Program Analysis for Adaptivity Analysis

Contents

1 `Query While` Language - Extended

1.1 Labeled Language

Arithmetic Operators	$\oplus_a$	$::=$	$+ \mid - \mid \times \mid \div \mid \max \mid \min$	
Boolean Operators	$\oplus_b$	$::=$	$\vee \mid \wedge$	
Relational Operators	$\sim$	$::=$	$< \mid \leq \mid ==$	
Arithmetic Expression	a	$::=$	$n \mid x \mid a \oplus_a a \mid \mathtt{log}\ a \mid \mathtt{sign}\ a$	
Boolean Expression	b	$::=$	$\mathtt{true} \mid \mathtt{false} \mid \neg b \mid b \oplus_b b \mid a \sim a$	
Expression	e	$::=$	$v \mid a \mid b \mid [e, \dots, e]$	
Value	v	$::=$	$n \mid \mathtt{true} \mid \mathtt{false} \mid [] \mid [v, \dots, v]$	
			$\mid (r, x_1, \dots, x_n) := c$	
Query Expression	ψ	$::=$	$\alpha \mid a \mid \psi \oplus_a \psi \mid \chi[a]$	
Query Value	α	$::=$	$n \mid \chi[n] \mid \alpha \oplus_a \alpha \mid n \oplus_a \chi[n] \mid \chi[n] \oplus_a n$	
Label	l	$::=$	$(n \in \mathbb{N} \cup \{\mathtt{in}, \mathtt{ex}\}) \mid (l, n)$	
Labeled Command	c	$::=$	$[x \leftarrow e]^l \mid [x \leftarrow \mathtt{query}(\psi)]^l \mid [\mathtt{skip}]^l \mid \mathtt{while}\ [b]^l\ \mathtt{do}\ c \mid \mathtt{if}\ ([b]^l, c, c)$	
			$\mid [\ \mathtt{fun}\ ]^l : x(r, x_1, \dots, x_n) := c \mid [x \leftarrow\ \mathtt{call}\ (x, e_1, \dots, e_n)]^l \mid c; c$	
Event	ϵ	$::=$	$(x, l, v, \bullet) \mid (x, l, v, \alpha)$	Assignment Event
			$\mid (b, l, v, \bullet)$	Testing Event
Trace	τ	$::=$	$[] \mid \tau :: \epsilon$	

We use following notations to represent the set of corresponding terms:

$\mathcal{VAR}$	:	Set of Variables
$\mathcal{VAL}$	:	Set of Values
$\mathcal{QVAL}$	:	Set of Query Values
$\mathcal{C}$	:	Set of Commands
$\mathcal{E}$	:	Set of Events
$\mathcal{E}^{\mathtt{asn}}$	:	Set of Assignment Events
$\mathcal{E}^{\mathtt{test}}$	:	Set of Testing Events
$\mathcal{L}$	:	Set of Labels
$\mathcal{VAL}$	:	Set of Labeled Variables
$\mathcal{DB}$	:	Set of Databases
$\mathcal{T}$	:	Set of Traces
$\mathcal{T}_0(c)$	:	Set of Initial Traces, where all the input variables of the program c are initialized.
$\mathcal{QD}$	:	Domain of Query Results

Environment $\rho : \mathcal{T} \to \mathcal{VAR} \to \mathcal{VAL} \cup \{\perp\}$

$$\rho(\tau::(x,l,v,\bullet))x \triangleq v \quad \rho(\tau::(y,l,v,\bullet))x \triangleq \rho(\tau)x, y \neq x \quad \rho(\tau::(b,l,v,\bullet))x \triangleq \rho(\tau)x$$
$$\rho(\tau::(x,l,v,\alpha))x \triangleq v \quad \rho(\tau::(y,l,v,\alpha))x \triangleq \rho(\tau)x, y \neq x \quad \rho([])x \triangleq \perp$$

1.2 Trace-based Operational Semantics for `Query While` Language

$$\boxed{\langle \tau, a\rangle \Downarrow_a v : \text{Trace} \times \text{Arithmetic Expr} \Rightarrow \text{Arithmetic Value}}$$

$$\frac{}{\langle \tau, n\rangle \Downarrow_a n} \qquad \frac{\rho(\tau)x = v}{\langle \tau, x\rangle \Downarrow_a v} \qquad \frac{\langle \tau, a_1\rangle \Downarrow_a v_1 \qquad \langle \tau, a_2\rangle \Downarrow_a v_2 \qquad v_1 \oplus_a v_2 = v}{\langle \tau, a_1 \oplus_a a_2\rangle \Downarrow_a v}$$

$$\frac{\langle \tau, a\rangle \Downarrow_a v' \qquad \texttt{log}\ v' = v}{\langle \tau, \texttt{log}\ a\rangle \Downarrow_a v} \qquad \frac{\langle \tau, a\rangle \Downarrow_a v' \qquad \texttt{sign}\ v' = v}{\langle \tau, \texttt{sign}\ a\rangle \Downarrow_a v}$$

$$\boxed{\langle \tau, b\rangle \Downarrow_b v : \text{Trace} \times \text{Boolean Expr} \Rightarrow \text{Boolean Value}}$$

$$\frac{}{\langle \tau, \texttt{false}\rangle \Downarrow_b \texttt{false}} \qquad \frac{}{\langle \tau, \texttt{true}\rangle \Downarrow_b \texttt{true}} \qquad \frac{\langle \tau, b\rangle \Downarrow_b v' \qquad \neg v' = v}{\langle \tau, \neg b\rangle \Downarrow_b v}$$

$$\frac{\langle \tau, b_1\rangle \Downarrow_b v_1 \qquad \langle \tau, b_2\rangle \Downarrow_b v_2 \qquad v_1 \oplus_b v_2 = v}{\langle \tau, b_1 \oplus_b b_2\rangle \Downarrow_b v} \qquad \frac{\langle \tau, a_1\rangle \Downarrow_a v_1 \qquad \langle \tau, a_2\rangle \Downarrow_a v_2 \qquad v_1 \sim v_2 = v}{\langle \tau, a_1 \sim a_2\rangle \Downarrow_b v}$$

$$\boxed{\langle \tau, e\rangle \Downarrow_e v : \text{Trace} \times \text{Expression} \Rightarrow \text{Value}}$$

$$\frac{\langle \tau, a\rangle \Downarrow_a v}{\langle \tau, a\rangle \Downarrow_e v} \qquad \frac{\langle \tau, b\rangle \Downarrow_b v}{\langle \tau, b\rangle \Downarrow_e v} \qquad \frac{\langle \tau, e_1\rangle \Downarrow_e v_1 \cdots \langle \tau, e_n\rangle \Downarrow_e v_n}{\langle \tau, [e_1, \cdots, e_n]\rangle \Downarrow_e [v_1, \cdots, v_n]} \qquad \frac{}{\langle \tau, v\rangle \Downarrow_e v}$$

$$\boxed{\langle \tau, \psi\rangle \Downarrow_q \alpha : \text{Trace} \times \text{Query Expr} \Rightarrow \text{Query Value}}$$

$$\frac{\langle \tau, a\rangle \Downarrow_a n}{\langle \tau, a\rangle \Downarrow_q n} \qquad \frac{\langle \tau, \psi_1\rangle \Downarrow_q \alpha_1 \qquad \langle \tau, \psi_2\rangle \Downarrow_q \alpha_2}{\langle \tau, \psi_1 \oplus_a \psi_2\rangle \Downarrow_q \alpha_1 \oplus_a \alpha_2} \qquad \frac{\langle \tau, a\rangle \Downarrow_a n}{\langle \tau, \chi[a]\rangle \Downarrow_q \chi[n]} \qquad \frac{}{\langle \tau, \alpha\rangle \Downarrow_q \alpha}$$

The trace based operational semantics rules are defined in Figure 1.

Definition 1 (Label Increase)**.** *Label Increase $+ : \mathcal{L} \to \mathbb{N} \to \mathcal{L}$, increase a label l by a natural number n:*

$$n + n' \triangleq n'' \ n, n' \in \mathbb{N} \wedge \langle [], n + n'\rangle \Downarrow_a n'' \qquad (l, n) + n' \triangleq (l + n', n'') \ n, n' \in \mathbb{N} \wedge \langle [], n + n'\rangle \Downarrow_a n''$$

The case of $(l, n) + n'$ will never happen during evaluation. By Operational semantics, the only place the label increase is in rule **fun-def**, $c' = (c)^{+n}$, where c is the function body. By the rule **fun-call**, and the label augment in Definition 3, the function body c will never be augmented.

$$\boxed{\text{Command} \times \text{Trace} \to \text{Command} \times \text{Trace}} \qquad \boxed{\langle c,\tau\rangle \to \langle c',\tau'\rangle}$$

$$\frac{}{\langle [\texttt{skip}]^l, \tau\rangle \to \langle [\texttt{skip}]^l,\tau\rangle}\ \textbf{skip} \qquad \frac{\epsilon = (x,l,v,\bullet)}{\langle [x \leftarrow a]^l,\tau\rangle \to \langle[\texttt{skip}]^l, \tau::\epsilon\rangle}\ \textbf{assn}$$

$$\frac{\langle\tau,\psi\rangle \Downarrow_q \alpha \qquad \texttt{query}(\alpha)=v \qquad \epsilon=(x,l,v,\alpha)}{\langle [x\leftarrow \texttt{query}(\psi)]^l,\tau\rangle \to \langle[\texttt{skip}]^l,\tau::\epsilon\rangle}\ \textbf{query}$$

$$\frac{\langle\tau,b\rangle\Downarrow_b \texttt{true} \qquad \epsilon=(b,l,\texttt{true},\bullet)}{\langle \texttt{while}\ [b]^l\ \texttt{do}\ c,\tau\rangle \to \langle c; \texttt{while}\ [b]^l\ \texttt{do}\ c, \tau::\epsilon\rangle}\ \textbf{while-t}$$

$$\frac{\langle\tau,b\rangle\Downarrow_b \texttt{false} \qquad \epsilon=(b,l,\texttt{false},\bullet)}{\langle \texttt{while}\ [b]^l,\ \texttt{do}\ c,\tau\rangle \to \langle[\texttt{skip}]^l,\tau::\epsilon\rangle}\ \textbf{while-f} \qquad \frac{\langle c_1,\tau\rangle\to\langle c_1',\tau'\rangle}{\langle c_1;c_2,\tau\rangle \to \langle c_1';c_2,\tau'\rangle}\ \textbf{seq1}$$

$$\frac{\langle c_2,\tau\rangle\to\langle c_2',\tau'\rangle}{\langle[\texttt{skip}]^l;c_2,\tau\rangle\to\langle c_2',\tau'\rangle}\ \textbf{seq2} \qquad \frac{\langle\tau,b\rangle\Downarrow_b \texttt{true} \qquad \epsilon=(b,l,\texttt{true},\bullet)}{\langle \texttt{if}\ ([b]^l, c_1,c_2),\tau\rangle \to \langle c_1,\tau::\epsilon\rangle}\ \textbf{if-t}$$

$$\frac{\langle\tau,b\rangle\Downarrow_b \texttt{false} \qquad \epsilon=(b,l,\texttt{false},\bullet)}{\langle \texttt{if}\ ([b]^l, c_1,c_2),\tau\rangle \to \langle c_2,\tau::\epsilon\rangle}\ \textbf{if-f}$$

$$\frac{c' = c^{+n} \qquad \epsilon = (x,l,(r,x_1,\dots,x_n):=c',\bullet)}{\langle[\ \texttt{fun}\ ]^l : x(r,x_1,\dots,x_n):=c,\tau\rangle \to \langle[\texttt{skip}]^l,\tau::\epsilon\rangle}\ \textbf{fun-def}$$

$$\frac{\begin{array}{c}\langle\tau,f\rangle\Downarrow_e (r,x_1,\dots,x_n):=c \qquad \langle[x_1\leftarrow e_1]^{(l,1)};\dots;[x_n\leftarrow e_n]^{(l,n)},\tau\rangle \to^* \langle[\texttt{skip}]^{(l,n)},\tau_1\rangle \\ \langle[c]^{(l)},\tau_1\rangle \to^* \langle[\texttt{skip}]^l,\tau'\rangle \qquad \langle\tau',r\rangle\Downarrow_e v \qquad \epsilon=(x,l,v,\bullet)\end{array}}{\langle[x\leftarrow\ \texttt{call}\ (f,e_1,\dots,e_n)]^l,\tau\rangle \to \langle[\texttt{skip}]^l,\tau' :: \epsilon\rangle}\ \textbf{fun-call}$$

Figure 1: Trace-based Operational Semantics for Language.

Definition 2 (Command Label Increase)**.** *Command Label Increase* $(\cdot)^{+n} : \mathcal{C} \to \mathcal{C}$*, increase the label in command by* n*.*

$$
\begin{array}{ll}
([x \leftarrow e]^{l})^{+n} & \triangleq [x \leftarrow e]^{l+n} \\
([x \leftarrow \texttt{query}(\psi)]^{l})^{+n} & \triangleq [x \leftarrow \texttt{query}(\psi)]^{l+n} \\
([\texttt{skip}]^{l})^{+n} & \triangleq [\texttt{skip}]^{l+n} \\
(\texttt{while}\ [b]^{l}\ \texttt{do}\ c')^{+n} & \triangleq \texttt{while}\ [b]^{l+n}\ \texttt{do}\ (c')^{+n} \\
(\texttt{if}\ ([b]^{l}, c_1, c_2))^{+n} & \triangleq \texttt{if}\ ([b]^{l+n}, (c_1)^{+n}, (c_2)^{+n}) \\
([\ \texttt{fun}\ ]^{l} : x(r^{l}, x_1, \dots, x_n) := c)^{+n} & \triangleq [\ \texttt{fun}\ ]^{l+n} : x(r^{l}, x_1, \dots, x_n) := (c)^{+n} \\
([x \leftarrow\ \texttt{call}\ (x, e_1, \dots, e_n)]^{l})^{+n} & \triangleq [x \leftarrow\ \texttt{call}\ (x, e_1, \dots, e_n)]^{l+n} \\
(c_1; c_2)^{+n} & \triangleq (c_1)^{+n}; (c_2)^{+n}
\end{array}
$$

Definition 3 (Command Label Augment)**.** *Command Label Augment* $[\cdot]^{l} : \mathcal{C} \to \mathcal{C}$*, augment the label in command with a label* l *in order to record the calling site.*

$$
\begin{array}{ll}
\left[[x \leftarrow e]^{l'}\right]^{l} & \triangleq [x \leftarrow e]^{(l,l')} \\
\left[[x \leftarrow \texttt{query}(\psi)]^{l'}\right]^{l} & \triangleq [x \leftarrow \texttt{query}(\psi)]^{(l,l')} \\
\left[[\texttt{skip}]^{l'}\right]^{l} & \triangleq [\texttt{skip}]^{(l,l')} \\
\left[\texttt{while}\ [b]^{l'}\ \texttt{do}\ c'\right]^{l} & \triangleq \texttt{while}\ [b]^{(l,l')}\ \texttt{do}\ (c')^{l} \\
\left[\texttt{if}\ ([b]^{l'}, c_1, c_2)\right]^{l} & \triangleq \texttt{if}\ ([b]^{(l,l')}, (c_1)^{l}, (c_2)^{l}) \\
\left[[\ \texttt{fun}\ ]^{l'} : x(r^{l}, x_1, \dots, x_n) := c\right]^{l} & \triangleq [\ \texttt{fun}\ ]^{(l,l')} : x(r^{l}, x_1, \dots, x_n) := c \\
\left[[x \leftarrow\ \texttt{call}\ (x, e_1, \dots, e_n)]^{l'}\right]^{l} & \triangleq [x \leftarrow\ \texttt{call}\ (x, e_1, \dots, e_n)]^{(l,l')} \\
[c_1; c_2]^{l} & \triangleq [c_1]^{l}; [c_2]^{l}
\end{array}
$$

The labeled variables and assigned variables are set of variables annotated by a label. We use $\mathcal{LV}$ represents the universe of all the labeled variables and $\mathbb{AV}_c \in \mathcal{P}(\mathcal{VAR} \times \mathbb{N}) \subset \mathcal{LV}$ and $\mathbb{LV}_c \in \mathcal{P}(\mathcal{VAR} \times \mathcal{L}) \subseteq \mathcal{LV}$, represents the the set of assigned variables and labeled variables for a labeled command c, defined in Definition 5 and 4.
$FV : e \to \mathcal{P}(\mathcal{VAR})$, 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 ψ 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 4 (Assigned Variables ($\mathbb{AV} : \mathcal{C} \to \mathcal{P}(\mathcal{VAR} \times \mathbb{N})$))**.**

$$
\mathbb{AV}_c \triangleq \begin{cases}
\{x^{l}\} & c = [x \leftarrow e]^{l} \\
\{x^{l}\} & c = [x \leftarrow \texttt{query}\ (\psi)]^{l} \\
\mathbb{AV}_{c_1} \cup \mathbb{AV}_{c_2} & c = c_1; c_2 \\
\mathbb{AV}_{c} \cup \mathbb{AV}_{c_2} & c = \texttt{if}\ ([b]^{l}, c_1, c_2) \\
\mathbb{AV}_{c'} & c = \texttt{while}\ ([b]^{l}, c')
\end{cases}
$$

Definition 5 (labelled Variables ($\mathbb{LV} : \mathcal{C} \to \mathcal{P}(\mathcal{LV})$)**.**

$$
\mathbb{LV}_c \triangleq \begin{cases}
\{x^{l}\} \cup FV(e)^{in} & c = [x \leftarrow e]^{l} \\
\{x^{l}\} \cup FV(\psi)^{in} & c = [x \leftarrow \texttt{query}\ (\psi)]^{l} \\
\mathbb{LV}_{c_1} \cup \mathbb{LV}_{c_2} & c = c_1; c_2 \\
\mathbb{LV}_{c} \cup \mathbb{LV}_{c_2} \cup FV(b)^{in} & c = \texttt{if}\ ([b]^{l}, c_1, c_2) \\
\mathbb{LV}_{c'} \cup FV(b)^{in} & c = \texttt{while}\ ([b]^{l}, c')
\end{cases}
$$

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 6.

Definition 6 (Query Variables ($\mathbb{QV} : \mathcal{C} \to \mathcal{P}(\mathcal{LV})$))**.** *Given a program c, its query variables $\mathbb{QV}(c)$ is the set of variables set to the result of a query in the program. It is defined as follows:*

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

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

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

$$\forall c \in \mathcal{C}, x^{i}, y^{j} \in \mathcal{L}\ .\ x^{i}, y^{j} \in \mathbb{LV}(c) \implies x^{i} \neq y^{j}.$$

2 Event and Trace

2.1 Event

Event projection operators π_i projects the ith element from an event:
$\pi_i : \mathcal{E} \to \mathcal{VAR} \cup \text{Boolean Expression} \cup \mathbb{N} \cup \mathcal{VAL} \cup \mathcal{QVAL}$
Free Variables: $FV : e \to \mathcal{P}(\mathcal{VAR})$, the set of free variables in an expression.
$FV(\psi)$ is the set of free variables in the query expression ψ.

Definition 7 (Equivalence of Query Expression)**.** *Two query expressions ψ_1, ψ_2 are equivalent, denoted as $\psi_1 =_q \psi_2$, if and only if*

$$\begin{array}{l} \forall \tau \in \mathcal{T} \, . \, \exists \alpha_1, \alpha_2 \in \mathcal{QVAL} \, . \, (\langle \tau, \psi_1 \rangle \Downarrow_q \alpha_1 \wedge \langle \tau, \psi_2 \rangle \Downarrow_q \alpha_2) \\ \quad \wedge (\forall D \in \mathcal{DB}, r \in D \, . \, \exists v \in \mathcal{VAL} \, . \, \langle \tau, \alpha_1[r/\chi] \rangle \Downarrow_a v \wedge \langle \tau, \alpha_2[r/\chi] \rangle \Downarrow_a v) \end{array} .$$

where $r \in D$ 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 8 (Event Equivalence)**.** *Two events $\epsilon_1, \epsilon_2 \in \mathcal{E}$ are equivalent, denoted as $\epsilon_1 = \epsilon_2$ if and only if:*

$$\pi_1(\epsilon_1) = \pi_1(\epsilon_2) \wedge \pi_2(\epsilon_1) = \pi_2(\epsilon_2) \wedge \pi_3(\epsilon_1) = \pi_3(\epsilon_2) \wedge \pi_4(\epsilon_1) =_q \pi_4(\epsilon_2)$$

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

Definition 9 (Events Different up to Value (`Diff`))**.** *Two events $\epsilon_1, \epsilon_2 \in \mathcal{E} are$* Different up to Value*, denoted as `Diff`(ϵ_1, ϵ_2) if and only if:*

$$\begin{array}{l} \pi_1(\epsilon_1) = \pi_1(\epsilon_2) \wedge \pi_2(\epsilon_1) = \pi_2(\epsilon_2) \\ \wedge \big((\pi_3(\epsilon_1) \neq \pi_3(\epsilon_2) \wedge \pi_4(\epsilon_1) = \pi_4(\epsilon_2) = \bullet) \vee (\pi_4(\epsilon_1) \neq \bullet \wedge \pi_4(\epsilon_2) \neq \bullet \wedge \pi_4(\epsilon_1) \neq_q \pi_4(\epsilon_2)) \big) \end{array}$$

2.2 Trace

Definition 10 (Trace Concatenation, $+\!+ : \mathcal{T} \to \mathcal{T} \to \mathcal{T}$)**.** *Given two traces $\tau_1, \tau_2 \in \mathcal{T}$, the trace concatenation operator $+\!+$ is defined as:*

$$\tau_1 +\!+ \tau_2 \triangleq \begin{cases} \tau_1 & \tau_2 = [] \\ (\tau_1 +\!+ \tau_2') :: \epsilon & \tau_2 = \tau_2' :: \epsilon \end{cases}$$

Definition 11. *(An Event Belongs to A Trace) An event $\epsilon \in \mathcal{E}$ belongs to a trace τ, i.e., $\epsilon \in \tau$ are defined as follows:*

$$\epsilon \in \tau \triangleq \begin{cases} \texttt{true} & \tau = \tau' :: \epsilon' \wedge \epsilon = \epsilon' \\ \epsilon \in \tau' & \tau = \tau' :: \epsilon' \wedge \epsilon \neq \epsilon' \\ \texttt{false} & \tau = [] \end{cases} \tag{1}$$

As usual, we denote by $\epsilon \notin \tau$ that the event ϵ doesn't belong to the trace τ.

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

$$\begin{array}{lll} \texttt{cnt}(\tau :: (x, l, v, \bullet), l) \triangleq \texttt{cnt}(\tau, l) + 1 & \texttt{cnt}(\tau :: (b, l, v, \bullet), l) \triangleq \texttt{cnt}(\tau, l) + 1 & \texttt{cnt}(\tau :: (x, l, v, \alpha), l) \triangleq \texttt{cnt}(\tau, l) + 1 \\ \texttt{cnt}(\tau :: (x, l', v, \bullet), l) \triangleq \texttt{cnt}(\tau, l), l' \neq l & \texttt{cnt}(\tau :: (b, l', v, \bullet), l) \triangleq \texttt{cnt}(\tau, l), l' \neq l & \texttt{cnt}(\tau :: (x, l', v, \alpha), l) \triangleq \texttt{cnt}(\tau, l), l' \neq \\ \texttt{cnt}([], l) \triangleq 0 & & \end{array}$$

We introduce an operator $\iota : \mathcal{T} \to \mathcal{VAR} \to \mathcal{L} \cup \{\bot\}$, 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,

$$\iota(\tau :: (x, l, \_, \_))x \triangleq l \quad \iota(\tau :: (y, l, \_, \_))x \triangleq \iota(\tau)x, y \neq x \quad \iota(\tau :: (b, l, v, \bullet))x \triangleq \iota(\tau)x \quad \iota([])x \triangleq \bot$$

The operator $\mathbb{TL} : \mathcal{T} \to \mathcal{P}(\mathcal{L})$ gives the set of labels in every event belonging to a trace, whoes behavior is defined as follows,

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

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, 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 $\langle c, \tau \rangle \to^* \langle \texttt{skip}, \tau' \rangle$, then there exists a trace $\tau'' \in \mathcal{T}$ with $\tau ++ \tau'' = \tau'$*

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

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 7. So we have the following Corollary 2.0.1 with proof in Appendix A.

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 ϵ and ϵ' 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 Adaptivity

3.1 Dependency

To define the may dependency relation on two labeled variables, we rely on the limited information at hand - the trace generated by the operational semantics. In this end, we first define the *May-Dependency* between events, and use it as a foundation of the variable may-dependency relation.

We compare two events by defining the $\mathtt{Diff}(\epsilon_1,\epsilon_2)$, we use $\psi_1 =_q \psi_2$ and $\psi_1 \neq_q \psi_2$ to notate query expression equivalence and inquivalence.

Definition 12 (Events Different up to Value ($\mathtt{Diff}$))**.** *Two events* $\epsilon_1,\epsilon_2 \in \mathcal{E}$ *are* Different up to Value*, denoted as* $\mathtt{Diff}(\epsilon_1,\epsilon_2)$ *if and only if:*

$$\begin{array}{l}\pi_1(\epsilon_1) = \pi_1(\epsilon_2) \land \pi_2(\epsilon_1) = \pi_2(\epsilon_2) \\ \land\big((\pi_3(\epsilon_1) \neq \pi_3(\epsilon_2) \land \pi_4(\epsilon_1) = \pi_4(\epsilon_2) = \bullet) \lor (\pi_4(\epsilon_1) \neq \bullet \land \pi_4(\epsilon_2) \neq \bullet \land \pi_4(\epsilon_1) \neq_q \pi_4(\epsilon_2))\big)\end{array}$$

For a program, its labeled variables and assigned variables are sub set of the labeled variables $\mathcal{LV}$. We use $\mathbb{AV}(c) \in \mathcal{P}(\mathcal{VAR} \times \mathbb{N}) \subset \mathcal{LV}$ and $\mathbb{LV}(c) \in \mathcal{P}(\mathcal{VAR} \times \mathcal{L}) \subseteq \mathcal{LV}$ for them. $FV : e \to \mathcal{P}(\mathcal{VAR})$, computes the set of free variables in an expression. We also define the set of query variables for a program c, $\mathbb{QV} : \mathcal{C} \to \mathcal{P}(\mathcal{LV})$. It is easy to see that a program c's query variables is a subset of its labeled variables, $\mathbb{QV}(c) \subseteq \mathbb{LV}(c)$. We have the operator $\mathbb{TL} : \mathcal{T} \to \mathcal{L}$, which gives the set of labels in every event belonging to a trace. Then we introduce a counting operator $\mathtt{cnt} : \mathcal{T} \to \mathbb{N} \to \mathbb{N}$, which counts the occurrence of of a labeled variable in the trace, whose behavior is defined as follows,

$$\mathtt{cnt}(\tau :: (\_, l, \_, \_), l) \triangleq \mathtt{cnt}(\tau, l) + 1 \quad \mathtt{cnt}(\tau :: (\_, l', \_, \_), l) \triangleq \mathtt{cnt}(\tau, l), l' \neq l \quad \mathtt{cnt}([], l) \triangleq 0$$

The full definitions of these above operators can be found in the appendix.

