sufficient to show that:

\[
\forall k_{\tau} \in \mathcal{W}X(G_{\text{trace}}(c)), \exists k_{\tau} \in \mathcal{W}X(G_{\text{est}}(c)) . \ \forall \tau \in T . \ 1\text{en}^3(k_{\tau}), \tau \not\subseteq n \iff n \geq 1\text{en}^3(k_{\tau}(\tau))
\]

Let \(k_{\tau} \in \mathcal{W}X(G_{\text{trace}}(c))\) be an arbitrary walk in \(G_{\text{trace}}(c)\), and \(\tau \in T\) be arbitrary trace.
Then, let \((e_{p1}, \ldots, e_{p(n-1)})\) and \((v_1, \ldots, v_n)\) be the edges and vertices sequence for \(k_{\tau}(\tau)\).

By Lemma B.1 and Lemma B.2, we know

\[
\forall e_i \in k_{\tau} . \ e_i = (v_i, v_{i+1}) \implies \exists e_{p_i} . \ e_{p_i} = (v_i, v_{i+1}) \land e_{p_i} \in E_{\text{est}}
\]

Then we construct a walk \(k_p\) with an edge sequence \((e_{p1}, \ldots, e_{p(n-1)})\) with a vertices sequence \((v_1, \ldots, v_n)\) where \(e_{p_i} = (v_i, v_{i+1}) \in E_{\text{est}}\) for all \(e_{p_i} \in (e_{p1}, \ldots, e_{p(n-1)})\).

Let \(n \in \mathbb{N}\) such that \(1\text{en}^3(k_p), \tau \not\subseteq n\), then, it is sufficient to show

\[
k_p \in E_{\text{est}}(c) \land n \geq 1\text{en}^3(k_{\tau}(\tau))
\]

To show \(k_p \in E_{\text{est}}(c)\), by Definition 12 for finite walk, we know

\[
\forall v_i \in (v_1, \ldots, v_n), (v_i, w_i) \in W_{\text{trace}}(c) . \ \text{visit}(v_1, \ldots, v_n), (v_i) \leq w_i(\tau)
\]

By Lemma B.3, we know for every

\[
\forall v_i \in (v_1, \ldots, v_n), (v_i, w_i) \in W_{\text{est}}(c), n_i \in \mathbb{N} . \ \langle w_i, \tau \rangle \not\subseteq n_i \iff w_i(\tau) \leq n_i (*)
\]

Then, by Definition 25, we know the occurrence times for every \(v_i \in (v_1, \ldots, v_n)\) is bound by the arithmetic expression \(w_i\) where \((v_i, w_i) \in W_{\text{est}}(c)\).
So we have \(k_p \in \mathcal{W}X(G_{\text{est}})\) proved.

In order to show \(n \geq 1\text{en}^3(k_{\tau}(\tau))\), it is sufficient to show

\[
\forall v_i \in (v_1, \ldots, v_n), (v_i, w_i) \in W_{\text{est}}(c), (v_i, w_{i}') \in W_{\text{trace}}(c), n_i \in \mathbb{N} . \ \langle w_i, \tau \rangle \not\subseteq n_i \implies \sum_{Q_{\text{trace}}(\tau)(v_i)=1} w_{i}'(\tau) \leq \sum_{Q_{\text{est}}(c)(v_i)=1} n_i
\]

By Lemma B.4 and Definition 26, we know for every \(v_i, Q_{\text{trace}}(\tau)(v_i) = Q_{\text{est}}(c)(v_i)\)
Then by (*), we know \(\sum_{Q_{\text{trace}}(\tau)(v_i)=1} w_{i}'(\tau) \leq \sum_{Q_{\text{est}}(c)(v_i)=1} n_i\).
Then we have \(n \geq 1\text{en}^3(k_{\tau}(\tau))\) proved.
This theorem is proved.

The following are the four lemmas used above, showing the correspondence properties between the program based graph and trace based graph.

**Lemma B.1 (One-on-One Mapping of vertices from \(G_{\text{trace}}\) to \(G_{\text{est}}\)).** Given a program \(c\) with its program-based graph \(G_{\text{est}}(c) = (V_{\text{est}}, E_{\text{est}}, W_{\text{est}}, Q_{\text{est}})\) and trace-based graph \(G_{\text{trace}}(c) = (V_{\text{trace}}, E_{\text{trace}}, W_{\text{trace}}, Q_{\text{trace}})\), then for every \(v \in \mathcal{VAR} \times \mathbb{N}\), \(v \in V_{\text{est}}\) if and only if \(v \in V_{\text{trace}}\).

\[
\forall c \in C, \forall v \in \mathcal{VAR} \times \mathbb{N} . \ G_{\text{est}}(c) = (V_{\text{est}}, E_{\text{est}}, W_{\text{est}}, Q_{\text{est}}) \land G_{\text{trace}}(c) = (V_{\text{trace}}, E_{\text{trace}}, W_{\text{trace}}, Q_{\text{trace}}) \implies v \in V_{\text{est}} \iff v \in V_{\text{trace}}
\]
Proof. Proof Summary: Proving by Definition 24 and Definition 8

Taking arbitrary program \( c \), by Definition 24 and Definition 8 we have its program-based graph \( G_{\text{est}}(c) = (V_{\text{est}}, E_{\text{est}}, W_{\text{est}}, Q_{\text{est}}) \) and trace-based graph \( G_{\text{trace}}(c) = (V_{\text{trace}}, E_{\text{trace}}, W_{\text{trace}}, Q_{\text{trace}}) \).

By the two definitions, we also know \( V_{\text{trace}} = \emptyset \cup V_c \) and \( V_{\text{est}} = \emptyset \cup V_c \).

Then we know \( V_{\text{trace}} = V_{\text{est}} \), i.e., for arbitrary \( v \in \mathbb{V} \times \mathbb{N} \), \( v \in V_{\text{est}} \iff v \in V_{\text{trace}}. \)

Lemma B.2 (Mapping from Edges of \( G_{\text{trace}} \) to \( G_{\text{est}} \)). Given a program \( c \) with its program-based graph \( G_{\text{est}}(c) = (V_{\text{est}}, E_{\text{est}}, W_{\text{est}}, Q_{\text{est}}) \) and trace-based graph \( G_{\text{trace}}(c) = (V_{\text{trace}}, E_{\text{trace}}, W_{\text{trace}}, Q_{\text{trace}}) \), then for every \( e = (u, v) \in E_{\text{trace}} \), there exists an edge \( \tilde{e} = (u', v') \in E_{\text{est}} \) with \( u_1 = v_1 \wedge v_2 = u_2. \)

\[
\forall e \in E_{\text{trace}}. G_{\text{est}}(c) = (V_{\text{est}}, E_{\text{est}}, W_{\text{est}}, Q_{\text{est}}) \land G_{\text{trace}}(c) = (V_{\text{trace}}, E_{\text{trace}}, W_{\text{trace}}, Q_{\text{trace}}) \rightarrow \exists e' \in E_{\text{est}}. \tilde{e} = (v_1, v_2)
\]

Proof. Proof Summary: Proving by Lemma B.1 Lemma C.1 Definition 24 and Definition 8

Taking arbitrary program \( c \), by Definition 24 and Definition 8 we have its program-based graph \( G_{\text{est}}(c) = (V_{\text{est}}, E_{\text{est}}, W_{\text{est}}, Q_{\text{est}}) \) and trace-based graph \( G_{\text{trace}}(c) = (V_{\text{trace}}, E_{\text{trace}}, W_{\text{trace}}, Q_{\text{trace}}) \).

By Lemma B.1 we know \( x', y' \in V_{\text{est}}. \)

By definition of \( E_{\text{trace}} \), we know \( \text{DEP}_{\text{var}}(x', y', c). \)

By Theorem C.1 we know \( \forall n \in \mathbb{N}, z_1^n, \cdots, z_n^n \in \mathbb{V} \cup V_c \cdot n \geq 0 \text{flowsTo}(x', z_1^n, c) \land \cdots \land \text{flowsTo}(z_n^n, y', c). \)

Then by definition of \( E_{\text{est}} \), we know \( (x', y') \in E_{\text{est}}. \) This Lemma is proved.

Lemma B.3 (Weights are bounded). Given a program \( c \) with its program-based graph \( G_{\text{est}}(c) = (V_{\text{est}}, E_{\text{est}}, W_{\text{est}}, Q_{\text{est}}) \) and trace-based graph \( G_{\text{trace}}(c) = (V_{\text{trace}}, E_{\text{trace}}, W_{\text{trace}}, Q_{\text{trace}}) \), for every \( v \in V_{\text{trace}} \), there is \( v \in V_{\text{est}} \) and \( w_{\text{trace}}(v) \leq w_{\text{est}}(v). \)

\[
\forall e \in E_{\text{trace}}. G_{\text{est}}(c) = (V_{\text{est}}, E_{\text{est}}, W_{\text{est}}, Q_{\text{est}}) \land G_{\text{trace}}(c) = (V_{\text{trace}}, E_{\text{trace}}, W_{\text{trace}}, Q_{\text{trace}}) \rightarrow \forall (x', w_i) \in W_{\text{trace}}, (x', w_i) \in W_{\text{est}}. \nu, \tau \in \mathbb{N}, \nu \leq \nu \leftarrow w_i(\nu) \leq \nu
\]

Proof. Proof Summary: Proving by Definition 24 Definition 8 and relying on the soundness of Reachability Bound Analysis.

By soundness of reachability bound analysis in Theorem D.2 we know \( \text{cnt}(\nu', \ell) \leq \nu. \)

By definition \( \text{8} \) we know \( w_i(\nu) = \text{cnt}(\nu', \ell), \) then we have \( w_i(\nu) \leq \nu \) and this is proved.

Lemma B.4 (One-On-One Mapping for Query Vertices). Given a program \( c \) with its program-based graph \( G_{\text{est}}(c) = (V_{\text{est}}, E_{\text{est}}, W_{\text{est}}, Q_{\text{est}}) \) and trace-based graph \( G_{\text{trace}}(c) = (V_{\text{trace}}, E_{\text{trace}}, W_{\text{trace}}, Q_{\text{trace}}) \), then for every \( (x', n) \in \mathbb{V} \times \mathbb{N} \times \{0, 1\} \), \( (x', n) \in Q_{\text{trace}} \) if and only if \( (x', n) \in Q_{\text{est}}. \)

\[
\forall c \in C, (x', n) \in \mathbb{V} \times \mathbb{N} \times \{0, 1\}. \quad G_{\text{est}}(c) = (V_{\text{est}}, E_{\text{est}}, W_{\text{est}}, Q_{\text{est}}) \land G_{\text{trace}}(c) = (V_{\text{trace}}, E_{\text{trace}}, W_{\text{trace}}, Q_{\text{trace}}) \rightarrow (x', n) \in Q_{\text{trace}} \iff (x', n) \in Q_{\text{est}}.
\]
Proof. Proving by Definition \[24\] Definition \[8\],
Taking arbitrary program \(c\), by Definition \[24\] and Definition \[8\] we have
its program-based graph \(G_{est}(c) = (V_{est}, E_{est}, W_{est}, Q_{est})\)
and trace-based graph \(G_{trace}(c) = (V_{trace}, E_{trace}, W_{trace}, Q_{trace})\).
By the two definitions, we also know \(Q_{trace} = Q_{est}\), i.e., for arbitrary \((x^i, n) \in \mathcal{VAR} \times \mathbb{N} \times \{0, 1\}\),
\((x^i, n) \in Q_{trace} \iff (x^i, n) \in Q_{est}\).
This lemma is proved. ■
C Soundness of Edge Estimation

C.1 Main Theorem

Theorem C.1 (DEP _var_ implies flowsTo). Given a program c, for all \(x^i, y^j \in \text{\texttt{\texttt{lv}}}_c\), if \(\text{DEP}_{\text{\texttt{var}}}(x^i, y^j, c)\), then there exist \(z_{1}^{i}, \ldots, z_{n}^{i} \in \text{\texttt{\texttt{lv}}}_c\) with \(n \geq 0\) such that \(\text{flowsTo}(x^i, z_{1}^{i}, c) \land \cdots \land \text{flowsTo}(z_{n}^{i}, y^j, c)\)

\[
\forall x^i, y^j \in \text{\texttt{\texttt{lv}}}_c. \text{DEP}_{\text{\texttt{var}}}(x^i, y^j, c) \\
\implies \left( \exists n \in \mathbb{N}, z_{1}^{i}, \ldots, z_{n}^{i} \in \text{\texttt{\texttt{lv}}}_c . \ n \geq 0 \land \text{flowsTo}(x^i, z_{1}^{i}, c) \land \cdots \land \text{flowsTo}(z_{n}^{i}, y^j, c) \right)
\]

Proof Summary, for arbitrary two \(x^i, y^j \in \text{\texttt{\texttt{lv}}}_c\), with Variable May-Dependency relation, in order to show there exists a "flows-to chain" relation from the static analysis results from \(x^i\) to \(y^j\), it is sufficient to show:

1. \(x^i\) is directly used in the expression of the assignment command associated to \(y^j\), or a boolean expression of the guard for a if or while command with the assignment command associated to \(y^j\) showing up in the body of that command, we call it, \(x^i\) directly flows to \(y^j\), i.e., \(\text{flowsTo}(x^i, y^j, c)\);
2. otherwise, there exists another labelled variable \(z^i\) with variable May-Dependency relation on \(x^i\) and \(z^i\) directly flows to \(y^j\), where the variable May-Dependency relation between \(x^i\) and \(z^i\) implies a "sub flows-to-chain" from \(z^i\) to \(z^j\), i.e.,

\[
\exists z^i \in \text{\texttt{\texttt{lv}}}_c . (\text{DEP}_{\text{\texttt{var}}}(x^i, z^i, c) \implies \exists n \in \mathbb{N}, z_{1}^{i}, \ldots, z_{n}^{i} \in \text{\texttt{\texttt{lv}}}_c . \ n \geq 0 \land \text{flowsTo}(x^i, z_{1}^{i}, c) \land \cdots \land \text{flowsTo}(z_{n}^{i}, z^j, c) \land \text{flowsTo}(z^i, y^j, c))
\]

By definition of \(\text{DEP}_{\text{\texttt{var}}}(x^i, y^j, c)\), let \(D \in \text{\texttt{\texttt{db}}}\) be the dataset, and \(\tau \in \mathcal{T}\), \(\epsilon_x, \epsilon_y\) be the trace and two events satisfying the definition, with \(\pi_1(\epsilon_x) = x\) and \(\pi_1(\epsilon_y) = y\).

\[
(\text{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c) \\
\lor \exists \epsilon \in \tau^i \cdot \epsilon \in \text{\texttt{\texttt{eass}}} \land \text{DEP}(\epsilon_x, \epsilon_y, \tau[\epsilon_x : \epsilon_y], c, D) \\
\implies \exists n \in \mathbb{N}, z_{1}^{i}, \ldots, z_{n}^{i} \in \text{\texttt{\texttt{lv}}}_c . \ n \geq 0 \land \text{flowsTo}(x^i, z_{1}^{i}, c) \land \cdots \land \text{flowsTo}(z_{n}^{i}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c))
\]

\land \begin{align*}
\text{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)
\end{align*}

It is clearer to show it in two lemmas:

1. Existence of a middle event: in Lemma C.3
2. The middle event with a sub-trace implies a "sub flows-to-chain", by induction on the trace \(\tau\)

\[
\forall D \in \text{\texttt{\texttt{db}}}, c \in C, \tau \in \mathcal{T} . \forall \epsilon_1, \epsilon_2 \in \mathcal{E} . \epsilon_1, \epsilon_2 \in \text{\texttt{\texttt{eass}}} \land \exists \tau' \in \mathcal{T} . \tau = [\epsilon_1], \ldots, \tau', [\epsilon_2] \implies \text{DEP}(\epsilon_1, \epsilon_2, \tau, c, D) \\
\implies \exists n \in \mathbb{N}, z_{1}^{i}, \ldots, z_{n}^{i} \in \text{\texttt{\texttt{lv}}}_c . \ n \geq 0 \land \text{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, z_{1}^{i}, c) \land \cdots \land \text{flowsTo}(z_{n}^{i}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)
\]

with the induction hypothesis:

\[
\forall \epsilon_{i_1h_1}, \epsilon_{i_2h_2} \in \tau . \epsilon_{i_1h_1}, \epsilon_{i_2h_2} \in \text{\texttt{\texttt{eass}}} \land \exists \tau' \in \mathcal{T} . \tau[\epsilon_{i_1h_1} : \epsilon_{i_2h_2}] = [\epsilon_{i_1h_1}], \ldots, \tau', [\epsilon_{i_2h_2}] \implies \text{DEP}(\epsilon_{i_1h_1}, \epsilon_{i_2h_2}, \tau[\epsilon_{i_1h_1} : \epsilon_{i_2h_2}], c, D) \\
\implies \exists n \in \mathbb{N}, z_{1}^{i}, \ldots, z_{n}^{i} \in \text{\texttt{\texttt{lv}}}_c . \ n \geq 0 \land \text{flowsTo}(\pi_1(\epsilon_{i_1h_1})^{\pi_2(\epsilon_{i_1h_1})}, z_{1}^{i}, c) \land \cdots \land \text{flowsTo}(z_{n}^{i}, \pi_1(\epsilon_{i_2h_2})^{\pi_2(\epsilon_{i_2h_2})}, c)
\]

Proved in Theorem C.2 with the main proof logic:

1. obtaining the existence of \(\epsilon_x \in \text{\texttt{\texttt{eass}}}\) with dependency on \(\epsilon_x\), and a "direct flowsto" from \(\epsilon_x\) to \(\epsilon_y\) from step 1.
2. from the dependency of the \(\epsilon_x\) with \(\epsilon_y\) on the subtrace, obtaining a "sub flowsto-chain" by induction hypothesis;
3. composing the "sub flowsto-chain" from (2) with the "direct flowsto" from (1), and getting the conclusion of the complete "flowsto chain".
Proof. Taking arbitrary \(x^i, y^j \in \mathbb{L}_c\), by definition of \(\text{DEP}_{\text{var}}(x^i, y^j, c)\), let \(D \in \mathbb{D} \mathbb{B}\) be the dataset, and \(\tau \in \mathbb{T}, \epsilon_x, \epsilon_y\) be the trace and two events satisfying the definition, with \(\pi_1(\epsilon_x)^{\pi_2(\epsilon_x)} = x^i \) and \(\pi_1(\epsilon_y)^{\pi_2(\epsilon_y)} = y^j\), it is sufficient to show:

\[
\text{DEP}_e(\epsilon_x, \epsilon_y, \tau, c, D) \Rightarrow \exists n \in \mathbb{N}, z_1^n, \ldots, z_n^n \in \mathbb{L}_c. n \geq 0 \land \text{flowsTo}(x^i, z_1^n, c) \land \cdots \land \text{flowsTo}(z_n^n, y^j, c)
\]

By Theorem C.2 we have this theorem proved. \(\square\)

### C.2 Soundness of flowsTo w.r.t. the Event

For concise of the proof, we introduce some conventional operators as follows.

**Definition 28 (Subtrace).** Subtrace: \([:]:\mathbb{T} \rightarrow \mathbb{E} \rightarrow \mathbb{E} \rightarrow \mathbb{T}\)

\[
\tau[\epsilon_1: \epsilon_2] \triangleq \begin{cases} \tau'[\epsilon_1: \epsilon_2] & \tau = \epsilon' \land \epsilon \neq \epsilon_c \\ \epsilon_1 :: \tau'[ : ] & \tau = \epsilon' \land \epsilon = \epsilon_c \\ [ ] & \tau = [ ] \end{cases}
\]

For any trace \(\tau\) and two events \(\epsilon_1, \epsilon_2 \in \mathbb{E}\), \(\tau[\epsilon_1: \epsilon_2]\) takes the subtrace of \(\tau\) starting with \(\epsilon_1\) and ending with \(\epsilon_2\) including \(\epsilon_1\) and \(\epsilon_2\).

We use \(\tau[: \epsilon_2]\) as the shorthand of subtrace starting from head and ending with \(\epsilon_2\), and similarly for \(\tau[\epsilon_1: ]\).

**Program Entry Point:** \(\text{entry}_c: \text{Command} \rightarrow \mathbb{N}\)

\[
\text{entry}_c \triangleq \begin{cases} l & c = [\text{skip}]\,^l \\ l & c = [x \leftarrow \epsilon_1] \,^l \\ l & c = [x \leftarrow \text{query}(\psi_1)] \,^l \\ l & c_1 = [\text{if} ([b] \,^l, c_t, c_f)] \,^l \\ l & c = [\text{while} [b] \,^l \rightarrow c' \,^l ]\,^l \\ \text{entry}_c & c = c_1; c_2 \end{cases}
\]

**Theorem C.2 (DEP\(_e\) implies flowsTo).** For every \(D \in \mathbb{D} \mathbb{B}, c \in \mathbb{E}, \tau \in \mathbb{T}\). \(\forall \epsilon_1, \epsilon_2 \in \mathbb{E} . \epsilon_1, \epsilon_2 \in \mathbb{E}^{\text{ann}}\), if \(\exists \tau' \in \mathbb{T} . \tau = [\epsilon_1] \cdots [\epsilon_2]\) and \(\text{DEP}_e(\epsilon_1, \epsilon_2, \tau, c, D)\), then \(z_1^n, \ldots, z_n^n \in \mathbb{L}_c\) with \(n \geq 0\) such that \(\text{flowsTo}(x^i, z_1^n, c) \land \cdots \land \text{flowsTo}(z_n^n, y^j, c)\)

\[
\forall D \in \mathbb{D} \mathbb{B}, c \in \mathbb{E}, \tau \in \mathbb{T} . \exists \epsilon_1, \epsilon_2 \in \mathbb{E} . \epsilon_1, \epsilon_2 \in \mathbb{E}^{\text{ann}} \land \exists \tau' \in \mathbb{T} . \tau = [\epsilon_1] \cdots [\epsilon_2] \Rightarrow \text{DEP}_e(\epsilon_1, \epsilon_2, \tau, c, D) \Rightarrow \exists n \in \mathbb{N}, z_1^n, \ldots, z_n^n \in \mathbb{L}_c. n \geq 0 \land \text{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, z_1^n, c) \land \cdots \land \text{flowsTo}(z_n^n, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)
\]

**Proof Summary:** I. Vacuously True cases, where trace doesn’t satisfy the hypothesis
II. Base case where \(\tau = [\epsilon_1; \epsilon_2]\)
III. Inductive case where \(\tau = [\epsilon_1, \cdots, \epsilon_2]\).

1. Existence of a middle event:
   - Proved by showing a contradiction, with detail in Lemma C.3
2. The middle event with a sub-trace implies a "sub flowsto-chain", informally:

46
(1) obtaining the existence of $e_z \in \mathcal{E}^{\text{ann}}$ with dependency on $e_x$, and a "direct flowsto" from $e_z$ to $e_y$ by Lemma C.3.

(2) from the dependency of the $e_z$ with $e_x$ on the subtrace, obtaining a "sub flowsto-chain" by induction hypothesis;

(3) composing the "sub flowsto-chain" from (2) with the "direct flowsto" from (1), and getting the conclusion of the complete "flowsto chain".

**Proof.** Taking arbitrary $D \in \mathcal{D}B, c \in C$, by induction on the trace $\tau$ we have the following cases:

**Case 1.** $(\tau = [])$

Since for all $e_1, e_2 \in \mathcal{E}^{\text{ann}}$ it holds $[], [e_1] \supseteq [e_1] \supseteq [e_2]$, the theorem is vacuously true.

**Case 2.** $(c \in \mathcal{E}, \tau = [c])$

Since for all $e_1, e_2 \in \mathcal{E}^{\text{ann}}$ it holds $\forall \mathcal{T}, [e_1] \supseteq [e_1] \supseteq [e_2]$, the theorem is vacuously true.