Definition 13 (Value Sequence $\mathtt{seq}(\tau, x^l)$)**.**

$$\begin{array}{lll}\mathtt{seq}(\tau :: (x, l, v, \bullet), x^l) \triangleq \mathtt{seq}(\tau) :: v & \mathtt{seq}(\tau :: (x, l, v, \alpha), x^l) \triangleq \mathtt{seq}(\tau) :: \alpha & \mathtt{seq}([]) \triangleq [] \\ \mathtt{seq}(\tau :: (y, j, \_, \_), x^l) \triangleq \mathtt{seq}(\tau) \quad y \neq x \lor j \neq l & & \end{array}$$

Definition 14 (Difference Sequence $\mathtt{Diff}_{\mathtt{seq}}(\tau_1, \tau_2, x^l)$)**.** *Let* $s_1 = \mathtt{seq}(\tau_1, x^l) \land s_2 = \mathtt{seq}(\tau_2, x^l)$ *be the value sequence of* x^l *on* τ_1 *and* τ_2*, and* s_{max} *be the sequence with longer length and* s_{min} *the shorter one, then their difference sequence is defined as follows,*

$$\mathtt{Diff}_{\mathtt{seq}}(\tau_1, \tau_2, x^l) \triangleq \begin{array}{l}\{(s_{min}[k], s_{max}[k]) \mid s_{min}[k] \neq s_{max}[k], k = 0, \ldots, len(s_{min})\} \\ \cup\{(\cdot, s_{max}[k]) \mid \mathtt{len}(s_{min}) \leq \mathtt{len}(s_{max}) k = len(s_{min}), \ldots \mathtt{len}(s_{max})\}\end{array}$$

Definition 15 (Event May-Dependency)**.** .
An event ϵ_2 *is in the* event may-dependency *relation with an assignment event* $\epsilon_1 \in \mathcal{E}^{\mathtt{asn}}$ *in a program* c *with a hidden database* D *and a trace* $\tau \in \mathcal{T}$ *denoted as* $\mathsf{DEP}_{\mathsf{e}}(\epsilon_1, \epsilon_2, [\epsilon_1]{+\!+}\tau{+\!+}[\epsilon_2], c, D)$*, iff*

$$\begin{array}{l}\exists \tau_0, \tau_1, \tau' \in \mathcal{T}, \epsilon_1' \in \mathcal{E}^{\mathtt{asn}}, c_1, c_2 \in \mathcal{C} \,.\, \mathtt{Diff}(\epsilon_1, \epsilon_1') \land \\ \quad (\exists \epsilon_2' \in \mathcal{E} \,.\, \left(\begin{array}{ll} & \langle c, \tau_0 \rangle \to^* \langle c_1, \tau_1{+\!+}[\epsilon_1] \rangle \to^* \langle c_2, \tau_1{+\!+}[\epsilon_1]{+\!+}\tau{+\!+}[\epsilon_2] \rangle \\ \bigwedge & \langle c_1, \tau_1{+\!+}[\epsilon_1'] \rangle \to^* \langle c_2, \tau_1{+\!+}[\epsilon_1']{+\!+}\tau'{+\!+}[\epsilon_2'] \rangle \\ \bigwedge & \mathtt{Diff}(\epsilon_2, \epsilon_2') \land \mathtt{cnt}(\tau, \pi_2(\epsilon_2)) = \mathtt{cnt}(\tau', \pi_2(\epsilon_2')) \end{array}\right) \\ \quad \lor \exists \tau_3, \tau_3' \in \mathcal{T}, \epsilon_b \in \mathcal{E}^{\mathtt{test}} \,. \\ \quad \left(\begin{array}{ll} & \langle c, \tau_0 \rangle \to^* \langle c_1, \tau_1{+\!+}[\epsilon_1] \rangle \to^* \langle c_2, \tau_1{+\!+}[\epsilon_1]{+\!+}\tau{+\!+}[\epsilon_b]{+\!+}\tau_3 \rangle \\ \bigwedge & \langle c_1, \tau_1{+\!+}[\epsilon_1'] \rangle \to^* \langle c_2, \tau_1{+\!+}[\epsilon_1']{+\!+}\tau'{+\!+}[(\neg \epsilon_b)]{+\!+}\tau_3' \rangle \\ \bigwedge & \mathbb{TL}_{\tau_3} \cap \mathbb{TL}_{\tau_3'} = \emptyset \land \mathtt{cnt}(\tau', \pi_2(\epsilon_b)) = \mathtt{cnt}(\tau, \pi_2(\epsilon_b)) \land \epsilon_2 \in \tau_3 \land \epsilon_2 \not\in \tau_3' \end{array}\right))\end{array}$$

Our event *May-Dependency* relation of two events $\epsilon_1 \in \mathcal{E}^{\mathtt{asn}}$ and $\epsilon_2 \in \mathcal{E}$, for a program c and hidden data base D is w.r.t to a trace $[\epsilon_1]_{++}\tau_{++}[\epsilon_2]$. $\epsilon_1 \in \mathcal{E}^{\mathtt{asn}}$ is an assignment event, because only a change on the assignment event will affect the execution trace, according to our operational semantics. In order to observe the changes of ϵ_2 under the modification of ϵ_1, this trace $[\epsilon_1]_{++}\tau_{++}[\epsilon_2]$ starts with ϵ_1 and ending with ϵ_2. The *May-Dependency* relation considers both the value dependency and value control dependency as discussed above. The relation can be divided into two parts naturally in Definition 15(line $2-4$, $5-8$ respectively, we think it start from line 1). The idea of the event ϵ_1 may depend on ϵ_2 can be briefly described: We have one execution of the program as reference (See line 2 and 6 , for the two kinds of dependency). When the value assigned to the first variable in ϵ_1 is modified, the reference trace $\tau_{1++}[\epsilon_1]$ is modified correspondingly as $\tau_{1++}[\epsilon_1']$. We use $\mathtt{Diff}(\epsilon_1,\epsilon_1')$ at line 1 to express this modification, which guarantees that ϵ_1 and ϵ_1' only differ in their assigned value and are equal on variable name and label. We perform a second run of the program by continuing the execution of the same program from the same execution point, but with the modified trace $\tau_{1++}[\epsilon_1']$ (See line 3, 7). The expected may dependency will be caught by observing two different possible changes (See line 4, 8 respectively) when comparing the second execution with the reference one.

In the first part, (line $2-4$ of Definition 15) we witness the appearance of ϵ_2' in the second execution, and a variation between ϵ_2 and ϵ_2' on their value. We have special requirement $\mathtt{Diff}(\epsilon_2,\epsilon_2')$, , which guarantees that they have the same variable name and label but only differ in their evaluated values. In particularly for query, if ϵ_2 and ϵ_2' are generated from query requesting, then $\mathtt{Diff}(\epsilon_2,\epsilon_2)$ guarantees that they differ in their query value rather than the query requesting result. Additionally, in order to handle the multiple occurrence of the same event through iterations of the while loop, where ϵ_2 and ϵ_2' could be in different while loop, we restrict the occurrence times of ϵ_2's label in the first(reference) trace equals to the occurrence times of ϵ_2''s label in the second trace, through $\mathtt{cnt}(\tau,\pi_2(\epsilon_2)) = \mathtt{cnt}(\tau',\pi_2(\epsilon_2'))$.

In the second part (line $5-8$ of Definition 15) , we witness the disappearance of ϵ_2 through observing the change of a testing event ϵ_b. In order to change the appearance of 5yhan event, the command that generates ϵ_2 must not be executed in the second execution. The only way to control whether a command will be executed, is through the change of a guard's evaluation result in the if or while command. So we first observe the testing event ϵ_b changes into $\neg\epsilon_b$ in the second execution, following with the disappearance of ϵ_2 in the second trace.
In the same way, we restrict the occurrence times of ϵ_b's label in the two traces being equal through $\mathtt{cnt}(\tau',\pi_2(\epsilon_b)) = \mathtt{cnt}(\tau,\pi_2(\epsilon_b))$ to handle the while loop. Again, in particularly for query, we observe the disappearance based on the query value equivalence. Considering a program's all possible executions(with respect to initialized user input), among all events generated during these executions and the variables and labels of these events are corresponding to the two labeled variables, as long as there is one pair of events satisfying the *Event May-Dependency* relation in Definition 15, then we say the two variables satisfy *Variable May-Dependency* relation in Definition 16.

Definition 16 (Variable May-Dependency)**.** .
A variable $x_2^{l_2} \in \mathbb{LV}(c)$ *is in the* variable may-dependency *relation with another variable* $x_1^{l_1} \in \mathbb{LV}(c)$ *in a program c, denoted as* $\mathsf{DEP}_{\mathsf{var}}(x_1^{l_1}, x_2^{l_2}, c)$*, if an only if.*

$$\exists \epsilon_1, \epsilon_2 \in \mathcal{E}^{\mathtt{asn}}, \tau \in \mathcal{T}, D \in \mathcal{DB} \;.\; \pi_1(\epsilon_1)^{\pi_2(\epsilon_1)} = x_1^{l_1} \wedge \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)} = x_2^{l_2} \wedge \mathsf{DEP}_{\mathsf{e}}(\epsilon_1, \epsilon_2, \tau, c, D)$$

A variable $y^j \in \mathbb{LV}(c)$ *is in the* may-dependency *relation with another variable* $x^i \in \mathbb{LV}(c)$ *in a program c, w.r.t. an initial trace* $\tau_0 \in \mathcal{T}(c)$ *and two witness traces* $\tau_1, \tau_2 \in \mathcal{T}$*, denoted as* $\mathsf{DEP}(x^i, y^j, \tau_1, \tau_2, \tau_0, c)$*, if an only if*

$$\exists D \in \mathcal{DB}, \tau_0' \in \mathcal{T} \;.\; (\forall z^l \neq x^i \;.\; \rho(\tau_0, z^l) = \rho(\tau_0', z^l)) \wedge \langle c, \tau_0 \rangle \to^* \langle [\mathtt{skip}]^l, \tau_{0++}\tau_1 \rangle \wedge \langle c, \tau_0' \rangle \to^* \langle [\mathtt{skip}]^l, \tau'_{0++}\tau_2 \rangle$$
$$\wedge \mathtt{Diff}_{\mathtt{seq}}(\tau_1, \tau_2, y^j) \neq \emptyset$$

We denote $\mathcal{T}_0(c)$ as the set of initial traces in which all the input variables in c are initialized.

3.2 Execution Based Dependency Graph

The variable *May-Dependency* relation gives us the edges, we define the execution based dependency graph.

Definition 17 (Execution Based Dependency Graph)**.** *Given a program c, its* Execution-Base Dependency Graph $\mathtt{G_{trace}}(c) = (\mathtt{V_{trace}}(c), \mathtt{E_{trace}}(c), \mathtt{W_{trace}}(c), \mathtt{Q_{trace}}(c))$ *is defined as follows,*

$$\begin{array}{rcl}
\mathtt{V_{trace}}(c) & := & \{(x^l, w) \mid w : \mathcal{T} \to \mathbb{N} \land x^l \in \mathbb{LV}(c) \\
& & \land \forall \tau \in \mathcal{T}_0(c), \tau' \in \mathcal{T} \,.\, \langle c, \tau \rangle \to^* \langle \mathtt{skip}, \tau {+\!+} \tau' \rangle \implies w(\tau) = \mathtt{cnt}(\tau', l)\} \\
\mathtt{E_{trace}}(c) & := & \{(x^i, w, y^j) \mid x^i, y^j \in \mathbb{LV}(c) \land w \in \mathcal{P}(\mathcal{T} \to \mathbb{N}) \land \exists \tau \in \mathcal{T}_0(c), \tau_1, \tau_2 \in \mathcal{T} \,.\, \mathsf{DEP}(x^i, y^j, \tau_1, \tau_2, \tau_0, c) \\
& & \land \forall \tau_0 \in \mathcal{T}_0(c) \,.\, w(\tau_0) = \max\{|\mathtt{Diff_{seq}}(\tau_1, \tau_2, y)| \forall \tau_1, \tau_2 \in \mathcal{T} \,.\, \mathsf{DEP}(x^i, y^j, \tau_1, \tau_2, \tau_0, c)\}\}
\end{array}$$

There are two components of the execution-based dependency graph.
The vertices $\mathtt{V_{trace}}(c)$ is a set of pairs, $(x^l, w) \in \mathcal{LV} \times (\mathcal{T} \to \mathbb{N})$, with a labeled variable as first component and its weight w the second component. Weight w for x^l is a function $w : \mathcal{T} \to \mathbb{N}$ mapping from a starting trace to a natural number. When program executes under this starting trace τ, $\langle c, \tau \rangle \to^* \langle \mathtt{skip}, \tau {+\!+} \tau' \rangle$, it generates an execution trace τ'. This natural number is the evaluation times of the labeled command corresponding to the vertex, computed by the counter operator $w(\tau) = \mathtt{cnt}(\tau', l)$. We can see in the execution-based dependency graph of `twoRounds` in Figure 3(b) in main paper, the weight of vertices in the while loop is $\rho(\tau)k$, which depends on the value of the user input k specified in the starting trace τ.
The directed edges $\mathtt{E_{trace}}(c)$ is a set of triples $(x^i, w, y^j) \in \mathcal{LV} \times (\mathcal{T}_0 \to \mathbb{N}) \times \mathcal{LV}$, with two labeled variables (from x^i pointing to y^j) and a weight w for this edge. The edges are constructed directly from our variable may-dependency relation. For any two vertices x^i and y^j in $\mathtt{V_{trace}}(c)$, if there exists two witness traces τ_1, τ_2 and an initial trace $\tau_0 \in \mathcal{T}_0$ such that, they satisfy the variable may-dependency relation $\mathsf{DEP}(x^i, y^j, \tau_1, \tau_2, \tau_0, c)$, there is a direct edge. The weight of the edge is a function $w : \mathcal{T}_0 \to \mathbb{N}$, where given an initial trace τ_0, it of the edge is the maximum length of their difference sequence between all pairs of the witness traces τ_1, τ_2 of their dependency relation. In most data analysis programs c we are interested, there are usually some user input variables, such as k in `twoRounds`. We denote $\mathcal{T}_0(c)$ as the set of initial traces in which all the input variables in c are initialized, it is also reflected in $\mathtt{W_{trace}}(c)$.

3.3 Trace-based Adaptivity

Given a program c's execution-based dependency graph $\mathtt{G_{trace}}(c)$, 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 18 (Finite Walk (k))**.** .
Given the execution-based dependency graph $\mathtt{G_{trace}}(c) = (\mathtt{V_{trace}}(c), \mathtt{E_{trace}}(c), \mathtt{W_{trace}}(c), \mathtt{Q_{trace}}(c))$ *of a program c, a* finite walk k *in* $\mathtt{G_{trace}}(c)$ *is a function* $k : \mathcal{T} \to$ *sequence of edges. For a initial trace* $\tau_0 \in \mathcal{T}_0(c)$, $k(\tau_0)$ *is a sequence of edges* $(e_1 \dots e_{n-1})$ *for which there is a sequence of vertices* $(v_1, \dots, v_n)$ *such that:*

- *$e_i = (v_i, w_i, v_{i+1}) \in \mathtt{E}_{\mathtt{trace}}(c)$ for every $1 \leq i < n$, and e_i appears in $(e_1 \ldots e_{n-1})$ at most $w_i(\tau_0)$ times.*

- *every $(v_i, w_i) \in \mathtt{V}_{\mathtt{trace}}(c)$ and v_i appears in $(v_1, \ldots, v_n)$ at most $w_i(\tau_0)$ times.*

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

We use $\mathcal{WK}(\mathtt{G}_{\mathtt{trace}}(c))$ to denote the set containing all finite walks k in $\mathtt{G}_{\mathtt{trace}}(c)$; and $k_{v_1 \to v_2} \in \mathcal{WK}(\mathtt{G}_{\mathtt{trace}}(c))$ with $v_1, v_2 \in \mathtt{V}_{\mathtt{trace}}(c)$ denotes the walk from vertex v_1 to v_2 .
We are interested in queries, so we need to recover the variables corresponding to queries from the walk. We define the query length of a walk, instead of counting all the vertices in k's vertices sequence, we just count the number of vertices which correspond to query variables in this sequence.

Definition 19 (Query Length of the Finite Walk($\mathtt{len}^{\mathtt{q}}$))**.** .
Given the execution-based dependency graph $\mathtt{G}_{\mathtt{trace}}(c) = (\mathtt{V}_{\mathtt{trace}}(c), \mathtt{E}_{\mathtt{trace}}(c))$ of a program c, and a finite walk *$k \in \mathcal{WK}(\mathtt{G}_{\mathtt{trace}}(c))$. The query length of k is a function $\mathtt{len}^{\mathtt{q}}(k) : \mathcal{T} \to \mathbb{N}$, such that with an initial trace $\tau_0 \in \mathcal{T}_0(c)$, $\mathtt{len}^{\mathtt{q}}(k)(\tau_0)$ is the number of vertices which correspond to query variables in the vertices sequence of the walk $k(\tau_0)$ $(v_1, \ldots, v_n)$ as follows,*

$$\mathtt{len}^{\mathtt{q}}(k)(\tau_0) = |\big(v \mid v \in (v_1, \ldots, v_n) \wedge v \in \mathbb{QV}(c)\big)|.$$

3.4 Example From Limitation

Example 3.1 (Accurate Adapativity for Multiple Rounds Single Example)**.** *The program's adaptivity in our formal model, in Definition 20 also comes across an over-approximation on the program's intuitive adaptivity rounds. It is resulted from 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 witness the* variable may-dependency *relation.*

As the program in Figure 2(a), which is a variant of the multiple rounds strategy, named `multipleRoundSingle(k)` *with input k. In this algorithm, at line 7 of every iteration, a query $\mathtt{query}(\chi[y] + p)$ based on previous query results stored in p and y is asked by the analyst like in the multiple rounds strategy. The difference is that only the query answers from the one single iterations ($j = k-2$) are used in this query $\mathtt{query}(\chi[y] + p)$. Because the execution trace updates p using the constant 0 for all the iterations where ($j \neq k-2$) at line 10 after the query request at line 7. In this way, all the query answers stored in p will not be accessed in next query request at line 7 in the iterations where ($j \neq k-2$). Only query answer at one single iteration where ($j = k-2$) will be used in next query request $\mathtt{query}(\chi[y] + p)$ at line 7. So the adaptivity for this example is 2. However, our adaptivity model fails to realize that there is only dependency relation between p^7 and p^7 in one single iteration, not the others. As shown in the execution-based dependency graph in Figure 2(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 $w_k(\tau_0)$ with a initial trace τ_0. As a result, the walk with the longest query length is $p^7 \to \cdots \to p^7 \to y^4 \to z^1$ with the vertex p^7 visited $w_k(\tau_0)$, as the dotted arrows. The adaptivity based on this walk is 2. The* AdaptFun *is able to give us 2, as an accurate bound w.r.t this definition.*

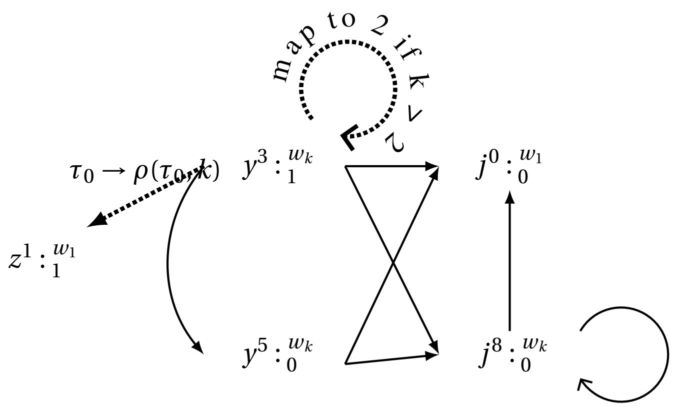
3.5 Trace-based Adaptivity

Definition 20 (Adaptivity of a Program)**.** .
Given a program c, its adaptivity $A(c)$ is function $A(c) : \mathcal{T} \to \mathbb{N}$ such that for an initial trace $\tau_0 \in \mathcal{T}_0(c)$,

$$A(c)(\tau_0) = \max\big\{\mathtt{len}^{\mathtt{q}}(k)(\tau_0) \mid k \in \mathcal{WK}(\mathtt{G}_{\mathtt{trace}}(c))\big\}$$

```
multipleRoundsSingle(k)
$[j \leftarrow k]^0; [z \leftarrow \texttt{query}(0)]^1;$
while $[j > 0]^2$ do
$\Big([y \leftarrow \texttt{query}(\chi[z] + y)]^3;$
if $([j \neq 2]^4, [y \leftarrow 0]^5, [\texttt{skip}]^6)$
$[j \leftarrow j - 1]^7\Big);$
```

(a)



(b)

Figure 2: (a) The multi rounds single example (b) The execution-based dependency graph.

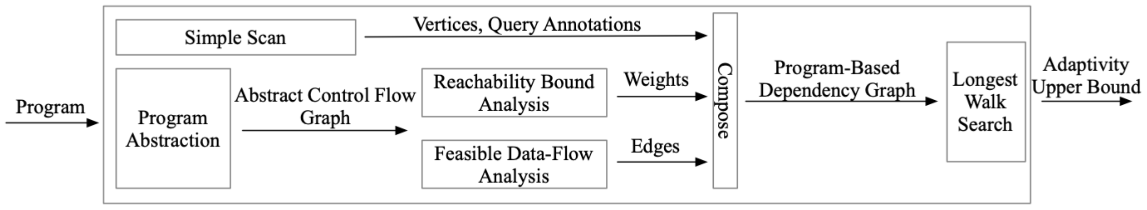

Program
Simple Scan
Vertices, Query Annotations
Program Abstraction
Abstract Control Flow Graph
Reachability Bound Analysis
Feasible Data-Flow Analysis
Weights
Edges
Compose
Program-Based Dependency Graph
Longest Walk Search
Adaptivity Upper Bound

Figure 3: The overview of AdaptFun

4 AdaptFun

In this section, we present our static program analysis for computing an upper bound on the adaptivity of an arbitrary program c, as we define in last section.

4.1 A guide to the static program analysis framework

In order to have the upper bound of the adaptivity of a program c, we design a program analysis framework AdaptFun. It can be divided as two steps: 1) to construct a weighted depdenency graph based on c. 2) to find a path in this graph, which is used to estimate an upper bound on the adaptivity of c.

4.1.1 Graph Estimation

According to the dependency graph we use in adaptivity definition, we want to build a similar graph to over-approximate the Execution-Based Dependency Graph (in Definition 17). The construction considers the vertices, edges, and the weight of every node, as well as some annotations which marks the query usage. The overall picture of this step is organized as follows.

1. Vertices are the assigned variables with unique labels, which is extracted directly from the program, see Section 4.2 without extra static analysis technique
2. The edge between vertices considers both control flow and data flow, See Section 4.3.2
3. Every vertex and edge come with a weight, which tells the maximal times each vertex and edge can be visited in realistic execution. This weight is estimated by a reachability bound analysis on each vertex, See Section 4.3.3.
4. Finally, with all the ingredients ready, we construct the final approximated program-based dependency graph in Section 4.4

Overall, this program-based graph has a similar topology structure as the Execution-Based Dependency Graph. It has the same vertices and query annotations, but approximated edges and weights. We call the graph generated by static analysis techniques, static analysis dedendency graph.

4.1.2 Adaptivity Computation

Likewise the adaptivity is defined as a finite walk in the execution based depdenency graph, our static estimation on this adaptivity also relies on finding a path in the static analysis depdenency graph.

The construction of the stastic analysis dependency graph is of great value of showing some useful properties of the target program, such as dependency between variables, the execution upper bound of a certian command, while the key novelty is our path searching algorithm, which connects all the information we need in the static anlaysis dependency graph and provides us a sound over-estimation of adaptivity! In order to get a sound but precise upper bound, we will discuss some challenges in finding the 'appropriate' path in the graph, and how our algorithm responds to these challengs. We present the path seaching algorithm in Section 4.5.

4.2 Vertices Estimationn

The first component of every vertex in the static analysis dependency graph are actually identical as the Execution-Based Dependency Graph, which are assigned variables in the program annotated with the unique label(line number). These vertices are collected by statically scanning the program, like what we do for vertices of its Execution-Based Dependency Graph. The vertices are defined formally as follows.

$$\mathtt{V_{prog}}^0(c) \triangleq \left\{(x^l, w) \in \mathcal{LV} \times \mathcal{A}_{\mathtt{in}} \;\middle|\; x^l \in \mathbb{LV}(c)\right\}$$

where $\mathcal{A}_{\mathtt{in}}$ is the set of arithmetic expressions over $\mathbb{N}$ and program's input variables. The weight w for every vertex will be computed in following step in Section 4.3.3.

4.3 Edge and Weight Estimation

Since the edges of the execution-based graph of a program relies on the dependency relation, which handles both control flow and data flow, as an over-approximation of this graph, the edges of our static anlaysis dependency graph also covers these two kind of flows. We develop a feasible data flow relation to catch these two flows, in Section 4.3.2.

The weight of every vertice in the execution-based graph is built on all possible execution traces. In order to over-approximate the weight statically but still tightly, we present a symbolic reachability bound analysis for estimation of the weight of each vertice(label) in Section 4.3.3, in spirit of some reachablility bound techiniques.

The edges and weight estimation are both performed on basis of an abstract control flow graph of the program, we first show how to generate this abstract execution control flow graph before the introduction of the edge and weight estimation.

4.3.1 Abstract Execution Control Flow graph

We discuss the vertices and edge of the abstract control flow graph for a program c, $\mathtt{absG}(c)$.

Every vertex corresponds to the unique label. Specifically, the vertices of this graph is the set of c's labels with an exit label l_{ex},

$$\mathtt{absV}(c) = labels(c) \cup \{l_{ex}\}$$

The edge in the abstract control flow graph comes from the abstract execution trace of the program. The abstract execution trace, an abstract representation of the execution, consists of a list of abstract transitions. Then, every abstract transition in the abstraction execution trace corresponds to an edge in the abstract control flow graph. In another word, the edge (l_1, dc, l_2) in the abstract control flow graph, represents an abstract transition from l_1 to l_2, with a set of difference constraints dc. Also notice, the difference constraints generated during the abstract transition appears in the edge as annotation.

Overall, the vertices can be easily collected and the key point of construction of the abstract execution control flow graph for a program is the abstract execution trace, which relies on the abstraction of expression and abstract transition (we also call it abstract event), we will discuss in the following section. To make it easy to understand, abstract control flow graph is a control flow graph, with difference constraints on every edge.

Expression Abstraction The expression assigned to the variable on the left hand of the assignment command is abstracted to an abstract value: (adopted from the expression abstraction method in paper [2]). The abstract value is expressed in the form of Difference constraint, denotated as DC : $\mathcal{VAR} \cup \mathcal{SMBCST} \to \mathcal{VAR} \times (\mathcal{VAR} \cup \mathcal{SMBCST}) \times (\mathbb{Z} \cup \{\infty\})$. $\mathcal{SMBCST}$ is called the Symbolic Constant defined as $\mathcal{SMBCST} \triangleq \mathbb{N} \cup \mathcal{VAR}_{\texttt{in}} \cup \{\max(\mathcal{DB})\}$, which consists of natural numbers $\mathbb{N}$, the program's input variables $\mathcal{VAR}_{\texttt{in}}$ and a constant value Q_m for estimating the upper bound of variables which are assigned by queries.

Give an instance of difference constraint used here, $DC(\mathcal{VAR} \cup \mathcal{SMBCST}) \cup \{\top\}$ represents all the difference constraints over variable and symbolic constants. It is a set of the inequality of form $x \leq y + v$ where $x \in \mathcal{VAR}$, $y \in \mathcal{VAR} \cup \mathcal{SMBCST}$ and $v \in \mathbb{Z}$. This difference constraint is defined in the same way as [2]. For concise, we use $\mathcal{DC}^{\top}$ to represent the $DC(\mathcal{VAR} \cup \mathcal{SMBCST}) \cup \{\top\}$.

We show the expression abstraction $\texttt{absexpr} : e \to \mathcal{VAR} \to DC(\mathcal{VAR} \cup \mathcal{SMBCST}) \cup \{\top\}$ below.

$$
\begin{array}{ll}
\texttt{absexpr}(x - v, x) = x' \leq x - v & x \in \mathcal{VAR}_{\texttt{guard}} \wedge v \in \mathbb{N} \\
\texttt{absexpr}(y + v, x) = x' \leq y + v & x \in \mathcal{VAR}_{\texttt{guard}} \wedge v \in \mathbb{Z} \wedge y \in (\mathcal{VAR}_{\texttt{guard}} \cup \mathcal{SMBCST}) \\
\texttt{absexpr}(v, x) = x' \leq v + 0 & x \in \mathcal{VAR}_{\texttt{guard}} \wedge v \in (\mathcal{VAR}_{\texttt{guard}} \cup \mathcal{SMBCST}) \\
\texttt{absexpr}(y + v, x) = x' \leq y + v & \\
\mathcal{VAR}_{\texttt{guard}} = \mathcal{VAR}_{\texttt{guard}} \cup \{y\} & x \in \mathcal{VAR}_{\texttt{guard}} \wedge v \in \mathbb{Z} \wedge y \notin (\mathcal{VAR}_{\texttt{guard}} \cup \mathcal{SMBCST}) \\
\texttt{absexpr}(\psi, x) = x' \leq 0 + Q_m & x \in \mathcal{VAR}_{\texttt{guard}} \wedge \psi \text{ is a query expression} \\
\texttt{absexpr}(b, x) = x' \leq 0 + 1 & x \in \mathcal{VAR}_{\texttt{guard}} \wedge b \text{ is a boolean expression} \\
\texttt{absexpr}(e, x) = x' \leq \infty & x \in \mathcal{VAR}_{\texttt{guard}} \wedge e \text{ doesn't have any of the forms as above} \\
\texttt{absexpr}(e, x) = \top & x \notin \mathcal{VAR}_{\texttt{guard}}
\end{array}
$$

$\mathcal{VAR}_{\texttt{guard}}$ 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 $\mathcal{VAR}_{\texttt{guard}}$ is updated by a variable y, which isn't in this set, we add y into the set $\mathcal{VAR}_{\texttt{guard}}$ and repeat above procedure until $\mathcal{VAR}_{\texttt{guard}}$ and $\texttt{absexpr}(e, x)$ is stabilized.