**Case 3.** $(e_1', e_2' \in \mathcal{E}, \tau = [e_1', e_2'])$

To show:

\[
\forall e_1, e_2 \in \mathcal{E}^{\text{ann}} \cdot \exists \mathcal{T} \in \mathcal{T}. [e_1'] \supseteq [e_1] \supseteq [e_2] \\
\quad \implies \text{DEP}(e_1, e_2, e_1, e_2), [c, D] \implies \text{flowsTo}(\pi_1(e_1)^{\pi_2(e_1)}, \pi_1(e_2)^{\pi_2(e_2)}, c)
\]

Taking arbitrary $e_1, e_2 \in \mathcal{E}^{\text{ann}}$, by law of excluded middle, there are 2 cases:

\[
e_1 = e_1' \land e_2 = e_2'.
\]

In case of $e_1 = e_1'$ and $e_2 = e_2'$, let $\tau' = [\cdot]$, we know $\forall \mathcal{T} \in \mathcal{T}$ satisfying $[e_1; e_2] = [e_1] \supseteq [e_2]$. Then it is sufficient to show:

\[
\text{DEP}(e_1, e_2, e_1, e_2), [c, D] \implies \text{flowsTo}(\pi_1(e_1)^{\pi_2(e_1)}, \pi_1(e_2)^{\pi_2(e_2)}, c)
\]

By Lemma C.1 we have this case proved.

**Case 4.** $(e_1', e_2' \in \mathcal{E}, \tau_{ih} \in \mathcal{T}, \tau = [e_1'] \supseteq [e_1] \supseteq [e_2'] \supseteq [\tau_{ih} \neq []])$

It is sufficient to show:

\[
\forall e_1, e_2 \in \mathcal{E} \cdot e_1, e_2 \in \mathcal{E}^{\text{ann}} \land \exists \mathcal{T} \in \mathcal{T}. [e_1] \supseteq [e_1] \supseteq [e_2] \\
\quad \implies \text{DEP}(e_1, e_2, e_1, e_2), [c, D] \\
\quad \implies \exists n \in \mathbb{N}, z_{1, n}^1, \ldots, z_{1, n}^r \in \mathcal{L}V_c \cdot n \geq 0 \land \text{flowsTo}(\pi_1(e_1)^{\pi_2(e_1)}, z_{1, n}^i, c) \land \ldots \land \text{flowsTo}(z_{n, n}^r, \pi_1(e_2)^{\pi_2(e_2)}, c)
\]

with the induction hypothesis:

\[
\forall e_{ih1}, e_{ih2} \in \mathcal{E} \cdot e_{ih1}, e_{ih2} \in \mathcal{E}^{\text{ann}} \land \exists \mathcal{T} \in \mathcal{T}. [e_{ih1}] \supseteq [e_{ih1}] \supseteq [e_{ih2}] \\
\quad \implies \text{DEP}(e_{ih1}, e_{ih2}, e_{ih1}, e_{ih2}), [c, D] \\
\quad \implies \exists n \in \mathbb{N}, z_{ih1}^1, \ldots, z_{ih2}^n \in \mathcal{L}V_c \cdot n \geq 0 \land \text{flowsTo}(\pi_1(e_{ih1})^{\pi_2(e_{ih1})}, z_{ih1}^i, c) \land \ldots \land \text{flowsTo}(z_{ih2}^r, \pi_1(e_{ih2})^{\pi_2(e_{ih2})}, c)
\]

Taking arbitrary $e_1, e_2 \in \mathcal{E}^{\text{ann}}$, by law of excluded middle, there are 2 cases:

\[
e_1 = e_1' \land e_2 = e_2'.
\]

In case of $e_1 = e_1'$ and $e_2 = e_2'$, let $\tau = \tau_{ih}$, we know $\exists \mathcal{T} \in \mathcal{T}$ satisfying $[e_1'] \supseteq [e_1] \supseteq [e_2'] \supseteq [\tau_{ih}]$. To show:

\[
\text{DEP}(e_1, e_2, e_1, e_2), [c, D] \\
\quad \implies \exists n \in \mathbb{N}, z_{1, n}^1, \ldots, z_{1, n}^r \in \mathcal{L}V_c \cdot n \geq 0 \land \text{flowsTo}(\pi_1(e_1)^{\pi_2(e_1)}, z_{1, n}^i, c) \land \ldots \land \text{flowsTo}(z_{n, n}^r, \pi_1(e_2)^{\pi_2(e_2)}, c)
\]
By Lemma C.3 we know:

\[ \text{flowsTo}(\pi_1(e_1)_{\pi_2(e_1)}, \pi_1(e_2)_{\pi_2(e_2)}, c) \]
\[ \forall \varepsilon \in \tau_{ih} \cdot \text{DEP}_e(e_1, e_2, \tau[e_1; c], c, D) \land \text{flowsTo}(\pi_1(e_1)_{\pi_2(e_1)}, \pi_1(e_2)_{\pi_2(e_2)}, c) \]

In first case, we have \( \text{flowsTo}(\pi_1(e_1)_{\pi_2(e_1)}, \pi_1(e_2)_{\pi_2(e_2)}, c) \) proved directly.
In the second case, let \( \varepsilon_{ih} \) be this event, from the induction hypothesis, we know:

\[ \exists n \in \mathbb{N}, z^n_1, \ldots, z^n_n \in \mathbb{L} \cdot n \geq 0 \land \text{flowsTo}(\pi_1(e_1)_{\pi_2(e_1)}, z^n_1, c) \land \cdots \land \text{flowsTo}(z^n_n, \pi_1(e_{ih})_{\pi_2(e_{ih})}, c) \]

Then we know:

\[ \exists n \in \mathbb{N}, z^n_1, \ldots, z^n_n \in \mathbb{L} \cdot n \geq 0 \land \text{flowsTo}(\pi_1(e_1)_{\pi_2(e_1)}, z^n_1, c) \land \cdots \land \text{flowsTo}(z^n_n, \pi_1(e_{ih})_{\pi_2(e_{ih})}, c) \]
\[ \land \text{flowsTo}(\pi_1(e)_{\pi_2(e)}, \pi_1(e_2)_{\pi_2(e_2)}, c) \]

This case is proved.

C.3 Inversion Lemmas and Helper Lemmas

The following are the inversion lemmas and helper lemmas used in the proof of Theorem C.2 above, showing the correspondence properties between the trace based semantics and the program analysis results.

Lemma C.1 (The One-Step Event Dependency Inversion). For every \( c \in E, D \in D \cdot B \) and two assignment events \( e_1, e_2 \in E^{\text{ass}}, \) if \( \text{DEP}_e(e_1, e_2, [e_1; e_2], c, D), \) then, \( \text{flowsTo}(\pi_1(e_1)_{\pi_2(e_1)}, \pi_1(e_2)_{\pi_2(e_2)}, c) \).

\[ \forall e_1, e_2 \in E^{\text{ass}}, c \in E, D \in D \cdot B \cdot \text{DEP}_e(e_1, e_2, [e_1; e_2], c, D) \]
\[ \Rightarrow \text{flowsTo}(\pi_1(e_1)_{\pi_2(e_1)}, \pi_1(e_2)_{\pi_2(e_2)}, c) \]

Proof Summary:
1. case of (the labelled unique assignment command associated to the \( e_2 \) is executed but the value assigned to the variable in this event is changed in second execution)
   show \( x \) directly used by the assignment of the second event
2. (the labelled unique assignment command associated to the \( e_2 \) isn’t executed in second execution)
   show \( x \) is directly used by the boolean expression for a conditional command and second event shows in the body of that conditional command

Proof. By the Definition[10] for \( \text{DEP}_e(e_1, e_2, [e_1; e_2], c, D) \), we know there are 2 cases:

**case 1**
(the labelled unique assignment command associated to the \( e_2 \) is executed but the value assigned to the variable in this event is changed in second execution).

**Proof of the Basecase: Case 1.** We have the following by the definition \( \text{DEP}_e(e_1, e_2, [e_1; e_2], c, D) \) for case 1:

\[
\begin{align*}
\exists \tau_0, \tau_1, \tau' \in \mathcal{T}, e'_1 &\in E^{\text{ass}}, e'_2 \in E, c_1, e_2 \in E \cdot \text{Diff}(e_1, e'_1) \land \\
&\left( \begin{array}{l}
\langle c, \tau_0 \rangle \rightarrow^* \langle c_1, \tau_1 \rangle \cdots \langle e_1 \rangle \rightarrow^* \langle c_2, \tau_1 \rangle \cdots \langle e_2 \rangle \\
\land \langle c_1, \tau_1 \rangle \rightarrow^* \langle e'_1 \rangle \rightarrow^* \langle c_2, \tau_1 \rangle \cdots \langle e'_2 \rangle \\
\land \text{Diff}(e_1, e_2) \land \text{cnt}(\tau) \pi_2(e_2) = \text{cnt}(\tau') \pi_2(e_2)
\end{array} \right)
\end{align*}
\]
Let \( \tau_0, \tau_1, \tau' \in \mathcal{T}, e_1, e_2 \in \mathcal{E}, e_1', e_2' \in \mathcal{E}_\text{assn}, c_1, c_2 \) be the traces, events and commands satisfying the executions, by Inversion Lemma C.10 on \( e_1, e_2 \), we have the following instance of the first execution in Eq. 4.

\[
\langle c, \tau_0 \rangle \rightarrow^* \langle [x_1 \leftarrow e_1 / \text{query}(\psi_1)] / \pi_2(e_1) ; c_1, \tau_1 \rangle \rightarrow_{\text{assn}} \langle c_1, \tau_1 + [e_1] \rangle
\]

\[
\quad -^* \langle [x_2 \leftarrow e_2 / \text{query}(\psi_2)] / \pi_2(e_2) ; c_2, \tau_1 + [e_1] \rangle \rightarrow_{\text{assn}} \langle c_2, \tau_1 + [e_1; e_2] \rangle
\]

(5)

, where \( x_1 = \pi_1(e_1), l_1 = \pi_2(e_1), x_2 = \pi_1(e_2), l_2 = \pi_2(e_2) \), and \( e_1 / \psi_1, e_2 / \psi_2 \) are the expressions of the assignment commands associated to the \( e_1 \) and \( e_2 \) from Lemma C.7.

By \( \text{Diff}(e_2, e'_2) \) and the command label consistency, we also have the instance of second execution in Eq. 4 as follows:

\[
\langle c_1, \tau_1 + [e_1'] \rangle \rightarrow^* \langle [x_2 \leftarrow e_2 / \text{query}(\psi_2)] / \pi_2(e_2) ; c_2, \tau_1 + [e_1'] \cdot \tau'_2 + [e_2'] \rangle \rightarrow_{\text{assn}} \langle c_2, \tau_1 + [e_1'] \cdot \tau'_2 + [e_2'] \rangle
\]

(6)

From Eq. 4 we also have

\[
\text{cnt}(\tau') l_2 = \text{cnt}([]) l_2 = 0
\]

(7)

By Inversion Lemma C.10 and the execution in Eq. 5 we know:

\[
c_1 = c \cdot [\text{skip}]^* ; [x_2 \leftarrow e_2 / \text{query}(\psi_2)] / l_2 ; c_2
\]

By substituting \( c_1 \) in Eq. 6 the following subproof shows there is only 1 qualified instance of the execution in Eq. 6.

**Subproof.** There are two possibilities by the law of excluded middle:

\[
[x_2 \leftarrow e_2 / \text{query}(\psi_2)] / l_2 \in_c c_2
\]
or \( [x_2 \leftarrow e_2 / \text{query}(\psi_2)] / l_2 \notin_c c_2 \).

1. \( [x_2 \leftarrow e_2 / \text{query}(\psi_2)] / l_2 \notin_c c_2 \)

   In this case, we have the following execution instance:

   \[
   \langle c_1, \tau_1 + [e_1'] \rangle \rightarrow_{\text{skip}}^* \langle [x_2 \leftarrow e_2 / \text{query}(\psi_2)] / l_2 ; c_2, \tau_1 + [e_1'] \rangle \rightarrow_{\text{assn}}^* \langle c_2, \tau_1 + [e_1'] ; c_2 \rangle
   \]

   (8)

2. \( [x_2 \leftarrow e_2 / \text{query}(\psi_2)] / l_2 \in_c c_2 \)

   By Inversion Lemma C.9 we have a while conditional command \( (\text{while } [b_w]_w \text{ do } c_w) \) in \( c_2 \), where \( [x_2 \leftarrow e_2 / \text{query}(\psi_2)] / l_2 \in_c c_w \). Then, we have the following possible execution instances:

   \[
   \langle c_1, \tau_1 + [e_1'] \rangle \rightarrow_{\text{skip}} \langle [x_2 \leftarrow e_2 / \text{query}(\psi_2)] / l_2 ; c_2, \tau_1 + [e_1'] \rangle \rightarrow_{\text{assn}} \langle c_2, \tau_1 + [e_1'] ; c_2 \rangle
   \]

   \[
   \langle c_1, \tau_1 + [e_1'] \rangle \rightarrow_{\text{skip}} \langle [x_2 \leftarrow e_2 / \text{query}(\psi_2)] / l_2 ; c_2, \tau_1 + [e_1'] \rangle \rightarrow_{\text{assn}} \langle c_2, \tau_1 + [e_1'] ; [x_2, l_2, v_2] \rangle
   \]

   \[
   \langle c_2, \tau_1 + [e_1'] \rangle \rightarrow_{\text{assn}} \langle c_2, \tau_1 + [e_1'] \rangle \rightarrow_{\text{assn}} \langle c_2, \tau_1 + [e_1'] \rangle \rightarrow_{\text{assn}} \langle c_2, \tau_1 + [e_1'] \rangle \rightarrow_{\text{assn}} \langle c_2, \tau_1 + [e_1'] \rangle \rightarrow_{\text{assn}} \langle c_2, \tau_1 + [e_1'] \rangle \rightarrow_{\text{assn}} \langle c_2, \tau_1 + [e_1'] \rangle \rightarrow_{\text{assn}} \langle c_2, \tau_1 + [e_1'] \rangle \rightarrow_{\text{assn}} \langle c_2, \tau_1 + [e_1'] \rangle
   \]

   (9)

   , where each execution instance iterates the conditional command \( (\text{while } [b_w]_w \text{ do } c_w) \) in \( c_2 \) 0, 1 or more times.

   For each execution instance, we have the corresponding instance of \( \tau' \) as follows:

---

\( x \leftarrow e / \text{query}(\psi) \) denotes variable \( x \) is assigned by either an expression \( e \) or query \( \text{query}(\psi) \)

\( \left[ \text{skip} \right]^* \) denotes a sequence command only composed of \( \text{skip} \) commands.

\( _-\text{skip}^* \) denotes the rule applied on every evaluation step of this execution is the \text{skip} rule.
\[ \tau' = [] \]
\[ \tau' = [(x_2, l_2, v'_2)] \ldots \tau_3 \]

By Eq. 7 where \( \text{cnt}(\tau')l_2 = 0 \), we know only the first execution instance with 0 iteration of\( \text{while} \) command in \( c_2 \) satisfies this restriction, i.e., \( \tau' = [] \).

In conclusion, we have the only qualified execution instance as follows where \( \tau' = [] \).

\[ \langle e_1, \tau_1, \ldots, [e'_1] \rangle \rightarrow \text{skip}' \langle [x_2 \leftarrow e_2/\text{query}(\psi_2)]^2; c_2, \tau_1, \ldots, [e'_1] \rangle \rightarrow \text{assn/query} \langle e_2, \tau_1, \ldots, [e'_1] \ldots [e'_2] \rangle \]

Then we know by the environment definition, \( \rho \) obtains different values only for variable \( x_1 \) from trace \( \tau_1, \ldots, [e_1] \) and \( \tau_1, \ldots, [e'_1] \), i.e.,

\[ \forall z' \in L \cup \{ x^l_1 \}, \rho(\tau_1, \ldots, [e_1])(z) = \rho(\tau_1, \ldots, [e'_1])(z) \]

By Inversion Lemma C.5 of arithmetic expression evaluation, we have

\[ x_1 \in \text{VAR}(e_2/\psi_2) \]

Since \( t(\tau_1, \ldots, [e_1])x_1 = l_1 \), by Inversion Lemma C.8 we know \( x^l_1 \in \text{RD}(l_2, c) \).

By \text{flowsTo} definition, we have:

\[ \text{flowsTo}(x^l_1, x^l_2, c) \]

i.e.,

\[ \text{flowsTo}(\pi_1(e_1)^{\pi_2(e_1)}, \pi_1(e_2)^{\pi_2(e_2)}, c) \]

This case is proved.

\noindent \textbf{case: 2 (the labelled unique assignment command associated to the \( e_2 \) isn’t executed in second execution).}

\noindent \textbf{Proof Summary:}

1. Let \( e_b \) be the testing event, in the same way of case 1, we get: \( \pi_1(e_1) \in \text{VAR}(\pi_1(e_b)) \land \pi_1(e_1)^{\prime} \in \text{RD}(l^b_2, c) \)

2. By Lemma C.2 we know: \( \forall z \in \text{VAR}(\pi_1(e_b)) \cdot \exists i \in \mathbb{N} \cdot \text{flowsTo}(z, \pi_1(e)^{\pi_2(e)}, c) \)

3. By \text{flowsTo} definition we have: \( \text{flowsTo}(\pi_1(e_1)^{\pi_2(e_1)}, \pi_1(e_2)^{\pi_2(e_2)}, c) \)

\noindent \textbf{Proof of the Basecase: Case 2.} We have the following by the definition \( \text{DEP}_e(e_1, e_2, [e_1; e_2], c, D) \) of case 2:

\[ \exists \tau_0, \tau_1, \tau', \tau_3, \tau'_3 \in \mathcal{T}, \epsilon \in \mathcal{E}^{\text{ass}}, e_1, e_2 \in \mathcal{E}, e_b \in \mathcal{E}^{\text{test}} . \]
\[ \text{Diff}(e_1, e'_1) \land \langle c, \tau_0 \rangle \rightarrow^* \langle c_1, \tau_1, \ldots, [e_1] \rangle \rightarrow^* \langle c_2, \tau_1, \ldots, [e_1; e_b], \ldots \tau_3 \rangle \]
\[ \land \langle c_1, \tau_1, \ldots, [e'_1] \rangle \rightarrow^* \langle c_2, \tau_1, \ldots, [e'_1], \ldots \tau'_1 \rangle \]
\[ \land \mathbb{T}_{\epsilon_1} \cap \mathbb{T}_{\epsilon_2} = \emptyset \land \text{cnt}(\tau') \pi_2(e_b) = \text{cnt}(\tau) \pi_2(e_b) \land \epsilon_2 \epsilon_3 \land \epsilon_2 \not\in \epsilon_3 \]

Let \( \tau_0, \tau_1, \tau', \tau_3, \tau'_3 \in \mathcal{T}, \epsilon \in \mathcal{E}, e'_1 \in \mathcal{E}^{\text{ass}}, e_b, e_1, e_2 \) be the traces, events and commands satisfying the executions, by Inversion Lemma C.7 on \( e_1, e_2, \) and \( e_b \), we have the following instance of the first
This case is proved.

From Eq. 8, we also have

\[ c, t_0 \rightarrow^* \langle x_1 \leftarrow e_1/\text{query}(\psi_1) \rangle^{b_1}; c_1, t_1 \rangle \rightarrow^{\text{assign}} \langle c_1, t_1 \cdots | e_1 \rangle \]

\[ \rightarrow^* \langle \text{if } [(b)|^{b_1}, c_1, f] / \text{while } (b)|^{b_1} \text{ do } c_w; c_3, t_1 \cdots | e_1 \rangle \]

\[ \rightarrow^* \langle \text{if } b / \text{while } b \langle (c_i; c_i')/^{c_4} / c_6/w; \text{while } (b)|^{b_1} \text{ do } c_w; c_3, t_1 \cdots | e_1; e_b \rangle \]

\[ \rightarrow^* \langle c_3, t_1 \cdots | e_1; e_b \cdots t_3 \rangle \]  \hspace{1cm} (9)

, where \( x_1 = \pi_1(e_1) \), \( l_1 = \pi_2(e_1) \), and \( \text{if } [(b)|^{b_1}, c_1, f] / \text{while } (b)|^{b_1} \text{ do } c_w \) is the conditional command of the assignment commands associated to the \( e_b \) from Inversion Lemma \( C.7 \) of testing event.

By the command label consistency, we also have the instance of second execution in Eq. 8 as follows:

\[ \langle c, t_0 \rangle \rightarrow^* \langle x_1 \leftarrow e_1/\text{query}(\psi_1) \rangle^{b_1}; c_1, t_1 \rangle \rightarrow^{\text{assign}} \langle c_1, t_1 \cdots | e_1 \rangle \]

\[ \rightarrow^* \langle \text{if } [(b)|^{b_1}, c_1, f] / \text{while } (b)|^{b_1} \text{ do } c_w; c_3, t_1 \cdots | e_1 \cdots t_3 \rangle \]

\[ \rightarrow^* \langle \text{if } b / \text{while } b \langle (c_i; c_i')/^{c_4} / c_6/w; \text{while } (b)|^{b_1} \text{ do } c_w; c_3, t_1 \cdots | e_1 \cdots t_3 \cdots | e_b \rangle \]  \hspace{1cm} (10)

From Eq. 8, we also have \( \text{cnt}(t')|^{l_b} = \text{cnt}(|) = 0 \).

By the same proof steps from case 1 in Subproof C.3, we have

\[ x_1 \in \text{VAR}(b) \land x_1|^{l_i} \in \text{RD}(l_b, c) \]

By Lemma \( C.2 \) we also know:

\[ \forall z \in \text{VAR}(\pi_1(e_b)) \exists i \in \mathbb{N} . \text{flowsTo}(z^i, \pi_1(e)^{\pi_2(e)}, c) \]

Then by \( \text{flowsTo} \) definition, we have \( \text{flowsTo}(x_1|^{l_i}, x_2|^{l_j}, c) \) i.e.,

\[ \text{flowsTo}(\pi_1(e)^{\pi_2(e)}, \pi_1(e)^{\pi_2(e)}, c) \]

This case is proved.

Lemma \( C.2 \) (Control Dependency Inversion). For every \( c \in \mathcal{C} \), \( D \in \mathcal{DB}, t \in \mathcal{T} \) and two assignment events \( e_1, e_2 \in \mathcal{E}^{\text{ann}}, \) if they are in the second case of the Event May-Dependency relation from Definition \( C.7 \) \( \text{DEP}_e(c, c, t, D) \) as Eq. 11 then for all \( z \in \text{VAR}(\pi_1(e_b)) \) there exists a label \( l \in \mathbb{N} \) such that \( \text{flowsTo}(z^j, \pi_1(e)^{\pi_2(e)}, c) \)

\[ \forall D \in \mathcal{DB}, c \in \mathcal{C}, t \in \mathcal{T}, e_1, e_2 \in \mathcal{E}^{\text{ann}} . \]

\[ \exists \tau_0, \tau_1, \tau_2, \tau_3, \tau_4, \in \mathcal{T}, e_1 \in \mathcal{E}^{\text{ann}}, c_1, c_2 \in \mathcal{C}, e_b \in \mathcal{E}^{\text{test}}, \tau|^{b_1} \in \mathcal{T} . \tau = | e_1 \cdots t_{|^{b_1}} \cdots | e_2 \]

\[ \Longrightarrow \langle c, \tau_0 \rangle \rightarrow^* \langle c_1, t_1 \cdots | e_1 \rangle \rightarrow^* \langle c_2, t_1 \cdots | e_1 \cdots t_3 \rangle \]

\[ \land \langle c_1, t_1 \cdots | e_1 \rangle \rightarrow^* \langle c_2, t_1 \cdots | e_1 \cdots t_3 \rangle \]

\[ \land \forall \tau \in \mathcal{T} \cup \mathcal{T}_3 \phi = \emptyset \land \text{cnt}(t') \pi_2(e_b) = \text{cnt}(t) \pi_2(e_b) \land e_e \land e_3 \land e_2 \notin \theta \tau_3 \]

\[ \Longrightarrow \forall z \in \text{VAR}(\pi_1(e_b)) \exists i \in \mathbb{N} . \text{flowsTo}(z^i, \pi_1(e)^{\pi_2(e)}, c) \]

Proof Summary:

Proving by using the Inversion Lemmas \( C.5 \), \( C.6 \), \( C.7 \) and \( C.8 \) and the Event May-Dependency definition of the second case.
Proof. Take arbitrary $D \in \mathcal{DB}, c \in \mathcal{C}, r \in \mathcal{T}, e_1, e_2 \in \mathcal{E}^{\text{asn}}$, let $\tau_0, \tau_1, \tau', \tau_3, \tau'_3 \in \mathcal{T}, e'_2 \in \mathcal{E}, e'_1 \in \mathcal{E}^{\text{asn}}, c_1, c_2$ be the traces, events and commands satisfying the executions, by Inversion Lemma C.7 on $e_2$, and $e_b$, we have the following instance of the first execution in Eq. [11]

$$
\langle c, \tau_0 \rangle \rightarrow^* \langle |[b|b; c_1, c_f], \text{while}[b|b; c_0; c_1, \tau_1]; \tau \rangle
$$

$$
\rightarrow^* \langle |[x_2 \rightarrow \text{query}(\psi_2)|b|c_1, c_3]|b|c_0; c_1, \tau_1]; \tau \rangle
$$

$$
\rightarrow^* \langle |\text{asl}|\text{query}(c_3|b|c_1, c_3); c_0; c_1, \tau_1]; \tau \rangle
$$

where $\tau_3 = \tau_3 \rightarrow \tau_3 b \rightarrow \tau_3 b$, $x_2 = \pi_1(e_2)$, $I_2 = \pi_2(e_2)$, and $|[(b|b; c_1, c_f); \text{while}[b|b; c_0; c_1, \tau_1]; \tau \rangle$ do the conditional command of the assignment commands associated to the $e_b$ from Inversion Lemma C.7 of testing event.

The notation $(c_1|c_2|/c_3)/c_4; c_5 \rightarrow \text{query}(\psi_2)|b|c_0; c_1, \tau_1]; \tau \rangle$ represents:

In case of $|[(b|b; c_1, c_f), \pi_2(e_b) = \text{true}, we have the evaluation:

$$
\langle c, \tau_0 \rangle \rightarrow^* \langle |[b|b; c_1, c_f], \text{while}[b|b; c_0; c_1, \tau_1]; \tau \rangle
$$

$$
\rightarrow^* \langle |[x_2 \rightarrow \text{query}(\psi_2)|b|c_1, c_3]|b|c_0; c_1, \tau_1]; \tau \rangle
$$

$$
\rightarrow^* \langle |\text{asl}|\text{query}(c_3|b|c_1, c_3); c_0; c_1, \tau_1]; \tau \rangle
$$

The same for case of $|[(b|b; c_1, c_f) with \pi_2(e_b) = \text{false}, and case of while [b|b; c_0; c_1, \tau_1]; \tau \rangle do the conditional command of the assignment commands associated to the $e_b$ from Inversion Lemma C.7 of testing event.
Proof. Taking arbitrary $D \in \mathcal{DB}, c \in \mathcal{C}, \tau \in \mathcal{T}$ and two events $e_1, e_2 \in \mathcal{E}_{\text{ann}}$, where $\tau$ has the form $\tau = [e_1] \ldots \tau' \ldots [e_2]$ for some $\tau' \in \mathcal{T}$ and $\text{DEP}_{\mathcal{E}}(e_1, e_2, \tau, c, D)$

Assume

$$
\neg \text{flowsTo}(\pi_1(e_1)^{\#2(e_1)}, \pi_1(e_2)^{\#2(e_2)}, c) \ (1)
$$

$$
\land \forall e \in \tau'. (\neg \text{DEP}_{\mathcal{E}}(e_1, c, \tau(e_1 : e), c, D) \lor \neg \text{flowsTo}(\pi_1(e_1)^{\#2(e_1)}, \pi_1(e_2)^{\#2(e_2)}, c)) \ (2)
$$

Then, by Lemma C.4 and (2), we know

$$
\text{flowsTo}(\pi_1(e_1)^{\#2(e_1)}, \pi_1(e_2)^{\#2(e_2)}, c)
$$

, which is contradict to (1).

This Lemma is proved. $\blacksquare$

Lemma C.4 (Independent Events Doesn’t Block flowsTo). For every $D \in \mathcal{DB}, c \in \mathcal{C}, \tau \in \mathcal{T}$, one assignment events $e_1 \in \mathcal{E}_{\text{ann}}$, and another event $e_2 \in \mathcal{E}$, if the trace $\tau$ has the form $\tau = [e_1] \ldots \tau' \ldots [e_2]$ with $\tau' \in \mathcal{T}$, and $\text{DEP}_{\mathcal{E}}(e_1, e_2, \tau, c, D)$, then the following two conclusions hold when $e_2$ is an assignment event and a testing event respectively.

- If $e_2 \in \mathcal{E}_{\text{ann}}$, then for every $e \in \tau'$, if it either doesn’t have the Event May-Dependency relation on $e_1$, or $\pi_1(e)^{\#2(e)}$ doesn’t have the flowsTo relation with $\pi_1(e_2)^{\#2(e_2)}$, then the labelled variable $\pi_1(e_1)^{\#2(e_1)}$ directly flows to the other one $\pi_1(e_2)^{\#2(e_2)}$, i.e., $\text{flowsTo}(\pi_1(e_1)^{\#2(e_1)}, \pi_1(e_2)^{\#2(e_2)}, c)$.

- If $e_2 \in \mathcal{E}_{\text{test}}$, then for every $e \in \tau'$, if it either doesn’t have the Event May-Dependency relations on $e_1$, or $\pi_1(e) \notin \text{VAR}(\pi_1(e_2))$, then $\pi_1(e_1) \notin \text{VAR}(\pi_1(e_2))$, and $\pi_2(e_1) = i(\tau)$

Proof. Taking arbitrary $D \in \mathcal{DB}, c \in \mathcal{C}$, and an assignment events $e_1 \in \mathcal{E}_{\text{ann}}$ and another event $e_2 \in \mathcal{E}$. Without loss of generalization, taking arbitrary trace has the form $\tau = [e_1] \ldots \tau' \ldots [e_2]$ for arbitrary $\tau_2 \in \mathcal{T}$, then we know $\exists \tau' \in \mathcal{T}. \tau = [e_1] \ldots \tau' \ldots [e_2]$, let $\tau_2$ be this $\tau'$.

Case: $e_2 \in \mathcal{E}_{\text{ann}}$

By the definition of $\text{DEP}_{\mathcal{E}}(e_1, e_2, \tau, c, D)$, taking $e_1', e_2' \in \mathcal{E}_{\text{ann}}, \tau_2' \in \mathcal{T}, c_1, c_2 \in \mathcal{C}$ as the events, traces and commands satisfying the definition, we have following two executions:

$$
\langle c, \tau_0 \rangle \rightarrow^* \langle c_1, \tau_1 \ldots [e_1] \rangle \rightarrow^* \langle c_2, \tau_1 \ldots [e_1] \ldots \tau_2 \ldots [e_2] \rangle
$$

$$
\langle c_1, \tau_1 \ldots [e_1] \rangle \rightarrow^* \langle c_2, \tau_1 \ldots [e_1'] \ldots \tau_2' \ldots [e_2'] \rangle
$$

By inversion Lemma C.7 on $e_2$ and $e_2'$, in the two executions and Diff($e_2, e_2'$), we have the following two execution instances:

$$
\langle c_1, \tau_1 \ldots [e_1] \rangle \rightarrow^* \langle [\pi_1(e_2) - e_2/\text{query}(\psi_2)]^{\#2(e_2)}, c_2, \tau_1 \ldots [e_1] \ldots \tau_2 \ldots [e_2] \rangle
$$

$$
\langle c_1, \tau_1 \ldots [e_1'] \rangle \rightarrow^* \langle [\pi_1(e_2) - e_2/\text{query}(\psi_2)]^{\#2(e_2)}, c_2, \tau_1 \ldots [e_1'] \ldots \tau_2' \ldots [e_2'] \rangle
$$

53
where \( e_2/\psi_2 \) is the expression of the assignment command associated to the \( e_2 \) and \( e'_2 \) by the Inversion Lemma. 

Taking arbitrary \( e_2 \in \tau_2 \), we know \( \neg \text{DEP}_e(e_1,e,\tau[e_1:e_2],c,D) \lor \pi_1(e_2) \in \text{VAR}(e_2/\psi_2) \).

In case of \( \neg \text{DEP}_e(e_1,e,\tau[e_1:e_2],c,D) \), by Definition \[10\] we know \( e_2 \in \tau'_2 \) and

\[
\rho(\tau_1 \cdots \tau(e_1:e_2))\pi_1(e_2) = \rho(\tau_1 \cdots \tau(e'_1:e_2))\pi_1(e_2)
\]

In case of \( \pi_1(e_2) \in \text{VAR}(e_2/\psi_2) \), by Inversion Lemma \[C.5\] of arithmetic and query expression cases, we know:

\[
\forall \bar{x}^i \in \mathbb{L}V, \tau, \tau' \in \mathbb{T}, v, v' : (\forall z^j \in \mathbb{L}V / [\pi_1(e_2)^{\tau_2}(e_2)]) . \rho(\tau)z = \rho(\tau')z \land (\tau, e_2/\psi_2) \Downarrow a v \land (\tau', e_2) \Downarrow a v' \implies v = v'
\]

\[
\forall \bar{x}^i \in \mathbb{L}V, \tau, \tau' \in \mathbb{T}, \alpha, \alpha' : (\forall z^j \in \mathbb{L}V / [\pi_1(e_2)^{\tau_2}(e_2)]) . \rho(\tau)z = \rho(\tau')z \land (\tau, \psi_2) \Downarrow q \alpha \land (\tau', \psi_2) \Downarrow q \alpha' \implies \alpha = q \alpha'
\]

for \( e_2 \) or \( \psi_2 \) respectively .

Let \( \text{use}_{\tau_2} \) a subset of the events in \( \tau_2 \), satisfying:

\[
\forall e \in \mathcal{E}^{\text{ann}} . \ e \in \text{use}_{\tau_2} \iff e \in \tau_2 \land \pi_1(e) \in \text{VAR}(e_2/\psi_2)
\]

Then we also know for every \( e_2 \in \text{use}_{\tau_2} \), \( \neg \text{DEP}_e(e_1,e_2,\tau[e_1:e_2],c,D) \), i.e.:

\[
\forall z^j \in \mathbb{L}V \setminus \left[ \mathbb{L}V_{\tau_2} \setminus \mathbb{L}V_{\text{use}_{\tau_2}} \cup [\pi_1(e_2)^{\tau_2}(e_2)] \right] . \rho(\tau_1 \cdots \tau(e_1:e_2))z = \rho(\tau_1 \cdots [e'_1] \cdots \tau_2')z \tag{1}
\]

and

\[
\forall z^j \in \mathbb{L}V \setminus \left[ \mathbb{L}V_{\tau_2} \setminus \mathbb{L}V_{\text{use}_{\tau_2}} \right] , \tau, \tau' \in \mathbb{T}, v, v' : \rho(\tau)z = \rho(\tau')z \land (\tau, e_2) \Downarrow a v \land (\tau', e_2) \Downarrow a v' \implies v = v' \tag{2a};
\]

\[
\forall z^j \in \mathbb{L}V \setminus \left[ \mathbb{L}V_{\tau_2} \setminus \mathbb{L}V_{\text{use}_{\tau_2}} \right] , \tau, \tau' \in \mathbb{T}, \alpha, \alpha' : \rho(\tau)z = \rho(\tau')z \land (\tau', \psi_2) \Downarrow q \alpha \land (\tau', \psi_2) \Downarrow q \alpha' \implies \alpha = q \alpha' \tag{2q},
\]

where \( \mathbb{L}V_{\tau_2} \) and \( \mathbb{L}V_{\text{use}_{\tau_2}} \) are the sets of labelled variables of every event in \( \tau_2 \) and \( \text{use}_{\tau_2} \) respectively.

Since \( \text{Diff}(e_2,e'_2) \), we also know:

\[
(\tau_1 \cdots [e_1] \cdots \tau_2, e_2) \Downarrow a \pi_3(e_2) \land (\tau_1 \cdots [e'_1] \cdots \tau_2', e_2) \Downarrow a \pi_3(e'_2) \land \pi_3(e_2) \neq \pi_3(e'_2)
\]

We know \( e_1 \) is the only cause of the difference in \( e_y \) and \( e'_y \) when evaluating \( [\pi_1(e_2) \leftarrow e_2/\text{query}(\psi_2)]^{\pi_2(e_2)} \).

By inversion Lemma \[C.6\] of arithmetic and query expression cases, given the two traces \( \tau_1 \cdots [e'_1] \cdots \tau_2' \) and \( \tau_1 \cdots [e'_1] \cdots \tau_2' \) satisfying this lemma by (1), (2a) and (2q), we know

\[
\pi_1(e_1) \in \text{VAR}(e_2/\psi_2) \land \pi_2(e_1) = i(\tau_1 \cdots [e_1] \cdots \tau_2)\pi_1(e_1)
\]

By \text{flowsTo} definition:

\[
\text{flowsTo}(\pi_1(e_1)^{\pi_2(e_1)}, \pi_1(e_y)^{\pi_2(e'_y)}, c)
\]

This case is proved.

case: \( e_2 \in \mathcal{E}^{\text{test}} \)

By the definition of \( \text{DEP}_e(e_1,e_2,\tau,c,D) \), taking \( e'_1 \in \mathcal{E}^{\text{ann}}, \tau'_2 \in \mathbb{T}, c_1, c_2 \in \mathcal{E} \) and \( e'_2 \in \mathcal{E}^{\text{test}} \) as the events, traces and commands satisfying the definition, we have following two executions:

\[
\begin{align*}
\langle c, \tau_0 \rangle & \rightarrow^* \langle c_1, \tau_1 \cdots [e_1] \rangle & \rightarrow^* \langle c_2, \tau_1 \cdots [e_1] \cdots \tau_2, \cdots [e_2] \rangle \\
\langle c_1, \tau_1 \cdots [e'_1] \rangle & \rightarrow^* \langle c_2, \tau_1 \cdots [e'_1] \cdots \tau_2', \cdots [e'_2] \rangle 
\end{align*}
\]
Taking arbitrary \( e_2 \in \tau_2 \), we know \( \neg \text{DEP}_e(e_1, e, \tau(e_1), c) \) \( \forall c(x) \in \text{VAR}(e_2) \). This case is proved trivially in the same way as the case of the arithmetic expression.

Then by the same proof in case: \( e_2 \in \mathcal{E}^{\text{ann}} \), and applying the Inversion Lemma \( \text{CS.5} \) and \( \text{CS.6} \) of the boolean expression case, we have:

\[
\pi_1(e_1) \in \text{VAR}(\pi_1(e_2)) \land \pi_2(e_1) = i(\tau)
\]

This case is proved.

**Lemma C.5 (Expression Inversion).** For all \( x^1 \in \mathcal{L} \), and \( \tau, \tau' \in \mathcal{T} \), and an expression \( e \) if \( \forall x^1 \in \mathcal{L} \setminus \{x^1\} \cdot \rho(\tau)z = \rho(\tau')z \), and if

- \( e \) is an arithmetic expression \( a \), and \( \langle \tau, a \rangle \not\in \mathcal{L} \) \( a \) \( \mid \) \( a' \) \( x \) \( = \) \( v' \neq v \), then \( x \) is in the free variables of \( a \) and \( i \) is the latest label for \( x \) in \( \tau \), i.e., \( x \in \text{VAR}(a) \) and \( i = i(\tau) \).

- \( e \) is a boolean expression \( b \), and \( \langle \tau, b \rangle \not\in \mathcal{L} \) \( b \) \( \mid \) \( b' \) \( x \) \( = \) \( v' \neq v \), then \( x \) is in the free variables of \( b \) and \( i \) is the latest label for \( x \) in \( \tau \), i.e., \( x \in \text{VAR}(b) \) and \( i = i(\tau) \).

- \( e \) is a query expression \( \psi \), and \( \langle \tau, \psi \rangle \not\in \mathcal{L} \) \( \psi \) \( \mid \) \( \psi' \) \( x \) \( = \) \( v' \neq v \), then \( x \) is in the free variables of \( \psi \) and \( i \) is the latest label for \( x \) in \( \tau \), i.e., \( x \in \text{VAR}(\psi) \) and \( i = i(\tau) \).

Proof Summary:
To show \( x \in \text{VAR}(a) \), by showing contradiction \( \forall \tau, \tau' \) in second hypothesis \( v = v' \) if \( x \not\in \text{VAR}(a) \).
To show \( i = i(\tau) \), by showing contradiction \( \forall \tau, \tau' \) in second hypothesis \( v = v' \).

**Proof.** Take two arbitrary traces \( \tau, \tau' \in \mathcal{T} \), and an expression \( e \) satisfying \( \forall x^1 \in \mathcal{L} \setminus \{x^1\} \cdot \rho(\tau)z = \rho(\tau')z \), we have the following three cases.

**case: \( e \) is an arithmetic expression \( a \)**
We have \( \langle \tau, b \rangle \not\in \mathcal{L} \) \( b \) \( \mid \) \( b' \) \( v \) \( = \) \( v' \neq v \) from the lemma hypothesis.
To show \( x \in \text{VAR}(\psi) \) and \( i = i(\tau) \).
Assuming \( x \in \text{VAR}(a) \), since \( \forall x^1 \in \mathcal{L} \setminus \{x^1\} \cdot \rho(\tau)z = \rho(\tau')z \), we know \( v = v' \), which is contradicted to \( v' \neq v \).

Then we know \( x \in \text{VAR}(\psi) \).
Assuming \( i = i(\tau) \) \( x \) \( \neq j \), by \( \forall x^1 \in \mathcal{L} \setminus \{x^1\} \cdot \rho(\tau)z = \rho(\tau')z \), we know \( \rho(\tau)x = \rho(\tau')x \), i.e., \( \forall x^1 \in \mathcal{L} \setminus \{x^1\} \cdot \rho(\tau)x = \rho(\tau')x \).

Then by the determination of the evaluation, we know \( v = v' \), which is contradicted to \( v' \neq v \).

Then we know \( i = i(\tau) \).

**case: \( e \) is a boolean expression \( b \)**
This case is proved trivially in the same way as the case of the arithmetic expression.

**case: \( e \) is a query expression \( \psi \)**
This case is proved trivially in the same way as the case of the arithmetic expression.

**Lemma C.6 (Expression Inversion Generalization).** For all subset of the labelled variables \( \text{Diff} \subset \mathcal{L} \), and \( x^1 \in (\mathcal{L} \setminus \text{Diff}) \), and an expression \( e \), if

- \( e \) is an arithmetic expression \( a \), and for all \( z^1 \in \mathcal{L} \setminus \text{Diff} \), \( \tau, \tau' \in \mathcal{T} \), \( v, v' \) such that \( \rho(\tau)z = \rho(\tau')z \), and \( \langle \tau, a \rangle \not\in \mathcal{L} \) \( a \) \( \mid \) \( a' \) \( v \) \( = \) \( v' \neq v \), and for all \( z^1 \in \mathcal{L} \setminus (\text{Diff} \cup \{x^1\}) \) there exist.

55
\( \tau, \tau' \in \mathcal{I}, v, v' \) such that \( \rho(\tau)z = \rho(\tau')z \) and \( \langle \tau, a \rangle \downarrow_a v \) and \( \langle \tau', a \rangle \downarrow_a v' \) with \( v \neq v' \), then \( x \in \text{VAR}(a) \) and \( i = i(\tau)x \).

\[
\forall \text{DIFF} \subseteq \mathcal{L}, x' \in (\mathcal{L} \setminus \text{DIFF}), a .
\]

\[
\forall z' \in \mathcal{L} \setminus \text{DIFF}, \tau, \tau' \in \mathcal{I}, v, v' . \rho(\tau)z = \rho(\tau')z \land \langle \tau, b \rangle \downarrow_b v \land \langle \tau', b \rangle \downarrow_b v' \land v = v' . \Rightarrow \forall z' \in \mathcal{L} \setminus (\text{DIFF} \cup \{x'\}) . \exists r, r' \in \mathcal{I}, v, v' . \rho(\tau)z = \rho(\tau')z \land \langle \tau, b \rangle \downarrow_b v \land \langle \tau', b \rangle \downarrow_b v' \land v \neq v' \Rightarrow x \in \text{VAR}(a) \land i = i(\tau)x
\]

**Proof Summary:**

To show \( x \in \text{VAR}(a) \), by showing contradiction \( \forall r, r' \) in second hypothesis \( v = v' \) if \( x \notin \text{VAR}(a) \).

To show \( i = i(\tau) \), by showing contradiction \( \forall r, r' \) in second hypothesis \( v = v' \) if \( j = i(\tau)x \) and \( i \neq j \).

**Proof.** Taking an arbitrary expression \( e \), we have the following three cases.

**case: \( e \) is an arithmetic expression**

Taking an arbitrary set of labelled variables \( \text{DIFF} \subseteq \mathcal{L}, x' \in (\mathcal{L} \setminus \text{DIFF}) \) satisfies:

\[
\forall z' \in \mathcal{L} \setminus \text{DIFF}, \tau, \tau' \in \mathcal{I}, v, v' . \rho(\tau)z = \rho(\tau')z \land \langle \tau, a \rangle \downarrow_a v \land \langle \tau', a \rangle \downarrow_a v' \land v = v' (1)
\]

and \( \forall z' \in \mathcal{L} \setminus \text{DIFF} \cup \{x'\}) . \exists r, r' \in \mathcal{I}, v, v' . \rho(\tau)z = \rho(\tau')z \land \langle \tau, b \rangle \downarrow_b v \land \langle \tau', b \rangle \downarrow_b v' \land v \neq v' (2) \).

Let \( \tau, \tau' \in \mathcal{I}, v, v' \) be the two traces and values satisfies hypothesis (2).

To show: \( x \in \text{VAR}(a) \land i = i(\tau)x \).

Assuming \( x \notin \text{VAR}(a) \), we know from the Inversion Lemma of the arithmetic expression case, \( \forall z' \in \mathcal{L} \setminus \{x'\}) . \tau, \tau' \in \mathcal{I}, v, v' . \rho(\tau)z = \rho(\tau')z \land \langle \tau, a \rangle \downarrow_a v \land \langle \tau', a \rangle \downarrow_a v' \land v = v'. \) Then with the hypothesis (1), we know:

\[
\forall z' \in \mathcal{L} \setminus \{x'\} . \tau, \tau' \in \mathcal{I}, v, v' . \rho(\tau)z = \rho(\tau')z \land \langle \tau, a \rangle \downarrow_a v \land \langle \tau', a \rangle \downarrow_a v' \land v = v'.
\]

This is contradicted to the hypothesis (2).

Then we know \( x \in \text{VAR}(e) \).

Assuming \( j = i(\tau)x \land i \neq j \), by hypothesis (2) where \( \forall z' \in \mathcal{L} \setminus \{x'\} . \rho(\tau)z = \rho(\tau')z \), we know \( \rho(\tau)x = \rho(\tau')x \), i.e.,

\[
\forall z' \in \mathcal{L} \setminus \{\text{DIFF}\} . \rho(\tau)z = \rho(\tau')z.
\]

Then we have \( v' = v \) by hypothesis (1), which is contradicted to \( v' \neq v \).