Specifically we handle a normalized guard expression ($x > 0$ for $x^l \in \mathbb{LV}_c$) in `while` , and the counter variables only increase, decrease or reset by simple arithmetic expression (mainly multiplication, division, minus and plus (able to extend to max and min)). This is the same as in paper [2].

For more complex expression assignments, where the counter reset, or calculated from `log` , multiplication or division, and expressions involving multiple variables, the constraint is approximated as reset of ∞.

This approximation strategy doesn't affect our analysis results in our examples. It is easy to extend the normalized expression into more complex forms as in [2], as well as the counter variable manipulation with more advanced expressions.

Program Event Abstraction We show the abstract event definition, which is generated when computing its abstract execution trace.

Definition 21 (Abstract Event)**.** *Abstract Event:* $\overset{\alpha}{\epsilon} \in \mathcal{L} \times \mathcal{DC}^{\top} \times \mathcal{L}$ *is a triple where the first and third components are labels, second component is a constraint from* $\mathcal{DC}^{\top}$.

Specifically, in an abstract event, the first label correspond to an initial state, and the second label and the constraint correspond to an abstract final state. The abstract initial state is a label from $\mathcal{L}$. The abstract final state is a pair from $\mathcal{L} \times \mathcal{DC}^{\top}$, where first component is a label from $\mathcal{L}$ and the second component is a constraint from $\mathcal{DC}^{\top}$.

Given a program c, its abstract initial state, and the set of its abstract final state is computed as follows,

$$\begin{array}{ll}
\texttt{absinit}([x \leftarrow e]^{l}) & = l \\
\texttt{absinit}([x \leftarrow e]^{l}) & = l \\
\texttt{absinit}(\left[\texttt{skip}\right]^{l}) & = l \\
\texttt{absinit}(\,\texttt{if}\ [b]^{l}\ \texttt{then}\ c_1\ \texttt{else}\ c_2) & = l \\
\texttt{absinit}(\,\texttt{while}\ [b]^{l}\ \texttt{do}\ c) & = l \\
\texttt{absinit}(c_1 ; c_2) & = \texttt{absinit}(c_1)
\end{array}$$

Final State Abstraction: $\texttt{absfinal} : \mathcal{C} \to \mathcal{P}(\mathcal{L} \times \mathcal{DC}^{\top})$, computes the set of Abstract Final State for the command.

$$\begin{array}{ll}
\texttt{absfinal}([x \leftarrow e]^{l}) & = \{(l, \texttt{absexpr}\,(e, x))\} \\
\texttt{absfinal}(\left[x \leftarrow \texttt{query}(\psi)\right]^{l}) & = \{(l, x' \leq 0 + Q_m)\} \\
\texttt{absfinal}(\left[\texttt{skip}\right]^{l}) & = \{(l, \top)\} \\
\texttt{absfinal}(\,\texttt{if}\ [b]^{l}\ \texttt{then}\ c_1\ \texttt{else}\ c_2) & = \texttt{absfinal}(c_1) \cup \texttt{absfinal}(c_2) \\
\texttt{absfinal}(\,\texttt{while}\ [b]^{l}\ \texttt{do}\ c) & = \{(l, \top)\} \\
\texttt{absfinal}(c_1 ; c_2) & = \texttt{absfinal}(c_2)
\end{array}$$

Abstract Execution Trace Now, we extract the abstract execution trace $\texttt{abstrace}(c)$ for a program, which computes the Abstract Execution Trace for program c, as a set of the abstract events $\overset{\alpha}{\epsilon}$.

Definition 22 (Abstract Execution Trace)**.** $\texttt{abstrace} \in \mathcal{C} \to \mathcal{P}(\mathcal{L} \times DC(\mathcal{VAR} \cup \mathcal{SMBCST}) \cup \{\top\}) \times \mathcal{L})$

We now show how to compute the abstract execution trace. For simplicity, we use $\mathcal{P}(\overset{\alpha}{\epsilon})$ represent the power set of all abstract events, and we have $\texttt{abstrace}(c) \in \mathcal{P}(\overset{\alpha}{\epsilon})$. We first append a skip command with the exist label l_{ex}, i.e., $\left[\texttt{skip}\right]^{l_{ex}}$ at the end of the program c, and compute the $\texttt{abstrace}(c) = \texttt{abstrace}'(c')$ for c', where $c' = c; \left[\texttt{skip}\right]^{l_{ex}}$ as follows,

$$\begin{array}{ll}
\texttt{abstrace}'([x \leftarrow e]^{l}) & = \emptyset \\
\texttt{abstrace}'(\left[x \leftarrow \texttt{query}(\psi)\right]^{l}) & = \emptyset \\
\texttt{abstrace}'([\texttt{skip}]^{l}) & = \emptyset \\
\texttt{abstrace}'(\,\texttt{if}\ [b]^{l}\ \texttt{then}\ c_t\ \texttt{else}\ c_f) & = \texttt{abstrace}'(c_t) \cup \texttt{abstrace}'(c_f) \cup \{(l, \top, \texttt{absinit}(c_t)), (l, \top, \texttt{absinit}(c_f))\} \\
\texttt{abstrace}'(\,\texttt{while}\ [b]^{l}\ \texttt{do}\ c_w) & = \texttt{abstrace}'(c_w) \cup \{(l, \top, \texttt{absinit}(c_w))\} \cup \{(l', dc, l) | (l', dc) \in \texttt{absfinal}(c_w)\} \\
\texttt{abstrace}'(c_1 ; c_2) & = \texttt{abstrace}'(c_1) \cup \texttt{abstrace}'(c_2) \cup \{(l, dc, \texttt{absinit}(c_2)) | (l, dc) \in \texttt{absfinal}(c_1)\}
\end{array}$$

Notice $\texttt{abstrace}'([x := e]^{l})$, $\texttt{abstrace}'([x := \texttt{query}(\psi)]^{l})$ and $\texttt{abstrace}'([\texttt{skip}]^{l})$ are all empty set. For every event ϵ with label l in an execution trace τ of program c, there is an abstract event in program's abstract execution trace of form $(l, \_, \_)$. We also show the soundness of the abstract execution trace in Appendix.

Lemma 4.1 (Soundness of the Abstract Execution Trace)**.** *Given a program c, we have:*

$$\begin{array}{c}
\forall \tau_0, \tau \in \mathcal{T}, \epsilon = (\_, l, \_) \in \mathcal{E}\ .\ \langle c, \tau_0 \rangle \to^{*} \langle \texttt{skip}, \tau_{0} {+\!+} \tau \rangle \land \epsilon \in \tau \\
\implies \exists \overset{\alpha}{\epsilon} = (l, \_, \_) \in (\mathcal{L} \times \mathcal{DC}^{\top} \times \mathcal{L})\ .\ \overset{\alpha}{\epsilon} \in \texttt{abstrace}(c)
\end{array}$$

This lemma is proved formally in Lemma E.1 in Appendix E.

For every labeled variable in program c, $x^l \in \mathbb{LV}_c$, there is a unique abstract event in program's abstract execution trace $\overset{\alpha}{\epsilon} \in \texttt{abstrace}(c)$ of form $(l, \_, \_)$.

Lemma 4.2 (Uniqueness of the Abstract Execution Trace)**.** *Given a program c, we have:*

$$\forall \tau_0, \tau \in \mathcal{T}, \epsilon = (\_, l, \_, \_) \in \mathcal{E}^{\texttt{asn}} . \langle c, \tau_0 \rangle \to^* \langle \texttt{skip}, \tau_{0 {+}{+} \tau} \rangle \wedge \epsilon \in \tau$$
$$\implies \exists! \overset{\alpha}{\epsilon} = (l, \_, \_) \in (\mathcal{L} \times \mathcal{DC}^{\top} \times \mathcal{L}) . \overset{\alpha}{\epsilon} \in \texttt{abstrace}(c)$$

This lemma and proof is also formalized in Lemma E.3 in Appendix E.

Then, we build the edge for c's abstract control flow graph as follos,

$$\texttt{absE}(c) = \{(l_1, dc, l_2) | (l_1, dc, l_2) \in \texttt{abstrace}(c)\}$$

Abstract Control Flow Graph With the vertices $\texttt{absV}(c)$ and edges $\texttt{absE}(c)$ ready, we construct the abstract control flow graph, formally defined in Definition 23.

Definition 23 (Abstract Control Flow Graph)**.** *Given a program c, with its abstract control flow* $\texttt{abstrace}(c)$ *its abstract control flow graph* $\texttt{absG}(c) = (\texttt{absV}(c), \texttt{absE}(c), \texttt{absW}(c))$ *is defined as follows,*

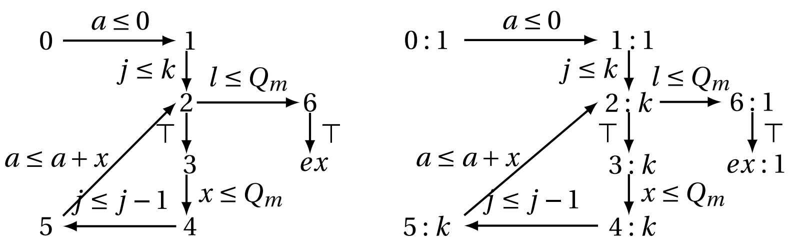
$\texttt{absE}(c) = \{(l_1, dc, l_2) | (l_1, dc, l_2) \in \texttt{abstrace}(c)\}$,
$\texttt{absV}(c) = labels(c) \cup \{l_{ex}\}$
$\texttt{absW}(c) \triangleq \left\{(l, w) \in \mathbb{L} \times EXPR(\mathcal{SMBCST})\right\}$.

Notice we also define the $\texttt{absW}(c)$ in this graph without giving an actual value. This $\texttt{absW}(c)$ is the set of weight for every label. The weight is a symbolic expression over the symbolic constant, which is the estimated upper bound on the number of visiting time for every control location through the reachability bound analysis as follows.

Example Let us look at the two-round example, its generated abstract control flow graph is shown as in Figure 4(b). For example, the edge $(0, a \leq 0, 1)$ on the top, tells us the command $[a \leftarrow 0]^0$ is executed with next continuation location 1, where the command $\left[j \leftarrow k\right]^1$ will be executed next. The constraint $a \leq 0$ is a difference constraint, generated by abstracting from the assignment command $a \leftarrow 0$, representing that value of a is less than or equals to 0 after location 0 before executing command at line 1. The difference constraint is an inequality relation between, the left-hand side of the inequality talks about the variable before the execution and the right-hand side ascribes those after the execution. Look at the $a < a + x$ on the edge 5 to 2, which describes the execution of the command at line 5, which is an assignment $a = a + x$. The a on the left side of $a < a + x$ represents the value of a after the assignment, while the right-hand side a stores the value before the assignment. Also, we have while loop, which is a circle $2 \to 4 \to 5 \to 2$ in Figure 4(b). Please also look at the edge from 3 to 4, which talks about the query! The $x < Q_m$ describes the execution of a query request (the command at line 3), the query results stored in x is bounded by Q_m. Q_m is the maximal value for query requesting result from the database DB. top means there is no assignment executed, for example, we have the difference constraint $\top$ on the edge 2 to 6, means at line 2, there is no assignment (it is a testing guard $j > 0$.) The same way for the rest edges' constructions.

```
[a ← 0]^0; [j ← k]^1;
while [j > 0]^2 do
([x ← query(χ[j])]^3;
[j ← j − 1]^4;
[a ← x + a]^5);
[l ← query(χ[k] * a)]^6
```

(a)



(b) (c)

Figure 4: (a) The same `towRounds(k)` program as Figure **??** (b) The abstract control flow graph for `towRounds(k)` (c) The abstract control flow graph with the reachability bound for `towRounds(k)`.

4.3.2 Edge Estimation with Interprocedure Call

We show how to estimate the directed edges in the static analysis dependency graph. We develop a variant of data flow analysis, called Feasible Data-Flow Generation, which considers both the control flow and data flow and is a sound approximation of the edges in the execution based dependency graph.

Also, worth to mention, we use the result of reaching definition on the abstract control flow graph in feasible data-flow generation to have a more precise approximation. Let us see a simple example, a program $[x = 0]^1; [x = 2]^2; [y = x + 1]^3$. The standard data flow analysis tells us that both the labeled variable x^1 and $x2$ 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.

In the first step, it performs the standard reaching definition analysis given a program c, on every label in $\texttt{absV}(c)$. This step generates set of all the reachable variables at location of label l in the program c. The $\texttt{RD}(l, c)$ 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 entry point of executing the command of label l. The block, is either the command of the form of assignment, skip, or a test of the form of $[b]^l$, denoted by $\mathsf{blocks}(c)$ the set of all the blocks in program c, where $\mathsf{blocks} : \mathcal{C} \to \mathcal{P}(\mathcal{C} \cup [b]^l)$. Then it generates the set of feasible data-flow between labeled variables with detail in Definition 24, based on $\texttt{RD}(l, c)$ for every label in a program c and its blocks blocks.
The details are as follows.

Reaching definition analysis A block is either the command of the form of assignment, skip, or test of the form of $[b]^l$.
The operator $\mathsf{blk} : \mathcal{C} \to blocks$ gives all the blocks in program c.
Set ? to be undefined:
The operator kill: $blocks \to \mathcal{P}(\mathcal{VAR} \times \mathcal{L} \cup \{?\})$ produces the set of labelled variables of assignment destroyed by the block.
The operator gen: $blocks \to \mathcal{P}(\mathcal{VAR} \times \mathcal{L} \cup \{?\})$ generates the set of labelled variables generated by the block.
The operator $in(l)$, $out(l)$: $\mathcal{L} \to \mathcal{LV} \cup \{?\}$ for every block in program c is defined as follows,

$$\begin{array}{ll} in(l) & = \{x^? | x^l \in \mathbb{LV}_c \wedge l = \texttt{absinit}(c)\} \cup \{out(l') || (l', \_, l) \in \texttt{absE}(c) \wedge l \neq \texttt{absinit}(c)\} \\ out(l) & = gen(B^l) \cup \{in(l) \setminus kill(B^l)\} \end{array}$$

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

of $in(l)$ at label l in program c.
The stabilized $in(l)$ and $out(l)$ for program c, as well as $\mathtt{RD}(l,c)$, is computed by the standard worklist algorithm with detail as below.

1. initial in[l]=out[l]=∅
2. initial in[entry label] = ∅
3. initialize a work queue, contains all the blocks in C
4. while |W| != 0
   pop l in W
   old = out[l]
   in(l) = out(l') where $(l', \_, l) \in \mathtt{absE}(c)$
   out(l) = gen(b^l) ∪ (in(l) - kill(b^l)) where b^l in $\mathsf{blk}(c)$
   if (old != out(l)) W= W ∪ {l'| (l,l') in $(l', \_, l) \in \mathtt{absE}(c)$}
   end while

Feasible Data-Flow Generation by using the results of Reaching definition analysis results, specifically $\mathtt{RD}(l,c)$ for every label in a program c, we refine the vertices and edges in the $\mathtt{absG}$ graph by generating the set of feasible data-flow between labeled variables as follows,

Definition 24 (Feasible Data-Flow)**.** *Given a program c and two labeled variables x^i, y^j in this program,* $\mathtt{flowsTo}(x^i, y^j, c)$ *is*

$$
\begin{array}{ll}
\mathtt{flowsTo}(x^i, y^j, [x \leftarrow e]^l) & \triangleq (x^i, y^j) \in \{(y^i, x^l) | y \in \mathsf{FV}(e) \land y^i \in \mathtt{RD}(l, [x \leftarrow e]^l)\} \\
\mathtt{flowsTo}(x^i, y^j, \left[x \leftarrow \mathtt{query}(\psi)\right]^l) & \triangleq (x^i, y^j) \in \{(y^i, x^l) | y \in \mathsf{FV}(\psi) \land y^i \in \mathtt{RD}(l, \left[x \leftarrow \mathtt{query}(\psi)\right]^l)\} \\
\mathtt{flowsTo}(x^i, y^j, [\mathtt{skip}]^l) & = \emptyset \\
\mathtt{flowsTo}(x^i, y^j, \mathtt{if}\ ([b]^l, c_1, c_2)) & \triangleq \mathtt{flowsTo}(x^i, y^j, c_1) \lor \mathtt{flowsTo}(x^i, y^j, c_2) \\
 & \lor (x^i, y^j) \in \{(x^i, y^j) | x \in \mathsf{FV}(b) \land x^i \in \mathtt{RD}(l, \mathtt{if}\ ([b]^l, c_1, c_2)) \land y^j \in \mathbb{LV}(c_1) \\
 & \lor (x^i, y^j) \in \{(x^i, y^j) | x \in \mathsf{FV}(b) \land x^i \in \mathtt{RD}(l, \mathtt{if}\ ([b]^l, c_1, c_2)) \land y^j \in \mathbb{LV}(c_2) \\
\mathtt{flowsTo}(x^i, y^j, \mathtt{while}\ [b]^l\ \mathsf{do}\ c_w) & \triangleq \mathtt{flowsTo}(x^i, y^j, c_w) \lor \\
 & (x^i, y^j) \in \{(x^i, y^j) | x \in \mathsf{FV}(b) \land x^i \in \mathtt{RD}(l, \mathtt{while}\ [b]^l\ \mathsf{do}\ c_w) \land y^j \in \mathbb{LV}(c_w) \\
\mathtt{flowsTo}(x^i, y^j, c_1; c_2) & \triangleq \mathtt{flowsTo}(x^i, y^j, c_1) \lor \mathtt{flowsTo}(x^i, y^j, c_2) \\
[\ \mathtt{fun}\ ]^l : x(r, x_1, \dots, x_n) := c & \emptyset \\
\left[x \leftarrow\ \mathtt{call}\ (f, e_1, \dots, e_n)\right]^l & \triangleq \mathtt{flowsTo}(x^i, y^j, [x_i \leftarrow e_i]^{(l,i)}) \lor \mathtt{flowsTo}(x^i, y^j, [c^{+n}]^l) \lor \mathtt{flowsTo}(x^i, y^j, \left[x \leftarrow r^{l_r+n}\right]^l) \\
 & \land f(r^{l_r}, x_1, \dots, x_n) := c \in \mathtt{RD}(l, c)
\end{array}
$$

We prove that this *Feasible Data-Flow* relation is a sound approximation of the *Variable May-Dependency* relation over labeled variables for every program, in Appendix D.

Edges Estimation Then we define the estimated directed edges between vertices (x_1^i, w_1) and (x_2^j, w_2) where $x_1^i, x_2^j \in \mathbb{LV}(c)$, as a set of triples $\mathtt{E}_{\mathtt{prog}}(c) \in \mathcal{P}(\mathcal{LV} \times \mathcal{A}_{\mathtt{in}} \times \mathcal{LV})$ indicating a directed edge from the first vertex to the second one in each pair as follows,

$$
\mathtt{E}_{\mathtt{prog}}{}^0(c) \triangleq \left\{ (x_1^i, w, x_2^j) \in \mathcal{LV} \times \mathcal{A}_{\mathtt{in}} \times \mathcal{LV} \;\middle|\; \begin{array}{l} x_1^i, x_2^j \in \mathbb{LV}(c) \land \exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c\ .\ n \geq 0 \land \\ \mathtt{flowsTo}(x^i, z_1^{r_1}, c) \land \cdots \land \mathtt{flowsTo}(z_n^{r_n}, y^j, c) \end{array} \right\}
$$

The weight for every edge will be computed as next step in Section 4.3.3. We prove that this estimated directed edge set $\mathtt{E}_{\mathtt{prog}}(c)$ is a sound approximation of the edge set in c's Execution-Based Dependency Graph in Appendix B.

Example Still looking at the Figure 3(c) in main paper, and taking the edge (l^6, a^5) for example. By $\texttt{flowsTo}(l^6, a^5, c)$, we can see a is used directly in the query expression $\chi[k] * a$, in the assignment command $[l \leftarrow \texttt{query}(\chi[k] * a)]^l$, i.e., $a \in FV(\chi[k] * a)$. Also, from the Reaching definition analysis, we know $a^5 \in \texttt{RD}(6, two - round)$. Then we have $\texttt{flowsTo}(l^6, a^5, c)$ and construct the edge (l^6, a^5). And same way for constructing the rest edges. Also, the edge (x^3, j^5) in the same graph represents the control flow, caught by our `flowsTo` relation.

4.3.3 Weight Estimation via Path Sensitive Reachability Bound Analysis

In order to estimate weight for every vertex in the static analysis dependency graph($\texttt{V}_{\texttt{prog}}(c)$), we want to find out the upper bound on the number of times the labeled command (uniquely associated with a vertex in $\texttt{V}_{\texttt{prog}}(c)$) may be executed when running the program. This information can be obtained by computing the reachability bound for every vertex in the abstract control flow graph ($\texttt{absW}(c)$), because the vertices in the two graph share the same unique label, the line number. We can easily show that the reachability bound on one vertex of the abstract control flow graph is also the upper bound for the corresponding vertex in the static analysis dependency graph, both vertices share the same unique line number.

We perform the symbolic reachability bound analysis on the abstract control flow graph, through the edges in $\texttt{absG}(c)$, which correspond to c's abstract transition between labels. We infer the invariant for every variable, and compute the transition closure for every abstract transition. By solving the closure with the invariants of variables involved in this closure for every transition, we compute the symbolic reachability bound of every commands corresponding to this transition. Specifically, this analysis can be performed in four steps: Variable Modification Tracking, Local Bounds Computation, Invariant Inference and Closure Generation, and Reachability Bound Computation, with details as follows.

Variable Modification Tracking Identify the abstract events where each variable is increased, decreased and reset:
$\texttt{inc} : \mathcal{VAR} \rightarrow \mathcal{P}(\overset{\alpha}{\epsilon})$ the set of the abstract events where the variable increase.
$\texttt{inc}(x) = \{(\overset{\alpha}{\epsilon}, c) \mid \overset{\alpha}{\epsilon} = (l, l', x' \leq x + v)\}$
$\texttt{re} : \mathcal{VAR} \rightarrow \mathcal{P}(\overset{\alpha}{\epsilon})$ The set of the abstract events where the variable is reset.
$\texttt{dec} : \mathcal{VAR} \rightarrow \mathcal{P}(\overset{\alpha}{\epsilon})$ The set of abstract events where the variable decrease.
$Incr(v) \triangleq \sum_{(\overset{\alpha}{\epsilon}, c) \in \texttt{inc}(v)} \{\texttt{Tclosure}(\overset{\alpha}{\epsilon}) \times v\}$

Local Bounds Given a program c with its abstract control flow graph $\texttt{absG}(c) = (\texttt{absV}, \texttt{absE})$
Local Bounds Computation: $\texttt{locb} : \overset{\alpha}{\epsilon} \rightarrow \mathcal{VAR} \cup \mathcal{SMBCST}$.

$$
\begin{array}{ll}
\texttt{locb}(\overset{\alpha}{\epsilon}) \triangleq 1 & \overset{\alpha}{\epsilon} \notin SCC(\texttt{absG}(c)) \\
\texttt{locb}(\overset{\alpha}{\epsilon}) \triangleq (x, v) & \overset{\alpha}{\epsilon} \in SCC(\texttt{absG}(c)) \wedge \overset{\alpha}{\epsilon} \in \texttt{dec}(x) \wedge \overset{\alpha}{\epsilon} = (\_, \_, x' \leq x - v) \\
\texttt{locb}(\overset{\alpha}{\epsilon}) \triangleq (x, \max(\texttt{dec}(x))) & \overset{\alpha}{\epsilon} \in SCC(\texttt{absG}(c)) \wedge \overset{\alpha}{\epsilon} \notin \bigcup_{x \in \mathcal{VAR}} \texttt{dec}(x) \wedge \overset{\alpha}{\epsilon} \notin SCC(\texttt{absG}(c) \setminus \texttt{dec}(x))
\end{array}
$$

The first case is straightforward. Since variable's visiting time outside of any while loop is at most 1, we do not need to analyze the visiting times of every node in the graph from phase 1. The second and third step is guaranteed by the *Discussion on Soundness* in Section 4 of [2]. Then soundness proof is in Lemma E.2 in Appendix E.

Invariant Inference and Closure Generation Then, computing the bound invariants for variables and the transition closures for abstract events:
$\texttt{Vinvar} : \mathcal{VAR} \cup \mathcal{SMBCST} \rightarrow EXPR(\mathcal{SMBCST})$
$\texttt{Tclosure} : \overset{\alpha}{\epsilon} \rightarrow EXPR(\mathcal{SMBCST})$
$EXPR(\mathcal{SMBCST})$ is symbolic expression over $\mathcal{SMBCST}$, which is a subset of arithmetic expressions over $\mathbb{N}$ with input variables and . We use $\mathcal{A}_{\texttt{in}}$ denotes the arithmetic expression over the symbolic variables, (i.e., $\mathbb{N}$ with input variables and). Then, the symbolic invariant for each variable as well as the symbolic transition closure for each transition is calculated as follows:

$$\begin{array}{lll} \texttt{Vinvar}(x) & \triangleq c & c \in \mathcal{SMBCST} \\ \texttt{Vinvar}(x) & \triangleq Incr(v) + \max(\{\texttt{Vinvar}(a) + c \mid (t, a, c) \in \texttt{re}(x)\}) & c \notin \mathcal{SMBCST} \end{array}$$

Definition 25.

$$\begin{array}{ll} \texttt{Tclosure}(\overset{\alpha}{\epsilon}) & \triangleq x / v \\ & \texttt{locb}(\overset{\alpha}{\epsilon}) = (x, v) \in \mathcal{SMBCST} \times \mathbb{N} \\ \texttt{Tclosure}(\overset{\alpha}{\epsilon}) & \triangleq (Incr(x) + \sum_{(\overset{\alpha'}{\epsilon}, y, v') \in \texttt{re}(x)} \texttt{Tclosure}(\overset{\alpha'}{\epsilon}) \times \max(\texttt{Vinvar}(y) + v', 0)) / v \\ & \texttt{locb}(\overset{\alpha}{\epsilon}) = (x, v) \wedge x \notin \mathcal{SMBCST} \end{array}$$

Improved Variable Modification Tracking Instead of just identifying the abstract events where each variable is reset, this improvement identifies the chain of the events where a given variable is reset by the variables of the abstract events through the chain.
$\texttt{rechain} : \mathcal{VAR} \rightarrow \mathcal{P}(\mathcal{P}(\overset{\alpha}{\epsilon}))$ The set of the chain of abstract events where the variable is reset through the chain.

Improved Invariant Inference and Closure Generation Then, computing the bound invariants for variables and the transition closures for abstract events:
$\texttt{Vinvar} : \mathcal{VAR} \cup \mathcal{SMBCST} \rightarrow \mathcal{A}_{\texttt{in}}$
$\texttt{Tclosure} : \overset{\alpha}{\epsilon} \rightarrow \mathcal{A}_{\texttt{in}}$
Then, the symbolic invariant for each variable as well as the symbolic transition closure for each transition is calculated as follows:

$$\begin{array}{lll} \texttt{Vinvar}(x) & \triangleq c & c \in \mathcal{SMBCST} \\ \texttt{Vinvar}(x) & \triangleq Incr(v) + \max(\{\texttt{Vinvar}(a) + c \mid (t, a, c) \in \texttt{re}(x)\}) & c \notin \mathcal{SMBCST} \end{array}$$

Definition 26.

$$\begin{array}{ll} \texttt{Tclosure}(\overset{\alpha}{\epsilon}) & \triangleq x / v \\ & \texttt{locb}(\overset{\alpha}{\epsilon}) = (x, v) \in \mathcal{SMBCST} \times \mathbb{N} \\ \texttt{Tclosure}(\overset{\alpha}{\epsilon}) & \triangleq \Big(\sum_{y \in \{y \mid ch \in \texttt{rechain}(x), (l_1, x, y, v, l_2) \in ch\}} Incr(x) \\ & \quad + \sum_{ch \in \texttt{rechain}(x)} \big(\min_{\overset{\alpha'}{\epsilon} \in ch} (\texttt{Tclosure}(\overset{\alpha'}{\epsilon})) \times \max(\texttt{Vinvar}(y) + \sum_{(l_1, x, y, v, l_2) \in ch} v, 0) \big) \Big) / v \\ & \texttt{locb}(\overset{\alpha}{\epsilon}) = (x, v) \wedge x \notin \mathcal{SMBCST} \end{array}$$

$\texttt{W}_{\texttt{prog}}(x^{l}) \triangleq \texttt{Tclosure}(\overset{\alpha}{\epsilon})$

Reachability Bound Computation Through the transition closure computed above, The weight of every label in the program c's abstract control flow graph, $\mathtt{absG}(c) = (\mathtt{absV}, \mathtt{absE}, \mathtt{absW})$ is computed as the maximum over all the abstract events $\overset{\alpha}{e} \in \mathtt{absE}$ heading out from this vertex, formally as follows. $\mathtt{absW} \triangleq \left\{(l, w) \in \mathbb{N} \times \mathcal{A}_{\mathtt{in}} | w = \max\left\{\mathtt{Tclosure}(\overset{\alpha}{e}) \mid \overset{\alpha}{e} \in \mathtt{abstrace}(c) \land \overset{\alpha}{e} = (l, \_, \_)\right\}\right\}$.

Example We perform the symbolic reachability bound analysis on the abstract control flow graph as follows. 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 $(4, j < j-1, 5)$, $\mathtt{Tclosure}(4,5) = \mathtt{Vinvar}(j)$. The invariant for variable j, $\mathtt{Vinvar}(j)$ used here is $\mathtt{Vinvar}(j) = k * \mathtt{Tclosure}(1,2)$, which is generated by all the difference constraints involving j in the graph. Notice the k in $\mathtt{Vinvar}(j)$ comes from considering both difference constraint $j <= k$ from edge (1,2) and $j <= j-1$ from (4,5), 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 obtains the reachability bound of every edge. We decide the weight for every vertex in the abstract control flow graph by using the bound of the edges which head out from this vertex, by taking the max of the bound from these involving edges. For instance, By the constraint on the edge $(4, j \leq j-1, 5)$, we get bound k for this edge. Then, we assign vertex 4 by reachability bound k, as in Figure 4(c). Another interesting vertex is 2, which has more than one edge heading out from it, $(2, \top, 3)$ and $(2, \top, 6)$. For the weight for vertex 2, we choose the max between the bound k from $(2, \top, 3)$ and 1 from $(2, \top, 6)$. The same way for the rest weights' computation. We use $\mathtt{absW}(c)$ for the set of weights we just computed for each label in the abstract control flow graph of c. The same way for the rest weights' computation.

Vertex Weight Computation Then we compute the weight for each vertex in $\mathtt{V}_{\mathtt{prog}}(c)$, as a set of pairs mapping each vertex $x^l \in \mathbb{LV}(c)$ to a symbolic expression over $\mathcal{SMBCST}$. $\mathtt{W}_{\mathtt{prog}}(c) \in \mathcal{P}(\mathcal{LV} \times \mathcal{A}_{\mathtt{in}})$ is formally computed as follows,

$$\mathtt{V}_{\mathtt{prog}}(c) \triangleq \left\{(x^l, w) \mid x^l \in \mathtt{V}_{\mathtt{prog}}{}^0(c) \land (l, w) \in \mathtt{absW}(c)\right\}.$$

We prove that this symbolic expression for $x^l \in \mathtt{V}_{\mathtt{prog}}(c)$ is a sound upper bound of the weight for the same vertex x^l in Program's execution-based dependency graph in Appendix E. The maximum visiting times of x^l over all execution traces of c in Appendix E.

Theorem 4.1 (Soundness of the Vertex Weight Estimation). *Given a program c with its program-based dependency graph* $\mathtt{G}_{\mathtt{prog}} = (\mathtt{V}_{\mathtt{prog}}, \mathtt{E}_{\mathtt{prog}})$, $\mathtt{G}_{\mathtt{trace}} = (\mathtt{V}_{\mathtt{trace}}, \mathtt{E}_{\mathtt{trace}})$, *we have:*

$$\forall (x^l, w_t) \in \mathtt{V}_{\mathtt{trace}}, (x^l, w_p) \in \mathtt{V}_{\mathtt{prog}}, \tau_0 \in \mathcal{T}_0(c), \tau' \in \mathcal{T}, v \in \mathbb{N}\,.$$
$$\langle c, \tau_0 \rangle \to^* \langle \mathtt{skip}, \tau_0 +\!+ \tau' \rangle \land \langle w^p, \tau_0 \rangle \Downarrow_e v \implies w_t(\tau) \leq v$$

Example Now let's go back to the Program-Based Dependency Graph which we aim to build for approximating the Execution-Based Dependency graph for two-round example, as in Figure **??**(c). Every vertex from $\mathtt{V}_{\mathtt{prog}}(c)$ in this graph corresponds to a labeled variable, for example a^5, and this label 5 is also a vertex 5 in the abstract control flow graph in Figure 4(b). Then, it is straight forward, that the reachability bound for the label 5, is also the maximum visiting times bound of the labeled variable a^5. So, we estimate the visiting time for labeled variable a^5 in Program-Based Dependency Graph in Figrue 4(c) as k as well. The same way for the rest weights' computation.

Edges Weight Computation Then we compute the weight for each edge in $\mathtt{E}_{\mathtt{prog}}(c)$ computed above,

$$\mathtt{E}_{\mathtt{prog}}(c) \triangleq \left\{(x^{i}, w, y^{j}) \mid (x^{i}, w, y^{j}) \in \mathtt{E}_{\mathtt{prog}}{}^{0}(c) \land w = \max\left\{\mathtt{Tclosure}(\overset{\alpha}{\epsilon}) \mid \overset{\alpha}{\epsilon} \in \mathtt{abstrace}(c) \land \overset{\alpha}{\epsilon} = (i, \_, j)\right\}\right\}.$$

We prove that this symbolic expression w for edge $(x^{i}, w, y^{j}) \in \mathtt{E}_{\mathtt{prog}}(c)$ is a sound upper bound of the weight for the same edge (x^{i}, w', y^{j}) in Program's execution-based dependency graph in Appendix F.

Theorem 4.2 (Soundness of the Edge Weight Estimation)**.** *Given a program c with its program-based dependency graph* $\mathtt{G}_{\mathtt{prog}} = (\mathtt{V}_{\mathtt{prog}}, \mathtt{E}_{\mathtt{prog}})$, $\mathtt{G}_{\mathtt{trace}} = (\mathtt{V}_{\mathtt{trace}}, \mathtt{E}_{\mathtt{trace}})$, *we have:*

$$\forall (x^{l}, w_{t}) \in \mathtt{W}_{\mathtt{trace}}, (x^{l}, w_{p}) \in \mathtt{W}_{\mathtt{prog}}, \tau \in \mathcal{T} \,.\, \langle c, \tau \rangle \to^{*} \langle \mathtt{skip}, \tau_{0++}\tau' \rangle \land \langle w_{p}, \tau \rangle \Downarrow_{e} v \implies \leq w_{t}(\tau) \leq v$$

Example Now let's go back to the Program-Based Dependency Graph which we aim to build for approximating the Execution-Based Dependency graph for two-round example, as in Figure **??**(c).

4.4 Program-Based Data Dependency Graph Generation

Finally we build the estimated data dependency graph based on the above program static analysis as follows:

$$\mathtt{G}_{\mathtt{prog}}(c) = (\mathtt{V}_{\mathtt{prog}}(c), \mathtt{E}_{\mathtt{prog}}(c))$$

with $\mathtt{V}_{\mathtt{prog}}(c)$ and $\mathtt{E}_{\mathtt{prog}}(c)$ as computed in each steps above. This program-based graph program-based graph has a similar topology structure as the Execution-Based Dependency Graph. It has the same vertices but approximated edges and weights. It is formally defined in Definition 27.

Definition 27 (Program-Based Dependency Graph)**.** *Given a program c, with its abstract weighted control flow graph* $\mathtt{absG}(c) = (\mathtt{absV}, \mathtt{absE}, \mathtt{absW})$ *and feasible data flow relation* $\mathtt{flowsTo}(x^{i}, y^{j}, c)$ *for every* $x^{i}, y^{j} \in \mathbb{LV}_{c}$, *its Program-Based Weighted Data Dependency Graph* $\mathtt{G}_{\mathtt{prog}}(c) = (\mathtt{V}_{\mathtt{prog}}, \mathtt{E}_{\mathtt{prog}})$, *is generated as follows,*

$$\begin{array}{lcl}
\mathtt{V}_{\mathtt{prog}}(c) & \triangleq & \left\{(x^{l}, w) \in \mathcal{LV} \times \mathcal{A}_{in} \mid x^{l} \in \mathbb{LV}_{c} \land (l, w) \in \mathtt{absW}(c)\right\} \\
\mathtt{E}_{\mathtt{prog}}(c) & \triangleq & \Big\{(x^{i}, w, y^{j}) \in \mathcal{LV} \times \mathcal{A}_{\mathtt{in}} \times \mathcal{LV} \mid \\
& & \quad x^{i}, y^{j} \in \mathbb{LV}(c) \land \mathtt{flowsTo}(x^{i}, y^{j}, c) \land \exists n \in \mathbb{N}, z_{1}^{r_1}, \cdots, z_{n}^{r_n} \in \mathbb{LV}_{c} \,.\, n \geq 0 \land \cdots \land \mathtt{flowsTo}(z_{n}^{r_n}, y^{j}, c) \\
& & \quad \land w = \max\left\{\mathtt{Tclosure}(\overset{\alpha}{\epsilon}) \mid \overset{\alpha}{\epsilon} \in \mathtt{abstrace}(c) \land \overset{\alpha}{\epsilon} = (i, \_, j)\right\}\Big\}.
\end{array}$$

4.5 Adaptivity Upper Bound Computation

This phase computes the adaptivity upper bound for a program c.
With c's program-based data dependency graph $\mathtt{G}_{\mathtt{prog}}(c)$ approximated above, its adaptivity upper bound is estimated as the maximum query length over all finite walks in $\mathcal{WK}(\mathtt{G}_{\mathtt{prog}}(c))$ formally in Definition 30, and computed in Algorithm 1.

Different from the finite walk on a program c's execution based graph, the finite walk in $\mathtt{G}_{\mathtt{prog}}(c)$ 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 $(v_i, w_i) \in \mathtt{V}_{\mathtt{prog}}(c)$, is v_i's estimated weight. Formally defined as follows.

Definition 28 (Finite Walk on Program-Based Dependency Graph (k))**.** .
Given a program c's program-based dependency graph $\mathtt{G}_{\mathtt{prog}}(c) = (\mathtt{V}_{\mathtt{prog}}(c), \mathtt{E}_{\mathtt{prog}}(c))$ *a* finite walk k *in* $\mathtt{G}_{\mathtt{trace}}(c)$ *is a sequence of edges* $(e_1 \dots e_{n-1})$ *for which there is a sequence of vertices* $(v_1, \dots, v_n)$ *such that:*

- *$e_i = (v_i, w_i, v_{i+1}) \in \mathtt{E}_{\mathtt{prog}}(c)$ for every $1 \leq i < n$, and occurrence times of e_i smaller than w_i.*
- *every vertex $(v_i, w_i) \in \mathtt{V}_{\mathtt{prog}}(c)$, v_i appears in $(v_1, \dots, v_n)$ at most w_i times.*

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

We abuse the notation $\mathcal{WK}(\mathtt{G}_{\mathtt{prog}}(c))$ represents the walks over the program-based dependency graph for c. Different from the walks on a program c's execution based graph, $k \in \mathcal{WK}(\mathtt{G}_{\mathtt{trace}}(c))$, $k \in \mathcal{WK}(\mathtt{G}_{\mathtt{prog}}(c))$ 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 $(v_i, w_i) \in \mathtt{V}_{\mathtt{prog}}(c)$, is v_i's estimated weight. The length of a finite walk $k \in \mathcal{WK}(\mathtt{G}_{\mathtt{prog}}(c))$ is an arithmetic expression as well, i.e., $\mathtt{len}(k) \in \mathcal{A}_{in}$

Then the query length of a finite walk in $\mathtt{G}_{\mathtt{prog}}(c)$ is an arithmetic expression as well as follows,

Definition 29 (Query Length of the Finite Walk on Program-Based Dependency Graph ($\mathtt{len}^{\mathtt{q}}$))**.** *Given a program c's execution-based dependency graph $\mathtt{G}_{\mathtt{prog}}(c) = (\mathtt{V}_{\mathtt{prog}}(c), \mathtt{E}_{\mathtt{prog}}(c), \mathtt{W}_{\mathtt{prog}}(c), \mathtt{Q}_{\mathtt{prog}}(c))$, and a* finite walk $k \in \mathcal{WK}(\mathtt{G}_{\mathtt{prog}}(c))$*, The query length of k, $\mathtt{len}^{\mathtt{q}}(k) \in \mathcal{A}_{in}$ is the number of vertices which correspond to query variables in the vertices sequence of the this walk k $(v_1, \dots, v_n)$ as follows,*

$$\mathtt{len}^{\mathtt{q}}(k) = |\big(v \mid v \in (v_1, \dots, v_n) \land v \in \mathbb{QV}(c)\big)|.$$

Definition 30 (Program-Based Adaptivity)**.** .
Given a program c and its program-based graph $\mathtt{G}_{\mathtt{prog}}(c)$ the program-based adaptivity for c is defined as

$$A_{\mathtt{prog}}(c) \triangleq \max\big\{\mathtt{len}^{\mathtt{q}}(k) \mid k \in \mathcal{WK}(\mathtt{G}_{\mathtt{prog}}(c))\big\}.$$

Based on our soundness of the program-based adaptivity, our program-based adaptivity is a sound upper bound of its adaptivity in Definition 20.

Theorem 4.3 (Soundness of AdaptFun)**.** *For every program c, its program-based adaptivity is a sound upper bound of its adaptivity.*

$$A_{\mathtt{prog}}(c) \geq A(c)$$

For $A_{\mathtt{prog}}(c) \geq A(c)$ comparing between function and arithmetic expression, we are specifically comparing, $\forall \tau \in \mathcal{T} \,.\, \langle A(c), \tau \rangle \Downarrow_e n \implies n \geq A(c)(\tau)$. To estimate a sound and precise upper bound on adaptivity, we develop an adaptivity estimation algorithm called AdaptSearch (in Apdix Algorithm I), which uses both the deep first search and breath first search strategy to find the walk. We also show that the estimated adaptivity from our AdaptSearch is sound with respect to the program-based adaptivity.

Theorem 4.4 (Soundness of AdaptSearch)**.** *For every program c.*

$$\mathsf{AdaptSearch}(\mathtt{G}_{\mathtt{prog}}(c)) \geq A_{\mathtt{prog}}(c).$$

The full details of all the soundness can be found in the appendix.

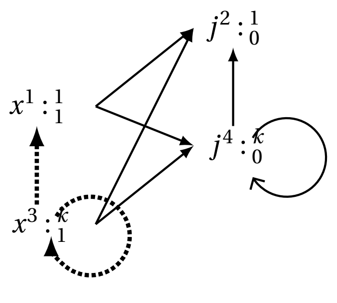
As indicated by our definition of prograpm-based adaptivity, the key point is to find the walks in the program-based dependency graph. We develop some walk-finding algorithms, Algorithm 1 and Algorithm 2, which use both the deep first search and breath first search strategy.

By Definition 18, this finite walk isn't easy to find. We first discuss two challenges when we try to find the walks in the dependency graph, and show that how we solve them using our algorithms.

Non-Termination Challenge: One naive walk finding method is to simply traverse on this graph by decreasing the weight of every node by one after every visiting. However, this simple traversing strategy leads to non-termination dilemma for most programs we are interested in. Specifically, this

$\texttt{whileSim(k)} \triangleq$
$[j \leftarrow k]^0;$
$[x \leftarrow \texttt{query}(\chi[0])]^1;$
$\texttt{while}\ [j > 0]^2\ \texttt{do}$
$\big([x \leftarrow \texttt{query}(\chi[x])]^3;$
$[j \leftarrow j - 1]^4\big)$

(a)


$j^2 : \substack{1 \\ 0}$
$x^1 : \substack{1 \\ 1}$
$j^4 : \substack{k \\ 0}$
$x^3 : \substack{k \\ 1}$

(b)

Figure 5: (a) Simple While Loop Example, (b) The Program-Based Dependency Graph generated from AdaptFun.

challenge comes from the weight of each vertex estimated in program's Program-Based Dependency Graph, which is not only a number but also can be a symbolic expression.

It is difficult to tell when to terminate the recursion when the domain of this symbolic expression isn't finite, some the walk may also be infinite. While, in most of our cases, the programs' Program-Based Dependency Graphs are having symbolic weights with infinite domains on vertices. Look at the simple example in Figure 5, where k is the input variable from domain $\mathbb{N}$. If we traverse on the program-based 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: we first find a longest path in the program-based dependency graph and then approximate the walk with the path. Through a simple deep first search algorithm, we find the longest weighted path as the dotted arrow in Figure 5, $x^3 : \substack{k \\ 1} \to x^1 : \substack{1 \\ 1}$. Then, by summing up the weights on this path where the vertices has query annotation 1, deep first search algorithm gives the adaptivity bound $1 + k$. This is a the tight bound for this program's adaptivity.

Approximation Challenge: When we adopt a deep first strategy to search for the longest weighted path, and then use the path to approximate the adaptivity. We find that this gives us over-approximation to a large extend. This over-approximation could result in a ∞ adaptivity upper bound on the program with actual adaptivity 2. Look at the two-round example in overview, it is easy to find that the longest weighted path is $x^3 : \substack{k \\ 1} \to a^5 : \substack{k \\ 0} \to l^6 : \substack{1 \\ 0}$ with weighted query length $1 + k$. If we use this path to approximate a finite walk, and weight of each vertex as its visiting time, then it isn't a qualified walk. In the approximated walk, we have the vertices as $x^3 \to \cdots \to x^3 \to a^5 \to \cdots \to a^5 \to l^6$. Because l^6 can only be visited as most once by its weight, resulting in the restriction on the maximum visiting time of x^3, such that x^3 is only able to be visited at most once as well. However, x^3 is visited k times in this approximated walk. In order to have x^3 be visited k time, we need to go back to x^3 on this walk from either a^5 or l^6 for k time. This is impossible since there is no edge going back to x^3 in $\mathtt{G}_{\mathtt{prog}}(twoRound)$. Obviously, its weighted query length, $1 + k$, over approximates the adaptivity of this example to a large extend, which supposed to be 2.

These challenges motivate us to design a walk search algorithm through a combination of deep first search and breath first search strategy. This walk search algorithm consists of two components: the path searching algorithm, AdaptSearch (in Algorithm 1) which search for a 'suitable' path relying on the strong connected components of the program based dependency graph, and $\mathsf{AdaptSearch}_{\mathtt{scc}}(\mathtt{G})$ (in Algorithm 2) which approximates the path. The AdaptSearch as shown in Appendix Algorithm I, takes our program-based dependency graph as input, and outputs the estimated adaptivity by two steps. 1. Process the input graph to a simplified graph 2. Perform the standard breath first search strategy to find the longest weighted path on this simplified graph and return the length as adaptivity. The step 2 is not interesting, we now discuss step 1. The input dependency graph may contain circle due to the while loop, we simplify (shrank) the input graph by replacing every strong connected components(circle) of the graph with, the vertex whose weight is the adaptivity of the SCC (a subgraph of the input one) calculated by the $\mathsf{AdaptSearch}_{\mathsf{scc}}$. The SCC is found by using the Kosaraju's algorithm. The details of this algorithm is explained as follows.

Algorithm 1 Adaptivity Computation Algorithm ($\mathsf{AdaptSearch}$)

```
Require: $G = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{Q})$ #{The program based dependency graph}
1: $\mathsf{AdaptSearch}(\mathtt{G})$:
2: init
   $q$: empty queue.
   $\mathtt{adapt}$ : the adaptivity of this graph initialize with 0.
3: Find all Strong Connected Components (SCC) in $G$: $\mathtt{SCC_1}, \cdots, \mathtt{SCC_n}, 0 \leq n \leq |\mathtt{V}|$,
4: for every SCC: $\mathtt{SCC_i}$, compute its Adaptivity $\mathtt{SCC_i}$:
5:    $\mathtt{adapt}_{\mathtt{scc}}[\mathtt{SCC_i}] = \mathsf{AdaptSearch}_{\mathsf{scc}}(\mathtt{SCC_i})$;
6: for every $\mathtt{SCC_i}$:
7:       $q.append(\mathtt{SCC_i})$;
8:       $\mathtt{adapt}_{\mathtt{tmp}} = 0$;
9:       while $q$ isn't empty:
10:            $\mathtt{s} = q.pop()$; #{take the top SCC from head of queue}
11:            $\mathtt{adapt}_{\mathtt{tmp_0}} = \mathtt{adapt}_{\mathtt{tmp}}$; #{record the adaptivity of last level}
12:            $\mathtt{SCC}_{\mathtt{max}}$; #{record the SCC with longest walk in this level}
13:            for every different SCC, $\mathtt{s}'$ connected by $\mathtt{s}$ by a directed edge from $\mathtt{s}$:
14:                  if $(\mathtt{adapt}_{\mathtt{tmp}} < \mathtt{adapt}_{\mathtt{tmp_0}} + \mathtt{adapt}_{\mathtt{scc}}[\mathtt{s}'])$:
15:                        $\mathtt{adapt}_{\mathtt{tmp}} = \mathtt{adapt}_{\mathtt{tmp_0}} + \mathtt{adapt}_{\mathtt{scc}}[\mathtt{s}']$;
16:                        $\mathtt{SCC}_{\mathtt{max}} = \mathtt{s}'$; #{update the SCC with longest walk in this level}
17:                  $q.append(\mathtt{SCC}_{\mathtt{max}})$;
18:       $\mathtt{adapt} = \max(\mathtt{adapt}, \mathtt{adapt}_{\mathtt{tmp}})$;
19: return $\mathtt{adapt}$.
```

The Adaptivity Computation Algorithm ($\mathsf{AdaptSearch}$) This algorithm first finds all the strong connected components (SCC) of $\mathtt{G}_{\mathtt{prog}}(c)$ using the Kosaraju's algorithm in line:3. Every $\mathtt{SCC_1}, \cdots, \mathtt{SCC_n}$ where $0 \leq n \leq |\mathtt{V}|$ is a sub-graph of $\mathtt{G}_{\mathtt{prog}}(c)$, where $\mathtt{SCC_i} = (\mathtt{V}_i, \mathtt{E}_i, \mathtt{W}_i, \mathtt{Q}_i)$. Then, it computes the adaptivity on every SCC in line:4-5 by Algorithm 2. We guarantee the soundness of the adaptivity on SCC by Lemma G.1 with proof in Appendix G. The $\mathtt{G}_{\mathtt{prog}}(c)$ is then shrunk into an acyclic directed graph where $\mathtt{SCC_1}, \cdots, \mathtt{SCC_n}$ are vertices with their adaptivities as weights. For every $(v_i, v_j) \in \mathtt{E}$ such that $v_1 \in \mathtt{V}_i$, $v_j \in \mathtt{V}_j$ and $i \neq j$, there is a edge (s_i, s_j) in this shrank graph.
Then, we use the standard breath first search strategy to find the longest weighted path on this shrank graph and return the length as adaptivity.
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 a sound computation of the finite walk having the longest query length on c's program based dependency graph, in Theorem G.1 in Appendix. We also guarantee the conditional completeness of the adaptivity computation for graphs under the case that c's Program-Based Dependency Graph $\mathtt{G}_{\mathtt{prog}}(c)$ is acyclic directed in Theorem H.1 in Appendix H.

Algorithm 2 Adaptivity Computation Algorithm on SCC Graph

```
Require: $G = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{Q})$ #{An Strong Connected program based dependency Graph}
 1: AdaptSearch$_{\mathsf{scc}}$(G):
 2: init
    $\mathtt{r}_{\mathsf{scc}}$: $EXPR(\mathcal{SMBCST})$, initialized 0, the Adaptivity of this SCC
 3:     init
        visited : {0,1} List,
        #{length |V|, initialize with 0 for every vertex, recording whether a vertex is visted.}
        r : $EXPR(\mathcal{SMBCST})$ List,
        #{length |V|, initialize with Q($v$) for every vertex, recording the adaptivity reaching each
    vertex.}
        flowcapacity: $EXPR(\mathcal{SMBCST})$ List,
        #{length |V|, initialize with $\infty$ for every vertex, recording the minimum weight when the walk
    reaching that vertex, inside a cycle}
        querynum: INT List,
        #{length |V|, initialize with Q($v$) for every vertex, recording the query numbers when the path
    reaching that vertex, inside a cycle}
 4: if |V| = 1 and |E| = 0:
 5:     return Q($v$)
 6: def dfs(G, c, visited):
 7:     for every vertex $v$ connected by a directed edge from $c$:
 8:         if visited[$v$] = false:
 9:             flowcapacity[v] = min(W(v), flowcapacity[c]);
10:             querynum[v] = querynum[c] + Q(v);
11:             r[v] = max(r[v], flowcapacity[v] × querynum[v]);
12:             visited[$v$] = 1;
13:             dfs(G, v, visited);
14:         else: #{There is a cycle finished}
15:             r[v] = max(r[v], r[c] + min(W(v), flowcapacity[c]) * (querynum[c] + Q(v)));
    #{update the length of the longest walk reaching this vertex on this cycle}
16:     return r[c]
17: for every vertex $v$ in V:
18:     initialize the visited, r, flowcapacity, querynum;
19:     $\mathtt{r}_{\mathsf{scc}}$ = max($\mathtt{r}_{\mathsf{scc}}$, dfs(G, v, visited)) ;
20: return $\mathtt{r}_{\mathsf{scc}}$
```

Adaptivity Computation Algorithm on SCC Graph ($\mathsf{AdaptSearch}_{\mathsf{scc}}(\mathtt{G})$**)** This algorithm takes a subgraph of the program-based dependency graph as input, to be precise, the input graph is SCC, and the output is the adaptivity of this SCC. 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. find out all the paths in the input SCC 2. Calculate the adaptivity of every path using our designed adaptivity counting method. 3. Return the maximal adaptivity among all the paths. The step 3 is trivial. Because our input graph is SCC, when we start traversing from a vertex, we will finally go back to this vertex. The paths we find in step 1 are all those with the same starting and ending vertex. The most interesting part is step 2. We discuss as follows.

This algorithm first check if an SCC contains only one vertex without any edge, as in line:4-5 in Algorithm 2. Again, for the SCC containing only one vertex without any edge, as in line:4-5 in Algorithm 2. The adaptivity on this SCC is at most one if it is a query vertex, and zero otherwise. $\mathsf{AdaptSearch}_{\mathsf{scc}}(\mathtt{G})$ return query annotation directly as in line:4-5.
For the SCC containing at least one edge, we compute the adaptivity for each path on the fly of searching for the paths in the recursion algorithm `dfs` designed based on a deep first search strategy from line: 6-16 in $\mathsf{AdaptSearch}_{\mathsf{scc}}(\mathtt{G})$ in Algorithm 2.
As the **Approximation Challenge** discussed above, we want to guarantee the visiting time of each vertex smaller than its weight and compute the adaptivity accurately, in the meantime guarantee the algorithm termination. It uses a capacity limitation and special parameters to achieve it, specifically as follows. Additionally, we are computing the query length rather than sum of the weights. We design a deep first search strategy from line: 6-16 in Algorithm 2, with a capacity limitation and use special parameter to compute the adaptivity.
In order to guarantee the termination, $\mathsf{AdaptSearch}_{\mathsf{scc}}(\mathtt{G})$ terminates the recursion if monitored a cycle, as in line:8 and line:14, through a boolean list `visited`. This guaranteed the termination and solved the **Challenge II.** discussed above.
In order to solve the **Approximation Challenge**, specifically guarantee the visiting times of each vertex by its weight and compute the adaptivity accurately, we use a special parameter `flowcapacity` to track the minimum weight along the path during the searching procedure, and a parameter `querynum` to track the total number of vertices with query annotation 1 along the path in order to compute the query length.