Then we know \( i = i(\tau)x \).
case: $e$ is a boolean expression $b$
This case is proved trivially in the same way as the case of the arithmetic expression.

case: $e$ is a query expression $\psi$
This case is proved trivially in the same way as the case of the arithmetic expression.

Lemma C.7 (Event Inversion). For all $c \in \mathcal{C}, \tau_0 \in \mathcal{T}, c \in E$ such that $\langle c, \tau_0 \rangle \rightarrow^* \langle \text{skip}, \tau_0 \cdots \tau_1 \rangle$, and $e \in \text{asn}$, if

- $e \in \mathcal{E}^{\text{asn}}$, then either
  - there exists $\tau'_1 \in \mathcal{T}, c' \in \mathcal{C}, \tau$ such that
    $$\langle c, \tau_0 \rangle \rightarrow^* \langle [x \leftarrow e]^l; c', \tau_0 \cdots \tau' \rangle \rightarrow^{\text{asn}} \langle c', \tau_0 \cdots \tau'_1 \cdots [e] \rangle \rightarrow^* \langle \text{skip}, \tau_0 \cdots \tau_1 \rangle$$
  - or there exists $\tau'_1 \in \mathcal{T}, c' \in \mathcal{C}, \psi$ such that
    $$\langle c, \tau_0 \rangle \rightarrow^* \langle [x \leftarrow \text{query}(\psi)]^l; c', \tau_0 \cdots \tau'_1 \rangle \rightarrow^{\text{query}} \langle c', \tau_0 \cdots \tau'_1 \cdots [e] \rangle \rightarrow^* \langle \text{skip}, \tau_0 \cdots \tau_1 \rangle$$

- $e \in \mathcal{E}^{\text{test}}$, then either
  - there exists $\tau'_1 \in \mathcal{T}, c', c_t, c_f, c'' \in \mathcal{C}, b$ such that
    $$\langle c, \tau_0 \rangle \rightarrow^* \langle \text{if} ([b]^l; c_t, c_f); c', \tau_0 \cdots \tau'_1 \rangle \rightarrow_{\text{if}}^* b \langle c'', \tau_0 \cdots \tau'_1 \cdots [e] \rangle \rightarrow^* \langle \text{skip}, \tau_0 \cdots \tau_1 \rangle$$
  - or there exists $\tau'_1 \in \mathcal{T}, c', c_w, c'' \in \mathcal{C}, b$ such that
    $$\langle c, \tau_0 \rangle \rightarrow^* \langle \text{while} ([b]^l; c_w); c', \tau_0 \cdots \tau'_1 \rangle \rightarrow_{\text{while}}^* b \langle c'', \tau_0 \cdots \tau'_1 \cdots [e] \rangle \rightarrow^* \langle \text{skip}, \tau_0 \cdots \tau_1 \rangle$$

Proof Summary: trivially by induction on $c$ and enumerate all operational semantic rules.

Proof. Take arbitrary $\tau_0 \in \mathcal{T}$, by induction on $c$, we have following cases:

case: $c = [x \leftarrow e]^l$
By the evaluation rule $\text{asn}$, we have $\langle [x \leftarrow a]^l, \tau \rangle \rightarrow \langle \text{skip}, \tau \cdots [(x, l, v)] \rangle$.
Then we know $\tau_1 = [(x, l, v)]$ and there is only 1 event $(x, l, v) \in \tau_1$.
Then we have $\tau'_1 = []$ and $c' = \text{skip}$ satisfying
$$\langle c, \tau_0 \rangle \rightarrow^* \langle [x \leftarrow e]^l; c', \tau_0 \cdots \tau' \rangle \rightarrow^{\text{asn}} \langle c', \tau_0 \cdots \tau'_1 \cdots [e] \rangle \rightarrow^* \langle \text{skip}, \tau_0 \cdots \tau_1 \rangle.$$
This case is proved.

case: $c = [x \leftarrow \text{query}(\psi)]^l$
This case is proved trivially in the same way as case: $c = [x \leftarrow e]^l$.

case: $c = c_{s1}; c_{s2}$
This case is proved trivially by the induction hypothesis on $c_{s1}$ and $c_{s2}$ separately, we have this case proved.

case: while $[b]^l$ do $c$
If the rule applied to is while-t, we have:
$$\langle \text{while} ([b]^l; c_w); \tau \rangle \rightarrow \langle \text{while} ([b]^l; c_w, \tau \cdots [(b, l, true)]) \rangle \rightarrow^* \langle \text{skip}, \tau \cdots \tau_1 \rangle,$$
$$(b, l, true) \in \mathcal{E}^{\text{test}} \text{ and } \tau_1 \in \mathcal{T}.$$
Let $\tau' = [], c' = \text{skip}$ and $c'' = c_w, \text{ while } [b]^l \text{ do } c_w$, we know that they satisfy
$$\langle c, \tau_0 \rangle \rightarrow^* \langle \text{while} ([b]^l; c_w); c', \tau_0 \cdots \tau'_1 \rangle \rightarrow^{\text{while}} b \langle c'', \tau_0 \cdots \tau'_1 \cdots [e] \rangle \rightarrow^* \langle \text{skip}, \tau_0 \cdots \tau_1 \rangle$$
This case is proved.
If the rule applied to is while-f, we have
\(\langle \text{while}[b]\rangle \rightarrow \text{while-f}(\langle \text{skip},\tau \rightarrow (b,l,\text{false}))\rangle,\langle b,l,\text{true}\rangle \in \epsilon^\text{test},\) and \(\langle b,l,\text{true}\rangle \in \tau^1.\)

Let \(\tau' = [];\) \(c' = \text{skip}\) and \(c'' = \text{skip},\) we know that they satisfy
\(\langle c,\tau_0 \rangle \rightarrow^* \langle \text{while}(\langle b\rangle,c_0);c',\tau_0;\tau_1' \rangle \rightarrow^* \text{while-f}(\langle c'',\tau_0;\tau_1' \rangle \rightarrow^* (\text{skip},\tau_0;\tau_1)\).

This case is proved.

case: \(\langle [b]\rangle\)

This case is proved in the same way as \textbf{case:} \(c = [x - \text{query}(\psi)]^l.\)

\[\text{Lemma C.8 (Reachable Variabile Inversion). For all } c \in \mathbb{C}_t,\ t' \in \mathcal{T},\ \text{if } \langle c,\tau \rangle \rightarrow^* \langle c',\tau'\rangle,\ \text{and for all } x^l \in \mathbb{L}_c\ \text{such that } i(\tau')x = l,\ \text{then } x^l \in \text{RD}(\text{absinit}(c),c).\]

**Proof.** Take arbitrary \(c \in \mathbb{C}_t,\ t',\ t'' \in \mathcal{T}\) satisfying \(\langle c,\tau \rangle \rightarrow^* \langle c',\tau'\rangle,\) and an arbitrary \(x^l \in \mathbb{L}_c\) satisfying \(i(\tau')x = l.\)

By definition of \(i,\) we know \(\tau'\) has the form \(\tau'_a;[(x,l,v)];\tau'_b\) for some \(\tau'_a,\tau'_b \in \mathcal{T}\) and \(v.\)

And the variable \(x\) doesn’t show up in all the events in \(\tau'_b.\)

Then, by the environment definition, we know: \(p(\tau')x = v,\) i.e., \(x^l\) is reachable at the point of \(\text{absinit}(c).\)

By the \textit{in(l)} operator define in Section \[4.3.2\] we know \(x^l\) is in the \textit{in(absinit(c))} for program \(c.\)

Since \(\text{RD}(\text{absinit}(c),c)\) is a stabilized closure of \textit{in(l)} for \(c,\) we know \(x^l \in \text{RD}(\text{absinit}(c),c).\)

This lemma is proved.

\[\text{Lemma C.9 (While Loop Inversion). For every } \tau,\ t' \in \mathcal{T},\ c, c_1, c_2, c \in \mathbb{C}\ \text{if } \langle c,\tau \rangle \rightarrow^* \langle c_1; c_2, \tau' \rangle\) and \(c_1 \in c c_2.\) then there exist a \textit{while} command in \(c_2\) and \(c_1\) must shows up in the body of that \textit{while} command, i.e., \(\exists l \in \mathbb{N},\ b \in \mathbb{B},\ c_\omega \in \mathbb{C}.\) \(\langle \text{while}[b]\rangle \rightarrow \langle c_1; c_2 \rangle \Rightarrow c_1 \in c c_2.\)

**Proof.** Trivially by induction on \(c\) and enumerate all operational semantic rules.

\textbf{Proof:} Take arbitrary \(c \in \mathcal{T},\) by induction on \(c,\) we have following cases:

case: \(c = [x - e]^l\)

By the evaluation rule \textit{assn}, we have \(\langle [x - a]^l,\tau \rangle \rightarrow (\text{skip},\tau \rightarrow (x,l,v))\).

Since there doesn’t exist \(c_1, c_2 \in \mathbb{C}\) satisfying \(\text{skip} = c_1; c_2,\) this theorem is vacuously true.

case: \(c = [x - \text{query}(\psi)]^l\)

By the evaluation rule \textit{query}, we have \(\langle [x - \text{query}(\psi)]^l,\tau \rangle \rightarrow (\text{skip},\tau \rightarrow (x,l,a,v))\).

Since there doesn’t exist \(c_1, c_2 \in \mathbb{C}\) satisfying \(\text{skip} = c_1; c_2,\) this theorem is vacuously true.

case: \(c = \text{if } \langle [b]\rangle,\ c_1, c_2\)

By the evaluation rule \textit{query} and \textit{if-f}, and the label consistency, we know:

for all possible \(c_{f1}\) and \(c_{f2}\) such that \(c_f\) has the form \(c_f = c_{f1}; c_{f2};\)

all possible \(c_{f1} \) and \(c_{f2}\) such that \(c_{f}\) has the form \(c_f = c_{f1}; c_{f2}.\)

58
c_{f1} \not\in c_{f2}.
Then this theorem is vacuously true.

**case:** c = c_{s1}; c_{s2}
By label consistency, we know for every c'_1 \in_c c_{s1}, c'_1 \not\in c_{s2}.
Then by the induction hypothesis on c_{s1} and c_{s2} separately, we have this case proved.

**case:** while [b]^l do c
By rule while-l, we have:

\[(\text{while } [b]^l \text{ do } c_w, \tau) \rightarrow (c_w; \text{while } [b]^l \text{ do } c_w, \text{skip}), \tau_{\rightarrow [c]}]\]

If c_w is a sequence command, let c_1 = c_{w1} be the any possible command in this sequence, for all possible c_{w1} and c_{w2} such that c_w has the form c_w = c_{w1}; c_{w2}.
Then we have c_2 = c_{w2}; while [b]^l do c_w, \text{skip) and } c_1 \in_c c_2.
And we also have the existence of l = l_b, b and c_w, and while [b]^l do c_w \in_c c_2 and c_1 \in c_{w}.
If c_w isn’t a sequence command, let c_1 = c_{w1}, then we have c_2 = while [b]^l do c_w, \text{skip) and } c_1 \in_c c_2.
And we also have the existence of l = l_b, b and c_w, and while [b]^l do c_w \in_c c_2 and c_1 \in c_{w}.
This case is proved.
By the evaluation rule while-l, we have \( (\text{while } [b]^l \text{ do } c_w, \tau) \rightarrow (\text{while } [b]^l, \tau_{\rightarrow [((b, l, \text{false})]})) \).
Since there doesn’t exist c_1, c_2 \in C satisfying skip = c_1; c_2, this theorem is vacuously true. \(\square\)

**Lemma C.10** (Only skip Command doesn’t Produce Event). For all trace \(\tau \in \mathcal{T}\), and c, c' \in C, \langle c, \tau \rangle \rightarrow \langle c', \tau \rangle \text{ if and only if } c = \text{[skip]; } c'.

**Proof.** Proved trivially by induction on c and enumerate all operational semantic rules. \(\square\)

**Lemma C.11**. (Event Dependency Transitivity) For every D \in \mathcal{D}, c \in C, \tau \in \mathcal{T}, and e_1, e_2, e_3 \in E_{\text{ann}}, \tau_{12}, \tau_{23} \in \mathcal{T}, if DEP_e(e_1, e_2, \tau_{12}, c, D) \text{ and } DEP_e(e_2, e_3, \tau_{23}, c, D), then DEP_e(e_1, e_3, \tau_{12} \cdots \tau_{23}, c, D).

\[\forall D \in \mathcal{D}, \exists C, e_1, e_2, e_3 \in E_{\text{ann}}, \tau_{12}, \tau_{23} \in \mathcal{T}, \text{ DEP}_e(e_1, e_2, \tau_{12}, c, D) \land \text{DEP}_e(e_2, e_3, \tau_{23}, c, D) \Rightarrow \text{DEP}_e(e_1, e_3, \tau_{12} \cdots \tau_{23}, c, D)\]

**Lemma C.12** (Variable May-Dependency Transitivity). For every c \in C, x^i, y^j, z^l \in \mathbb{L}_c , if DEP_var(x^i, y^j, c) \text{ and } DEP_var(y^j, z^l, c), then DEP_var(x^i, z^l, c).

\[\forall c \in C, x^i, y^j, z^l \in \mathbb{L}_c \cdot \text{DEP}_\text{var}(x^i, y^j, c) \land \text{DEP}_\text{var}(y^j, z^l, c) \Rightarrow \text{DEP}_\text{var}(x^i, z^l, c)\]
D Soundness of The Weight Estimation

D.1 Proof of Lemma[4.1]

Lemma (Soundness of the Abstract Events Computation). For every program $c$ and an execution trace $\tau \in \mathcal{T}$ that is generated w.r.t. an initial trace $\tau_0 \in \mathcal{T}_0(c)$, there is an abstract event $\hat{e} = (l,\_\_\_) \in \text{abstrace}(c)$ for every event $e \in \tau$ having the label $l$, i.e., $e = (l,\_\_\_)$.

$$\forall c \in \mathcal{C}, \tau_0 \in \mathcal{T}_0(c), \tau \in \mathcal{T}, e = (l,\_\_\_) \in \mathcal{E} \Rightarrow (c,\tau_0) \rightarrow^* (\text{skip},\tau_0 \cdots \tau) \wedge e \in \tau \Rightarrow \exists \hat{e} = (l,\_\_\_) \in (\mathcal{L} \times \mathcal{D}_0^\top \times \mathcal{L} \times \mathcal{L}) \cdot \hat{e} \in \text{abstrace}(c)$$

Proof. Taking arbitrary $\tau_0 \in \mathcal{T}$, and an arbitrary event $e = (l,\_\_\_) \in \mathcal{E}$, it is sufficient to show:

$$\forall \tau \in \mathcal{T} \cdot (c,\tau_0) \rightarrow^* (\text{skip},\tau_0 \cdots \tau) \wedge e \in \tau \Rightarrow \exists \hat{e} = (l,\_\_\_) \in (\mathcal{L} \times \mathcal{D}_0^\top \times \mathcal{L} \times \mathcal{L}) \cdot \hat{e} \in \text{abstrace}(c)$$

By induction on program $c$, we have the following cases:

case: $c = [x \rightarrow e]^l$

By the evaluation rule assn, we have $(\langle x \rightarrow e \rangle^l, \tau) \rightarrow (\text{skip}, \tau_0 \cdots \langle x, l', v \rangle)$, for some $v \in \mathbb{N}$ and $\tau = \langle [x, l', v] \rangle$.

There are 2 cases, where $l' = l$ and $l' \neq l$.

In case of $l' \neq l$, we know that $e \not\in_\pi \tau$, then this Lemma is vacuously true.

In case of $l' = l$, by the abstract Execution Trace computation, we know $\text{abstrace}(c) = \text{abstrace}'([x := e]^l; [\text{skip}]^l) = ([l, \text{absexpr}(e), l_e])$

Then we have $\hat{e} = (l, \text{absexpr}(e), l_e)$ and $\hat{e} \in \text{abstrace}(c)$.

This case is proved.

case: $c = [x \rightarrow \text{query}(\psi)]^l$

This case is proved in the same way as case: $c = [x \rightarrow e]^l$.

case: while $[b]^l_w$ do $c$

If the rule applied to is while-t, we have

$(\langle \text{while} [b]^l_w \text{ do } c_w, \tau \rangle) \rightarrow (\langle c_w, \text{while} [b]^l_w \text{ do } c_w, \tau_0 \cdots ([b, l, \text{true}]) \rangle) \sim (\langle \text{skip}, \tau_0 \cdots ([b, l, \text{true}]) \cdots \tau_w \cdots \tau \rangle)$

Then we have the following execution:

$(\langle \text{while} [b]^l_w \text{ do } c_w, \tau \rangle) \rightarrow (\langle c_w, \text{while} [b]^l_w \text{ do } c_w, \tau_0 \cdots ([b, l, \text{true}]) \rangle) \sim (\text{while} [b]^l_w \text{ do } c_w, \tau_0 \cdots ([b, l, \text{true}]) \cdots \tau_w \cdots \tau_1 )$ for some $\tau_1 \in \mathcal{T}$ and $\tau = ([b, l, \text{true}]) \cdots \tau_w \cdots \tau_1$.

Then we have 3 cases: (1) $e = (b, l, \text{true})$, (2) $e \in \tau_w$ or (3) $e \in \tau_1$.

In case of (1), $e = (b, l, \text{true})$, since $\text{abstrace}(c) = \text{abstrace}'(c; [\text{skip}]^l) = ([l, \tau, \text{init}(c_w)]) \cup \cdots$, we have $\hat{e} = (l, \tau, \text{init}(c_w))$ and this case is proved.

In case of (2), $e \in \tau_w$, by induction hypothesis on $c_w$ with the execution $(c_w, \tau_0 \cdots ([b, l, \text{true}]) \cdots \tau_w)$ and trace $\tau_w$, we know there is an abstract event of the form $\hat{e}' = (l,\_\_\_) \in \text{abstrace}(c_w)$ where $\text{abstrace}(c_w) = \text{abstrace}'(c_w; [\text{skip}]^l)$.

Let $\hat{e} = (l, \delta, c, l')$ for some $\delta$ and $l'$ such that $\hat{e} \in \text{abstrace}(c)$.

By definition of $\text{abstrace}'$, we have $\text{abstrace}'(c_w; [\text{skip}]^l) = \text{abstrace}'(c_w) \cup \{(l', \delta, c) \mid (l', \delta, c) \in \text{absfinal}(c_w)\}$.

There are 2 subcases: (2.1) $\hat{e}' \in \text{abstrace}'(c_w)$ or (2.2) $\hat{e}' \in (l', \delta, c, l') \in \text{absfinal}(c_w)$.

60
**Theorem D.1**

This case is proved.

**sub-case: (2.1)**

Since \( \text{absG} \), we know the abstract event \( \hat{e} \in \text{absG} \).

This case is proved.

**sub-case: (2.2)** \( \hat{e} \in \{l', dc, l_w\} \) (\( \text{absfinal}(c_w) \))

In this case, we know \((l, dc) \in \text{absfinal}(c_w)\).

Since \( \text{absG} \), we know \((l', dc, l_w) \in \text{absfinal}(c_w)\).

This case is proved.

In case of (3), \( \hat{e} \in \text{absG} \), we know \( \hat{e} \in \text{absG} \).

Then this case is proved by repeating the proof in case (1) and case (2).

If the rule applied to is while-f, we have

\[
\text{while-f} \langle b, l_w, \_\rangle \rightarrow \text{while-f} \langle \text{skip}, l_w, \_\rangle \cap \ldots
\]

In this case, we have \( r = ((b, l_w, \text{false}) \) and \( \hat{e} = (b, l_w, \text{false}) \) (o.w., \( \hat{e} \not\in r \) and this lemma is vacuously true) with \( l = l_w \).

By the abstract trace computation, \( \text{absG} \) has the form \( \{l, \_\} \).

This case is proved.

**case:** if \( (b) \), \( c_i, c_f \)

In this case is proved in the same way as \( c = \text{while} \langle b \rangle \) do \( c \).

**case:** \( c = c_{i1}; c_{i2} \)

By the induction hypothesis on \( c_{i1} \) and \( c_{i2} \) separately, and the same step as case (2). of \( c = \text{while} \langle b \rangle \) do \( c \), we have this case proved.

\[ \square \]

**D.2 Proof of Lemma D.2**

**Lemma** (Uniqueness of the Abstract Events Computation). For every program \( c \) and an execution trace \( r \in T \) that is generated w.r.t. an initial trace \( r_0 \in T_0(c) \), there is a unique abstract event \( \hat{e} = (l, \_\_\_) \in \text{abstrace}(c) \) for every assignment event \( e \in \text{E}_{\text{ann}} \) in the execution trace having the label \( l \), i.e., \( e = (l, \_\_\_) \) and \( e \in r \).

\[
\forall c \in C, r_0 \in T_0(c), r \in T, e = (l, \_\_\_) \in \text{E}_{\text{ann}}, (c, r_0) \rightarrow * (\text{skip}, r_0 + r) \wedge e \in r \\
\implies \exists \hat{e} = (l, \_\_\_) \in (L \times \mathcal{D}L \times L) \cdot \hat{e} \in \text{abstrace}(c)
\]

**Proof.** This is proved trivially by induction on the program \( c \).

\[ \square \]

**D.3 Soundness of Weight Estimation, Theorem 4.1**

Preliminary Theorem from paper [6].

**Theorem D.1** (Soundness of the Transition Bound). For each program \( c \) and an edge \( \hat{e} = (l, \_\_\_) \in \text{absG}(c) \), if \( l \) is the label of an assignment command, then its path-insensitive transition bound \( TB(\hat{e}, c) \) is a sound upper bound on the execution times of this assignment command in \( c \).

\[
\forall c \in C, l \in L \cup (c), r_0 \in T_0(c), r \in T, \nu \in \mathbb{N}, (c, r_0) \rightarrow * (\text{skip}, r_0 + r) \wedge (TB(\hat{e}, c), r_0) \downarrow \nu \wedge \text{cnt}(r, l) \leq \nu
\]
**Theorem D.2** (Soundness of the Weight Estimation). *Given a program c with its program-based dependency graph G_west \( = (V_west, E_west, W_west, Q_west) \), we have:

\[
\forall (x', w) \in W_west, (c', \tau') \in V, \nu \in \mathbb{N} : (c, \tau) \rightarrow \ast \langle \text{skip}, \tau, \tau' \rangle \land \langle \tau, w \rangle \downarrow v \land \text{cnt}(\tau', l) \leq \nu
\]

*Proof.* Taking an arbitrary a program c with its program-based dependency graph G_west = (V, E, W, Q), and an arbitrary pair of labeled variable and weights \( (x', w) \in W_west \) and arbitrary \( \tau, \tau' \in V, \nu \in \mathbb{N} \) satisfying \( (c, \tau) \rightarrow \ast \langle \text{skip}, \tau, \tau' \rangle \land \langle \tau, w \rangle \downarrow v \).

By Definition of \( W_west \) in G_west, we know \( w = \text{abs}\tilde{w}(l) = \text{max}(\text{TB}(\tilde{e})) \in (l, \_). \)

Let \( (\tilde{e}) = (l, dc, l') \in \text{abstrace}(c) \) be this event for some \( dc \) and \( l' \) such that \( (\tilde{e}) = (l, dc, l') \in \text{abstrace}(c) \), by the last step of phase 2, we know \( W_west(x') \triangleq \text{TB}(\tilde{e}) \). Then, it is sufficient to show:

\[
\forall v \in \mathbb{N} : \langle \text{TB}(\tilde{e}), \tau \rangle \downarrow v \land \text{cnt}(\tau', l) \leq v \text{TB}(\tilde{e})
\]

By definition of \( \text{TB}(\tilde{e}) \):

\[
\text{locb}(\tilde{e}) \quad \text{locb}(\tilde{e}) \in \text{SMBCST}
\]

\[
\text{Incr}(\text{locb}(\tilde{e})) + \sum \{ \text{TB}(\tilde{e}') \times \text{max}(\text{Vinvar}(a) + c, 0) \} | (\tilde{e}', a, c) \in \text{re}(\text{locb}(\tilde{e})))
\]

By the soundness of Local bound in Lemma [D.1].

*case: \( \text{locb}(\tilde{e}) \in \text{SMBCST} \)

To show:

\[
\text{max}\{\text{cnt}(\tau')l \mid \forall \tau \in J : (c, \tau) \rightarrow \ast \langle \text{skip}, \tau, \tau' \rangle \}
\]

\[
\leq \text{Incr}(\text{locb}(\tilde{e})) + \sum \{ \text{TB}(\tilde{e}') \times \text{max}(\text{Vinvar}(a) + c, 0) \} | (\tilde{e}', a, c) \in \text{re}(\text{locb}(\tilde{e})))
\]

Taking an arbitrary initial trace \( \tau_0 \in J \), executing \( c \) with \( \tau_0 \), let \( \tau \) be the trace after evaluation, i.e., \( (c, \tau_0) \rightarrow \ast \langle \text{skip}, \tau \rangle \), it is sufficient to show:

\[
\text{cnt}(\tau')l \leq \text{Incr}(\text{locb}(\tilde{e})) + \sum \{ \text{TB}(\tilde{e}') \times \text{max}(\text{Vinvar}(a) + c, 0) \} | (\tilde{e}', a, c) \in \text{re}(\text{locb}(\tilde{e})))
\]

By the soundness of the (1) Transition Bound and (2) Variable Bound Invariant in [6] Theorem 1 (attached above in Theorem [4.1]). This case is proved.

**Lemma D.1** (Soundness of the Local Bound). *Given a program c, we have:

\[
\forall \tilde{e} = (l, dc, l') : \text{max}\{\text{cnt}(\tau')l \mid \forall \tau \in J : (c, \tau) \rightarrow \ast \langle \text{skip}, \tau, \tau' \rangle \} \leq \text{locb}(\tilde{e})
\]

*Proof.*

**sub-case: l \notin \text{SCC(\text{abs}(G(c)))}**

In this case, we know variable \( x^l \) isn’t involved in the body of any while command.

Taking an arbitrary \( \tau_0 \in J \), let \( \tau \in J \) be of resulting trace of executing \( c \) with \( \tau \), i.e., \( (c, \tau_0) \rightarrow \ast \langle \text{skip}, \tau \rangle \), we know the assignment command at line \( l \) associated with the abstract event \( \tilde{e} \) will be executed at most once, i.e., \( \text{cnt}(\tau) \leq 1 \)

By locb definition, we know \( \text{locb}(\tilde{e}) = 1 \).

This case is proved.
**sub-case:** \( l \in \text{SCC}(\text{absG}(c))\land \varepsilon \in \text{dec}(x) \)

in this case, we know \( \text{locb}(\varepsilon) \triangleq x \).

**sub-case:** \( l \in \text{SCC}(\text{absG}(c))\land \varepsilon \notin \bigcup_{x \in \text{VAR}} \text{dec}(x) \land \varepsilon \notin \text{SCC}(\text{absG}(c)/\text{dec}(x)) \)

in this case, we know \( \text{locb}(\varepsilon) \triangleq x \).

In the two cases above, the soundness is discussed in [6] Section 4 of Paragraph *Discussion on Soundness* in Page 25.
E Soundness of Adaptivity Computation Algorithm

**Theorem E.1** (Soundness of AdaptSearch). *For every program* $c$, *given its Program-Based Dependency Graph* $G_{\text{est}}$.

$$\text{AdaptSearch}(G_{\text{est}}) \geq A_{\text{est}}(G_{\text{est}}).$$

**proof Summary:**

1. for every two vertices $x, y$ with a walk $k_{x,y}$ from $x$ to $y$ on $G_{\text{est}}$, 
2. if they are on the same SCC, 
   2.1 Then this walk must also be in this SCC. (By the property that each SCC are single direct connected, otherwise they are the same SCC) 
   2.2 By Lemma [E.1] $\text{l}en^q$ of this walk is bound by the longest walk of this SCC. 
   2.3 The output of $\text{AdaptSearch}(G_{\text{est}})$ is greater than longest walk of a single SCC. 
3. if they are on different SCC. 
   3.1 Then this walk can be split into $n, 2 \leq n$ sub-walks, and each sub-walk belongs to a different SCC. (Also by the property of SCC) 
   3.2 By Lemma [E.1] $\text{l}en^q$ of each sub-walk is bound by the longest walk of the SCC it belongs to. 
   3.3 By line: in algorithm, the output of $\text{AdaptSearch}(G_{\text{est}})$ is greater than sum up the $\text{l}en^q$ of longest walk in every SCC that each sub-walk belongs to. 
4. Then we have $\text{AdaptSearch}(G_{\text{est}}(c)) \geq A_{\text{est}}(c)$.

**Proof.** Taking arbitrary program $c \in C$, let $G_{\text{est}}(c) = (V_{\text{est}}, E_{\text{est}}, W_{\text{est}}, Q_{\text{est}})$ be its program based dependency graph.

Taking an arbitrary walk $k_{x,y} \in W X(G_{\text{est}})$, with vertices sequence $(x, s_1, \cdots, y)$, it is sufficient to show:

$$\text{l}en^q(k_{x,y}) = \text{l}en(s) | s \in (x, s_1, \cdots, y) \land Q(s) = 1 \leq \text{AdaptSearch}(G_{\text{est}}(c))$$

By line:3 of $\text{AdaptSearch}(G_{\text{est}})$ algorithm defined in Algorithm ??, let $G_{\text{SCC}}^1, \cdots, G_{\text{SCC}}^n$ be all the strong connected components on $G_{\text{est}}$ with $0 \leq n \leq |V|$, where each $G_{\text{SCC}}^i = (V_i, E_i, W_i, Q_i)$.

By line:5-6 in Algorithm ??, let $\text{adapt}_{\text{acc}}[G_{\text{SCC}}^i]$ be the result of $\text{AdaptSearch}_{\text{acc}}(G_{\text{SCC}}^i)$ for each $G_{\text{SCC}}^i$.

There are 2 cases:

**case: $x, y$ on the same SCC** 
Let $G_{\text{SCC}}^i$ be this SCC where vertices $x$ and $y$ on, by Lemma [E.1] we know

$$\text{l}en^q(k_{x,y}) = \max\{\text{l}en^q(k) | k \in W X(G_{\text{SCC}}^i)\} \leq \text{AdaptSearch}_{\text{acc}}(G_{\text{SCC}}^i)$$

By line:15 and line:18 in $\text{AdaptSearch}(G_{\text{est}})$ algorithm in Algorithm ??, let $\text{adapt}_{\text{est}}$ be the output value, we know $\text{AdaptSearch}(G_{\text{est}}(c)) = \text{adapt}_{\text{est}} \geq \text{adapt}_{\text{acc}}(\text{SCC})$.

i.e.,

$$\text{l}en^q(k_{x,y}) \leq \text{AdaptSearch}(G_{\text{est}}(c))$$

This case is proved.

**case: $x, y$ on different SCC** 
Let $G_{\text{SCC}}^1, G_{\text{SCC}}^2, \cdots, G_{\text{SCC}}^m, G_{\text{SCC}}^y, 0 \leq m$ be all the SCC this walk pass by, where each vertex in $(x, s_1, \cdots, s_n, y)$ belongs to a single SCC number.

By the property of SCC, we know every 2 SCCs are single direct connected. Then we can divide this walk into $m + 2$ sub-walks:
$k_x = (x, s_1, \cdots, s_{\text{SCC}})$;  
$k_1 = (s_{\text{SCC}}, \cdots, s_{\text{SCC}})$;  
\[ \vdots \]
$k_y = (s_{\text{SCC}}, \cdots, s_y)$;

where $k_x \in \mathcal{WK}(G^{\text{SCC}}_x), \ldots, k_y \in \mathcal{WK}(G^{\text{SCC}}_y)$.

By Lemma [E.1] we know for each walk $k_i$:

$$\text{len}^q(k_i) \leq \max\{\text{len}^q(k_i) | k_i \in \mathcal{WK}(G^{\text{SCC}}_i)\} \leq \text{AdaptSearch}_{\text{sc}}(G^{\text{SCC}}_i) = \text{adapt}_{\text{sc}}[G^{\text{SCC}}_i]$$

Then we have:

$$\text{len}^q(k_{x,y}) = \text{len}^q(k_x) + \cdots + \text{len}^q(k_y) \leq \text{adapt}_{\text{sc}}[G^{\text{SCC}}] + \cdots + \text{adapt}_{\text{sc}}[G^{\text{SCC}}] = \text{adapt}$$

where adapt is the output of AdaptSearch($G_{\text{est}}$). This case is proved. \[\blacksquare\]

**Lemma E.1 (Soundness of AdaptSearch$_{\text{sc}}$).** For every program $c$, given its Program-Based Dependency Graph $G_{\text{est}}$, if $G^{\text{SCC}}$ is a strong connected sub-graph of $G_{\text{est}}$, then $\max\{\text{len}^q(k) | k \in \mathcal{WK}(G^{\text{SCC}})\} \leq \text{AdaptSearch}_{\text{sc}}(G^{\text{SCC}})$.

**Proof Summary:**

1. For each node $x$ on SCC, by property of SCC, for every walk on SCC $k_{x,x} = (x, s_1, \cdots, x)$, with set of unique vertex $\{v_1, \cdots, x\}$ there are $\mathcal{PATH}(p_{x,x})$ on $G^{\text{SCC}}$.
2. For every path $p^i_{x,x} = (x, v_1, \cdots, x) \in \mathcal{PATH}(p_{x,x})$, flowcapacity($p^i_{x,x}$) is the maximum visiting times for every $v \in (x, v_1, \cdots, x)$, visit(s)(s_1, \cdots, x) \leq flowcapacity(p^i_{x,x});
3. querynum($p^i_{x,x}$) * flowcapacity($p^i_{x,x}$) \geq \text{len}(s) = (s_1, \cdots, x) \land Q(s) = 1 = \text{len}^q(k)$,
4. Then, the $\max_{p^i_{x,x} \in \mathcal{PATH}(p_{x,x})} \text{len}^q(k_{x,x}) | k_{x,x} \in \mathcal{WK}(k_{x,x})$.
5. Then, $\max\{\text{querynum}(p^i_{x,x}) \times \text{flowcapacity}(p^i_{x,x}) | x \in G^{\text{SCC}} \land p^i_{x,x} \in \mathcal{PATH}(p_{x,x})\} \geq \text{max}\{\text{len}^q(k^i_{x,x}) | x \in G^{\text{SCC}} \land k^i_{x,x} \in \mathcal{WK}(k_{x,x})\}$
6. We also know by the property of SCC, $\forall x, y \in G^{\text{SCC}}$, let $k_{x,y}$ be arbitrary walk on $G^{\text{SCC}}$, $\text{len}^q(k_{x,y}) \leq \text{max}\{\text{len}^q(k^i_{x,x}) | k^i_{x,x} \in \mathcal{WK}(k_{x,x})\}$.
7. Then, $\max\{\text{len}^q(k^i_{x,x}) | x \in G^{\text{SCC}} \land k^i_{x,x} \in \mathcal{WK}(k_{x,x})\} \geq \text{max}\{\text{len}^q(k^i_{x,x}) | x \in G^{\text{SCC}} \land k^i_{x,x} \in \mathcal{WK}(k_{x,x})\}$ i.e., $\text{max}\{\text{len}^q(k^i_{x,x}) | x \in G^{\text{SCC}} \land k^i_{x,x} \in \mathcal{WK}(k_{x,x})\} \geq \text{len}(k) \times \text{adapt}_{\text{sc}}(G^{\text{SCC}})$.
8. We also know $\text{AdaptSearch}_{\text{sc}}(G^{\text{SCC}}) = \max\{\text{querynum}(p^i_{x,x}) \times \text{flowcapacity}(p^i_{x,x}) | x \in G^{\text{SCC}} \land p^i_{x,x} \in \mathcal{PATH}(p_{x,x})\}$ by the $\text{AdaptSearch}_{\text{sc}}$ algorithm.

Then we have $\text{AdaptSearch}_{\text{sc}}(G^{\text{SCC}}) \geq \text{adapt}_{\text{sc}}(G^{\text{SCC}})$.

**Proof.** Taking arbitrary program $c \in \mathcal{C}$, let $G_{\text{est}}(c) = (V, E, W, Q)$ be its program based dependency graph and $G^{\text{SCC}} = (V_{\text{sc}}, E_{\text{sc}}, W_{\text{sc}}, Q_{\text{sc}})$ be an arbitrary sub SCC graph of $G_{\text{est}}$.

There are 2 cases:

**case:** $G^{\text{SCC}}$ contains no edge and only 1 vertex $v$, i.e., $|E| = 0 \land |V| = 1$

In this case there is no walk in this graph, i.e., $\mathcal{WK}(G^{\text{SCC}}) = \emptyset$.

The adaptivity is $Q(v)$.

This case is proved.
\textbf{case: }$G^{\text{SCC}}$ contains at least 1 edge and at least 1 vertex $v$, i.e., $1 \leq |E| \land 1 \leq |V|$

Taking arbitrary walk $k_{x,y} \in \mathcal{W}(G^{\text{SCC}})$, with vertices sequence $(x, s_1, \ldots, y)$, it is sufficient to show:

$$\text{len}^q(k_{x,y}) = \text{len}(s) s \in (x, s_1, \ldots, y) \land Q(s) = 1 \leq \text{AdaptSearch}_{\text{scce}}(G^{\text{SCC}})$$

By AdaptSearch$_{\text{scce}}(G^{\text{SCC}})$ algorithm line 19, in the iteration where $x$ is the starting vertex, we know AdaptSearch$_{\text{scce}}(G^{\text{SCC}}) = r_{\text{scce}} = \max(r_{\text{scce}}, \text{dfs}(G^{\text{SCC}}, x, v_{\text{visited}}))$, then it is sufficient to show:

$$\text{len}(s) s \in (x, s_1, \ldots, y) \land Q(s) = 1 \leq \text{dfs}(G^{\text{SCC}}, x, v_{\text{visited}}).$$

Let $(v_1, \ldots, x)$ be the set of all the distinct vertices of $k_{x,y}$'s vertices sequence $(x, s_1, \ldots, y)$, and $(v_1, \ldots, x)$ be a subsequence containing all the vertices in $(x, v_1, \ldots, y)$.

By the definition of walk, there is a path $p_{x,y}$ from $x$ to $y$ with this vertices sequence: $(x, v_1, \ldots, y)$.

By lines 13 of the dfs$(G^{\text{SCC}}, x, v_{\text{visited}})$ in Algorithm 2, we know dfs$(G^{\text{SCC}}, x, v_{\text{visited}}) = r[x]$ and $r[x] = \max(\text{flowcapacity}(p) \times \text{querynum}(p) | p \in \mathcal{P}(x,x)(G^{\text{SCC}}))$, where $\mathcal{P}(x,x)(G^{\text{SCC}})$ is a subset of $\mathcal{P}(x,x)(G^{\text{SCC}})$, in which every path starts from $x$ and goes back to $x$.

By the property of strong connected graph, we know in this case $\mathcal{P}(x,x)(G^{\text{SCC}}) \neq \emptyset$ and there are 2 cases, $x = y$ and $x \neq y$.

\textbf{case: }$x = y$

In this case, we know $p_{x,y} \in \mathcal{P}(x,x)(G^{\text{SCC}})$, then it is sufficient to show:

$$\text{len}(s) s \in (x, s_1, \ldots, y) \land Q(s) = 1 \leq \text{flowcapacity}(p_{x,y}) \times \text{querynum}(p_{x,y})$$

By line 7 and line 13 in Algorithm 2, we know $\text{flowcapacity}(p_{x,y})$ is the maximum visiting times for every $v \in (x, v_1, \ldots, y)$.

By line 8 and line 13 in Algorithm 2, we know $\text{querynum}(p_{x,y})$ is the number of vertices with $Q$ equal to 1.

Then we know

$$\text{len}(s) s \in (x, s_1, \ldots, y) \land Q(s) = 1 \leq \text{flowcapacity}(p_{x,y}) \times \text{querynum}(p_{x,y})$$

This case is proved.

\textbf{case: }$x \neq y$

we also have a path start from $y$ and go back to $x$.

Let $p_{y,x}$ be this path with vertices sequence $(y, y', \ldots, x)$, we have a path $p_{x,y}$, which is the path $p_{x,y}$ concatenated by path $p_{y,x}$ with vertices sequence $(x, v_1, \ldots, y', \ldots, v_m, x)$, where $m \geq 0$.

Then in this case, it is sufficient to show:

$$\text{len}(s) s \in (x, s_1, \ldots, y) \land Q(s) = 1 \leq \text{flowcapacity}(p_{x,y}) \times \text{querynum}(p_{x,y})$$

Since $\text{flowcapacity}(p_{x,y} + p_{y,x})$ is the maximum visiting times for every $v \in (x, v_1, \ldots, y', \ldots, x)$.

By line 7 in Algorithm 2, we know $\text{flowcapacity}(p_{x,y})$ is the maximum visiting times for every $v \in (x, v_1, \ldots, y)$.

By line 8 in Algorithm 2, we know $\text{querynum}(p_{x,y})$ is the number of vertices with $Q$ equal to 1.

Then we know

$$\text{len}(s) s \in (x, s_1, \ldots, y) \land Q(s) = 1 \leq \text{flowcapacity}(p_{x,y}) \times \text{querynum}(p_{x,y})$$

By line 13, we also know $r[y] = \max(r[x], r[y']) + \text{flowcapacity}(p_{x,y}) \times \text{querynum}(p_{x,y})$, and $r[y] \leq r[w']$ then we know $r[y] \leq r[x]$, i.e., $\text{len}(s) s \in (x, s_1, \ldots, y) \land Q(s) = 1 \leq r[x]$.

This case is proved.
F Conditional Completeness of Adativity Computation Algorithm

Theorem F.1 (Conditional Completeness of AdaptSearch). For every program $c$, given its Program-Based Dependency Graph $G_{est}$, if $G_{est}(c)$ is acyclic directed, then

$$\text{AdaptSearch}(G_{est}) = A_{est}(G_{est}).$$

The proof summary:
1. For every two vertices $x, y$ with a walk $k_{x,y}$ from $x$ to $y$ on $G_{est}$.
2. Since $G_{est}$ is acyclic directed, then this walk corresponds to a path $p_{x,y}$ where every vertex is visited exactly once.
3. The query length is the sum of the query annotation.

From Algorithm 2, every vertex is a SCC with only one vertex and zero edge, its adaptivity is exactly its query annotation.

$$\sum_{v \in \text{SCC}_i} \text{Adapt}[scc_i]$$

This is proved.

Proof. Taking arbitrary program $c \in C$, let $G_{est}(c) = (V_{est}, E_{est}, W_{est}, Q_{est})$ be its program-based dependency graph.

Let the walk $k_{max} \in \mathcal{W}_K(G_{est}(c))$ be the finite walk with the longest query length, and the vertices sequence $(s_1, \cdots, s_n)$, it is sufficient to show:

$$\text{len}^q(k_{max}) = \text{len}(s|s \in (s_1, \cdots, s_n) \land Q_{est}(s) = 1) = \text{AdaptSearch}(G_{est}(c))$$

In order to show the completeness, it is sufficient to show two following items,
1. By line: 15, AdaptSearch($G_{est}(c)$) can find a path $p_{max}$ such that $\text{adapt}_{p_{max}} = \text{len}^q(k_{max})$
2. This $p_{x,y}$ is the longest weighted path found by AdaptSearch($G_{est}(c)$), and $\text{adapt}_{p_{max}}$ is returned as the final output.

By the property of ACG, we know every $s_i \in (s_1, \cdots, s_n)$ shows up exactly once. Then we know this walk is a path and

$$\text{len}^q(k_{max}) = \sum_{s_i \in (s_1, \cdots, s_n)} Q_{est}(s_i)$$

By line: 13, through searching on all the vertices connected on $G_{est}(c)$ from the starting node $s_i$, we know that AdaptSearch($G_{est}(c)$) finds this path $p_{max} = (s_1, \cdots, s_n)$.

Then, it is sufficient to show

$$\text{adapt}_{p_{max}} = \sum_{s_i \in (s_1, \cdots, s_n)} Q_{est}(s_i).$$

By line: 15, let $G_{SCC}^{1}, \cdots, G_{SCC}^{m}$ be all the SCC, where each vertex in $(s_1, \cdots, s_n)$ belongs to, it is sufficient to show:

$$\sum_{G_{SCC} \in (G_{SCC}^{1}, \cdots, G_{SCC}^{m})} \text{adapt}_{SCC}[G_{SCC}^{1}] = \sum_{s_i \in (s_1, \cdots, s_n)} Q_{est}(s_i).$$

By line 3 in Algorithm 2, let $G_{SCC}^{1} = (V_i, E_i, W_i, Q_i)$ for $G_{SCC}^{1} \in (G_{SCC}^{1}, \cdots, G_{SCC}^{m})$ be the SCC found by the standard Algorithm.

Then, by the property of ACG, we know every $G_{SCC}^{1}$ is a single vertex $\nu_i$ without edge and $Q_i$ is the query annotation of $\nu_i$, i.e., $V_i = \{s_i\}$ and $Q_i = \{(s_i, Q_{est}(s_i))\}$.

So we know $n = m$. 67
Also by Algorithm 2 line: 4-5, we know $\text{adapt}_{\text{sc}}[G_{\text{SCC}_1}] = Q_{\text{est}}(s_i)$. Then we can conclude:

$$\sum_{G_{\text{SCC}} \in \{G_{\text{SCC}_1}, \ldots, G_{\text{SCC}_m}\} \text{adapt}_{\text{sc}}[G_{\text{SCC}}]} = \sum_{G_{\text{SCC}} \in \{G_{\text{SCC}_1}, \ldots, G_{\text{SCC}_m}\}} Q_{\text{est}}(s_i) = \sum_{s_i \in (s_1, \ldots, s_n)} Q_{\text{est}}(s_i).$$

So we have (1). "the existence" proved. In order to show $p_{\max}$ is the longest path found and $\text{adapt}_{p_{\max}}$ is returned by AdaptSearch($G_{\text{est}}(c)$), by line: 18, it is sufficient to show $\text{adapt} = \text{adapt}_{p_{\max}}$.

It is sufficient to show a contradiction if $\text{adapt} \neq \text{adapt}_{p_{\max}}$ in following two cases:

**case:** $\text{adapt} < \text{adapt}_{p_{\max}}$

. It is easy to show the contradiction by line: 18 where $\text{adapt} = \max(\text{adapt}, \text{adapt}_{p_{\max}}) \geq \text{adapt}_{p_{\max}}$.

**case:** $\text{adapt} > \text{adapt}_{p_{\max}}$

. Let $p'_{\max}$ be the path such that $\text{adapt} = \text{adapt}_{p'_{\max}} > \text{adapt}_{p_{\max}}$ with vertices sequence $(s'_1, \ldots, s'_n)$. Then we know $p'_{\max}$ corresponds to a walk $k'_{\max}$ with the same vertices sequence. Then by the same proof above, we know $\text{len}^q(k'_{\max}) = \text{adapt}_{p'_{\max}}$ and $\text{len}^q(k'_{\max}) > \text{len}^q(k_{\max})$. Then there is a contradiction that $k'_{\max}$ is the walk with the longest query length rather than $k_{\max}$.