The detail steps of this dfs strategy from line: 3-16 in Algorithm 2, particularly from line: 7-15 on how to use these two special parameters to resolve **Approximation Challenge** is described as follows. `flowcapacity` is a list of symbolic expressions for every vertex, recording the minimum weight when the path reaches that vertex, which is initialized by ∞.

`querynum` is a list of integer with length $|\mathtt{V}|$, which is initialized with $\mathtt{Q}(v)$ for every vertex. For every vertex, it records the total query numbers when the path reaching this vertex.

We maintain the minimum weight for the `flowcapacity`, number of query vertices `querynum` and update the adaptivity for this path `r` alone the path and update the adaptivity reaching this vertex, when traversing on this graph, as in Algorithm 2 from line: 8-13. At line: 15 where this vertex is visited, i.e., this path going back to its starting node, we only update the adaptivity `r` reaching this vertex.
The updating operations during the traversing (in line: 11) and at the end of the traverse (in line: 15), specifically the $\mathtt{flowcapacity[v]} \times \mathtt{querynum[v]}$ computes the query length for this path. it guarantees the visiting times of each vertex on the path reaching a vertex v is no more than the maximum visiting it can be on a qualified walk, through `flowcapacity[v]`, and in the same time compute the query length instead of weighted length accurately through `querynum[v]`. In this way, we resolve the **Approximation Challenge** and in the same time without losing the soundness,

We first initialize some parameters:
`visited` is initialized as a list of 0 for every vertex on this SCC, in order to guarantee the termination;
`r` is initialized as a list of integer with length $|\mathtt{V}|$, initialize with $\mathtt{Q}(v)$ for every vertex. The adaptivity reaching each vertex.
`flowcapacity` a list of symbolic expressions for every vertex, recording the minimum weight when the walk reaching that vertex, which is initialized by ∞.
`querynum` is a list of integer with length $|\mathtt{V}|$, which is initialized with $\mathtt{Q}(v)$ for every vertex. For every vertex, in order to record the total query numbers when the walk reaches a vertex.
Then from line: 5-11, we record the minimum weight and number of query vertices alone the path and update the adaptivity reaching this vertex, and then recursively dfs on all vertices heading out from this vertex.
At line: 12 where this vertex is visited, we only update the adaptivity reaching this vertex and neither recursion nor update the `flowcapacity` and `querynum`.
The updating operation in these two branches, specifically `flowcapacity[v]` $\times$ `querynum[v]` in line: 11 and line: 15 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, through `flowcapacity[v]`. In this way, we resolve the **Approximation Challenge** and in the same time without losing the soundness by using `flowcapacity[v]` $\times$ `querynum[v]` to compute the query length.
Notice here, another special operation we have in the second branch is Non-updating of `querynum` and `flowcapacity`. This guarantees both the accuracy and the soundness, formally in Lemma G.1 in Appendix G.

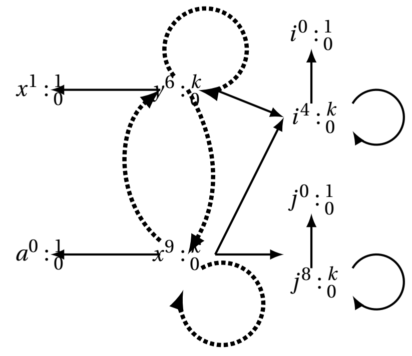
Now, we show an example illustrating how our two updating operations for adaptivity for each path can guarantee both the accuracy and the soundness. Look at a Nested While Loop example program in Figure 6. We first search for a path: $y^6 \to y^6$, and compute the adaptivity for this path as k. Notice here, another special operation we have in the second branch is Non-updating of `querynum` and `flowcapacity`. This guarantees both the accuracy and the soundness. Specifically, if this vertex is visited, it indicates that a cycle is monitored and the traversing on this cycle is finished by going back to this vertex. When we continuously search for walks heading out of this vertex, the minimum weight on this cycle does not affect the walks going out of this vertex that not pass this cycle. However, if we keep recording the minimum weight, then we restrict the visiting times of vertices on a walk by using the minimum weight of vertices not on this walk. Then, it is obviously that this leads to unsoundness. If we update the `flowcapacity[`y^6`]` as k after visiting y^6 the second time on this walk, and continuously visit x^9, then the `flowcapacity[k]` is updated as $\min(k, k^2)$. So the visiting times of x^9 is restricted by k on the walk $y^6 \to y^6 \to x^9$. This restriction excludes the finite walk $y^6 \to y^6 \to x^9 \to x^9$ where y^6 and x^9 visited by k^2 times in the computation. However, the finite walk $y^6 \to y^6 \to x^9 \to x^9$ where y^6 is visited k times and x^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 again, we guarantee that the visiting times og vertices on every searched walk will not be restricted by weights not on this walk, i.e., the soundness. In the last line of this dfs algorithm, line: 16, it returns the adaptivity heading out from its input vertex. By applying this deep first search strategy on every vertex on this SCC, we compute the adaptivity of this SCC by taking the maximum value over every vertex. The soundness is formally guaranteed in Lemma G.1 in Appendix G.

Theorem 4.5 (Soundness of AdaptSearch). *For every program c, given its* Program-Based Dependency Graph $\mathtt{G}_{\mathtt{prog}}$,

$$\mathsf{AdaptSearch}(\mathtt{G}_{\mathtt{prog}}) \geq A_{\mathtt{prog}}(\mathtt{G}_{\mathtt{prog}}).$$

```
nestedWhileMultiVarRecAcross(k) ≜
[i ← k]^0;
[x ← query(χ[0])]^1;
[y ← query(χ[1])]^2;
while [i > 0]^3 do
([i ← i − 1]^4;
[j ← k]^5;
[y ← query(χ(ln(x) + y))]^6;
while [j > 0]^7 do
([j ← j − 1]^8;
[x ← query(χ(ln(y)) + χ[x])]^9))
```

(a)



(b)

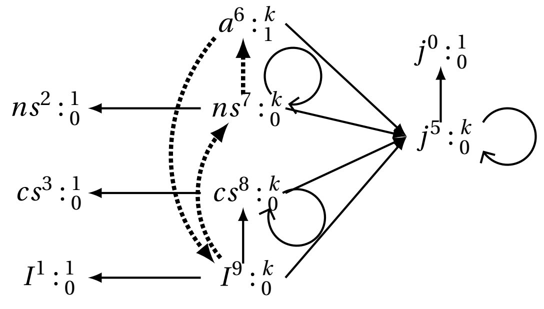
Figure 6: (a) Nested While Loop Example, (b) Execution-Based Dependency Graph, (c) The Static Program-Based Dependency graph.

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  $r[c]$
```

```
multipleRounds(k,c) $\triangleq$
$[j \leftarrow k]^0; [I \leftarrow []]^1;$
$[ns \leftarrow 0]^2; [cs \leftarrow 0]^3;$
while $[j > 0]^4$ do
$\big([j \leftarrow j-1]^5; [a \leftarrow \texttt{query}(I)]^6;$
$[ns \leftarrow \texttt{updnscore}(ns, a)]^7;$
$[cs \leftarrow \texttt{updcscore}(cs, a)]^8;$
$[I \leftarrow \texttt{updI}(I, ns, cs)]^9\big)$
```

(a)



(b)

Figure 7: (a) The simplified multiple rounds example (b) The program-based dependency graph from AdaptFun

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 - multiple rounds algorithm, as in Figure 7(a). 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* $\texttt{query}(\chi[I])$. *After r rounds, the algorithm returns the columns of the hidden database D not specified in the tracking list I. We use functions* $\texttt{updnscore}(p, a)$, $\texttt{updcscore}(p, a)$,$\texttt{update}(I, ns, cs)$ *to simplify the complex update computations of* $Nscore$, $Cscore$ *and the tracking list I, which will not affect our analysis.*

The interesting part here is the query asked in each iteration is not independent any more. The query in one iteration j now depends on the tracking list I from its previous iteration $j-1$, *which is updated by the query result in the same iteration* $j-1$. *The connection between queries from different iterations, which means these queries are adaptively chosen according to our discussion in overview.*

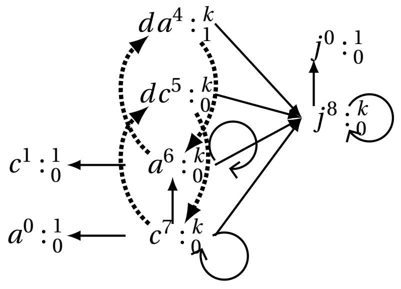
The program-based dependency graph is presented in Figure 7(b). Its execution-based dependency graph has the same graph, except different weight so we do not show it again. We can simply replaces k with a function w_k *which takes a trace and returns the value of k in this trace. The weight* 1 *is replaced as a constant function* w_1 *taking whatever trace and returns* 1 *for the execution-based dependency graph. For consistence, we use* w_k *and* w_1 *for all the examples in this section. As the adaptivity definition in our formal adaptivity model in Definition 20, there is a finite walk along the dashed arrows,* $a^6 \to I^9 \to ns^7 \to \cdots \to ns^7$, *where every vertex is visited* $w_k(\tau_0)$ *times for an initial trace* $\tau_0 \in \mathcal{T}_0(c)$. *There is one vertex* a^6 *visited* $w_k(\tau_0)$ *times with query annotation 1, So we have the adaptivity with* τ_0 *for this program as* $w_k(\tau_0)$.

Next, we show AdaptFun providing the tight upper bound for this example. If first finds a path $a^6 : {}^{k}_{1} \to I^9 : {}^{k}_{0} \to ns^7 : {}^{k}_{0}$ *with three weighted vertices, and then* AdaptSearch *approximate this path to a walk, in which* a^6, I^9, ns^7 *is visited k times. So the estimated adaptivity is k. We know for any initial trace* τ_0 *where* $\langle \tau_0, k \rangle \Downarrow_e v$ *and* $w_k(\tau_0) = v$. *So k from AdaptFun is a tight bound.*

Example 5.2 (Linear Regression Algorithm with Gradient Decent Optimization)**.** *The linear regression algorithm with gradient decent Optimization works well in our AdaptFun as well. Analysis Result:* $A_{\text{prog}}(\texttt{linearRegressionGD(k,rate)}) = k$

```
linearRegressionGD(k,rate) $\triangleq$
$[a \leftarrow 0]^0; [c \leftarrow 0]^1; [j \leftarrow k]^2;$
while $[j > 0]^3$ do
$\Big([da \leftarrow$ query$(-2 * (\chi[1] - (\chi[0] \times a + c)) \times (\chi[0]))]^4;$
$[dc \leftarrow$ query$(-2 * (\chi[1] - (\chi[0] \times a + c)))]^5;$
$[a \leftarrow a -$ rate $* da]^6; [c \leftarrow c -$ rate $* dc]^7;$
$[j \leftarrow j - 1]^8\Big);$
```

(a)



(b)

Figure 8: (a) The linear regression algorithm (b) The program-based dependency graph from AdaptFun

This linear regression algorithm aims to model a linear relationship between a dependent variable y, and an independent variable x, $y = a \times x + c$, specifically 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 every iteration. This training data set contains two columns (can extend to higher dimensional data sets), first column is used as the observed value for the independent variable x, second column is used as the observed label value for the dependent variable y. This algorithm is written in our `Query While` language in Figure 8(a) as `linearRegressionGD(k,rate)`.

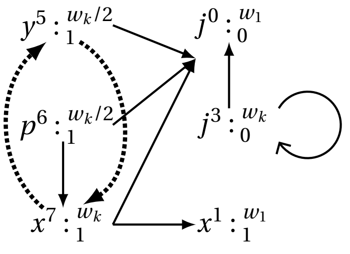
This linear regression algorithm starts from initializing the linear model parameters and the counter variable, and then goes into the training iterations. In each iteration, it computes the differential value w.r.t. parameter a and c respectively, through requesting two queries, query$(-2*(\chi[1]-(\chi[0]\times a+c))\times(\chi[0]))$ and query$(-2*(\chi[1]-(\chi[0]\times a+c)))$ at line 4 and 5. Then, it uses these two differential values stored in variable da and dc to update the linear model parameters a and c. Its the program-based dependency graph is shown in Figure 8(b). Its execution-based dependency graph share the same graph, only needs to change the weight, k into w_k and 1 for w_1 as we do in the previous example. In the execution-based dependency graph, there are multiple walks having the same longest query length. For example, the walk $c^7 \to dc^6 :\to c^7 \to \cdots \to dc^6$ along the dotted arrows, where each vertex is visited $w_k(\tau_0)$ times for an initial trace τ_0. There is actually other walks having the same query length k, the walk $a^7 \to da^6 \to a^7 \to \cdots \to da^6$ along the dotted arrows, where each vertex is visited $w_k(\tau_0)$ times. But it doesn't affect the adaptivity for this program, which is still the maximal query length $w_k(\tau_0)$ with respect to initial trace τ_0. Also, AdaptFun, estimates the adaptivity k for this example. Similarly as the multiple round example, we can show it is a tight bound.

Example 5.3 (Over-approximation Algorithm)**.** *The AdaptFun comes across an over-approximation on the estimation 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.*

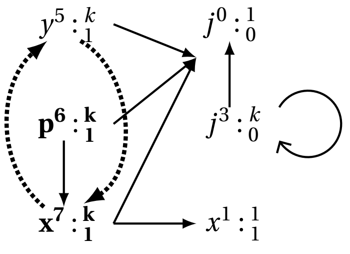
We show the over-approximation, in Figure 9(a), we call it a multiple rounds odd iteration algorithm. In this algorithm, at line 5 of every iteration, a query query$(\chi[x])$ *based on previous query results stored in x is asked by the analyst like in the multiple rounds strategy. The difference is that only the query answers from the even iterations ($i = 0, 2, \cdots$) are used in the query in line 7,* query$(\chi[\ln(y)])$. *Because the execution trace only updates x using the query answers in even iterations, so the answers from odd iterations do not affect the queries in even iterations. From the execution-based dependency graph in Figure 9(b), we can see that the weight for the vertex y^5 is $w_k/2$. a function which takes any initial trace τ_0, return the value of $k/2$ evaluated in τ_0. However, AdaptFun fails to realize that odd iteration will always execute the then branch and even iteration means else branch, so it considers both branches for every iteration. In this sense, the weight estimated for y^5 and p^6 are both k as in Figure 9(c). As a result, AdaptFun estimates the longest walk from Figure 9(c),*

```
multipleRoundsOdd(k) ≜
[j ← k]^0; [x ← query(χ[0])]^1;
while [j > 0]^2 do ([j ← j − 1]^3;
if ([j%2 == 0]^4,
[y ← χ[x]]^5, [p ← χ[x]]^6);
[x ← query(χ(ln(y)))]^7)
```

(a)



(b)

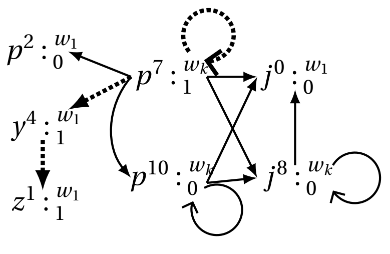

$y^5 : \overset{k}{1}$
$j^0 : \overset{1}{0}$
$\mathbf{p^6} : \overset{\mathbf{k}}{\mathbf{1}}$
$j^3 : \overset{k}{0}$
$\mathbf{x^7} : \overset{\mathbf{k}}{\mathbf{1}}$
$x^1 : \overset{1}{1}$

(c)

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

```
multipleRoundsSingle(k)
[j ← 0]^0; [z ← query(0)]^1; [p ← 0]^2;
 if ([k = 0]^3, [y ← query(z)]^4, [skip]^5);
 while [j ≠ k]^6 do
 ([p ← query(χ[y] + p)]^7; [j ← j + 1]^8
 if ([j ≠ k − 2]^9, [p ← 0]^10, [skip]^10));
```

(a)



(b)

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

$y^5 \to x^7 \to y^5 \to \cdots \to x^7$ *with each vertex visited k times, as the dotted arrows. And the adaptivity computed is* $1 + 2 * k$*, instead of* $1 + k$*.*

Example 5.4 (Over-Defined Adaptivtiy Example)**.** *The program's adaptivity in our formal model, in Definition 20 also comes across an over-approximation on the program's intuitive adaptivity rounds. It is resulted from 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 witness the* variable may-dependency *relation.*

As the program in Figure 10(a), which is a variant of the multiple rounds strategy, named multipleRoundSingle(k) *with input k. In this algorithm, at line 7 of every iteration, a query* query($\chi[y] + p$) *based on previous query results stored in p and y is asked by the analyst like in the multiple rounds strategy. The difference is that only the query answers from the one single iterations* ($j = k - 2$) *are used in this query* query($\chi[y] + p$)*. Because the execution trace updates p using the constant* 0 *for all the iterations where* ($j \neq k - 2$) *at line* 10 *after the query request at line* 7*. In this way, all the query answers stored in p will not be accessed in next query request at line* 7 *in the iterations where* ($j \neq k - 2$)*. Only query answer at one single iteration where* ($j = k - 2$) *will be used in next query request* query($\chi[y] + p$) *at line* 7*. So the adaptivity for this example is* 2*. However, our adaptivity model fails to realize that there is only dependency relation between* p^7 *and* p^7 *in one single iteration, not the others. As shown in the execution-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* $w_k(\tau_0)$ *with a initial trace* τ_0*. As a result, the walk with the longest query length is* $p^7 \to \cdots \to p^7 \to y^4 \to z^1$ *with the vertex* p^7 *visited* $w_k(\tau_0)$*, as the dotted arrows. The adaptivity based on this walk is* $2 + w(\tau_0)$*, instead of* 2*. Though the* AdaptFun *is able to give us* $2 + k$*, as an accurate bound w.r.t this definition.*

5.2 Implementation Results

We implemented AdaptFun as a tool which takes a labeled command as input and outputs an upper bound on the program adaptivity and on the number of query requests. This implementation consists of an abstract control flow graph generation, weight estimation (as presented in Section 4.3.3), edge estimation (as presented in Section 4.3.2) 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, 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 17 example programs with the evaluation results shown in Table 1. 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 $A(c)$ we defined through our formal semantic model above. In the third column, we use k represent the weight function w_k (in program's execution-based dependency graph) which return value of variable k from an initial trace τ_0, same for natural numbers. The last 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.

The first 3 programs we evaluated are `twoRoundsComplete(k)`, `multipleRoundsComplete(k)`, and the `linearRegressionGD(k,rate)` which we discussed in overview and above section. For these examples, $A(c)$ give the accurate adaptivity definition, simultaneously the AdaptFun outputs the tight bounds for both of the adaptivity and query requesting number as expected. But for the forth program `multipleRoundOdd(k)`, AdaptFun outputs an over-approximated upper bound $1 + 2 * k$ 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 in the table from `seq()` to `nestedWhileMultiPathMultiVarRecAcross(k)` are designed for testing the programs under different possible situitions. These programs contain control dependency, data value dependency, the nested while, dependency through multiple variables, dependency across nested loops, and so on. Overall for these examples, our system gives both the accurate adaptivity definition and adaptivity upper bound simultaneously through the dynamic analysis and static analysis. The full programs are defined below from Example 5.5 to Example 5.20.

Table 1: Experimental results of AdaptFun implementation

Program c	adaptivity rounds	$A(c)$	AdaptFun
`twoRoundsComplete(k)`	2	2	$2, k$
`multipleRoundsComplete(k)`	k	k	k, k
`linearRegressionGD(k,rate)`	k	k	$k, 2*k$
`multipleRoundsOdd(k)`	$1+k$	$1+k$	$1+2*k, 1+2*k$
`multipleRoundsSingle(k)`	2	$2+k$	$2+k, 2+k$
`seq()`	4	4	$4, 4$
`seqMultiVar()`	4	4	$4, 4$
`ifValueDependency`	3	3	$3, 3$
`ifControlDependency()`	3	3	$3, 3$
`whileRec(k)`	$1+k$	$1+k$	$1+k$
`whileMultipleVar(k)`	$1+2*k$	$1+2*k$	$1+2*k, 2+3*k$
`whileValueControlDependency(k)`	$1+2*k$	$1+2*k$	$1+2*k, 2+2*k$
`whileMultiplePathValueControlDependency(k)`	$2+k$	$2+k$	$2+k, 1+2*k$
`nestWhileValueDependency(k)`	$2+k^2$	$2+k^2$	$2+k^2, 1+k+k^2$
`nestedWhileRecAcross(k)`	$1+2*k$	$1+2*k$	$1+2*k, 1+k+k^2$
`nestedWhileMultiVarRecAcross(k)`	$1+k+k^2$	$1+k+k^2$	$1+k+k^2, 2+k+k^2$
`nestedWhileMultiPathMultiVarRecAcross(k)`	$1+k+k^2$	$1+k+k^2$	$1+k+k^2, 2+k+k^2$

Example 5.5 (Complete Two Round Algorithm)**.**

$$
\texttt{twoRoundsComplete(k)} \triangleq
\begin{array}{l}
[a \leftarrow []]^1; \\
\left[j \leftarrow k\right]^2; \\
\texttt{while } \left[j > 0\right]^3 \texttt{ do} \\
\Big(\left[x \leftarrow \texttt{query}(\chi[k-j] \cdot \chi[k])\right]^4; \\
\left[j \leftarrow j-1\right]^5; \\
[a \leftarrow x :: a]^6\Big); \\
\left[l \leftarrow (\mathrm{sign}\big(\sum_{i \in [k]} \chi[i] \times \ln \frac{1+a[i]}{1-a[i]}\big))\right]^7
\end{array}
$$

Algorithm 4 A two-round analyst strategy for random data (The example in [1])

```
Require: Mechanism $\mathcal{M}$ with a hidden data set $D \in \{-1,+1\}^{n\times(k+1)} \subset \mathcal{DB}$.
  for $j \in [k]$ do.
      define $q_j(d) = d(j) \cdot d(k)$ where $d \in \{D(i) \mid i = 0, \cdots, n\} \subseteq \{-1,+1\}^{k+1}$.
      let $a_j = \mathcal{M}(q_j)$
      {In the line above, $\mathcal{M}$ computes approx. the exp. value of $q_j$ over $D$. So, $a_j \in [-1,+1]$.}
  define $q_k(d) = d(k) \cdot \mathrm{sign}\big(\sum_{i\in[k]} x(i) \cdot \ln \frac{1+a_i}{1-a_i}\big)$ where $x \in \{-1,+1\}^{k+1}$.
  {In the line above, $\mathrm{sign}(y) = \begin{cases} +1 & \text{if } y \geq 0 \\ -1 & \text{otherwise} \end{cases}$.}
  let $a_{k+1} = \mathcal{M}(q_{k+1})$
  {In the line above, $\mathcal{M}$ computes approx. the exp. value of $q_{k+1}$ over $X$. So, $a_{k+1} \in [-1,+1]$.}
  return $a_{k+1}$.
Ensure: $a_{k+1} \in [-1,+1]$
```

We have seen the two round algorithm above. We show the multiple-round algorithm, which is an advanced algorithm.

Algorithm 5 A multi-round analyst strategy for random data base [1]

Example 5.6 (Complete Multiple Round Algorithm). **Require:** Mechanism $\mathcal{M}$ with a hidden state $X \in [N]^n$ sampled u.a.r., control set size c

```
Define control dataset $C = \{0, 1, \cdots, c-1\}$
Initialize $Nscore(i) = 0$ for $i \in [N]$, $I = \emptyset$ and $Cscore(C(i)) = 0$ for $i \in [c]$
for $j \in [k]$ do
    let $p = \texttt{uniform}(0, 1)$
    define $q(x) = \texttt{bernoulli}(p)$ .
    define $qc(x) = \texttt{bernoulli}(p)$ .
    let $a = \mathcal{M}(q)$
    for $i \in [N]$ do
        $Nscore(i) = Nscore(i) + (a - p) * (q(i) - p)$ if $i \notin I$
    for $i \in [c]$ do
        $Cscore(C(i)) = Cscore(C(i)) + (a - p) * (qc(i) - p)$
    let $I = \{i | i \in [N] \wedge Nscore(i) > \max(Cscore)\}$
    let $D = D \setminus I$
return $D$.
```

```
            $\texttt{multipleRoundsComplete(k, c, N)} \triangleq$
$[j \leftarrow N]^0; [cs \leftarrow 0]^1; [ns \leftarrow 0]^2; [I \leftarrow 0]^3; [w \leftarrow k]^4;$
while $[j > 0]^5$ do
$\Big([j \leftarrow j - 1]^6; [cs \leftarrow 0 + cs]^7; [ns \leftarrow 0 + ns]^8\Big);$
while $[w > 0]^9$ do
$\Big([w \leftarrow w - 1]^{10}; [p \leftarrow c]^{11}; [q \leftarrow c]^{12}; [a \leftarrow \texttt{query}(\chi[I])]^{13};$
$[i \leftarrow N]^{14};$ while $[i > 0]^{15}$ do
$\Big([i \leftarrow i - 1]^{16}; [cs(i) \leftarrow cs(i) + (a - p) * (q - p)]^{17};$
if $([I < i]^{18}, [ns(i) \leftarrow ns(i) + (a - p) * (q - p)]^{19}, [ns \leftarrow ns(i)]^{20})\Big);$
$[i2 \leftarrow N]^{21};$
while $[i2 > 0]^{22}$ do
$\Big([i2 \leftarrow i2 - 1]^{23};$ if $([ns(i2) > \texttt{max}(cs)]^{24}, [I \leftarrow i + I]^{25}, [I \leftarrow I]^{26})\Big)\Big)$
```

(a)

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

Example 5.7 (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.*

```
$\texttt{gradientDecent(step,rate,t,n)} \triangleq$
$[a_1 \leftarrow 0]^0;$
$[a_2 \leftarrow 0]^1;$
$[c \leftarrow 0]^2;$
$[j \leftarrow \texttt{step}]^3;$
$\texttt{while } [j > 0]^4 \texttt{ do}$
$\big([da1 \leftarrow \texttt{query}(-2 * (\chi[2] - (\chi[0]^2 \times a_1 + \chi[1] \times a_2 + c)) \times (\chi[0]))]^5;$
$[da2 \leftarrow \texttt{query}(-2 * (\chi[2] - (\chi[0]^2 \times a_1 + \chi[1] \times a_2 + c)) \times (\chi[1]))]^6;$
$[dc \leftarrow \texttt{query}(-2 * (\chi[2] - (\chi[0]^2 \times a_1 + \chi[1] \times a_2 + c)))]^5;$
$[a_1 \leftarrow a_1 - \texttt{rate} * da1]^7;$
$[a_2 \leftarrow a_2 - \texttt{rate} * da2]^8;$
$[c \leftarrow c - \texttt{rate} * dc]^9;$
$[j \leftarrow j - 1]^{10}\big);$
```

It is easy to see, this approach can be generalized to the regression of a variety of relations in machine learning area.

Example 5.8 (convex optimization Algorithm)**.**