Then, we have (2) proved. $\square$
### G  The Detail Evaluation Table

Table 2: Experimental results of *AdaptFun* implementation

<table>
<thead>
<tr>
<th>Program c</th>
<th>True Value</th>
<th><em>AdaptFun</em> (1/II)</th>
<th>performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lines</td>
<td>Okaml</td>
<td>Weight</td>
</tr>
<tr>
<td>twoRounds(k)</td>
<td>(k)</td>
<td>(2k)</td>
<td>8</td>
</tr>
<tr>
<td>multiRounds(k)</td>
<td>(k)</td>
<td>(k(k+1))</td>
<td>10</td>
</tr>
<tr>
<td>1RGD(k,r)</td>
<td>(k)</td>
<td>(k(k+1))</td>
<td>10</td>
</tr>
<tr>
<td>mBDdd(k)</td>
<td>(1+k)</td>
<td>(2k)</td>
<td>10</td>
</tr>
<tr>
<td>mRSingle(k)</td>
<td>(1+k)</td>
<td>(1+k)</td>
<td>9</td>
</tr>
<tr>
<td>ifCB()</td>
<td></td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>while(k)</td>
<td>(1+k)</td>
<td>(2^k)</td>
<td>7</td>
</tr>
<tr>
<td>whileM(k)</td>
<td>(1+k)</td>
<td>(2^k)</td>
<td>9</td>
</tr>
<tr>
<td>whileRV(k)</td>
<td>(1+k)</td>
<td>(2^k)</td>
<td>9</td>
</tr>
<tr>
<td>whileVCD(k)</td>
<td>(1+k)</td>
<td>(2^k)</td>
<td>6</td>
</tr>
<tr>
<td>whileRVCD(k)</td>
<td>(1+k)</td>
<td>(2^k)</td>
<td>9</td>
</tr>
<tr>
<td>nestWhileMCR(k)</td>
<td>(1+k)</td>
<td>(2^k)</td>
<td>11</td>
</tr>
<tr>
<td>nestWhileVCD(k)</td>
<td>(1+k)</td>
<td>(2^k)</td>
<td>10</td>
</tr>
<tr>
<td>nestWhileRV(k)</td>
<td>(1+k)</td>
<td>(2^k)</td>
<td>10</td>
</tr>
<tr>
<td>nestWhileRV(k)</td>
<td>(1+k)</td>
<td>(2^k)</td>
<td>10</td>
</tr>
<tr>
<td>nestWhileMCR(k)</td>
<td>(1+k)</td>
<td>(2^k)</td>
<td>10</td>
</tr>
<tr>
<td>mFCOMplete(k)</td>
<td>(k)</td>
<td>(k)</td>
<td>27</td>
</tr>
<tr>
<td>mFCOMpose(k)</td>
<td>(k)</td>
<td>(k)</td>
<td>46</td>
</tr>
<tr>
<td>seqCOMpose(k)</td>
<td>(2k)</td>
<td>(2k)</td>
<td>502</td>
</tr>
<tr>
<td>tRCompose(k)</td>
<td>(2k)</td>
<td>(2k)</td>
<td>42</td>
</tr>
<tr>
<td>jumboS(k)</td>
<td>(2^k)</td>
<td>(2^k)</td>
<td>71</td>
</tr>
<tr>
<td>jumbo(k)</td>
<td>(2^k)</td>
<td>(2^k)</td>
<td>502</td>
</tr>
<tr>
<td>big(k)</td>
<td>(2^k)</td>
<td>(2^k)</td>
<td>214</td>
</tr>
</tbody>
</table>

### H  The Programs and Codes Of The Evaluated Examples in Table 2

#### H.1  The Programs for Examples from line:6 - 15 in Table 2

**Example H.1** (The Complete Gradient Decent Optimization Algorithm). *This example is the gradient decent algorithm example is a generalization of the linear regression on a higher degree data relation.*
It uses gradient decent algorithm to minimize the mean square loss function for a two-degree relation $y = a_1 \times x_1^2 + a_2 \times x_2 + c$ on the dataset of two feature columns and one indicator column.

$$\text{gradientDecent}(\text{step}, \text{rate}, t, n) \triangleq$$

- $[a_1 \leftarrow 0]^9$;
- $[a_2 \leftarrow 0]^4$;
- $[c \leftarrow 0]^2$;
- $[j \leftarrow \text{step}]^3$;

While $[j > 0]^4$ do

- $[da1 \leftarrow \text{query}(-2 \ast (\chi[2] - (\chi[0]^2 \times a_1 + \chi[1] \times a_2 + c)) \times (\chi[0]))]^5$;
- $[da2 \leftarrow \text{query}(-2 \ast (\chi[2] - (\chi[0]^2 \times a_1 + \chi[1] \times a_2 + c)) \times (\chi[1]))]^6$;
- $[dc \leftarrow \text{query}(-2 \ast (\chi[2] - (\chi[0]^2 \times a_1 + \chi[1] \times a_2 + c)))]^7$;
- $[a_1 \leftarrow a_1 - \text{rate} \ast da1]^7$;
- $[a_2 \leftarrow a_2 - \text{rate} \ast da2]^8$;
- $[c \leftarrow c - \text{rate} \ast dc]^9$;
- $[j \leftarrow j - 1]^{10}$;

This approach can be generalized to the regression of a variety of relations in machine learning area.

**Example H.2** (Sequence with Linear Query Dependency). This example algorithm contains only sequence of four query commands. Each of them depends on a previous query. The longest dependency depth, i.e., the adaptivity is expectation to be 4.

$$\text{seq()} \triangleq \begin{cases} x \leftarrow \chi[0]^0; y \leftarrow \chi[x+1]^1; z \leftarrow \chi[y+1]^2; w \leftarrow \chi[z+1]^3 \\ \end{cases}$$

Evaluation Result: $A_{\text{est}}(\text{seq}) = 4$

**Example H.3** (Sequence with Query Dependency between Related Variables). This example algorithm contains a sequence of four query commands. Each of them depends on one or more of the previous queries. The longest dependency depth, i.e., the adaptivity is expectation to be 4.

$$\text{seqRV()} \triangleq \begin{cases} x \leftarrow \chi[0]^0; y \leftarrow \chi[x+1]^1; z \leftarrow \chi[y+x]^2; w \leftarrow \chi[z+1] \cdot \chi[y]^3 \\ \end{cases}$$

Evaluation Result: $A_{\text{est}}(\text{seqMultiVar}) = 4$

**Example H.4** (If with Data-Value Dependency Separated). This example algorithm contains a if command and a query requests in each branch. Only the query in the first branch depend on the query in the command 0, and the variable in the guard is not assigned by a query request. The longest dependency depth, i.e., the adaptivity is expectation to be 3.

$$\text{ifVD}(k) \triangleq \begin{cases} z \leftarrow \text{query}(\chi[0])^0; x \leftarrow k/2^1; \\ \text{if } ([x < 0]^2, y \leftarrow \text{query}(\chi[z])^3, y \leftarrow \text{query}(\chi[0])^4) \\ \end{cases}$$

Evaluation Result: $A_{\text{est}}(\text{ifVD}) = 3$

**Example H.5** (If with Data-Control Dependency Overlapped). This example algorithm contains a if command and a query requests in each branch. The variable in the guard is assigned by a query
request in command 1. The two queries in the branches depend on the second query in command 1 but not depend on the query in the command 0. Even though the variable x isn’t used in the query expression in the query 3 and 4, there are still dependency relation because x is in the guard. The longest dependency depth, i.e., the adaptivity is expectation to be 3.

\[
\text{ifCD}(k) \triangleq [z \leftarrow \text{query}(\chi(0))]^0; [x \leftarrow \text{query}(\chi(z))]^1; \\
\quad \text{if}([x < 0]^2, [y \leftarrow \text{query}(\chi(0) + \chi(1))]^3, [y \leftarrow \text{query}(\chi(0))]^4).
\]

**Evaluation Result:** \( A_{\text{est}}(\text{ifCD}(k)) = 3 \)

**Example H.6** (While with Nested Query Dependency). This example algorithm contains a simple while loop. There is one query requests in the loop body at command 3. In each iteration, the query request depend on the query result from previous iteration. The longest dependency depth, i.e., the adaptivity is expectation to be \( k \).

\[
\text{whileNested}(k) \triangleq \quad \text{while} \quad [j > 0]^2 \quad \text{do} \\
\quad \left([x \leftarrow \text{query}(\chi(a))]^3; [a \leftarrow x + a]^4; [j \leftarrow j - 1]^5\right)
\]

**The Evaluation Result:** \( A_{\text{est}}(\text{whileRec}(k)) = 1 + k \)

**Example H.7** (While with Multi-Path Query Dependency). This example algorithm contains a simple while loop and a if command in the loop body. Each branch has a query request (in the commands 5 and 6) depend on the query at command 1 and the query at command 7. Among the \( \frac{k}{2} \) iterations, result from previous iteration. The longest dependency depth, i.e., the adaptivity is expectation to be \( 1 + 2 \cdot \lfloor \frac{k}{2} \rfloor \).

\[
\text{whileM}(k) \triangleq \left([j \leftarrow j - 1]^3; \\
\quad \text{if}([j \%2 == 0]^4, [y \leftarrow \chi(x)]^5, [w \leftarrow \chi(x)]^6); \\
\quad [x \leftarrow \text{query}(\chi(\ln(y))))]^7\right)
\]

**The Evaluation Result:** \( A_{\text{est}}(\text{whileM}(k)) = 1 + 2 \cdot \lfloor \frac{k}{2} \rfloor \)

**Example H.8** (While with Query Dependency through Related Variables). This example algorithm contains a simple while loop and a sequence of three query requests in the loop body. In each iteration, every query request depend on one or more query results from previous iteration. The longest dependencydepth, i.e., the adaptivity is expectation to be \( 1 + 2 \cdot k \).

\[
\text{whileRV}(k) \triangleq \quad \text{while} \quad [j > 0]^3 \quad \text{do} \\
\quad \left([j \leftarrow j - 1]^3; [z \leftarrow \text{query}(\chi(x + \ln(y)))]^5; [x \leftarrow \text{query}(\chi(z))]^6; [y \leftarrow \text{query}(\chi(z))]^7\right)
\]

**The Evaluation Result:** \( A_{\text{est}}(\text{whileRV}(k)) = 1 + 2 \cdot k \)

**Example H.9** (While with Query Dependency through Control Flow and Data Flow). This example algorithm contains a simple while loop and a sequence of three query requests in the loop body. The variable in the guard is assigned by a query request in command 0. In each iteration, the query at
3 depends on either the query at line 1, and the query result at line 4 from the previous iteration. In each iteration, the query at 4 depends on either the query at line 0 and the query at line 3 in the same iteration. The longest dependency depth, i.e., the adaptivity is expectation to be $1 + 2 \times k$.

$$\text{whileVCD}(k) \triangleq \text{while} \left[ x \leftarrow \text{query}(\chi(0)) \right] ; \left[ z \leftarrow \text{query}(\chi(0)) \right] ;$$

$$\text{whileMPVCD}(k) \triangleq \text{while} \left[ x > 0 \right] \left[ x \leftarrow \text{query}(\chi(z)) \right] ; \left[ z \leftarrow \text{query}(\chi(x)) \right]$$

The Evaluation Result: $A_{est}(\text{whileVCD}(k)) = 1 + 2 \times k$

Example H.10 (While with Multiple Path Query Dependency Dependency). This example algorithm contains a simple while loop and a if command in the loop body. Each branch has a query request (in the commands 5 and 6) depend on either the query at command 1 or the query at command 7. The longest dependency depth, i.e., the adaptivity is expectation to be $2 + k$.

$$\text{while} \left[ x \leftarrow \text{query}(k) \right] \left[ y \leftarrow 0 \right] ; \text{while} \left[ x > 0 \right] \left[ y \leftarrow \text{query}(\chi(12)) \right] ; \left[ w \leftarrow \text{query}(\chi(9)) \right] ;$$

$$\text{whileMPVCD}(k) \triangleq \text{while} \left[ x > 0 \right] \left[ x \leftarrow x - 1 \right] \left[ y \leftarrow \text{query}(\chi(\ln(y))) \right]$$

The Evaluation Result: $A_{est}(\text{whileMPVCD}(k)) = 2 + k$

Example H.11 (Nested While with Nested Query Dependency). This example algorithm contains two nested while loops. The query in the outer loop at line 5 depends on either the query at line 1 or the query results at line 8 from the previous iteration of the inner loop. The longest dependency depth, i.e., the adaptivity is expectation to be $2 + k^2$.

$$\text{nestWhileVD}(k) \triangleq \text{while} \left[ i \leftarrow k \right] \left[ x \leftarrow \text{query}(\chi(0)) \right] ;$$

$$\text{nestWhileVCD}(k) \triangleq \text{while} \left[ i > 0 \right] \left[ i \leftarrow i - 1 \right] \left[ j \leftarrow k \right] \left[ y \leftarrow \text{query}(\chi(\ln(x))) \right] ;$$

The Evaluation Result: $A_{est}(\text{nestWhileVD}(k)) = 2 + k^2$

Example H.12 (Nested While with Query Dependency through Related Variables). This example algorithm contains two nested while loops, one query in the outer loop, and one query in the inner loop. The query in the outer loop at line 8 depends on only the query result at line 7 from the last iteration of the inner loop. However, the query at line 7 depends on either the query at line 1 the query results at line 8 from the previous iteration. The longest dependency depth, i.e., the adaptivity is expectation to be $1 + 2 \times k$.

$$\text{nestWhileRV}(k) \triangleq \text{while} \left[ i \leftarrow k \right] \left[ x \leftarrow \text{query}(\chi(0)) \right] ;$$

$$\text{nestWhileRV}(k) \triangleq \text{while} \left[ i > 0 \right] \left[ i \leftarrow i - 1 \right] \left[ j \leftarrow k \right] ;$$

The Evaluation Result: $A_{est}(\text{nestWhileRV}(k)) = 1 + 2 \times k$
Example H.13 (Nested While with Nest Query Dependency and Related Variable Accross Outer and Inner Loop). This example algorithm contains two nested while loops, one query in the outer loop, and one query in the inner loop as well. The two queries depend on both the query results assigned to themselves in previous iteration. The longest dependency depth, i.e., the adaptivity is expectation to be \(1 + k + k^2\).

\[
\text{nestWhileMR}(k) \triangleq \begin{cases} (i \leftarrow k)^0; & [x \leftarrow \text{query}(\chi(0))]^1; [y \leftarrow \text{query}(\chi(1))]^2; \text{while } [i > 0]^3 \text{ do} \\ (i \leftarrow i - 1)^4; & [j \leftarrow k]^5; [y \leftarrow \text{query}(\chi(\ln(x) + y))]^6; \text{while } [j > 0]^7 \text{ do} \left( (j \leftarrow j - 1)^8; [x \leftarrow \text{query}(\chi(\ln(y)) + \chi[x])]^9 \right) \end{cases}
\]

The Evaluation Result: \(A_{\text{est}}(\text{nestWhileMR}(k)) = 1 + k + k^2\)

Reachability Bound The Evaluation Result:

- weight for Variable: \(j\) of label 6 is: \(0 + 0 + 1 * k * k\)
- weight for Variable: \(y\) of label 7 is: \(0 + 0 + 1 * k * k\)
- weight for Variable: \(j\) of label 4 is: \(0 + 1 * k\)
- weight for Variable: \(i\) of label 3 is: \(0 + 1 * k\)
- weight for Variable: \(x\) of label 8 is: \(0 + 1 * k\)
- weight for Variable: \(x\) of label 1 is: \(1\)
- weight for Variable: \(i\) of label 0 is: \(1\)

Example H.14 (Nested While with MultiplePath and Nested Recursive Multiple Variable Data-Value Dependency Across Outer and Inner Loop). We then show a more complex example with nested while command and nested data-flow across the outer and inner while loop through multiple variables. This example also contains the if command with data dependency occurred through the if guard. The longest dependency depth, i.e., the adaptivity is expectation to be \(1 + k + k^2\).

\[
\text{nestWhileMPRV}(k) \triangleq \begin{cases} (i \leftarrow k)^0; & [x \leftarrow \text{query}(\chi(0))]^1; [y \leftarrow \text{query}(\chi(1))]^2; \text{while } [i > 0]^3 \text{ do} \\ (i \leftarrow i - 1)^4; & [j \leftarrow k]^5; [y \leftarrow \text{query}(\chi(\ln(x) + y))]^6; \text{if } ([x > 0]^6, [y \leftarrow \text{query}(\chi(\ln(x) + y))]^7, [y \leftarrow \text{query}(\chi(x))]^8); \text{while } [j > 0]^9 \text{ do} \left( (j \leftarrow j - 1)^10; [x \leftarrow \text{query}(\chi(\ln(y)) + \chi[x])]^11 \right) \end{cases}
\]

The Evaluation Result: \(A_{\text{est}}(\text{nestWhileMPRV}(k)) = 1 + k + k^2\)

Reachability Bound The Evaluation Result:

- weight for Variable: \(j\) of label 10 is: \(0 + 0 + 1 * k * k\)
- weight for Variable: \(x\) of label 11 is: \(0 + 0 + 1 * k * k\)
- weight for Variable: \(y\) of label 7 is: \(0 + 1 * k\)
- weight for Variable: \(y\) of label 8 is: \(0 + 1 * k\)
- weight for Variable: \(j\) of label 5 is: \(0 + 1 * k\)
- weight for Variable: \(i\) of label 4 is: \(0 + 1 * k\)
- weight for Variable: \(y\) of label 2 is: \(1\)
- weight for Variable: \(x\) of label 1 is: \(1\)
- weight for Variable: \(i\) of label 0 is: \(1\)

H.2 The Programs for Examples from line:16 - 20 in Table 2

Example H.15 (mRCompose). The composed multiple rounds program:
Example H.16 (tRCompose). The composed two rounds program:

```
[ j <- N ] 0 ;
[ l <- 0 ] 1 ;
[ cs <- -1 ] 2 ;
[ ns <- -1 ] 3 ;
while [ < (0, j) ] 4 do {
    [ j <- - ( j, 1 ) ] 5;
    [ cs <- + ( cs, 0 ) ] 6;
    [ ns <- + ( ns, 0 ) ] 7 ;
}[w <- k] 8;
while [< (0, w)] 9 do {
    [ w <- - ( w, 1 ) ] 10;
    [ p <- c ] 11;
    [ q <- c ] 12;
    [ a <- query ( l ) ] 13 ;
    [ i <- N ] 14;
    while [ < (0, i) ] 15 do {
        [ i <- - (i, 1) ] 16;
        [ csi <- + (csi, * (- (a, p), - (q, p))) ] 17;
        if [ > (i , l) ] 18
            then { [ nsi <- + (nsi, * (- (a, p), - (q, p)))) ] 19 }
        else { [ nsi <- nsi ] 20 };
    [ i2 <- N ] 21;
    while [ < (0, i2) ] 22 do {
        [ i2 <- - (i2, 1) ] 23;
        [ p <- c ] 24;
        [ q <- c ] 25;
        [ a <- query ( l ) ] 26;
        [ i <- N ] 27;
    while [ < (0, i) ] 28 do {
        [ i <- - (i, 1) ] 29;
        [ csi <- + (csi, * (- (a, p), - (q, p))) ] 30;
        if [ > (i , l) ] 31
            then { [ nsi <- + (nsi, * (- (a, p), - (q, p)))) ] 32 }
        else { [ nsi <- nsi ] 33 };
    [ i2 <- N ] 34;
    while [ < (0, i2) ] 35 do {
        [ i2 <- - (i2, 1) ] 36;
        [ p <- c ] 37;
        [ q <- c ] 38;
        [ a <- query ( l ) ] 39;
        [ i <- N ] 40;
    while [ < (0, i) ] 41 do {
        [ i <- - (i, 1) ] 42;
        [ csi <- + (csi, * (- (a, p), - (q, p))) ] 43;
        if [ > (i , l) ] 44
            then { [ nsi <- + (nsi, * (- (a, p), - (q, p)))) ] 45 }
        else { [ nsi <- nsi ] 46 };
    [ i2 <- N ] 47;
    while [ < (0, i2) ] 48 do {
        [ i2 <- - (i2, 1) ] 49;
        [ p <- c ] 50;
        [ q <- c ] 51;
        [ a <- query ( l ) ] 52;
        [ i <- N ] 53;
    while [ < (0, i) ] 54 do {
        [ i <- - (i, 1) ] 55;
        [ csi <- + (csi, * (- (a, p), - (q, p))) ] 56;
        if [ > (i , l) ] 57
            then { [ nsi <- + (nsi, * (- (a, p), - (q, p)))) ] 58 }
        else { [ nsi <- nsi ] 59 };
    [ i2 <- N ] 60;
    while [ < (0, i2) ] 61 do {
        [ i2 <- - (i2, 1) ] 62;
        [ p <- c ] 63;
        [ q <- c ] 64;
        [ a <- query ( l ) ] 65;
        [ i <- N ] 66;
    while [ < (0, i) ] 67 do {
        [ i <- - (i, 1) ] 68;
        [ csi <- + (csi, * (- (a, p), - (q, p))) ] 69;
        if [ > (i , l) ] 70
            then { [ nsi <- + (nsi, * (- (a, p), - (q, p)))) ] 71 }
        else { [ nsi <- nsi ] 72 };
    [ i2 <- N ] 73;
    while [ < (0, i2) ] 74 do {
        [ i2 <- - (i2, 1) ] 75;
        [ p <- c ] 76;
        [ q <- c ] 77;
        [ a <- query ( l ) ] 78;
        [ i <- N ] 79;
    while [ < (0, i) ] 80 do {
        [ i <- - (i, 1) ] 81;
        [ csi <- + (csi, * (- (a, p), - (q, p))) ] 82;
        if [ > (i , l) ] 83
            then { [ nsi <- + (nsi, * (- (a, p), - (q, p)))) ] 84 }
        else { [ nsi <- nsi ] 85 }
    ]
```

```
Example H.17 (seqCompose). The composed two rounds program:

```plaintext
[ x <- query ( 0 ) ] 0 ;
[ y <- query ( x ) ] 1 ;
[ z <- query ( y ) ] 2 ;
[ a <- + ( x, 0 ) ] 3 ;
[ b <- + ( a, z ) ] 4 ;
[ c <- + ( a, b ) ] 5 ;
[ w <- query ( a ) ] 6 ;
```
\[ \begin{align*}
\text{x} & \leftarrow \text{query(x)} \quad \text{8} \; \\
\text{y} & \leftarrow \text{query(y)} \quad \text{9} \\
\text{z} & \leftarrow \text{query(z)} \quad \text{10} \\
\text{d} & \leftarrow \text{query(d)} \quad \text{11} \\
\text{e} & \leftarrow \text{query(e)} \quad \text{12} \\
\text{f} & \leftarrow \text{query(f)} \quad \text{13} \\
\text{g} & \leftarrow \text{query(g)} \quad \text{14} \\
\text{h} & \leftarrow \text{query(h)} \quad \text{15} \\
\text{i} & \leftarrow \text{query(i)} \quad \text{16} \\
\text{j} & \leftarrow \text{query(j)} \quad \text{17} \\
\text{k} & \leftarrow \text{query(k)} \quad \text{18} \\
\text{l} & \leftarrow \text{query(l)} \quad \text{19} \\
\text{m} & \leftarrow \text{query(m)} \quad \text{20} \\
\text{n} & \leftarrow \text{query(n)} \quad \text{21} \\
\text{q} & \leftarrow \text{query(q)} \quad \text{22} \\
\text{r} & \leftarrow \text{query(r)} \quad \text{23} \\
\text{s} & \leftarrow \text{query(s)} \quad \text{24} \\
\text{t} & \leftarrow \text{query(t)} \quad \text{25} \\
\text{u} & \leftarrow \text{query(u)} \quad \text{26} \\
\text{v} & \leftarrow \text{query(v)} \quad \text{27} \\
\text{w} & \leftarrow \text{query(w)} \quad \text{28} \\
\text{x} & \leftarrow \text{query(x)} \quad \text{29} \\
\text{y} & \leftarrow \text{query(y)} \quad \text{30} \\
\text{z} & \leftarrow \text{query(z)} \quad \text{31} \\
\text{a} & \leftarrow \text{query(a)} \quad \text{32} \\
\text{b} & \leftarrow \text{query(b)} \quad \text{33} \\
\text{c} & \leftarrow \text{query(c)} \quad \text{34} \\
\text{d} & \leftarrow \text{query(d)} \quad \text{35} \\
\text{e} & \leftarrow \text{query(e)} \quad \text{36} \\
\text{f} & \leftarrow \text{query(f)} \quad \text{37} \\
\text{g} & \leftarrow \text{query(g)} \quad \text{38} \\
\text{h} & \leftarrow \text{query(h)} \quad \text{39} \\
\text{i} & \leftarrow \text{query(i)} \quad \text{40} \\
\text{j} & \leftarrow \text{query(j)} \quad \text{41} \\
\text{k} & \leftarrow \text{query(k)} \quad \text{42} \\
\text{l} & \leftarrow \text{query(l)} \quad \text{43} \\
\text{m} & \leftarrow \text{query(m)} \quad \text{44} \\
\text{n} & \leftarrow \text{query(n)} \quad \text{45} \\
\text{o} & \leftarrow \text{query(o)} \quad \text{46} \\
\text{p} & \leftarrow \text{query(p)} \quad \text{47} \\
\text{q} & \leftarrow \text{query(q)} \quad \text{48} \\
\text{r} & \leftarrow \text{query(r)} \quad \text{49} \\
\text{s} & \leftarrow \text{query(s)} \quad \text{50} \\
\text{t} & \leftarrow \text{query(t)} \quad \text{51} \\
\text{u} & \leftarrow \text{query(u)} \quad \text{52} \\
\text{v} & \leftarrow \text{query(v)} \quad \text{53} \\
\text{w} & \leftarrow \text{query(w)} \quad \text{54} \\
\text{x} & \leftarrow \text{query(x)} \quad \text{55} \\
\text{y} & \leftarrow \text{query(y)} \quad \text{56} \\
\text{z} & \leftarrow \text{query(z)} \quad \text{57} \\
\text{a} & \leftarrow \text{query(a)} \quad \text{58} \\
\text{b} & \leftarrow \text{query(b)} \quad \text{59} \\
\text{c} & \leftarrow \text{query(c)} \quad \text{60} \\
\text{d} & \leftarrow \text{query(d)} \quad \text{61} \\
\text{e} & \leftarrow \text{query(e)} \quad \text{62} \\
\text{f} & \leftarrow \text{query(f)} \quad \text{63} \\
\text{g} & \leftarrow \text{query(g)} \quad \text{64} \\
\text{h} & \leftarrow \text{query(h)} \quad \text{65} \\
\text{i} & \leftarrow \text{query(i)} \quad \text{66} \\
\text{j} & \leftarrow \text{query(j)} \quad \text{67} \\
\text{k} & \leftarrow \text{query(k)} \quad \text{68} \\
\text{l} & \leftarrow \text{query(l)} \quad \text{69} \\
\text{m} & \leftarrow \text{query(m)} \quad \text{70} \\
\text{n} & \leftarrow \text{query(n)} \quad \text{71} \\
\text{o} & \leftarrow \text{query(o)} \quad \text{72} \\
\text{p} & \leftarrow \text{query(p)} \quad \text{73} \\
\text{q} & \leftarrow \text{query(q)} \quad \text{74} \\
\text{r} & \leftarrow \text{query(r)} \quad \text{75} \\
\text{s} & \leftarrow \text{query(s)} \quad \text{76} \\
\end{align*} \]
[ cs <- -1 ] 64 ;
[ ns <- -1 ] 65 ;
[ y <- query ( chi : x :) ] 66 ;
[ j <- ( j, 1 ) ] 67 ;
[ cs <- + { cs, 0 } ] 68 ;
[ ns <- + { ns, 0 } ] 69 ;
[ w <- k ] 70 ;
[ w <- - ( w, 1 ) ] 71 ;
[ w <- - ( w, 1 ) ] 72 ;
[ p <- c ] 73 ;
[ q <- c ] 74 ;
[ a <- query ( l ) ] 75 ;
[ i <- N ] 76 ;
[ w <- - ( w, 1 ) ] 77 ;
[ i <- - ( i, 1 ) ] 78 ;
[ csi <- + ( csi, * ( a, p ), - ( q, p ) ) ] 79 ;
[ if [ > ( i, I ) ] 80
  then { [ nsi <- + ( nsi, * ( a, p ), - ( q, p ) ) ] 81 }
  else { [ nsi <- nsi ] 82 } ;
[ i2 <- N ] 83 ;
[ y <- query ( chi : x :) ] 84 ;
[ i2 <- - ( i2, 1 ) ] 85 ;
[ if [ > ( nsi, I ) ] 86
  then { [ l <- + ( l, i2 ) ] 87 }
  else { [ l <- l ] 88 } ;
[ x <- query ( cs ) ] 89 ;
[ y <- query ( x ) ] 90 ;
[ z <- query ( ns ) ] 91 ;
[ w <- query ( z ) ] 92 ;
[ a <- x ] 93 ;
[ c <- z ] 94 ;
[ j <- k ] 95 ;
[ y <- query ( chi : x :) ] 96 ;
[ da <- query ( * ( a, c ) ) ] 97 ;
[ dc <- query ( * ( a, c ) ) ] 98 ;
[ a <- - ( a, da ) ] 99 ;
[ c <- - ( c, dc ) ] 100 ;
[ j <- - ( j, 1 ) ] 101 ;
[ x <- query ( 0 ) ] 102 ;
[ y <- query ( cs ) ] 103 ;
[ z <- query ( c ) ] 104 ;
[ w <- query ( z ) ] 105 ;
[ x <- query ( 0 ) ] 106 ;
[ y <- query ( x ) ] 107 ;
[ z <- query ( y ) ] 108 ;
[ w <- query ( z ) ] 109 ;
[ x <- query ( 0 ) ] 110 ;
[ y <- query ( x ) ] 111 ;
[ z <- query ( * ( x, y ) ) ] 112 ;
[ w <- query ( * ( chi : y :, chi : z :) ) ] 113 ;
[ z <- query ( w ) ] 114 ;
[ x <- query ( e ) ] 115 ;
[ if [ > ( x, 0 ) ] 116
  then { [ y <- query ( 0 ) ] 117 }
  else { [ w <- query ( 0 ) ] 118 } ;
[ a <- x ] 119 ;
[ c <- z ] 120 ;
\[ \text{j} \leftarrow \text{k} \] 121 ;
\[ \text{cs} \leftarrow + \left( \text{cs}, 0 \right) \] 122 ;
\[ \text{ns} \leftarrow + \left( \text{ns}, 0 \right) \] 123 ;
\[ \text{i} \leftarrow \text{k} \] 124 ;
\[ x \leftarrow \text{query} \left( 0 \right) \] 125 ;
\[ y \leftarrow \text{query} \left( \text{cs} \right) \] 126 ;
\[ w \leftarrow - \left( w, 1 \right) \] 127 ;
\[ i \leftarrow - \left( i, 1 \right) \] 128 ;
\[ j \leftarrow k \] 129 ;
\[ \text{if} \left[ > \left( x , 0 \right) \right] 130 \]
\[ \text{then} \left[ y \leftarrow \text{query} \left( + \left( \text{chi} : x : , \text{chi} : y : \right) \right) \right] 131 \]
\[ \text{else} \left[ y \leftarrow \text{query} \left( \text{chi} : x : \right) \right] 132 ; \]
\[ y \leftarrow \text{query} \left( \text{chi} : x : \right) \] 133 ;
\[ j \leftarrow - \left( j, 1 \right) \] 134 ;
\[ x \leftarrow \text{query} \left( + \left( x, y \right) \right) \] 135 ;
\[ x \leftarrow \text{query} \left( z \right) \] 136 ;
\[ y \leftarrow \text{query} \left( \text{cs} \right) \] 137 ;
\[ z \leftarrow \text{query} \left( \text{c} \right) \] 138 ;
\[ w \leftarrow \text{query} \left( z \right) \] 139 ;
\[ y \leftarrow \text{query} \left( x \right) \] 140 ;
\[ z \leftarrow \text{query} \left( + \left( x, y \right) \right) \] 141 ;
\[ w \leftarrow \text{query} \left( * \left( \text{chi} : y : , \text{chi} : z : \right) \right) \] 142 ;
\[ z \leftarrow \text{query} \left( 0 \right) \] 143 ;
\[ \text{if} \left[ > \left( x , 0 \right) \right] 144 \]
\[ \text{then} \left[ y \leftarrow \text{query} \left( x \right) \right] 145 \]
\[ \text{else} \left[ w \leftarrow \text{query} \left( z \right) \right] 146 ; \]
\[ y \leftarrow \text{query} \left( \text{cs} \right) \] 147 ;
\[ z \leftarrow \text{query} \left( \text{c} \right) \] 148 ;
\[ w \leftarrow \text{query} \left( z \right) \] 149 ;
\[ x \leftarrow \text{query} \left( w \right) \] 150 ;
\[ y \leftarrow \text{query} \left( x \right) \] 151 ;
\[ z \leftarrow \text{query} \left( y \right) \] 152 ;
\[ w \leftarrow \text{query} \left( z \right) \] 153 ;
\[ x \leftarrow \text{query} \left( 0 \right) \] 154 ;
\[ y \leftarrow \text{query} \left( x \right) \] 155 ;
\[ z \leftarrow \text{query} \left( + \left( x, y \right) \right) \] 156 ;
\[ w \leftarrow \text{query} \left( * \left( \text{chi} : y : , \text{chi} : z : \right) \right) \] 157 ;
\[ z \leftarrow \text{query} \left( 0 \right) \] 158 ;
\[ x \leftarrow \text{query} \left( w \right) \] 159 ;
\[ i \leftarrow k \] 160 ;
\[ x \leftarrow \text{query} \left( \text{chi} : \text{cs} : \right) \] 161 ;
\[ w \leftarrow - \left( w, 1 \right) \] 162 ;
\[ i \leftarrow - \left( i, 1 \right) \] 163 ;
\[ j \leftarrow k \] 164 ;
\[ y \leftarrow \text{query} \left( \text{chi} : x : \right) \] 165 ;
\[ y \leftarrow \text{query} \left( \text{chi} : x : \right) \] 166 ;
\[ j \leftarrow - \left( j, 1 \right) \] 167 ;
\[ x \leftarrow \text{query} \left( \text{chi} : x : \right) \] 168 ;
\[ x \leftarrow \text{query} \left( \text{cs} \right) \] 169 ;
\[ y \leftarrow \text{query} \left( x \right) \] 170 ;
\[ z \leftarrow \text{query} \left( \text{ns} \right) \] 171 ;
\[ w \leftarrow \text{query} \left( z \right) \] 172 ;
\[ a \leftarrow x \] 173 ;
\[ c \leftarrow z \] 174 ;
\[ j \leftarrow k \] 175 ;
\[ y \leftarrow \text{query} \left( \text{chi} : x : \right) \] 176 ;
\[ \text{da} \leftarrow \text{query} \left( * \left( a, c \right) \right) \] 177 ;
[ dc <- query ( * ( a , c ) ) ] 178 ;
[ a <- - (a, da) ] 179 ;
[ c <- - (c, dc) ] 180 ;
[ j <- - (j, 1 ) ] 181 ;
[ x <- query ( 0 ) ] 182 ;
[ y <- query ( cs ) ] 183 ;
[ z <- query ( c ) ] 184 ;
[ w <- query ( z ) ] 185 ;
[ x <- query ( 0 ) ] 186 ;
[ y <- query ( x ) ] 187 ;
[ z <- query ( y ) ] 188 ;
[ w <- query ( z ) ] 189 ;
[ x <- query ( 0 ) ] 190 ;
[ y <- query ( x ) ] 191 ;
[ z <- query ( x , y ) ] 192 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 193 ;
[ b <- + ( a, z ) ] 194 ;
[ c <- + ( a, b ) ] 195 ;
[ w <- query ( a ) ] 196 ;
[ x <- query ( b ) ] 197 ;
[ y <- query ( c ) ] 198 ;
[ z <- query ( z ) ] 199 ;
[ w <- query ( z ) ] 200 ;
[ j <- - (w, 1 ) ] 201 ;
[ x <- query ( 0 ) ] 202 ;
[ y <- query ( cs ) ] 203 ;
[ z <- query ( c ) ] 204 ;
[ w <- query ( z ) ] 205 ;
[ x <- query ( 0 ) ] 206 ;
[ y <- query ( x ) ] 207 ;
[ z <- query ( x ) ] 208 ;
[ w <- query ( z ) ] 209 ;
[ x <- query ( 0 ) ] 210 ;
[ d <- + ( x, w ) ] 211 ;
[ e <- + ( c, z ) ] 212 ;
[ f <- + ( a, b ) ] 213 ;
[ x <- query ( w ) ] 214 ;
[ y <- query ( x ) ] 215 ;
[ z <- query ( y ) ] 216 ;
[ x <- query ( z ) ] 217 ;
[ g <- + ( f, w ) ] 218 ;
[ h <- + ( c, x ) ] 219 ;
[ i <- + ( w, e ) ] 220 ;
[ z <- query ( x ) ] 221 ;
[ cs <- + ( cs, 0 ) ] 222 ;
[ ns <- + ( ns, 0 ) ] 223 ;
[ i <- k ] 224 ;
[ x <- query ( z ) ] 225 ;
[ y <- query ( cs ) ] 226 ;
[ w <- - (w, 1 ) ] 227 ;
[ i <- - (i, 1) ] 228 ;
[ j <- - (j, 1 ) ] 229 ;
[ if [ > (x , 0) ] 230
then { [ y <- query ( + ( chi : x : , chi : y : ) ) ] 231 } [ y <- query ( chi : x :) ] 232 } ;
[ y <- query ( chi : x :) ] 233 ;
[ j <- - (j, 1) ] 234 ;
x <- query ( + (x, y) )

y <- query (z)

w <- query (z)
y <- query (x)
z <- query (+ (x, y))
w <- query (* (chi : y : , chi : z :))
z <- query (0)

if [ > (x, 0) ]
then { y <- query (x) } else { w <- query (z) }

y <- query (cs)
z <- query (c)
w <- query (z)
x <- query (w)
y <- query (x)
z <- query (0)
y <- query (x)
z <- query (0)
w <- query (* (chi : y : , chi : z :) )
z <- query (0)
x <- query (w)
i <- k
x <- query (chi : cs :)
w <- - (w, 1)
i <- - (i, 1)
j <- k
y <- query (chi : x :)
w <- - (w, 1)
j <- - (j, 1)
i <- k
y <- query (chi : x :)
y <- query (cs)
z <- query (ns)
w <- query (z)
a <- x
b <- query (cs)
...
[ x <- query ( cs ) ] 349 ;
[ y <- query ( x ) ] 350 ;
[ z <- query ( ns ) ] 351 ;
[ w <- query ( z ) ] 352 ;
[ a <- x ] 353 ;
[ c <- z ] 354 ;
[ j <- k ] 355 ;
[ y <- query ( chi : x :) ] 356 ;
[ da <- query ( * ( a , c ) ) ] 357 ;
[ dc <- query ( * ( a , c ) ) ] 358 ;
[ a <- - (a, da) ] 359 ;
[ c <- - (c, dc) ] 360 ;
[ j <- - (j, 1 ) ] 361 ;
[ x <- query ( 0 ) ] 362 ;
[ y <- query ( cs ) ] 363 ;
[ z <- query ( c ) ] 364 ;
[ w <- query ( z ) ] 365 ;
[ x <- query ( 0 ) ] 366 ;
[ y <- query ( x ) ] 367 ;
[ z <- query ( y ) ] 368 ;
[ w <- query ( z ) ] 369 ;
[ x <- query ( 0 ) ] 370 ;
[ y <- query ( x ) ] 371 ;
[ z <- query ( * (x, y) ) ] 372 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 373 ;
[ z <- query ( 0 ) ] 374 ;
[ x <- query ( w ) ] 375 ;
if [ > (x , 0) ] 376
then { [ y <- query ( x ) ] 377 }
else { [ w <- query ( 0 ) ] 378 } ;
[ a <- x ] 379 ;
[ c <- z ] 380 ;
[ j <- k ] 381 ;
[ cs <- + ( cs, 0 ) ] 382 ;
[ ns <- + ( ns, 0 ) ] 383 ;
[ i <- k ] 384 ;
[ x <- query ( 0 ) ] 385 ;
[ y <- query ( cs ) ] 386 ;
[ w <- - ( w, 1 ) ] 387 ;
[ i <- - (i, 1) ] 388 ;
[ j <- k ] 389 ;
if [ > (x , 0) ] 390
then { [ y <- query ( + ( chi : x : , chi : y : ) ) ] 391 }
else { [ y <- query ( chi : x :) ] 392 } ;
[ y <- query ( chi : x :) ] 393;
[ j <- - (j, 1) ] 394 ;
[ x <- query ( + ( x, y ) ) ] 395 ;
[ x <- query ( 0 ) ] 396 ;
[ y <- query ( cs ) ] 397 ;
[ z <- query ( c ) ] 398 ;
[ w <- query ( z ) ] 399 ;
[ y <- query ( x ) ] 400 ;
[ z <- query ( + (x, y) ) ] 401 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 402 ;
[ z <- query ( 0 ) ] 403 ;
if [ > (x , 0) ] 404
then { [ y <- query ( w ) ] 405 }
407 else { [ w <- query ( z ) ] 406 } ;
408 [ y <- query ( cs ) ] 407 ;
409 [ z <- query ( c ) ] 408 ;
410 [ w <- query ( z ) ] 409 ;
411 [ x <- query ( 0 ) ] 410 ;
412 if [ > ( w , 0) ] 411 then { [ y <- w ] 412 } ;
413 else { [ w <- query ( z ) ] 413 } ;
414 [ x <- query ( w ) ] 414 ;
415 [ y <- query ( x ) ] 415 ;
416 [ z <- query ( y ) ] 416 ;
417 [ x <- query ( z ) ] 417 ;
418 [ g <- + ( f, w ) ] 418 ;
419 [ h <- + ( c, x ) ] 419 ;
420 [ i <- + ( w, e ) ] 420 ;
421 [ z <- query ( x ) ] 421 ;
422 [ cs <- + ( cs, 0 ) ] 422 ;
423 [ ns <- + ( ns, 0 ) ] 423 ;
424 [ i <- k ] 424 ;
425 [ x <- query ( z ) ] 425 ;
426 [ y <- query ( cs ) ] 426 ;
427 [ w <- - ( w, 1 ) ] 427 ;
428 [ i <- - (i, 1) ] 428 ;
429 [ j <- k ] 429 ;
430 if [ > (x , 0) ] 430 then { [ y <- query ( + ( chi : x : , chi : y : ) ) ] 431 } ;
431 else { [ y <- query ( chi : x : ) ] 432 } ;
432 [ y <- query ( chi : x :) ] 433 ;
433 [ j <- - (j, 1) ] 434 ;
434 [ x <- query ( + (x, y) ) ] 435 ;
435 [ x <- query ( z ) ] 436 ;
436 [ y <- query ( cs ) ] 437 ;
437 [ z <- query ( c ) ] 438 ;
438 [ w <- query ( z ) ] 439 ;
439 [ y <- query ( w ) ] 440 ;
440 [ z <- query ( + (x,y) ) ] 441 ;
441 [ w <- query ( * (chi : y : , chi : z :) ) ] 442 ;
442 [ z <- query ( 0 ) ] 443 ;
443 if [ > (x , 0) ] 444 then { [ y <- query ( x ) ] 445 } ;
444 else { [ w <- query ( z ) ] 446 } ;
445 [ y <- query ( cs ) ] 447 ;
446 [ z <- query ( c ) ] 448 ;
447 [ w <- query ( z ) ] 449 ;
448 [ x <- query ( w ) ] 450 ;
449 [ y <- query ( x ) ] 451 ;
450 [ z <- query ( w ) ] 452 ;
451 [ w <- query ( z ) ] 453 ;
452 [ x <- query ( 0 ) ] 454 ;
453 [ y <- query ( x ) ] 455 ;
454 [ z <- query ( + (x, y) ) ] 456 ;
455 [ w <- query ( * (chi : y : , chi : z :) ) ] 457 ;
456 [ z <- query ( 0 ) ] 458 ;
457 [ x <- query ( w ) ] 459 ;
458 [ i <- k ] 460 ;
459 [ x <- query ( chi : cs : ) ] 461 ;
460 [ i <- - (i, 1) ] 462 ;
Example H.18 (jumboS). The composed program with nested loops.
[ y <- \text{query} (\ x \ ) \ ] \ 15 ;
[ z <- \text{query} (\ y \ ) \ ] \ 16 ;
[ x <- \text{query} (\ z \ ) \ ] \ 17 ;
[ g <- \text{+} (\ f, \ w \ ) \ ] \ 18 ;
[ h <- \text{+} (\ c, \ x \ ) \ ] \ 19 ;
[ i <- \text{+} (\ w, \ e \ ) \ ] \ 20 ;
[ z <- \text{query} (\ x \ ) \ ] \ 21 ;
\text{if} \ [ > (\ x, \ 0) \ ] \ 22
\text{then} \ \{ \ [ y <- \text{query} (\ 0 \ ) \ ] \ 23 \}
\text{else} \ \{ \ [ w <- \text{query} (\ 0 \ ) \ ] \ 24 \} ;
[ x <- - (\ y, \ w \ ) \ ] \ 25 ;
[ j <- 5 \ ] \ 26 ;
[ x <- \text{query} (\ \chi : \ x : \ ) \ ] \ 27 ;
\text{while} \ [ > (\ j, \ 0) \ ] \ 28 \ \text{do} \ \{ 
[ j <- - (\ j, \ 1) \ ] \ 29 ;
\text{if} \ [ < (\ j, \ 5) \ ] \ 30
\text{then} \ \{ \ [ y <- \text{query} (\ \chi : \ x :) \ ] \ 31 \}
\text{else} \ \{ \ [ w <- \text{query} (\ \chi : \ x :) \ ] \ 32 \} ;
\text{while} \ [ > (\ j, \ 0) \ ] \ 33 \ \text{do} \ \{ 
[ j <- - (\ j, \ 1) \ ] \ 34 ;
\text{if} \ [ < (\ j, \ 5) \ ] \ 35
\text{then} \ \{ \ [ y <- \text{query} (\ \chi : \ x :) \ ] \ 36 \}
\text{else} \ \{ \ [ w <- \text{query} (\ \chi : \ x :) \ ] \ 37 \} ;
[ x <- \text{query} (\ \chi : \ y :) \ ] \ 38 ;
[ y <- \text{query} (\ x \ ) \ ] \ 39 ;
[ z <- \text{query} (\ + (\ x, \ y) \ ) \ ] \ 40 ;
[ w <- \text{query} (\ * (\chi : \ y : , \chi : z :) \ ) \ ] \ 41 ;
[ z <- \text{query} (\ w \ ) \ ] \ 42 ;
[ g <- \text{+} (\ f, \ z \ ) \ ] \ 43 ;
[ h <- \text{+} (\ c, \ x \ ) \ ] \ 44 ;
[ i <- \text{+} (\ w, \ g \ ) \ ] \ 45 ;
[ z <- \text{query} (\ x \ ) \ ] \ 46 ;
[ e <- \text{+} (\ c, \ z \ ) \ ] \ 47 ;
[ f <- \text{+} (\ a, \ i \ ) \ ] \ 48 ;
[ x <- \text{query} (\ 0 \ ) \ ] \ 49 ;
[ y <- \text{query} (\ x \ ) \ ] \ 50 ;
[ z <- \text{query} (\ x \ ) \ ] \ 51 ;
[ i <- k \ ] \ 52 ;
[ x <- \text{query} (\ y \ ) \ ] \ 53 ;
[ y <- \text{query} (\ x \ ) \ ] \ 54 ;
\text{while} \ [ > (\ i, \ 0) \ ] \ 55 \ \text{do} \ \{ 
[ i <- - (\ i, \ 1) \ ] \ 56 ;
[ y <- \text{query} (\ + (\ z, \ y) \ ) \ ] \ 57 ;
[ j <- k \ ] \ 58 ;
\text{while} \ [ > (\ j, \ 0) \ ] \ 59 \ \text{do} \ \{ 
[ j <- - (\ j, \ 1) \ ] \ 60 ;
[ x <- \text{query} (\ + (\ x, \ y) \ ) \ ] \ 61 \} ;
[ j <- N \ ] \ 62 ;
[ l <- x \ ] \ 63 ;
[ cs <- -1 \ ] \ 64 ;
[ ns <- -1 \ ] \ 65 ;
\text{while} \ [ < (\ 0, \ j) \ ] \ 66 \ \text{do} \ \{ 
[ j <- - (\ j, \ 1) \ ] \ 67 ;
[ cs <- + (\ cs, \ 0) \ ] \ 68 ;
[ ns <- + (\ ns, \ 0) \ ] \ 69 \} ;
Example H.19 (jumbo). The composed program with multiple paths nested loops.
{ \textbf{i} \leftarrow k \}; 52
{ \textbf{x} \leftarrow \text{query}(\textbf{y})}; 53
{ \textbf{y} \leftarrow \text{query}(\textbf{x})}; 54
\textbf{while} \left\{ \begin{array}{l}
\textbf{i} \leftarrow (i, 0); 55 \\
\textbf{y} \leftarrow \text{query}(\text{+(z, y)}); 56 \\
\textbf{j} \leftarrow k; 57 \\
\textbf{while} \left\{ \begin{array}{l}
\textbf{j} \leftarrow -(j, 1); 58 \\
\textbf{y} \leftarrow \text{query}(\text{+(x, y)}); 59 \\
\textbf{j} \leftarrow -(j, 1); 60 \\
\textbf{x} \leftarrow \text{query}(\text{+(x, y)}); 61 \\
\end{array} \right. 62 \\
\end{array} \right. 63
\}; 64
{ \textbf{j} \leftarrow \text{N}; 65
{ \textbf{l} \leftarrow \text{x}; 66
{ \textbf{cs} \leftarrow \text{N} \}; 67
{ \textbf{ns} \leftarrow \text{N} \}; 68
\textbf{while} \left\{ \begin{array}{l}
\textbf{j} \leftarrow -(j, 1); 69 \\
\textbf{cs} \leftarrow +(\text{cs}, \text{N}); 70 \\
\textbf{ns} \leftarrow +(\text{ns}, \text{N}); 71 \\
\textbf{w} \leftarrow k; 72
\textbf{while} \left\{ \begin{array}{l}
\textbf{w} \leftarrow -(w, 1); 73 \\
\textbf{p} \leftarrow \text{c}; 74
\textbf{q} \leftarrow \text{c}; 75
\textbf{a} \leftarrow \text{query}(\text{1}); 76
{ \textbf{i} \leftarrow \text{N}; 77
\textbf{while} \left\{ \begin{array}{l}
{ \textbf{i} \leftarrow -(i, 1); 78 \\
\textbf{csi} \leftarrow +(\text{csi}, \text{+}(\text{a}, \text{p}), \text{-(q, p)})); 79 \\
\textbf{if} \left\{ \begin{array}{l}
\textbf{i} \leftarrow -(i, 1); 80 \\
\textbf{nsi} \leftarrow +(\text{nsi}, \text{+}(\text{a}, \text{p}), \text{-(q, p)})); 81 \\
\textbf{else} \left\{ \begin{array}{l}
\textbf{nsi} \leftarrow \text{nsi}; 82 \\
\end{array} \right. 83 \\
\end{array} \right. 84
\textbf{i2} \leftarrow \text{N}; 85
\textbf{while} \left\{ \begin{array}{l}
\textbf{i2} \leftarrow -(i2, 1); 86 \\
\textbf{i} \leftarrow -(i2, 1); 87 \\
\textbf{a} \leftarrow \text{x}; 88
{ \textbf{c} \leftarrow \text{z}; 89
{ \textbf{j} \leftarrow \text{k}; 90
\textbf{while} \left\{ \begin{array}{l}
\textbf{a} \leftarrow \text{-(a, da)}; 91 \\
{ \textbf{dc} \leftarrow \text{query}(\text{+}(\text{a}, \text{c})}; 92
\textbf{c} \leftarrow \text{-(c, dc)}; 93
{ \textbf{j} \leftarrow -(j, 1); 94
\textbf{x} \leftarrow \text{query}(\text{0}); 95
{ \textbf{y} \leftarrow \text{query}(\text{cs}); 96
\end{array} \right. 97
\}; 98
\}; 99
}; 100
}; 101
}; 102
}; 103
}; 104
}; 105
}; 106
}; 107
}; 108
}; 109
}; 110
}; 111
}; 112
}; 113
}; 114
}; 115
}; 116
}; 117
}
[ z <- query ( c ) ] 104 ;
[w <- query ( z ) ] 105 ;
[x <- query ( 0 ) ] 106 ;
y <- query ( x ) 107 ;
[z <- query ( y ) ] 108 ;
[w <- query ( z ) ] 109 ;
[x <- query ( 0 ) ] 110 ;
y <- query ( x ) 111 ;
[z <- query ( + (x, y) ) ] 112 ;
w <- query ( * (chi : y : , chi : z :) ) ] 113 ;
z <- query ( w ) 114 ;
x <- query ( e ) 115 ;
if [ > (x , 0) ] 116
then { [ y <- query ( 0 ) ] 117 }
else { [ w <- query ( 0 ) ] 118 } ;
a <- x 119 ;
c <- z 120 ;
j <- k 121 ;
cs <- + ( cs, 0 ) ] 122 ;
ns <- + ( ns, 0 ) ] 123 ;
i <- k ] 124 ;
[x <- query ( 0 ) ] 125 ;
y <- query ( cs ) 126 ;
while [ > (i , 0) ] 127 do {
i <- - (i, 1) 128 ;
j <- k 129 ;
if [ > (x , 0) ] 130
then { [ y <- query ( + ( chi : x : , chi : y : ) ) ] 131 }
else { [ y <- query ( chi : x : ) ] 132 } ;
while [ > (j , 0) ] 133 do
j <- - (j, 1) 134 ;
x <- query ( + ( x, y) ) 135 ;
} ;
[x <- query ( z ) ] 136 ;
y <- query ( cs ) ] 137 ;
z <- query ( c ) 138 ;
w <- query ( z ) ] 139 ;
y <- query ( x ) ] 140 ;
z <- query ( + (x, y) ) ] 141 ;
w <- query ( * (chi : y : , chi : z :) ) ] 142 ;
z <- query ( 0 ) ] 143 ;
if [ > (x , 0) ] 144
then { [ y <- query ( x ) ] 145 }
else { [ w <- query ( z ) ] 146 } ;
y <- query ( cs ) ] 147 ;
z <- query ( c ) 148 ;
w <- query ( z ) ] 149 ;
x <- query ( w ) ] 150 ;
y <- query ( x ) ] 151 ;
z <- query ( y ) ] 152 ;
w <- query ( z ) ] 153 ;
x <- query ( 0 ) ] 154 ;
y <- query ( x ) ] 155 ;
z <- query ( + (x, y) ) ] 156 ;
w <- query ( * (chi : y : , chi : z :) ) ] 157 ;
z <- query ( 0 ) ] 158 ;
x <- query ( w ) ] 159 ;
\[ i \leftarrow k \];
\[ x \leftarrow \text{query}(\chi:cs:) \];
while \[ i > 0 \] do {
\[ j \leftarrow k \];
\[ y \leftarrow \text{query}(\chi:x:) \];
while \[ j > 0 \] do {
\[ j \leftarrow (j-1) \];
\[ x \leftarrow \text{query}(\chi:x:) \];
}
\[ x \leftarrow \text{query}(\chi:x:) \];
\[ y \leftarrow \text{query}(x) \];
\[ z \leftarrow \text{query}(cs) \];
\[ w \leftarrow \text{query}(z) \];
while \[ < 0, j \] do {
\[ da \leftarrow \text{query}(*(a,c)) \];
\[ dc \leftarrow \text{query}(*(a,c)) \];
\[ a \leftarrow (a, da) \];
\[ c \leftarrow (c, dc) \];
\[ j \leftarrow (j-1) \];
\[ x \leftarrow \text{query}((0)) \];
\[ y \leftarrow \text{query}(cs) \];
\[ z \leftarrow \text{query}(c) \];
\[ w \leftarrow \text{query}(z) \];
\[ x \leftarrow \text{query}((0)) \];
\[ y \leftarrow \text{query}(x) \];
\[ z \leftarrow \text{query}(y) \];
\[ w \leftarrow \text{query}(z) \];
\[ x \leftarrow \text{query}((0)) \];
\[ y \leftarrow \text{query}(x) \];
\[ z \leftarrow \text{query}(y) \];
\[ w \leftarrow \text{query}(z) \];
\[ x \leftarrow \text{query}((0)) \];
\[ y \leftarrow \text{query}(x) \];
\[ z \leftarrow \text{query}(w) \];
\[ w \leftarrow \text{query}(z) \];
\[ j \leftarrow (w-1) \];
\[ x \leftarrow \text{query}((0)) \];
\[ y \leftarrow \text{query}(cs) \];
\[ z \leftarrow \text{query}(c) \];
\[ w \leftarrow \text{query}(z) \];
\[ x \leftarrow \text{query}((0)) \];
\[ y \leftarrow \text{query}(x) \];
\[ z \leftarrow \text{query}(y) \];
\[ w \leftarrow \text{query}(z) \];
\[ x \leftarrow \text{query}((0)) \];
\[ d \leftarrow (+x,w) \];
\[ e \leftarrow (+c,z) \];
\[ f \leftarrow (+a,b) \];
\[ x \leftarrow \text{query}(w) \];
[ y <- query ( x ) ] 215 ;
[ z <- query ( y ) ] 216 ;
[ x <- query ( z ) ] 217 ;
[ g <- + ( f, w ) ] 218 ;
[ h <- + ( c, x ) ] 219 ;
[ i <- + ( w, e ) ] 220 ;
[ z <- query ( x ) ] 221 ;
[ cs <- + ( cs, 0 ) ] 222 ;
[ ns <- + ( ns, 0 ) ] 223 ;
[ i <- k ] 224 ;
[ x <- query ( z ) ] 225 ;
[ y <- query ( cs ) ] 226 ;
[ g <- + ( f, w ) ] 218 ;
[ h <- + ( c, x ) ] 219 ;
[ i <- k ] 224 ;
while [ > ( i, 0 ) ] 227 do {
[ i <- - ( i, 1 ) ] 228 ;
[ j <- k ] 229 ;
if [ > ( x, 0 ) ] 230
then { [ y <- query ( + ( chi : x : , chi : y : ) ) ] 231 }
else { [ y <- query ( chi : x : ) ] 232 } ;
while [ > ( j, 0 ) ] 233 do
[ [ j <- - ( j, 1 ) ] 234 ;
[ x <- query ( + ( x, y ) ) ] 235 }
};
[ x <- query ( z ) ] 236 ;
[ y <- query ( cs ) ] 237 ;
[ z <- query ( c ) ] 238 ;
[ w <- query ( z ) ] 239 ;
[ y <- query ( x ) ] 240 ;
[ z <- query ( + ( x, y ) ) ] 241 ;
[ w <- query ( * ( chi : x : , chi : y : ) ) ] 242 ;
[ z <- query ( 0 ) ] 243 ;
if [ > ( x, 0 ) ] 244
then { [ y <- query ( x ) ] 245 }
else { [ w <- query ( z ) ] 246 } ;
[ y <- query ( cs ) ] 247 ;
[ z <- query ( c ) ] 248 ;
[ w <- query ( z ) ] 249 ;
[ x <- query ( w ) ] 250 ;
[ y <- query ( x ) ] 251 ;
[ z <- query ( w ) ] 252 ;
[ w <- query ( z ) ] 253 ;
[ x <- query ( 0 ) ] 254 ;
[ y <- query ( x ) ] 255 ;
[ z <- query ( + ( x, y ) ) ] 256 ;
[ w <- query ( * ( chi : x : , chi : y : ) ) ] 257 ;
[ z <- query ( 0 ) ] 258 ;
[ x <- query ( w ) ] 259 ;
[ i <- k ] 260 ;
[ x <- query ( chi : cs : ) ] 261 ;
while [ > ( i, 0 ) ] 262 do {
[ i <- - ( i, 1 ) ] 263 ;
[ j <- k ] 264 ;
[ y <- query ( chi : x : ) ] 265 ;
while [ > ( j, 0 ) ] 266 do
[ [ j <- - ( j, 1 ) ] 267 ;
[ x <- query ( chi : x : ) ] 268 }
] ;
[ x <- query ( cs ) ] 269 ;
}
282 \[ y \leftarrow \text{query} \left( x \right) \] 270 ;
283 \[ z \leftarrow \text{query} \left( ns \right) \] 271 ;
284 \[ w \leftarrow \text{query} \left( z \right) \] 272 ;
285 \[ a \leftarrow x \] 273 ;
286 \[ c \leftarrow z \] 274 ;
287 \[ j \leftarrow k \] 275 ;
288 \text{while} [ \left< (0, j) \right] 276 do {
289 \[ da \leftarrow \text{query} \left( * \left( a, c \right) \right) \] 277 ;
290 \[ dc \leftarrow \text{query} \left( * \left( a, c \right) \right) \] 278 ;
291 \[ a \leftarrow - (a, da) \] 279 ;
292 \[ c \leftarrow - (c, dc) \] 280 ;
293 \[ j \leftarrow - (j, 1) \] 281 ;
294 };
295 \[ x \leftarrow \text{query} \left( a \right) \] 282 ;
296 \[ y \leftarrow \text{query} \left( cs \right) \] 283 ;
297 \[ z \leftarrow \text{query} \left( c \right) \] 284 ;
298 \[ w \leftarrow \text{query} \left( z \right) \] 285 ;
299 \[ x \leftarrow \text{query} \left( w \right) \] 286 ;
300 \[ y \leftarrow \text{query} \left( x \right) \] 287 ;
301 \[ z \leftarrow \text{query} \left( y \right) \] 288 ;
302 \[ w \leftarrow \text{query} \left( z \right) \] 289 ;
303 \[ x \leftarrow \text{query} \left( 0 \right) \] 290 ;
304 \[ y \leftarrow \text{query} \left( x \right) \] 291 ;
305 \[ z \leftarrow \text{query} \left( * \left( x, y \right) \right) \] 292 ;
306 \[ w \leftarrow \text{query} \left( * \left( \chi : y : , \chi : z : \right) \right) \] 293 ;
307 \[ z \leftarrow \text{query} \left( 0 \right) \] 294 ;
308 \[ x \leftarrow \text{query} \left( w \right) \] 295 ;
309 \text{if} [ \left> (x, 0) \right] 296
310 \text{then} \{ \[ y \leftarrow \text{query} \left( 0 \right) \] 297 \}
311 \text{else} \{ \[ w \leftarrow \text{query} \left( 0 \right) \] 298 \} ;
312 \[ a \leftarrow x \] 299 ;
313 \[ c \leftarrow z \] 300 ;
314 \[ j \leftarrow k \] 301 ;
315 \[ cs \leftarrow + \left( cs, 0 \right) \] 302 ;
316 \[ ns \leftarrow + \left( ns, 0 \right) \] 303 ;
317 \[ i \leftarrow k \] 304 ;
318 \[ x \leftarrow \text{query} \left( 0 \right) \] 305 ;
319 \[ y \leftarrow \text{query} \left( cs \right) \] 306 ;
320 \text{while} [ \left> (i, 0) \right] 307 do {
321 \[ i \leftarrow - (i, 1) \] 308 ;
322 \[ j \leftarrow k \] 309 ;
323 \text{if} [ \left> (x, 0) \right] 310
324 \text{then} \{ \[ y \leftarrow \text{query} \left( + \left( \chi : x : , \chi : y : \right) \right) \] 311 \}
325 \text{else} \{ \[ y \leftarrow \text{query} \left( \chi : x : \right) \] 312 \} ;
326 \text{while} [ \left> (j, 0) \right] 313 do
327 \[ j \leftarrow - (j, 1) \] 314 ;
328 \[ x \leftarrow \text{query} \left( + \left( x, y \right) \right) \] 315 ;
329 };
330 \[ x \leftarrow \text{query} \left( c \right) \] 316 ;
331 \[ y \leftarrow \text{query} \left( cs \right) \] 317 ;
332 \[ z \leftarrow \text{query} \left( c \right) \] 318 ;
333 \[ w \leftarrow \text{query} \left( z \right) \] 319 ;
334 \[ y \leftarrow \text{query} \left( x \right) \] 320 ;
335 \[ z \leftarrow \text{query} \left( + \left( x, y \right) \right) \] 321 ;
336 \[ w \leftarrow \text{query} \left( * \left( \chi : y : , \chi : z : \right) \right) \] 322 ;
337 \[ z \leftarrow \text{query} \left( 0 \right) \] 323 ;
338 \text{if} [ \left> (x, 0) \right] 324
then { [ y <- query ( w ) ] 325 }
else { [ w <- query ( z ) ] 326 } ;
 [ y <- query ( cs ) ] 327 ;
 [ z <- query ( c ) ] 328 ;
 [ w <- query ( z ) ] 329 ;
 [ x <- query ( 0 ) ] 330 ;
 [ y <- query ( x ) ] 331 ;
 [ z <- query ( y ) ] 332 ;
 [ w <- query ( z ) ] 333 ;
 [ x <- query ( y ) ] 334 ;
 [ y <- query ( x ) ] 335 ;
 [ z <- query ( * (x, y) ) ] 336 ;
 [ w <- query ( * (chi : y : , chi : z :) ) ] 337 ;
 [ z <- query ( 0 ) ] 338 ;
 [ x <- query ( w ) ] 339 ;
 [ i <- k ] 340 ;
 [ x <- query ( chi : cs : ) ] 341 ;
while [ > (i , 0) ] 342 do {
 [ i <- - (i, 1) ] 343 ;
 [ j <- k ] 344 ;
 [ y <- query ( chi : x : ) ] 345 ;
while [ > (j , 0) ] 346 do
 [ j <- - (j, 1) ] 347 ;
 [ x <- query ( chi : x : ) ] 348 } 349
 [ x <- query ( cs ) ] 349 ;
 [ y <- query ( x ) ] 350 ;
 [ z <- query ( ns ) ] 351 ;
 [ w <- query ( z ) ] 352 ;
 [ a <- x ] 353 ;
 [ c <- z ] 354 ;
 [ j <- k ] 355 ;
while [ < (0, j) ] 356 do {
 [ da <- query ( * ( a , c ) ) ] 357 ;
 [ dc <- query ( * ( a , c ) ) ] 358 ;
 [ a <- - (a, da) ] 359 ;
 [ c <- - (c, dc) ] 360 ;
 [ j <- - (j, 1 ) ] 361 }
 [ x <- query ( 0 ) ] 362 ;
 [ y <- query ( cs ) ] 363 ;
 [ z <- query ( c ) ] 364 ;
 [ w <- query ( z ) ] 365 ;
 [ x <- query ( 0 ) ] 366 ;
 [ y <- query ( x ) ] 367 ;
 [ z <- query ( y ) ] 368 ;
 [ w <- query ( z ) ] 369 ;
 [ x <- query ( 0 ) ] 370 ;
 [ y <- query ( x ) ] 371 ;
 [ z <- query ( * (x, y) ) ] 372 ;
 [ w <- query ( * (chi : y : , chi : z :) ) ] 373 ;
 [ z <- query ( 0 ) ] 374 ;
 [ x <- query ( w ) ] 375 ;
if [ > (x , 0) ] 376
then { [ y <- query ( x ) ] 377 }
else { [ w <- query ( 0 ) ] 378 } ;
 [ a <- x ] 379 ;


```plaintext
396 [ c <- z ] 380 ;
397 [ j <- k ] 381 ;
398 [ cs <- + ( cs, 0 ) ] 382 ;
399 [ ns <- + ( ns, 0 ) ] 383 ;
400 [ i <- k ] 384 ;
401 [ x <- query ( 0 ) ] 385 ;
402 [ y <- query ( cs ) ] 386 ;
403 while [ > (i , 0) ] 387 do {
404 [ i <- - (i, 1) ] 388 ;
405 [ j <- k ] 389 ;
406 if [ > (x , 0) ] 390
407 then {[ y <- query ( + ( chi : x : , chi : y : ) ) ] 391 }
408 else {[ y <- query ( chi : x : ) ] 392 } ;
409 while [ > (j , 0) ] 393 do
410 {[ j <- - (j, 1) ] 394 ;
411 [ x <- query ( + ( x, y) ) ] 395 }
412 ;}
413 [ x <- query ( 0 ) ] 396 ;
414 [ y <- query ( cs ) ] 397 ;
415 [ z <- query ( c ) ] 398 ;
416 [ w <- query ( z ) ] 399 ;
417 [ y <- query ( x ) ] 400 ;
418 [ z <- query ( + (x, y) ) ] 401 ;
419 [ w <- query ( * (chi : y : , chi : z :) ) ] 402 ;
420 [ z <- query ( 0 ) ] 403 ;
421 if [ > (x , 0) ] 404
422 then {[ y <- query ( w ) ] 405 }
423 else {[ w <- query ( z ) ] 406 } ;
424 [ y <- query ( cs ) ] 407 ;
425 [ z <- query ( c ) ] 408 ;
426 [ w <- query ( z ) ] 409 ;
427 [ x <- query ( 0 ) ] 410 ;
428 if [ > (w , 0) ] 411
429 then {[ y <- w ] 412 }
430 else {[ w <- query ( z ) ] 413 } ;
431 [ x <- query ( w ) ] 414 ;
432 [ y <- query ( x ) ] 415 ;
433 [ z <- query ( y ) ] 416 ;
434 [ x <- query ( z ) ] 417 ;
435 [ g <- + ( z, w ) ] 418 ;
436 [ h <- + ( c, x ) ] 419 ;
437 [ i <- + ( w, e ) ] 420 ;
438 [ z <- query ( x ) ] 421 ;
439 [ cs <- + ( cs, 0 ) ] 422 ;
440 [ ns <- + ( ns, 0 ) ] 423 ;
441 [ i <- k ] 424 ;
442 [ x <- query ( z ) ] 425 ;
443 [ y <- query ( cs ) ] 426 ;
444 while [ > (i , 0) ] 427 do {
445 [ i <- - (i, 1) ] 428 ;
446 [ j <- k ] 429 ;
447 if [ > (x , 0) ] 430
448 then {[ y <- query ( + ( chi : x : , chi : y : ) ) ] 431 }
449 else {[ y <- query ( chi : x : ) ] 432 } ;
450 while [ > (j , 0) ] 433 do
451 {[ j <- - (j, 1) ] 434 ;
452 [ x <- query ( + ( x, y) ) ] 435 }
```

93
\[ \begin{align*}
\text{if } & \text{> (x, 0)} \quad \text{then } \{ \text{y <- query (x)} \} \\
\text{else } & \{ \text{w <- query (z)} \}; \\
\text{y <- query (cs)}; \\
\text{z <- query (c)}; \\
\text{w <- query (z)}; \\
\text{x <- query (w)}; \\
\text{y <- query (x)}; \\
\text{z <- query (y)}; \\
\text{w <- query (z)}; \\
\end{align*} \]
References