```
$\texttt{gradientDecent(step,rate,t,n)} \triangleq$
$[a \leftarrow []]^0;$
$[j \leftarrow \texttt{step}]^1;$
$\texttt{while } [j > 0 \land d < t]^3 \texttt{ do}$
$\big([d \leftarrow \texttt{query}(2 * (\chi[1] - (\chi[0] \times x)) * (-\chi[0]))]^4;$
$[x \leftarrow x - \texttt{rate} * d]^4;$
$[j \leftarrow j - 1]^5;$
$[a \leftarrow x :: a]^6\big);$
```

Example 5.9 (Sequence with Single Variable Linear Data Value Dependency)**.**

$$\texttt{seq()} \triangleq \begin{array}{l} [x \leftarrow \chi[0]]^0; \\ [y \leftarrow \chi[x+1]]^1; \\ [z \leftarrow \chi[y+1]]^2; \\ [w \leftarrow \chi[z+1]]^3 \end{array}$$

Analysis Result: $A_{\texttt{prog}}(\texttt{seq()}) = 4$

Example 5.10 (Sequence with Multiple Variables Data Value Dependency)**.**

$$\texttt{seqMultiVar()} \triangleq \begin{array}{l} [x \leftarrow \chi[0]]^0; \\ [y \leftarrow \chi[x+1]]^1; \\ [z \leftarrow \chi[y+x]]^2; \\ [w \leftarrow \chi[z+1] \cdot \chi[y]]^3 \end{array}$$

Analysis Result: $A_{\texttt{prog}}(\texttt{seqMultiVar()}) = 4$

Example 5.11 (If with Data-Value Dependency Separated).

$$\texttt{ifValueDependency}(k) \triangleq \begin{array}{l} \left[z \leftarrow \texttt{query}(\chi[0])\right]^0; \\ [x \leftarrow k/2]^1; \\ \texttt{if}\ ([x < 0]^2, \\ \left[y \leftarrow \texttt{query}(\chi[z])\right]^3, \\ \left[y \leftarrow \texttt{query}(\chi[0])\right]^4) \end{array}$$

Analysis Result: $A_{\texttt{prog}}(\texttt{ifControlDependency}()) = 3$

Example 5.12 (If with Data-Control Dependency Overlapped).

$$\texttt{ifControlDependency}() \triangleq \begin{array}{l} \left[z \leftarrow \texttt{query}(\chi[0])\right]^0; \\ \left[x \leftarrow \texttt{query}(\chi[z])\right]^1; \\ \texttt{if}\ ([x < 0]^2, \left[y \leftarrow \texttt{query}(\chi[0] + \chi[1])\right]^3, \left[y \leftarrow \texttt{query}\ (\chi[0])\right]^4) \end{array}$$

Analysis Result: $A_{\texttt{prog}}(\texttt{ifControlDependency}()) = 3$

Example 5.13 (Simple While with Recursive Data-Value Dependency).

$$\texttt{whileRec}(k) \triangleq \begin{array}{l} \left[j \leftarrow k\right]^0; \\ \left[a \leftarrow \texttt{query}(\chi[0])\right]^1; \\ \texttt{while}\ \left[j > 0\right]^2\ \texttt{do} \\ \Big(\left[x \leftarrow \texttt{query}(\chi[a])\right]^3; \\ [a \leftarrow x + a]^4; \\ \left[j \leftarrow j - 1\right]^5\Big) \end{array}$$

Analysis Results: $A_{\texttt{prog}}(\texttt{whileRec}(k)) = 1 + k$

Example 5.14 (Simple While with Multi-Path Data-Value Dependency).

$$\texttt{whileMultiplePath}(k) \triangleq \begin{array}{l} \left[j \leftarrow k\right]^0; \\ \left[x \leftarrow \texttt{query}(\chi[0])\right]^1; \\ \texttt{while}\ \left[j > 0\right]^2\ \texttt{do} \\ \Big(\left[j \leftarrow j - 1\right]^3; \\ \texttt{if}\ (\left[j\%2 == 0\right]^4, \left[y \leftarrow \chi[x]\right]^5, \left[w \leftarrow \chi[x]\right]^6); \\ \left[x \leftarrow \texttt{query}(\chi(\ln(y)))\right]^7\Big) \end{array}$$

Analysis Results: $A_{\texttt{prog}}(\texttt{whileMultiplePath}(k)) = 1 + 2 * k$ *–> Over-Approximated*

Example 5.15 (Simple While with Recursive Multiple-Variable Data-Value Dependency).

$$\texttt{whileMultipleVar}(k) \triangleq \begin{array}{l} \left[j \leftarrow k\right]^0; \\ \left[x \leftarrow \texttt{query}(\chi[0])\right]^1; \\ \left[y \leftarrow \texttt{query}(\chi[1])\right]^2; \\ \texttt{while}\ \left[j > 0\right]^3\ \texttt{do} \\ \Big(\left[j \leftarrow j - 1\right]^4; \\ \left[z \leftarrow \texttt{query}(\chi(x + \ln(y)))\right]^5; \\ \left[x \leftarrow \texttt{query}(\chi[z])\right]^6; \\ \left[y \leftarrow \texttt{query}(\chi[z])\right]^7\Big) \end{array}$$

Analysis Results: $A_{\texttt{prog}}(\texttt{whileMultipleVar}(k)) = 1 + 2 * k$

Example 5.16 (Simple While with Data-Value and Data-Control Dependency).

$$\texttt{whileValueControlDependency}() \triangleq \begin{array}{l} \left[x \leftarrow \texttt{query}(\chi[0])\right]^0; \\ \left[z \leftarrow \texttt{query}(\chi[0])\right]^1; \\ \texttt{while}\ [x > 0]^2\ \texttt{do} \\ \Big(\left[x \leftarrow \texttt{query}(\chi(z))\right]^3; \\ \left[z \leftarrow \texttt{query}(\chi(x))\right]^4\Big) \end{array}$$

Analysis Results: $A_{\texttt{prog}}(\texttt{whileValueControlDependency}(k)) = 1 + 2 * k$

Example 5.17 (Simple While with MultiplePath Data-Value and Data-Control Dependency).

$$\begin{array}{l} \texttt{whileMultiplePathValueControlDependency}(k) \triangleq \\ \left[x \leftarrow \texttt{query}(k)\right]^0; \\ \left[y \leftarrow 0\right]^1; \\ \texttt{while}\ [x > 0]^2\ \texttt{do} \\ \Big(\texttt{if}\ (\left[y > 0\right]^3, \left[y \leftarrow \texttt{query}(\chi[12])\right]^4, \left[w \leftarrow \texttt{query}(\chi[9])\right]^5); \\ [x \leftarrow x - 1]^6\Big); \\ \left[y \leftarrow \texttt{query}(\chi(\ln(y)))\right]^7 \end{array}$$

Analysis Results: $A_{\texttt{prog}}(\texttt{whileMultiplePathValueControlDependency}(k)) = 2 + k$

Example 5.18 (Nested While with Recursive Data-Value Dependency).

$$\texttt{nestWhileValueDependency}(k) \triangleq \begin{array}{l} [i \leftarrow k]^0; \\ \left[x \leftarrow \texttt{query}(\chi[0])\right]^1; \\ \texttt{while}\ [i > 0]^2\ \texttt{do} \\ \Big([i \leftarrow i - 1]^3; \\ \left[j \leftarrow k\right]^4; \\ \left[y \leftarrow \texttt{query}(\chi(\ln(x)))\right]^5; \\ \texttt{while}\ \left[j > 0\right]^6\ \texttt{do} \\ \Big(\left[j \leftarrow j - 1\right]^7; \\ \left[x \leftarrow \texttt{query}(\chi(\ln(x)))\right]^8\Big)\Big) \end{array}$$

Analysis Results: $A_{\texttt{prog}}(\texttt{nestWhileValueDependency}(k)) = 2 + k^2$

Example 5.19 (Nested While with Nested Recursive Data-Value Dependency Across Outer and Inner Loop).

$$\texttt{nestedWhileRecAcross}(k) \triangleq \begin{array}{l} [i \leftarrow k]^0; \\ \left[x \leftarrow \texttt{query}(\chi[0])\right]^1; \\ \texttt{while}\ [i > 0]^2\ \texttt{do} \\ \Big([i \leftarrow i - 1]^3; \\ \left[j \leftarrow k\right]^4; \\ \texttt{while}\ \left[j > 0\right]^5\ \texttt{do} \\ \Big(\left[j \leftarrow j - 1\right]^6; \\ \left[y \leftarrow \texttt{query}(\chi(x) + \chi(1))\right]^7\Big); \\ \left[x \leftarrow \texttt{query}(\chi(\ln(y)))\right]^8\Big) \end{array}$$

Analysis Results: $A_{\texttt{prog}}(\texttt{nestedWhileRecAcross}(k)) = 1 + 2 * k$

Example 5.20 (Nested While with Nested Recursive Multiple Variable Data-Value Dependency Across Outer and Inner Loop)**.**

$$
\texttt{nestedWhileMultiVarRecAcross}(k) \triangleq
\begin{array}{l}
[i \leftarrow k]^{0}; \\
\left[x \leftarrow \texttt{query}(\chi[0])\right]^{1}; \\
\left[y \leftarrow \texttt{query}(\chi[1])\right]^{2}; \\
\texttt{while } [i > 0]^{3} \texttt{ do} \\
\Big([i \leftarrow i - 1]^{4}; \\
\left[j \leftarrow k\right]^{5}; \\
\left[y \leftarrow \texttt{query}(\chi(\ln(x) + y))\right]^{6}; \\
\texttt{while } \left[j > 0\right]^{7} \texttt{ do} \\
\Big(\left[j \leftarrow j - 1\right]^{8}; \\
\left[x \leftarrow \texttt{query}(\chi(\ln(y)) + \chi[x])\right]^{9}\Big)\Big)
\end{array}
$$

Analysis Results: $A_{\texttt{prog}}(\texttt{nestedWhileMultiVarRecAcross}(k)) = 1 + k + k^2$
Reachability Bound Analysis Results:
*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 5.21 (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.*

$$
\begin{array}{l}
\texttt{nestedWhileMultiPathMultiVarRecAcross}(k) \triangleq \\
[i \leftarrow k]^{0}; \\
\left[x \leftarrow \texttt{query}(\chi[0])\right]^{1}; \\
\left[y \leftarrow \texttt{query}(\chi[1])\right]^{2}; \\
\texttt{while } [i > 0]^{3} \texttt{ do} \\
\Big([i \leftarrow i - 1]^{4}; \\
\left[j \leftarrow k\right]^{5}; \\
\texttt{if } ([x > 0]^{6}, \left[y \leftarrow \texttt{query}(\chi(\ln(x) + y))\right]^{7}, \left[y \leftarrow \texttt{query}(\chi(x))\right]^{8}); \\
\texttt{while } \left[j > 0\right]^{9} \texttt{ do} \\
\Big(\left[j \leftarrow j - 1\right]^{10}; \\
\left[x \leftarrow \texttt{query}(\chi(\ln(y)) + \chi[x])\right]^{11}\Big)\Big)
\end{array}
$$

Analysis Results: $A_{\texttt{prog}}(\texttt{nestedWhileMultiPathMultiVarRecAcross}(k)) = 1 + k + k^2$
Reachability Bound Analysis Results:

```
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
```

Appendices

A Proofs of Lemmas in Section 1, 2 and 3

Lemma A.1 (Uniqueness of the Labeled Variables)**.** *For every program $c \in \mathcal{C}$ and every two labeled variables such that $x^i, y^j \in \mathbb{LV}(c)$, then $x^i \neq y^j$.*

$$\forall c \in \mathcal{C}, x^i, y^j \in \mathcal{L} \; . \; x^i, y^j \in \mathbb{LV}(c) \implies x^i \neq y^j.$$

Proof. □

Lemma A.2 (Trace Non-Decreasing)**.** *For every program $c \in \mathcal{C}$ and traces $\tau, \tau' \in \mathcal{T}$, if $\langle c, \tau \rangle \to^* \langle \texttt{skip}, \tau' \rangle$, then there exists a trace $\tau'' \in \mathcal{T}$ with $\tau {++} \tau'' = \tau'$*

$$\forall \tau, \tau' \in \mathcal{T}, c \; . \; \langle c, \tau \rangle \to^* \langle \texttt{skip}, \tau' \rangle \implies \exists \tau'' \in \mathcal{T} \; . \; \tau {++} \tau'' = \tau'$$

Proof. Taking arbitrary trace $\tau \in \mathcal{T}$, by induction on program c, we have the following cases:

case: $c = [x \leftarrow e]^l$
By the evaluation rule assn, we have $\langle [x \leftarrow a]^l, \tau \rangle \to \langle \texttt{skip}, \tau :: (x, l, v, \bullet) \rangle$, for some $v \in \mathbb{N}$.
Picking $\tau' = \tau :: (x, l, v, \bullet)$ and $\tau'' = [(x, l, v, \bullet)]$, it is obvious that $\tau {++} \tau'' = \tau'$.
This case is proved.

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

case: $\texttt{while}\, [b]^{l_w}\, \texttt{do}\, c$
By the first rule applied to c, there are two cases:

sub-case: while-t
If the first rule applied to is while-t, we have
$\langle \texttt{while}\, [b]^{l_w}\, \texttt{do}\, c_w, \tau \rangle \to \langle c_w; \texttt{while}\, [b]^{l_w}\, \texttt{do}\, c_w, \tau :: (b, l_w, \texttt{true}, \bullet) \rangle$ (1).
Let $\tau'_w \in \mathcal{T}$ be the trace satisfying following execution:
$\langle c_w, \tau :: (b, l_w, \texttt{true}, \bullet) \rangle \xrightarrow{*} \langle \texttt{skip}, \tau'_w \rangle$
By induction hypothesis on sub program c_w with starting trace $\tau :: (b, l_w, \texttt{true}, \bullet)$ and ending trace τ'_w, we know there exist $\tau_w \in \mathcal{T}$ such that $\tau'_w = \tau :: (b, l_w, \texttt{true}, \bullet) {++} \tau_w$.
Then we have the following execution continued from (1):
$\langle \texttt{while}\, [b]^{l_w}\, \texttt{do}\, c_w, \tau \rangle \to \langle c_w; \texttt{while}\, [b]^{l_w}\, \texttt{do}\, c_w, \tau :: (b, l_w, \texttt{true}, \bullet) \rangle \xrightarrow{*} \langle \texttt{while}\, [b]^{l_w}\, \texttt{do}\, c_w, \tau :: (b, l_w, \texttt{true}, \bullet) {++} \tau_w \rangle$ (2
By repeating the execution (1) and (2) until the program is evaluated into skip, with trace $\tau^{i'}_w$ for $i = 1, \cdots, nn \geq 1$ in each iteration, we know in the $i - th$ iteration, there exists $\tau^i_w \in \mathcal{T}$ such that $\tau^{i'}_w = \tau^{(i-1)'}_w :: (b, l_w, \texttt{true}, \bullet) + + \tau^{i'}_w$
Then we have the following execution:
$\langle \texttt{while}\, [b]^{l_w}\, \texttt{do}\, c_w, \tau \rangle \to \langle c_w; \texttt{while}\, [b]^{l_w}\, \texttt{do}\, c_w, \tau :: (b, l_w, \texttt{true}, \bullet) \rangle \xrightarrow{*} \langle \texttt{while}\, [b]^{l_w}\, \texttt{do}\, c_w, \tau^{n'}_w \rangle \to^{\text{while-f}}$ $\langle \texttt{skip}, \tau^{n'}_w :: (b, l_w, \texttt{false}, \bullet) \rangle$ and $\tau^{n'}_w = \tau :: (b, l_w, \texttt{true}, \bullet) {++} \tau^1_w :: \cdots :: (b, l_w, \texttt{true}, \bullet) {++} \tau^n_w$.
Picking $\tau' = \tau^{n'}_w :: (b, l_w, \texttt{false}, \bullet)$ and $\tau'' = [(b, l_w, \texttt{true}, \bullet)] {++} \tau^1_w :: \cdots :: (b, l_w, \texttt{true}, \bullet) {++} \tau^n_w$, we have $\tau + + \tau'' = \tau'$.
This case is proved.

sub-case: while-f
If the first rule applied to c is while-f, we have
$\langle \texttt{while}\, [b]^{l_w}\, \texttt{do}\, c_w, \tau \rangle \to^{\text{while-f}} \langle \texttt{skip}, \tau :: (b, l_w, \texttt{false}, \bullet) \rangle$, In this case, picking $\tau' = \tau :: (b, l_w, \texttt{false}, \bullet)$

and $\tau'' = [(b, l_w, \texttt{false}, \bullet)]$, it is obvious that $\tau_{++}\tau'' = \tau'$.
This case is proved.

case: $\texttt{if}\ ([b]^l, c_t, c_f)$
This case is proved in the same way as **case:** $c = \texttt{while}\ [b]^l\ \texttt{do}\ c$.

case: $c = c_{s1}; c_{s2}$
By the induction hypothesis on c_{s1} and c_{s2} separately, we have this case proved. □

Corollary A.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 ϵ and ϵ' equivalent but may differ in their query value.*

$$\forall \epsilon \in \mathcal{E}, \tau \in \mathcal{T} . \epsilon \in \tau \implies \exists \tau_1, \tau_2 \in \mathcal{T}, \epsilon' \in \mathcal{E} . (\epsilon \in \epsilon') \wedge \tau_{1++}[\epsilon']_{++}\tau_2 = \tau$$

Proof. By unfolding the $\mathsf{aq} \in_{\mathsf{aq}} t$, we have the following cases:

case: $t = []$
The hypothesis is $\texttt{false}$, this case is proved.

case: $t = \mathsf{aq}' :: t' \wedge \mathsf{aq}' =_{\mathsf{aq}} \mathsf{aq}$
Let $t_1 = []$ and $t_2 = t'$, by unfolding the list concatenation operation, we have:

$$t_1 + +[\mathsf{aq}'] + +t_2 = [] + +[\mathsf{aq}'] + +t' = t$$

Since $\mathsf{aq}' =_{\mathsf{aq}} \mathsf{aq}$ by the hypothesis, this case is proved.

case: $t = \mathsf{aq}' :: t' \wedge \mathsf{aq}' \neq_{\mathsf{aq}} \mathsf{aq}$
By induction hypothesis on $\mathsf{aq} \in_{\mathsf{aq}} t'$, we know:

$$\exists t_1', t_2', \mathsf{aq}''. \ s.t., \ (\mathsf{aq} =_{\mathsf{aq}} \mathsf{aq}'') \wedge t_1' + +[\mathsf{aq}''] + +t_2' = t'$$

Let $t_1 = \mathsf{aq}' :: t_1'$ and $t_2 = t_2'$, by unfolding the list concatenation operation, we have:

$$t_1 + +[\mathsf{aq}''] + +t_2 = (\mathsf{aq}' :: t_1') + +[\mathsf{aq}''] + +t_2' = \mathsf{aq}' :: t' = t$$

Since $\mathsf{aq}'' =_{\mathsf{aq}} \mathsf{aq}$ by the hypothesis, this case is proved. □

B Soundness of AdaptFun

Theorem B.1 (Soundness of the AdaptFun). *Given a program c, we have:*

$$\forall \tau \in \mathcal{T} \,.\, \langle A_{\mathtt{prog}}(c), \tau \rangle \Downarrow_e n \implies n \geq A(c)(\tau)$$

Proof Summary:
construct the program-based graph $\mathtt{G}_{\mathtt{prog}}(c) = (\mathtt{V}_{\mathtt{prog}}, \mathtt{E}_{\mathtt{prog}}, \mathtt{W}_{\mathtt{prog}}, \mathtt{Q}_{\mathtt{prog}})$
and trace-based graph $\mathtt{G}_{\mathtt{trace}}(c) = (\mathtt{V}_{\mathtt{trace}}, \mathtt{E}_{\mathtt{trace}}, \mathtt{W}_{\mathtt{trace}}, \mathtt{Q}_{\mathtt{trace}})$
1. prove the one-on-one mapping from $\mathtt{V}_{\mathtt{prog}}$ to $\mathtt{V}_{\mathtt{trace}}$, in Lemma C.1;
2. prove the total map from $\mathtt{E}_{\mathtt{trace}}$ to $\mathtt{E}_{\mathtt{prog}}$, in Lemma C.2;
3. prove that the weight of every vertex in $\mathtt{G}_{\mathtt{trace}}$ is bounded by the weight of the same vertex in $\mathtt{G}_{\mathtt{prog}}$, in Lemma B.3;
4. prove the one-on-one mapping from $\mathtt{Q}_{\mathtt{prog}}$ to $\mathtt{Q}_{\mathtt{trace}}$, in Lemma B.4;
5. show every walk in $\mathcal{WK}(\mathtt{G}_{\mathtt{trace}})$ is bounded by a walk in $\mathcal{WK}(\mathtt{G}_{\mathtt{prog}})$ of the same $\mathtt{len}^{\mathtt{q}}$.
6. get the conclusion that $A(c)$ is bounded by the $A_{\mathtt{prog}}(c)$.

Proof. Given a program c, we construct its
program-based graph $\mathtt{G}_{\mathtt{prog}}(c) = (\mathtt{V}_{\mathtt{prog}}, \mathtt{E}_{\mathtt{prog}}, \mathtt{W}_{\mathtt{prog}}, \mathtt{Q}_{\mathtt{prog}})$ by Definition 27
and trace-based graph $\mathtt{G}_{\mathtt{trace}}(c) = (\mathtt{V}_{\mathtt{trace}}, \mathtt{E}_{\mathtt{trace}}, \mathtt{W}_{\mathtt{trace}}, \mathtt{Q}_{\mathtt{trace}})$ by Definition 17.
The parameter (c) for the components in the two graphs are omitted for concise.
According to the Definition 30 and Definition 20, it is sufficient to show:

$$\forall \tau \in \mathcal{T} \,.\, \langle \max\big\{\mathtt{len}^{\mathtt{q}}(k) \mid k \in \mathcal{WK}(\mathtt{G}_{\mathtt{prog}}(c))\big\}, \tau \rangle \Downarrow_e n \implies n \geq \max\big\{\mathtt{len}^{\mathtt{q}}(k)(\tau) \mid k \in \mathcal{WK}(\mathtt{G}_{\mathtt{trace}}(c))\big\}$$

Then it is sufficient to show that:

$$\forall k_t \in \mathcal{WK}(\mathtt{G}_{\mathtt{trace}}(c), \exists k_p \in \mathcal{WK}(\mathtt{G}_{\mathtt{prog}}(c)) \,.\, \forall \tau \in \mathcal{T} \,.\, \mathtt{len}^{\mathtt{q}}(k_p), \tau \Downarrow_e n \implies n \geq \mathtt{len}^{\mathtt{q}}(k_t(\tau))$$

Let $k_t \in \mathcal{WK}(\mathtt{G}_{\mathtt{trace}}(c))$ be an arbitrary walk in $\mathtt{G}_{\mathtt{trace}}(c)$, and $\tau \in \mathcal{T}$ be arbitrary trace.
Then, let $(e_{p1}, \cdots, e_{p(n-1)})$ and $(v_1, \cdots, v_n)$ be the edges and vertices sequence for $k_t(\tau)$.
By Lemma C.1 and Lemma C.2, we know

$$\forall e_i \in k_t \,.\, e_i = (v_i, v_{i+1}) \implies \exists e_{pi} \,.\, e_{pi} = (v_i, v_{i+1}) \land e_{pi} \in \mathtt{E}_{\mathtt{prog}}$$

Then we construct a walk k_p with an edge sequence $(e_{p1}, \cdots, e_{p(n-1)})$ with a vertices sequence $(v_1, \cdots, v_n)$ where $e_{pi} = (v_i, v_{i+1}) \in \mathtt{E}_{\mathtt{prog}}$ for all $e_{pi} \in (e_{p1}, \cdots, e_{p(n-1)})$.
Let $n \in \mathbb{N}$ such that $\langle \mathtt{len}^{\mathtt{q}}(k_p), \tau \rangle \Downarrow_e n$, then, it is sufficient to show

$$k_p \in \mathtt{G}_{\mathtt{prog}}(c) \land n \geq \mathtt{len}^{\mathtt{q}}(k_t)(\tau)$$

To show $k_p \in \mathtt{G}_{\mathtt{prog}}(c)$, by Definition 18 for finite walk, we know

$$\forall v_i \in (v_1, \cdots, v_n), (v_i, w_i) \in \mathtt{W}_{\mathtt{trace}}(c) \,.\, \mathtt{visit}((v_1, \cdots, v_n), (v_i)) \leq w_i(\tau)$$

By Lemma B.3, we know for every

$$\forall v_i \in (v_1, \cdots, v_n), (v_i, w_i) \in \mathtt{W}_{\mathtt{prog}}(c), n_i \in \mathbb{N} \,.\, \langle w_i, \tau \rangle \Downarrow_e n_i \implies w_i(\tau) \leq n_i \ (\star)$$

Then, by Definition 28, we know the occurrence times for every $v_i \in (v_1, \cdots, v_n)$ is bound by the arithmetic expression w_i where $(v_i, w_i) \in \mathtt{W}_{\mathtt{prog}}(c)$.

So we have $k_p \in \mathcal{WK}(\mathtt{G_{prog}})$ proved.
In order to show $n \geq \mathtt{len^q}(k_t)(\tau)$, it is sufficient to show

$$\forall v_i \in (v_1, \cdots, v_n), (v_i, w_i) \in \mathtt{W_{prog}}(c), (v_i, w_i') \in \mathtt{W_{trace}}(c), n_i \in \mathbb{N} \,.\, \langle w_i, \tau \rangle \Downarrow_e n_i$$
$$\implies \sum_{\mathtt{Q_{trace}}(c)(v_i)=1} w_i'(\tau) \leq \sum_{\mathtt{Q_{prog}}(c)(v_i)=1} n_i$$

By Lemma B.4 and Definition 29, we know for every v_i, $\mathtt{Q_{trace}}(c)(v_i) = \mathtt{Q_{prog}}(c)(v_i)$
Then by $(\star)$, we know $\sum_{\mathtt{Q_{trace}}(c)(v_i)=1} w_i'(\tau) \leq \sum_{\mathtt{Q_{prog}}(c)(v_i)=1} n_i$.
Then we have $n \geq \mathtt{len^q}(k_t)(\tau)$ proved.
This theorem is proved. □

The following are the four lemmas used in the proof of Theorem C.1 above, showing the correspondence properties between the program based graph and trace based graph.

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

$$\forall c \in \mathcal{C}, v \in \mathcal{VAR} \times \mathbb{N} \,.\, \mathtt{G_{prog}}(c) = (\mathtt{V_{prog}}, \mathtt{E_{prog}}, \mathtt{W_{prog}}, \mathtt{Q_{prog}}) \land \mathtt{G_{trace}}(c) = (\mathtt{V_{trace}}, \mathtt{E_{trace}}, \mathtt{W_{trace}}, \mathtt{Q_{trace}})$$
$$\implies v \in \mathtt{V_{prog}} \iff v \in \mathtt{V_{trace}}$$

Proof. Proof Summary: Proving by Definition 27 and Definition 17.
Taking arbitrary program c, by Definition 27 and Definition 17, we have
its program-based graph $\mathtt{G_{prog}}(c) = (\mathtt{V_{prog}}, \mathtt{E_{prog}}, \mathtt{W_{prog}}, \mathtt{Q_{prog}})$
and trace-based graph $\mathtt{G_{trace}}(c) = (\mathtt{V_{trace}}, \mathtt{E_{trace}}, \mathtt{W_{trace}}, \mathtt{Q_{trace}})$.
By the two definitions, we also know $\mathtt{V_{trace}} = \mathbb{LV}_c$ and $\mathtt{V_{prog}} = \mathbb{LV}_c$.
Then we know $\mathtt{V_{trace}} = \mathtt{V_{prog}}$, i.e., for arbitrary $v \in \mathcal{VAR} \times \mathbb{N}$, $v \in \mathtt{V_{prog}} \iff v \in \mathtt{V_{trace}}$. □

Lemma B.2 (Mapping from Egdes of $\mathtt{G_{trace}}$ to $\mathtt{G_{prog}}$). *Given a program c with its program-based graph $\mathtt{G_{prog}}(c) = (\mathtt{V_{prog}}, \mathtt{E_{prog}}, \mathtt{W_{prog}}, \mathtt{Q_{prog}})$ and trace-based graph $\mathtt{G_{trace}}(c) = (\mathtt{V_{trace}}, \mathtt{E_{trace}}, \mathtt{W_{trace}}, \mathtt{Q_{trace}})$, then for every $e = (v_1, v_2) \in \mathtt{E_{trace}}$, there exists an edge $e' = (v_1', v_2') \in \mathtt{E_{prog}}$ with $v_1 = v_1' \land v_2 = v_2'$.*

$$\forall c \in \mathcal{C} \,.\, \mathtt{G_{prog}}(c) = (\mathtt{V_{prog}}, \mathtt{E_{prog}}, \mathtt{W_{prog}}, \mathtt{Q_{prog}}) \land \mathtt{G_{trace}}(c) = (\mathtt{V_{trace}}, \mathtt{E_{trace}}, \mathtt{W_{trace}}, \mathtt{Q_{trace}})$$
$$\implies \forall e = (v_1, v_2) \in \mathtt{E_{trace}} \,.\, \exists e' \in \mathtt{E_{prog}} \,.\, e' = (v_1, v_2)$$

Proof. Proof Summary: Proving by Lemma C.1, Lemma D.1 Definition 27 and Definition 17
Taking arbitrary program c, by Definition 27 and Definition 17, we have
its program-based graph $\mathtt{G_{prog}}(c) = (\mathtt{V_{prog}}, \mathtt{E_{prog}}, \mathtt{W_{prog}}, \mathtt{Q_{prog}})$
and trace-based graph $\mathtt{G_{trace}}(c) = (\mathtt{V_{trace}}, \mathtt{E_{trace}}, \mathtt{W_{trace}}, \mathtt{Q_{trace}})$.
Taking arbitrary edge $e = (x^i, y^j) \in \mathtt{E_{trace}}$, it is sufficient to show $(x^i, y^j) \in \mathtt{E_{prog}}$.
By Lemma C.1, we know $x^i, y^j \in \mathtt{V_{prog}}$.
By definition of $\mathtt{E_{trace}}$, we know $\mathsf{DEP_{var}}(x^i, y^j, c)$.
By Theorem D.1, we know $\exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c \,.\, n \geq 0 \land \mathtt{flowsTo}(x^i, z_1^{r_1}, c) \land \cdots \land \mathtt{flowsTo}(z_n^{r_n}, y^j, c)$.
Then by definition of $\mathtt{E_{prog}}$, we know $(x^i, y^j) \in \mathtt{E_{prog}}$. This Lemma is proved. □

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

$$\forall c \in \mathcal{C} \,.\, \mathtt{G_{prog}}(c) = (\mathtt{V_{prog}}, \mathtt{E_{prog}}, \mathtt{W_{prog}}, \mathtt{Q_{prog}}) \land \mathtt{G_{trace}}(c) = (\mathtt{V_{trace}}, \mathtt{E_{trace}}, \mathtt{W_{trace}}, \mathtt{Q_{trace}})$$
$$\implies \forall v \in \mathtt{V_{trace}} \,.\, v \in \mathtt{V_{prog}} \land \mathtt{W_{trace}}(v) \leq \mathtt{W_{prog}}(v)$$

$$\forall c \in \mathcal{C} \,.\, \mathtt{G_{prog}}(c) = (\mathtt{V_{prog}}, \mathtt{E_{prog}}, \mathtt{W_{prog}}, \mathtt{Q_{prog}}) \land \mathtt{G_{trace}}(c) = (\mathtt{V_{trace}}, \mathtt{E_{trace}}, \mathtt{W_{trace}}, \mathtt{Q_{trace}})$$
$$\implies \forall (x^l, w_t) \in \mathtt{W_{trace}}, (x^l, w_p) \in \mathtt{W_{prog}}, \tau, \tau' \in \mathcal{T}, v \in \mathbb{N} \,.\, \langle w_p, \tau \rangle \Downarrow_e v \implies w_t(\tau) \le v$$

Proof. Proof Summary: Proving by Definition 27, Definition 17 and relying on the soundness of Reachability Bound Analysis.
Taking arbitrary program c, by Definition 27 and Definition 17, we have
its program-based graph $\mathtt{G_{prog}}(c) = (\mathtt{V_{prog}}, \mathtt{E_{prog}}, \mathtt{W_{prog}}, \mathtt{Q_{prog}})$
and trace-based graph $\mathtt{G_{trace}}(c) = (\mathtt{V_{trace}}, \mathtt{E_{trace}}, \mathtt{W_{trace}}, \mathtt{Q_{trace}})$.
Taking arbitrary $(x^l, w_t) \in \mathtt{W_{trace}}, (x^l, w_p) \in \mathtt{W_{prog}}, \tau, \tau' \in \mathcal{T}$, satisfying:
$\langle c, \tau \rangle \to^* \langle \mathtt{skip}, \tau {+\!+} \tau' \rangle \land \langle w_p, \tau \rangle \Downarrow_e v$
By soundness of reachability bound analysis in Theorem **??**, we know $\mathtt{cnt}(\tau', l) \le v$
By definition 17, we know $w_t(\tau) = \mathtt{cnt}(\tau', l)$, then we have $w_t(\tau) \le v$ and this is proved. □

Lemma B.4 (One-on-One Mapping for Query Vertices). *Given a program c with its program-based graph* $\mathtt{G_{prog}}(c) = (\mathtt{V_{prog}}, \mathtt{E_{prog}}, \mathtt{W_{prog}}, \mathtt{Q_{prog}})$ *and trace-based graph* $\mathtt{G_{trace}}(c) = (\mathtt{V_{trace}}, \mathtt{E_{trace}}, \mathtt{W_{trace}}, \mathtt{Q_{trace}})$, *then for every* $(x^i, n) \in \mathcal{VAR} \times \mathbb{N} \times \{0,1\}$, $(x^i, n) \in \mathtt{Q_{trace}}$ *if and only if* $(x^i, n) \in \mathtt{Q_{prog}}$.

$$\forall c \in \mathcal{C}, (x^i, n) \in \mathcal{VAR} \times \mathbb{N} \times \{0,1\} \,.$$
$$\mathtt{G_{prog}}(c) = (\mathtt{V_{prog}}, \mathtt{E_{prog}}, \mathtt{W_{prog}}, \mathtt{Q_{prog}}) \land \mathtt{G_{trace}}(c) = (\mathtt{V_{trace}}, \mathtt{E_{trace}}, \mathtt{W_{trace}}, \mathtt{Q_{trace}})$$
$$\implies (x^i, n) \in \mathtt{Q_{trace}} \iff (x^i, n) \in \mathtt{Q_{prog}}$$

Proof. Proving by Definition 27, Definition 17.
Taking arbitrary program c, by Definition 27 and Definition 17, we have
its program-based graph $\mathtt{G_{prog}}(c) = (\mathtt{V_{prog}}, \mathtt{E_{prog}}, \mathtt{W_{prog}}, \mathtt{Q_{prog}})$
and trace-based graph $\mathtt{G_{trace}}(c) = (\mathtt{V_{trace}}, \mathtt{E_{trace}}, \mathtt{W_{trace}}, \mathtt{Q_{trace}})$.
By the two definitions, we also know $\mathtt{Q_{trace}} = \mathtt{Q_{prog}}$, i.e., for arbitrary $(x^i, n) \in \mathcal{VAR} \times \mathbb{N} \times \{0,1\}$, $(x^i, n) \in \mathtt{Q_{trace}} \iff (x^i, n) \in \mathtt{Q_{prog}}$.
This lemma is proved. ■

C Soundness of AdaptFun with Dependency Graph and Adaptivity Extension

Theorem C.1 (Soundness of the AdaptFun). *Given a program c, we have:*

$$\forall \tau \in \mathcal{T} . \langle A_{\mathtt{prog}}(c), \tau\rangle \Downarrow_e n \implies n \geq A(c)(\tau)$$

Proof. Given a program c, we construct its
program-based graph $\mathtt{G}_{\mathtt{prog}}(c) = (\mathtt{V}_{\mathtt{prog}}, \mathtt{E}_{\mathtt{prog}})$ by Definition 27
and trace-based graph $\mathtt{G}_{\mathtt{trace}}(c) = (\mathtt{V}_{\mathtt{trace}}, \mathtt{E}_{\mathtt{trace}})$ by Definition 17.
The parameter (c) for the components in the two graphs are omitted for concise.
According to the Definition 30 and Definition 20, it is sufficient to show:

$$\forall \tau \in \mathcal{T} . \langle \max\{\mathtt{len}^{\mathtt{q}}(k) \mid k \in \mathcal{WK}(\mathtt{G}_{\mathtt{prog}}(c))\}, \tau\rangle \Downarrow_e n \implies n \geq \max\{\mathtt{len}^{\mathtt{q}}(k)(\tau) \mid k \in \mathcal{WK}(\mathtt{G}_{\mathtt{trace}}(c))\}$$

Then it is sufficient to show that:

$$\forall k_t \in \mathcal{WK}(\mathtt{G}_{\mathtt{trace}}(c)), \exists k_p \in \mathcal{WK}(\mathtt{G}_{\mathtt{prog}}(c)) . \forall \tau \in \mathcal{T} . \mathtt{len}^{\mathtt{q}}(k_p), \tau \Downarrow_e n \implies n \geq \mathtt{len}^{\mathtt{q}}(k_t(\tau))$$

Let $k_t \in \mathcal{WK}(\mathtt{G}_{\mathtt{trace}}(c))$ be an arbitrary walk in $\mathtt{G}_{\mathtt{trace}}(c)$, and $\tau \in \mathcal{T}$ be arbitrary trace.
Then, let $(e_{p1}, \cdots, e_{p(n-1)})$ and $(v_1, \cdots, v_n)$ be the edges and vertices sequence for $k_t(\tau)$.
By Lemma C.1 and Lemma C.2, we know

$$\forall e_i \in k_t . e_i = (v_i, w_i^t, v_{i+1}) \implies \exists e_{pi} . e_{pi} = (v_i, w_i^p, v_{i+1}) \land e_{pi} \in \mathtt{E}_{\mathtt{prog}}$$

Then we construct a walk k_p with an edge sequence $(e_{p1}, \cdots, e_{p(n-1)})$ with a vertices sequence $(v_1, \cdots, v_n)$ where $e_{pi} = (v_i, v_{i+1}) \in \mathtt{E}_{\mathtt{prog}}$ for all $e_{pi} \in (e_{p1}, \cdots, e_{p(n-1)})$.
Let $n \in \mathbb{N}$ such that $\langle \mathtt{len}^{\mathtt{q}}(k_p), \tau\rangle \Downarrow_e n$, then, it is sufficient to show

$$k_p \in \mathtt{G}_{\mathtt{prog}}(c) \land n \geq \mathtt{len}^{\mathtt{q}}(k_t)(\tau)$$

To show $k_p \in \mathtt{G}_{\mathtt{prog}}(c)$, by Definition 18 for finite walk, we know

$$\forall v_i \in (v_1, \cdots, v_n), (v_i, w_i) \in \mathtt{W}_{\mathtt{trace}}(c) . \mathtt{visit}((v_1, \cdots, v_n), (v_i)) \leq w_i(\tau)$$

By Lemma C.3, we know for every

$$\forall v_i \in (v_1, \cdots, v_n), (v_i, w_i) \in \mathtt{V}_{\mathtt{prog}}(c), n_i \in \mathbb{N} . \langle w_i, \tau\rangle \Downarrow_e n_i \implies w_i(\tau) \leq n_i \ (\star)$$

Then, by Definition 28, we know the occurrence times for every $v_i \in (v_1, \cdots, v_n)$ is bound by the arithmetic expression w_i where $(v_i, w_i) \in \mathtt{V}_{\mathtt{prog}}(c)$.
Also, by Lemma C.4, we know for every

$$\forall v_i \in (v_1, \cdots, v_n), (v_i, w_i) \in \mathtt{V}_{\mathtt{prog}}(c), n_i \in \mathbb{N} . \langle w_i, \tau\rangle \Downarrow_e n_i \implies w_i(\tau) \leq n_i \ (\star)$$

Then, by Definition 28, we know the occurrence times for every $v_i \in (v_1, \cdots, v_n)$ is bound by the arithmetic expression w_i where $(v_i, w_i) \in \mathtt{V}_{\mathtt{prog}}(c)$.
So we have $k_p \in \mathcal{WK}(\mathtt{G}_{\mathtt{prog}})$ proved.
In order to show $n \geq \mathtt{len}^{\mathtt{q}}(k_t)(\tau)$, it is sufficient to show

$$\forall v_i \in (v_1, \cdots, v_n), (v_i, w_i) \in \mathtt{W}_{\mathtt{prog}}(c), (v_i, w_i') \in \mathtt{W}_{\mathtt{trace}}(c), n_i \in \mathbb{N} . \langle w_i, \tau\rangle \Downarrow_e n_i$$
$$\implies \sum_{v_i \in \mathbb{LV}(c)} w_i'(\tau) \leq \sum_{v_i \in \mathbb{LV}(c)} n_i$$

Then by $(\star)$, we know $\sum\limits_{v_i \in \mathbb{LV}(c)} w'_i(\tau) \leq \sum\limits_{v_i \in \mathbb{LV}(c)} n_i$.
Then we have $n \geq \mathtt{len}^{\mathtt{q}}(k_t)(\tau)$ proved.
This theorem is proved. □

The following are the four lemmas used in the proof of Theorem C.1 above, showing the correspondence properties between the program based graph and trace based graph.

Lemma C.1 (One-on-One Mapping of vertices from $\mathtt{G}_{\mathtt{trace}}$ to $\mathtt{G}_{\mathtt{prog}}$)**.** *Given a program c with its program-based graph $\mathtt{G}_{\mathtt{prog}}(c) = (\mathtt{V}_{\mathtt{prog}}, \mathtt{E}_{\mathtt{prog}})$ and trace-based graph $\mathtt{G}_{\mathtt{trace}}(c) = (\mathtt{V}_{\mathtt{trace}}, \mathtt{E}_{\mathtt{trace}})$, then for every $v \in \mathcal{VAR} \times \mathbb{N}$, $v \in \mathtt{V}_{\mathtt{prog}}$ if and only if $v \in \mathtt{G}_{\mathtt{trace}}$.*

$$\begin{aligned}&\forall c \in \mathcal{C}, v \in \mathcal{LV} \,.\, \mathtt{G}_{\mathtt{prog}}(c) = (\mathtt{V}_{\mathtt{prog}}, \mathtt{E}_{\mathtt{prog}}) \land \mathtt{G}_{\mathtt{trace}}(c) = (\mathtt{V}_{\mathtt{trace}}, \mathtt{E}_{\mathtt{trace}}) \\ &\quad \Longrightarrow (v, \_) \in \mathtt{V}_{\mathtt{prog}} \Longleftrightarrow (v, \_) \in \mathtt{V}_{\mathtt{trace}}\end{aligned}$$

Proof. Proof Summary: Proving by Definition 27 and Definition 17.
Taking arbitrary program c, by Definition 27 and Definition 17, we have
its program-based graph $\mathtt{G}_{\mathtt{prog}}(c) = (\mathtt{V}_{\mathtt{prog}}, \mathtt{E}_{\mathtt{prog}})$
and trace-based graph $\mathtt{G}_{\mathtt{trace}}(c) = (\mathtt{V}_{\mathtt{trace}}, \mathtt{E}_{\mathtt{trace}})$.
By the two definitions, we know $\mathtt{V}_{\mathtt{trace}} = \{(v, w^t) \mid v \in \mathbb{LV}(c)\}$ and $\mathtt{V}_{\mathtt{prog}} = \{(v, w^p) \mid v \in \mathbb{LV}(c)\}$.
Then we know $(v, \_) \in \mathtt{V}_{\mathtt{prog}} \Longleftrightarrow (v, \_) \in \mathtt{V}_{\mathtt{trace}}$.
This theorem is proved. □

Lemma C.2 (Mapping from Egdes of $\mathtt{G}_{\mathtt{trace}}$ to $\mathtt{G}_{\mathtt{prog}}$)**.** *Given a program c with its program-based graph $\mathtt{G}_{\mathtt{prog}}(c) = (\mathtt{V}_{\mathtt{prog}}, \mathtt{E}_{\mathtt{prog}})$ and trace-based graph $\mathtt{G}_{\mathtt{trace}}(c) = (\mathtt{V}_{\mathtt{trace}}, \mathtt{E}_{\mathtt{trace}})$, then for every $e = (v_1, \_, v_2) \in \mathtt{E}_{\mathtt{trace}}$, there exists an edge $e' = (v'_1, \_, v'_2) \in \mathtt{E}_{\mathtt{prog}}$ with $v_1 = v'_1 \land v_2 = v'_2$.*

$$\begin{aligned}&\forall c \in \mathcal{C} \,.\, \mathtt{G}_{\mathtt{prog}}(c) = (\mathtt{V}_{\mathtt{prog}}, \mathtt{E}_{\mathtt{prog}}) \land \mathtt{G}_{\mathtt{trace}}(c) = (\mathtt{V}_{\mathtt{trace}}, \mathtt{E}_{\mathtt{trace}}) \\ &\quad \Longrightarrow \forall e = (v_1, \_, v_2) \in \mathtt{E}_{\mathtt{trace}} \,.\, \exists e' \in \mathtt{E}_{\mathtt{prog}} \,.\, e' = (v_1, \_, v_2)\end{aligned}$$

Proof. Proof Summary: Proving by Lemma C.1, Lemma D.1 Definition 27 and Definition 17
Taking arbitrary program c, by Definition 27 and Definition 17, we have
its program-based graph $\mathtt{G}_{\mathtt{prog}}(c) = (\mathtt{V}_{\mathtt{prog}}, \mathtt{E}_{\mathtt{prog}})$
and trace-based graph $\mathtt{G}_{\mathtt{trace}}(c) = (\mathtt{V}_{\mathtt{trace}}, \mathtt{E}_{\mathtt{trace}})$.
Taking arbitrary edge $e = (x^i, \_, y^j) \in \mathtt{E}_{\mathtt{trace}}$, it is sufficient to show $(x^i, \_, y^j) \in \mathtt{E}_{\mathtt{prog}}$.
By Lemma C.1, we know $(x^i, \_), (y^j, \_) \in \mathtt{V}_{\mathtt{prog}}$.
By definition of $\mathtt{E}_{\mathtt{trace}}$, we know there is an initial trace $\tau_0 \in \mathcal{T}_0(c)$ and two witness traces $\tau_1, \tau_2 \in \mathcal{T}$ such that $\mathsf{DEP}(x^i, y^j, \tau_0, \tau_1, \tau_2, c)$.
By Theorem D.1, we know $\exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c \,.\, n \geq 0 \land \mathtt{flowsTo}(x^i, z_1^{r_1}, c) \land \cdots \land \mathtt{flowsTo}(z_n^{r_n}, y^j, c)$.
Then by definition of $\mathtt{E}_{\mathtt{prog}}$, we know $(x^i, \_, y^j) \in \mathtt{E}_{\mathtt{prog}}$. This Lemma is proved. □

Lemma C.3 (Vertex Weights are bounded)**.** *Given a program c with its program-based graph $\mathtt{G}_{\mathtt{prog}}(c) = (\mathtt{V}_{\mathtt{prog}}, \mathtt{E}_{\mathtt{prog}})$ and trace-based graph $\mathtt{G}_{\mathtt{trace}}(c) = (\mathtt{V}_{\mathtt{trace}}, \mathtt{E}_{\mathtt{trace}})$, for every $(x^l, w_t) \in \mathtt{V}_{\mathtt{trace}}$, there is $(x^l, w_p) \in \mathtt{V}_{\mathtt{prog}}$ and w_p is a bound on w_t.*

$$\begin{aligned}&\forall c \in \mathcal{C} \,.\, \mathtt{G}_{\mathtt{prog}}(c) = (\mathtt{V}_{\mathtt{prog}}, \mathtt{E}_{\mathtt{prog}}) \land \mathtt{G}_{\mathtt{trace}}(c) = (\mathtt{V}_{\mathtt{trace}}, \mathtt{E}_{\mathtt{trace}}) \Longrightarrow \\ &\quad \forall (x^l, w^t) \in \mathtt{V}_{\mathtt{trace}}, (x^l, w^p) \in \mathtt{V}_{\mathtt{prog}}, \tau_0 \in \mathcal{T}_0(c), \tau' \in \mathcal{T}, v \in \mathbb{N} \,.\, \langle c, \tau_0 \rangle \to^* \langle \mathtt{skip}, \tau {+\!+} \tau' \rangle \land \langle w^p, \tau_0 \rangle \Downarrow_e v \\ &\quad \Longrightarrow w^t(\tau) \leq v\end{aligned}$$

Proof. Taking arbitrary program c, by Definition 27 and Definition 17, we have
its program-based graph $\mathtt{G}_{\mathtt{prog}}(c) = (\mathtt{V}_{\mathtt{prog}}, \mathtt{E}_{\mathtt{prog}})$
and trace-based graph $\mathtt{G}_{\mathtt{trace}}(c) = (\mathtt{V}_{\mathtt{trace}}, \mathtt{E}_{\mathtt{trace}})$.
Taking arbitrary $(x^l, w^t) \in \mathtt{V}_{\mathtt{trace}}, (x^l, w^p) \in \mathtt{V}_{\mathtt{prog}}, \tau_0 \in \mathcal{T}_0(c), \tau' \in \mathcal{T}$, satisfying:
$\langle c, \tau_0 \rangle \to^* \langle \mathtt{skip}, \tau_0 +\!\!+ \tau' \rangle \wedge \langle w^p, \tau_0 \rangle \Downarrow_e v$
By soundness of reachability bound analysis in Theorem E.1, we know $\mathtt{cnt}(\tau', l) \leq v$
By definition 17, we know $w^t(\tau) = \mathtt{cnt}(\tau', l)$, then we have $w^t(\tau) \leq v$ and this is proved. □

Lemma C.4 (Edge Weights are bounded). *Given a program c with its program-based graph $\mathtt{G}_{\mathtt{prog}}(c) = (\mathtt{V}_{\mathtt{prog}}, \mathtt{E}_{\mathtt{prog}})$ and trace-based graph $\mathtt{G}_{\mathtt{trace}}(c) = (\mathtt{V}_{\mathtt{trace}}, \mathtt{E}_{\mathtt{trace}})$, for every $e = (v_1, w^p, v_2) \in \mathtt{E}_{\mathtt{trace}}$ and $e' = (v_1, w^t, v_2) \in \mathtt{E}_{\mathtt{prog}}$, w^p is a bound on w^t.*

$$\begin{array}{l}
\forall c \in \mathcal{C} \,.\, \mathtt{G}_{\mathtt{prog}}(c) = (\mathtt{V}_{\mathtt{prog}}, \mathtt{E}_{\mathtt{prog}}) \wedge \mathtt{G}_{\mathtt{trace}}(c) = (\mathtt{V}_{\mathtt{trace}}, \mathtt{E}_{\mathtt{trace}}) \implies \\
\quad \forall (v_1, w^p, v_2) \in \mathtt{E}_{\mathtt{trace}}, (v_1, w^t, v_2) \in \mathtt{E}_{\mathtt{prog}}, \tau_0 \in \mathcal{T}_0(c), \tau' \in \mathcal{T}, v \in \mathbb{N} \,.\, \langle c, \tau_0 \rangle \to^* \langle \mathtt{skip}, \tau_0 +\!\!+ \tau' \rangle \wedge \langle w^p, \tau_0 \rangle \Downarrow_e v \\
\quad \implies w^t(\tau) \leq v
\end{array}$$

Proof. Taking arbitrary program c, by Definition 27 and Definition 17, we have
its program-based graph $\mathtt{G}_{\mathtt{prog}}(c) = (\mathtt{V}_{\mathtt{prog}}, \mathtt{E}_{\mathtt{prog}})$
and trace-based graph $\mathtt{G}_{\mathtt{trace}}(c) = (\mathtt{V}_{\mathtt{trace}}, \mathtt{E}_{\mathtt{trace}})$.
Taking arbitrary $e = (v_1, w^p, v_2) \in \mathtt{E}_{\mathtt{trace}}$, $e' = (v_1, w^t, v_2) \in \mathtt{E}_{\mathtt{prog}}$, and $\tau, \tau' \in \mathcal{T}$, satisfying:
$\langle c, \tau \rangle \to^* \langle \mathtt{skip}, \tau +\!\!+ \tau' \rangle \wedge \langle w^p, \tau \rangle \Downarrow_e v$
By soundness of reachability bound analysis in Theorem F.1, we know $\mathtt{cnt}(\tau', l) \leq v$
By definition 17, we know $w^t(\tau) = \mathtt{cnt}(\tau', l)$, then we have $w^t(\tau) \leq v$ and this is proved. □

D Soundness of `flowsTo` with Language and Adaptivity Extension

Theorem D.1 (DEP implies `flowsTo`)**.** *Given a program c, for all $x^i, y^j \in \mathbb{LV}_c$, if there exist two witness traces and an initial trace satisfying* $\mathsf{DEP}(x^i, y^j, \tau_1, \tau_2, \tau_0, c)$*, then there exist* $z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c$ *with* $n \geq 0$ *such that* $\mathtt{flowsTo}(x^i, z_1^{r_1}, c) \land \cdots \land \mathtt{flowsTo}(z_n^{r_n}, y^j, c)$

$$\begin{array}{l}\forall x^i, y^j \in \mathbb{LV}_c.(\exists \tau_1, \tau_2 \in \mathcal{T}, \tau_0 \in \mathcal{T}_0(c) \ . \ \mathsf{DEP}(x^i, y^j, \tau_1, \tau_2, \tau_0, c)) \\ \quad \Longrightarrow \Big(\exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c \ . \ n \geq 0 \land \mathtt{flowsTo}(x^i, z_1^{r_1}, c) \land \cdots \land \mathtt{flowsTo}(z_n^{r_n}, y^j, c)\Big)\end{array}$$

Proof Summary:
induction on c. Proved by induction hypothesis in if and while case, and cases analysis in seq case.

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

case: $c = [\mathtt{skip}]^l$
By $\mathbb{LV}$ in Definition 5, we know $\mathbb{LV}(c) = \emptyset$ and the theorem is vacuously true.

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

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

case: $c = [\ \mathtt{fun}\ ]^l : f(r^{l_r}, x_1, \ldots, x_n) := c$
This case is proved in the same way as **case:** $c = [x \leftarrow e]^l$.

case: $c = \mathtt{if}\ ([b]^l, c_1, c_2)$
Let $\tau, \tau' \in \mathcal{T}, \tau_0 \in \mathcal{T}_0(c)$ be the two witness traces and initial trace satisfying $\mathsf{DEP}(x^i, y^j, \tau_1, \tau_2, \tau_0, \mathtt{if}\ ([b]^l, c_1, c_2))$.
By *may-dependency* in Definition 16, let $\tau_0' \in \mathcal{T}_0(c)$ be the initial trace satisfying,
(1) $(\forall z^r \neq x^i \ . \ \rho(\tau_0, z^r) \neq \rho(\tau_0', z^r))$ and
(2) $\langle \mathtt{if}\ ([b]^l, c_1, c_2), \tau_0 \rangle \to^* \langle [\mathtt{skip}]^l, \tau_0 {+\!+} \tau \rangle$ and
(3) $\langle \mathtt{if}\ ([b]^l, c_1, c_2), \tau_0' \rangle \to^* \langle [\mathtt{skip}]^l, \tau_0' {+\!+} \tau' \rangle$ and
(4) $\mathtt{Diff}_{\mathtt{seq}}(\tau, \tau', y^j) \neq \emptyset$.
By the evaluation rules for $\mathtt{if}\ ([b]^l, c_1, c_2)$, we have the two following cases on the evaluation in (2).

sub-case: (2) : if-t
$\langle \mathtt{if}\ ([b]^l, c_1, c_2), \tau_0 \rangle \to^{\textsf{if-t}} \langle c_1, \tau_0 :: (b, l, \mathtt{true}, \bullet) \rangle \to^* \langle [\mathtt{skip}]^l, \tau_0 :: (b, l, \mathtt{true}, \bullet) {+\!+} \tau_1 \rangle$.
Accordingly, there are also two cases on the evaluation in (3) as follows,

subsub-case: (3) : if-f
$\langle \mathtt{if}\ ([b]^l, c_1, c_2), \tau_0' \rangle \to^{\textsf{if-f}} \langle c_2, \tau_0' :: (b, l, \mathtt{false}, \bullet) \rangle \to^* \langle [\mathtt{skip}]^l, \tau_0' :: (b, l, \mathtt{false}, \bullet) {+\!+} \tau_2' \rangle$
By (1) and Inversion Lemma D.1(3) of boolean expression evaluation, we know $x \in FV(b)$ and $x^l \in \mathtt{RD}(l, c)$.
By $\mathtt{Diff}_{\mathtt{seq}}(\tau, \tau', y^j) \neq \emptyset$, we know $y^j \in \mathbb{LV}(c_1) \cup \mathbb{LV}(c_2)$.
Then, by `flowsTo` in Definition 24, we know $\mathtt{flowsTo}(x^i, y^j, c)$, this case is proved.

subsub-case: (3) : if-t
$\langle \mathtt{if}\ ([b]^l, c_1, c_2), \tau_0' \rangle \to^{\textsf{if-t}} \langle c_1, \tau_0' :: (b, l, \mathtt{true}, \bullet) \rangle \to^* \langle [\mathtt{skip}]^l, \tau_0' :: (b, l, \mathtt{true}, \bullet) {+\!+} \tau_1' \rangle$
To show : $\Big(\exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c \ . \ n \geq 0 \land \mathtt{flowsTo}(x^i, z_1^{r_1}, c) \land \cdots \land \mathtt{flowsTo}(z_n^{r_n}, y^j, c)$.
We first have the following induction hypothesis,
(IH) $(\exists \tau_{ih}, \tau_{ih}' \in \mathcal{T}, \tau_{ih0} \in \mathcal{T}_0(c_1) \ . \ \mathsf{DEP}(x^i, y^j, \tau_{ih}, \tau_{ih}', \tau_{ih0}, c_1)) \Longrightarrow \Big(\exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_{c_1} \ . \ n \geq 0 \land \mathtt{flowsTo}(x^i, z_1^{r_1}, c_1) \land \cdots \land \mathtt{flowsTo}(z_n^{r_n}, y^j, c_1)\Big)$.

Constructing $\tau_{ih0} = \tau_0 :: (b, l, \texttt{true}, \bullet)$, $\tau_{ih} = \tau_1$ and $\tau'_{ih} = \tau'_1$.
By (1), we know $\forall z^r \neq x^i$,
(ih1)$(\rho(\tau_0 :: (b, l, \texttt{true}, \bullet), z^r) = \rho(\tau'_0 :: (b, l, \texttt{true}, \bullet), z^r))$.
We also have two following evaluations:
(ih2) $\langle c_1, \tau_0 :: (b, l, \texttt{true}, \bullet)\rangle \to^* \langle[\texttt{skip}]^l, \tau_{0++}\tau_1\rangle$
(ih3) $\langle c_1, \tau'_0 :: (b, l, \texttt{true}, \bullet)\rangle \to^* \langle[\texttt{skip}]^l, \tau'_{0++}\tau'_1\rangle$
By the determinism of evaluation, we have $\tau = (b, l, \texttt{true}, \bullet)_{++}\tau_1$ and $\tau' = (b, l, \texttt{true}, \bullet)_{++}\tau_1$.
By (4) and $\texttt{Diff}_{\texttt{seq}}$ in Definition 14, we then have
(ih4) $\texttt{Diff}_{\texttt{seq}}((b, l, \texttt{true}, \bullet)_{++}\tau_1, (b, l, \texttt{true}, \bullet)_{++}\tau'_1, y^j) = \texttt{Diff}_{\texttt{seq}}(\tau_1, \tau'_1, y^j) \neq \emptyset$
Then, by (ih1) - (ih4), we know
$(\exists \tau_{ih}, \tau'_{ih} \in \mathcal{T}, \tau_{ih0} \in \mathcal{T}_0(c_1) \,.\, \mathsf{DEP}(x^i, y^j, \tau_{ih}, \tau'_{ih}, \tau_{ih0}, c_1))$.
Then, by induction hypothesis (IH), we know
$\exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_{c_1} \,.\, n \geq 0 \wedge \texttt{flowsTo}(x^i, z_1^{r_1}, c_1) \wedge \cdots \wedge \texttt{flowsTo}(z_n^{r_n}, y^j, c_1)$
Then, by the $\texttt{flowsTo}$ in Definition 24, we have
$\exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c \,.\, n \geq 0 \wedge \texttt{flowsTo}(x^i, z_1^{r_1}, c) \wedge \cdots \wedge \texttt{flowsTo}(z_n^{r_n}, y^j, c)$.
This case is proved.

sub-case: (2) : if-f
This **sub-case** is proved in exactly the same way as **sub-case: (2) : if-t**.

case: $c = \texttt{while}\ [b]^l\ \mathsf{do}\ c$
This **case** is proved in exactly the same way as **case:** $c = \texttt{if}\ ([b]^l, c_1, c_2)$.

case: $c = c_1; c_2$
Let $\tau, \tau' \in \mathcal{T}, \tau_0 \in \mathcal{T}_0(c)$ be the two witness traces and initial trace satisfying $\mathsf{DEP}(x^i, y^j, \tau_1, \tau_2, \tau_0, c_1; c_2)$.
By *may-dependency* in Definition 16, let $\tau'_0 \in \mathcal{T}_0(c)$ be the initial trace satisfying,
(1) $(\forall z^r \neq x^i \,.\, \rho(\tau_0, z^r) = \rho(\tau'_0, z^r))$ and
(2) $\langle c_1; c_2, \tau_0\rangle \to^* \langle[\texttt{skip}]^l, \tau_{0++}\tau\rangle$ and
(3) $\langle c_1; c_2, \tau'_0\rangle \to^* \langle[\texttt{skip}]^l, \tau'_{0++}\tau'\rangle$ and
(4) $\texttt{Diff}_{\texttt{seq}}(\tau, \tau', y^j) \neq \emptyset$.
By the Evaluation rules for $c_1; c_2$, we have the following two concrete evaluations for (2) and (3):
(exe1) $\langle c_1; c_2, \tau_0\rangle \to^* \langle c_2, \tau_{0++}\tau_1\rangle \to^* \langle[\texttt{skip}]^l, \tau_{0++}\tau_{1++}\tau_2\rangle$
(exe2) $\langle c_1; c_2, \tau'_0\rangle \to^* \langle c_2, \tau'_{0++}\tau'_1\rangle \to^* \langle[\texttt{skip}]^l, \tau'_{0++}\tau'_{1++}\tau'_2\rangle$
where $\tau = \tau_{1++}\tau_2$ and $\tau' = \tau'_{1++}\tau'_2$. Then, we have two following **sub-cases**,

sub-case: $\texttt{Diff}_{\texttt{seq}}(\tau_1, \tau'_1, y^j) \neq \emptyset$
Then, by (exe1), (exe2) and the determinism of the program evaluation, we have the two following evaluations:
(ih1-2) $\langle c_1; c_2, \tau_0\rangle \to^* \langle[\texttt{skip}]^l, \tau_{0++}\tau_1\rangle$;
(ih1-3) $\langle c_1; c_2, \tau'_0\rangle \to^* \langle[\texttt{skip}]^l, \tau'_{0++}\tau'_1\rangle$.
Then, by (1), (ih1-2), (ih1-3) and the **sub-case** condition, according to the *may-dependency* in Definition 16, we know

$$\mathsf{DEP}(x^i, y^j, \tau_1, \tau'_1, \tau_0, c_1)$$

By induction hypothesis, we have
$\exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_{c_1} \,.\, n \geq 0 \wedge \texttt{flowsTo}(x^i, z_1^{r_1}, c_1) \wedge \cdots \wedge \texttt{flowsTo}(z_n^{r_n}, y^j, c_1)$
Then, by the $\texttt{flowsTo}$ in Definition 24, we have
$\exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c \,.\, n \geq 0 \wedge \texttt{flowsTo}(x^i, z_1^{r_1}, c) \wedge \cdots \wedge \texttt{flowsTo}(z_n^{r_n}, y^j, c)$.
This case is proved.

sub-case: $\texttt{Diff}_{\texttt{seq}}(\tau_1, \tau_1', y^j) = \emptyset$
By (4), we know
(ih2-4) $\texttt{Diff}_{\texttt{seq}}(\tau_2, \tau_2', y^j) \neq \emptyset$.
There are two cases,

subsub-case: $(\forall z^r \in \mathcal{L} \, . \, \rho(\tau_1, z^r) = \rho(\tau_1', z^r))$
Let $\tau_{ih0} = \tau_0{+\!+}\tau_1$ and $\tau_{ih0}' = \tau_0'{+\!+}\tau_1'$, we know the following by (1),
(ih2-1) $(\forall z^r \neq x^i \, . \, \rho(\tau_0{+\!+}\tau_1, z^r) = \rho(\tau_0'{+\!+}\tau_1, z^r))$
By (exe1), (exe2) and the determinism of the program evaluation, we have the two following evaluations:
(ih2-2) $\langle c_2, \tau_{ih0} \rangle \to^* \langle [\texttt{skip}]^l, \tau_{ih0}{+\!+}\tau_2 \rangle$;
(ih2-3) $\langle c_2, \tau_{ih0}' \rangle \to^* \langle [\texttt{skip}]^l, \tau_{ih0}'{+\!+}\tau_2' \rangle$.
Then, by (ih2-1), (ih2-2), (ih2-3) and the (ih2-4), according to the *may-dependency* in Definition 16, we know

$$\mathsf{DEP}(x^i, y^j, \tau_2, \tau_2', \tau_{ih0}, c_2)$$

By induction hypothesis, we have
$\exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_{c_2} \, . \, n \geq 0 \land \texttt{flowsTo}(x^i, z_1^{r_1}, c_2) \land \cdots \land \texttt{flowsTo}(z_n^{r_n}, y^j, c_2)$
Then, by the $\texttt{flowsTo}$ in Definition 24, we have
$\exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_{c} \, . \, n \geq 0 \land \texttt{flowsTo}(x^i, z_1^{r_1}, c) \land \cdots \land \texttt{flowsTo}(z_n^{r_n}, y^j, c)$.
This case is proved.

subsub-case: $\neg(\forall z^r \in \mathcal{L} \, . \, \rho(\tau_1, z^r) = \rho(\tau_1', z^r))$
According to the Definition 14, since $\texttt{Diff}_{\texttt{seq}}(\tau_2, \tau_2', y^j) \neq \emptyset$, there are two cases,
$|\texttt{seq}(\tau_2, y^j)| = |\texttt{seq}(\tau_2', y^j)|$
$|\texttt{seq}(\tau_2, y^j)| \neq |\texttt{seq}(\tau_2', y^j)|$

subsubsub-case: $|\texttt{seq}(\boldsymbol{\tau_2}, \boldsymbol{y^j})| = |\texttt{seq}(\boldsymbol{\tau_2'}, \boldsymbol{y^j})|$
, According to the Definition 13, and Diff Value Dependency Inversion LemmaD.6 we know there exist two events $(y, j, v_1, \alpha) \in \tau_2$ and $(y, j, v_1', \alpha') \in \tau_2'$.
Then we have the following two execution instances

$$\langle c_2, \tau_0{+\!+}\tau_1 \rangle \to^* \langle [y \leftarrow e/\texttt{query}(\psi)]^j, \tau_0{+\!+}\tau_1{+\!+}\tau_2^1 \rangle \to^* \langle \texttt{skip}, \tau_0{+\!+}\tau_1{+\!+}\tau_2^1 :: (y, j, v_1, \alpha){+\!+}\tau_2^2 \rangle \tag{2}$$

$$\langle c_2, \tau_0'{+\!+}\tau_1' \rangle \to^* \langle [y \leftarrow e/\texttt{query}(\psi)]^j, \tau_0'{+\!+}\tau_1'{+\!+}\tau_2'^1 \rangle \to^* \langle \texttt{skip}, \tau_0'{+\!+}\tau_1'{+\!+}\tau_2'^1 :: (y, j, v_1', \alpha'){+\!+}\tau_2'^2 \rangle, \tag{3}$$

where e_2/ψ_2 is the expression of the assignment command associated to the events (y, j, v_1, α) and (y, j, v_1', α') by the Inversion Lemma. D.4.
Let $\mathbb{LV}_{\texttt{Diff}}$ be the set of all the variables z^r on $\tau_1'{+\!+}\tau_2'^1$ and $\tau_1{+\!+}\tau_2^1$ satisfying $\rho(\tau_1{+\!+}\tau_2^1, z^r) \neq \rho(\tau_1'{+\!+}\tau_2'^1, z^r)$.
Then, by the $\texttt{Diff}_{\texttt{seq}}$ in Definition 14, we know for every $z^r \in \mathbb{LV}_{\texttt{Diff}}$, $\texttt{Diff}_{\texttt{seq}}(\tau_1{+\!+}\tau_2^1, \tau_1'{+\!+}\tau_2'^1, z^r) \neq \emptyset$.
Then we know the following for every $z^r \in \mathbb{LV}_{\texttt{Diff}}$,
(ih3-4) $\forall z^r \in \mathbb{LV}_{\texttt{Diff}} \, . \, \texttt{Diff}_{\texttt{seq}}(\tau, \tau', z^r) \neq \emptyset$.
Then, by (1), (ih1-2), (ih1-3), and (ih3-4), according to the *may-dependency* in Definition 16, we know for every $z^r \in \mathbb{LV}_{\texttt{Diff}}$,

$$\mathsf{DEP}(x^i, z^r, \tau, \tau', \tau_0, c_1)$$

By induction hypothesis, and the $\texttt{flowsTo}$ in Definition 24, we have
(cld-1) $\exists n \in \mathbb{N}, w_{1r}^{l_{1r}}, \cdots, w_{1r}^{l_{nr}} \in \mathbb{LV}_c \, . \, n \geq 0 \land \texttt{flowsTo}(x^i, w_{1r}^{l_{1r}}, c) \land \cdots \land \texttt{flowsTo}(w_{nr}^{l_{nr}}, z^r, c)$.
By the inversion Lemma D.2, we also know $\exists z^r \in (\mathbb{LV}_{\texttt{Diff}} \cup \{x^i\}) \, . \, z \in FV(e) \land r = \iota(\tau_0{+\!+}\tau_1{+\!+}\tau_2^1)z$.
Then by the reaching definition Inversion Lemma D.5, we know $z^r \in \texttt{RD}(j, c)$.
Then by $\texttt{flowsTo}$ in Definition 24, we know $\exists z^r \in (\mathbb{LV}_{\texttt{Diff}} \cup \{x^i\}) \, . \, \texttt{flowsTo}(z^r, y^j, c_2)$, i.e., $\texttt{flowsTo}(z^r, y^j, c)$.

Together with (cld-1), we conclude
$\exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c \, . \, n \geq 0 \land \texttt{flowsTo}(x^i, z_1^{r_1}, c_1) \land \cdots \land \texttt{flowsTo}(z_n^{r_n}, y^j, c)$.
This case is proved.

subsubsub-case: $|\texttt{seq}(\boldsymbol{\tau_2}, \boldsymbol{y^j})| \neq |\texttt{seq}(\boldsymbol{\tau'_2}, \boldsymbol{y^j})|$
According to the Definition 13, and Diff Control Dependency Inversion Lemma D.7, we know there exist two testing events $\epsilon_b \in \tau_2$ and $\neg\epsilon_b \in \tau'_2$ satisfying
$\forall z \in FV(\pi_1(\epsilon_b)), \exists r \in \mathcal{L} \, . \, \texttt{flowsTo}(z^r, y^j, c_2)$, i.e., $\texttt{flowsTo}(z^r, y^j, c)$.
In the same way as above by inversion on the testing event ϵ_b, we get similar execution instance as Equation 2 and 3.
Repeating the same proof steps under the two executions, similarly, by the expression evaluation inversion Lemma D.2, we know
$\exists z^r \in (\mathbb{LV}_{\texttt{Diff}} \cup \{x^i\}) \, . \, z \in FV(b) \land r = \iota(\tau_0{++}\tau_1{++}\tau_2^1)z$.
Then by the reaching definition Inversion Lemma D.5, we know $z^r \in \texttt{RD}(j, c)$.
Then by $\texttt{flowsTo}$ in Definition 24, we know $\exists z^r \in (\mathbb{LV}_{\texttt{Diff}} \cup \{x^i\}) \, . \, \texttt{flowsTo}(z^r, y^j, c_2)$, i.e., $\texttt{flowsTo}(z^r, y^j, c)$.
Then together with (cld-1), we know
$\exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c \, . \, n \geq 0 \land \texttt{flowsTo}(x^i, z_1^{r_1}, c_1) \land \cdots \land \texttt{flowsTo}(z_n^{r_n}, y^j, c)$.
This case is proved.

case: $c = [x \leftarrow \texttt{call}\ (x, e_1, \ldots, e_n)]^l$
By the evaluation rule, this case is proved in the same way as **case:** $c = c_1; c_2$ □

D.1 Inversion Lemmas and Helper Lemmas

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

Arithmetic Inversions The Inversion Lemmas on expression evaluations.

Lemma D.1 (Expression Inversion). *For all $x^i \in \mathbb{LV}$, and $\tau, \tau' \in \mathcal{T}$, and an expression e if $\forall z^j \in \mathbb{LV}/\{x^i\} \, . \, \rho(\tau)z = \rho(\tau')z$, and if*

- *e is an arithmetic expression a, and $\langle \tau, a \rangle \Downarrow_a v$ and $\langle \tau', a \rangle \Downarrow_a v'$ with $v' \neq v$, then x is in the free variables of a and i is the latest label for x in τ, i.e., $x \in VAR(a)$ and $i = \iota(\tau)x$.*
- *e is a boolean expression b, and $\langle \tau, b \rangle \Downarrow_b v$ and $\langle \tau', b \rangle \Downarrow_b v'$ with $v' \neq v$, then x is in the free variables of b and i is the latest label for x in τ, i.e., $x \in VAR(b)$ and $i = \iota(\tau)x$.*
- *e is a query expression ψ, and $\langle \tau, \psi \rangle \Downarrow_q \alpha$ and $\langle \tau', \psi \rangle \Downarrow_q \alpha'$ with $\alpha \neq_q \alpha'$, then x is in the free variables of ψ and i is the latest label for x in τ, i.e., $x \in VAR(\psi)$ and $i = \iota(\tau)x$.*

Proof Summary:
To show $x \in VAR(a)$, by showing contradiction ($\forall \tau, \tau'$ in second hypothesis $v = v'$) if $x \notin VAR(a)$.
To show $i = \iota(\tau)$, by showing contradiction ($\forall \tau, \tau'$ in second hypothesis $v = v'$) if $j = \iota(\tau)x$ and $i \neq j$.

Proof. Take two arbitrary traces $\tau, \tau' \in \mathcal{T}$, and an expression e satisfying $\forall z^j \in \mathbb{LV}/\{x^i\} \, . \, \rho(\tau)z = \rho(\tau')z$, we have the following three cases.

case: e is an arithmetic expression a
We have $\langle \tau, b\rangle \Downarrow_b v$ and $\langle \tau', b\rangle \Downarrow_b v'$ with $v' \neq v$ from the lemma hypothesis.
To show $x \in VAR(\psi)$ and $i = \iota(\tau)x$:
Assuming $x \notin VAR(a)$, since $\forall z^j \in \mathbb{LV}/\{x^i\} . \rho(\tau)z = \rho(\tau')z$, we know $v = v'$, which is contradicted to $v' \neq v$.
Then we know $x \in VAR(\psi)$.
Assuming $j = \iota(\tau)x \wedge i \neq j$, by $\forall z^j \in \mathbb{LV}/\{x^i\} . \rho(\tau)z = \rho(\tau')z$, we know $\rho(\tau)x = \rho(\tau')x$, i.e., $\forall z^j \in \mathbb{LV} . \rho(\tau)z = \rho(\tau')z$.
Then by the determination of the evaluation, we know $v = v'$, which is contradicted to $v' \neq v$.
Then we know $i = \iota(\tau)x$.

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 ψ
This case is proved trivially in the same way as the case of the arithmetic expression. □

Lemma D.2 (Expression Inversion Generalization)**.** *For all subset of the labelled variables $\mathbb{LV}_{\texttt{Diff}} \subset \mathbb{LV}$ and an expression e, if*

- *e is an arithmetic expression a, and for all $z^j \in \mathbb{LV} \setminus \mathbb{LV}_{\texttt{Diff}}$ there exist $\tau, \tau' \in \mathcal{T}, v, v'$ such that $\rho(\tau)z = \rho(\tau')z$, $\langle \tau, a\rangle \Downarrow_a v$ and $\langle \tau', a\rangle \Downarrow_a v'$ with $v \neq v'$, then $\exists x^i \in \mathbb{LV}_{\texttt{Diff}}$ such that $x \in FV(a)$ and $i = \iota(\tau)x$.*

$$\begin{array}{l} \forall \mathbb{LV}_{\texttt{Diff}} \subset \mathbb{LV}, a . \\ \quad \forall z^j \in \mathbb{LV} \setminus \mathbb{LV}_{\texttt{Diff}} . \exists \tau, \tau' \in \mathcal{T}, v, v' . \rho(\tau)z = \rho(\tau')z \wedge \langle \tau, a\rangle \Downarrow_a v \wedge \langle \tau', a\rangle \Downarrow_a v' \wedge v \neq v' \\ \quad \Longrightarrow \exists x^i \in \mathbb{LV}_{\texttt{Diff}} . x \in FV(a) \wedge i = \iota(\tau)x \end{array}$$

- *e is a boolean expression b, and for all $z^j \in \mathbb{LV} \setminus \mathbb{LV}_{\texttt{Diff}}$ there exist $\tau, \tau' \in \mathcal{T}, v, v'$ such that $\rho(\tau)z = \rho(\tau')z$, $\langle \tau, b\rangle \Downarrow_b v$ and $\langle \tau', b\rangle \Downarrow_b v'$ with $v \neq v'$, then $\exists x^i \in \mathbb{LV}_{\texttt{Diff}}$ such that $x \in FV(b)$ and $i = \iota(\tau)x$.*

$$\begin{array}{l} \forall \mathbb{LV}_{\texttt{Diff}} \subset \mathbb{LV}, a . \\ \quad \forall z^j \in \mathbb{LV} \setminus \mathbb{LV}_{\texttt{Diff}} . \exists \tau, \tau' \in \mathcal{T}, v, v' . \rho(\tau)z = \rho(\tau')z \wedge \langle \tau, b\rangle \Downarrow_b v \wedge \langle \tau', b\rangle \Downarrow_b v' \wedge v \neq v' \\ \quad \Longrightarrow \exists x^i \in \mathbb{LV}_{\texttt{Diff}} . x \in FV(b) \wedge i = \iota(\tau)x \end{array}$$

- *e is a query expression ψ, and for all $z^j \in \mathbb{LV} \setminus \mathbb{LV}_{\texttt{Diff}}$ there exist $\tau, \tau' \in \mathcal{T}, \alpha, \alpha'$ such that $\rho(\tau)z = \rho(\tau')z$, $\langle \tau, \psi\rangle \Downarrow_q \alpha$ and $\langle \tau', \psi\rangle \Downarrow_q \alpha'$ with $\alpha \neq \alpha'$, then $\exists x^i \in \mathbb{LV}_{\texttt{Diff}}$ such that $x \in FV(\psi)$ and $i = \iota(\tau)x$.*

$$\begin{array}{l} \forall \mathbb{LV}_{\texttt{Diff}} \subset \mathbb{LV}, a . \\ \quad \forall z^j \in \mathbb{LV} \setminus \mathbb{LV}_{\texttt{Diff}} . \exists \tau, \tau' \in \mathcal{T}, \alpha, \alpha' . \rho(\tau)z = \rho(\tau')z \wedge \langle \tau, \psi\rangle \Downarrow_q \alpha \wedge \langle \tau', \psi\rangle \Downarrow_q \alpha' \wedge \alpha \neq \alpha' \\ \quad \Longrightarrow \exists x^i \in \mathbb{LV}_{\texttt{Diff}} . x \in FV(\psi) \wedge i = \iota(\tau)x \end{array}$$

Proof. Proof by showing contradiction and Applying Lemma D.1. □

Lemma D.3 (Expression Inversion Generalization-II)**.** *For all subset of the labelled variables $\texttt{Diff} \subset \mathbb{LV}$, and $x^i \in (\mathbb{LV} \setminus \texttt{Diff})$, and an expression e, if*

- *e is an arithmetic expression a, and for all $z^j \in \mathbb{LV} \setminus \texttt{Diff}, \tau, \tau' \in \mathcal{T}, v, v'$ 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 $v = v'$; and for all $z^j \in \mathbb{LV}/(\texttt{Diff} \cup \{x^i\})$ there exist $\tau, \tau' \in \mathcal{T}, v, v'$ 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 $x \in VAR(a)$ and $i = \iota(\tau)x$.*

$$\begin{aligned}&\forall \texttt{Diff} \subset \mathbb{LV}, x^i \in (\mathbb{LV} \setminus \texttt{Diff}), a\ .\\ &\quad \forall z^j \in \mathbb{LV} \setminus \texttt{Diff}, \tau, \tau' \in \mathcal{T}, v, v'\ .\ \rho(\tau)z = \rho(\tau')z \land \langle \tau, a \rangle \Downarrow_a v \land \langle \tau', a \rangle \Downarrow_a v' \land v = v'\\ &\quad \implies \forall z^j \in \mathbb{LV}/(\texttt{Diff} \cup \{x^i\})\ .\ \exists \tau, \tau' \in \mathcal{T}, v, v'\ .\ \rho(\tau)z = \rho(\tau')z \land \langle \tau, a \rangle \Downarrow_a v \land \langle \tau', a \rangle \Downarrow_a v' \land v \neq v'\\ &\qquad \implies x \in VAR(a) \land i = \iota(\tau)x\end{aligned}$$

- *e is a boolean expression b, and for all $z^j \in \mathbb{LV} \setminus \texttt{Diff}, \tau, \tau' \in \mathcal{T}, v, v'$ such that $\rho(\tau)z = \rho(\tau')z \land \langle \tau, b \rangle \Downarrow_b v \land \langle \tau', b \rangle \Downarrow_b v' \land v = v'$; and for all $z^j \in \mathbb{LV}/(\texttt{Diff} \cup \{x^i\})\ .\ \exists \tau, \tau' \in \mathcal{T}, v, v'\ .\ \rho(\tau)z = \rho(\tau')z \land \langle \tau, b \rangle \Downarrow_b v \land \langle \tau', b \rangle \Downarrow_b v' \land v \neq v'$ then $x \in VAR(b) \land i = \iota(\tau)x$*

$$\begin{aligned}&\forall \texttt{Diff} \subset \mathbb{LV}, x^i \in (\mathbb{LV} \setminus \texttt{Diff}), b\ .\\ &\quad \forall z^j \in \mathbb{LV} \setminus \texttt{Diff}, \tau, \tau' \in \mathcal{T}, v, v'\ .\ \rho(\tau)z = \rho(\tau')z \land \langle \tau, b \rangle \Downarrow_b v \land \langle \tau', b \rangle \Downarrow_b v' \land v = v'\\ &\quad \implies \forall z^j \in \mathbb{LV}/(\texttt{Diff} \cup \{x^i\})\ .\ \exists \tau, \tau' \in \mathcal{T}, v, v'\ .\ \rho(\tau)z = \rho(\tau')z \land \langle \tau, b \rangle \Downarrow_b v \land \langle \tau', b \rangle \Downarrow_b v' \land v \neq v'\\ &\qquad \implies x \in VAR(b) \land i = \iota(\tau)x\end{aligned}$$

- *e is a query expression ψ, and for all $\texttt{Diff} \subset \mathbb{LV}, x^i \in (\mathbb{LV} \setminus \texttt{Diff}), \psi$ such that for all $z^j \in \mathbb{LV} \setminus \texttt{Diff}, \tau, \tau' \in \mathcal{T}, \alpha, \alpha'\ .\ \rho(\tau)z = \rho(\tau')z \land \langle \tau, \psi \rangle \Downarrow_q \alpha \land \langle \tau', \psi \rangle \Downarrow_q \alpha' \land \alpha =_q \alpha'$; and for all $z^j \in \mathbb{LV}/(\texttt{Diff} \cup \{x^i\})\ .\ \exists \tau, \tau' \in \mathcal{T}, \alpha, \alpha'\ .\ \rho(\tau)z = \rho(\tau')z \land \langle \tau, \psi \rangle \Downarrow_q \alpha \land \langle \tau', \psi \rangle \Downarrow_q \alpha' \land \alpha \neq_q \alpha'$, then $x \in VAR(\psi) \land i = \iota(\tau)x$.*

$$\begin{aligned}&\forall \texttt{Diff} \subset \mathbb{LV}, x^i \in (\mathbb{LV} \setminus \texttt{Diff}), \psi\ .\\ &\quad \forall z^j \in \mathbb{LV} \setminus \texttt{Diff}, \tau, \tau' \in \mathcal{T}, \alpha, \alpha'\ .\ \rho(\tau)z = \rho(\tau')z \land \langle \tau, \psi \rangle \Downarrow_q \alpha \land \langle \tau', \psi \rangle \Downarrow_q \alpha' \land \alpha =_q \alpha'\\ &\quad \implies \forall z^j \in \mathbb{LV}/(\texttt{Diff} \cup \{x^i\})\ .\ \exists \tau, \tau' \in \mathcal{T}, \alpha, \alpha'\ .\ \rho(\tau)z = \rho(\tau')z \land \langle \tau, \psi \rangle \Downarrow_q \alpha \land \langle \tau', \psi \rangle \Downarrow_q \alpha' \land \alpha \neq_q \alpha'\\ &\qquad \implies x \in VAR(\psi) \land i = \iota(\tau)x\end{aligned}$$

Proof Summary:
To show $x \in VAR(a)$, by showing contradiction ($\forall \tau, \tau'$ in second hypothesis $v = v'$) if $x \notin VAR(a)$.
To show $i = \iota(\tau)$, by showing contradiction ($\forall \tau, \tau'$ in second hypothesis $v = v'$) if $j = \iota(\tau)x$ and $i \neq j$.

Proof. Taking an arbitrary expression e, we have the following three cases.

case: e is an arithmetic expression a
Taking an arbitrary set of labelled variables $\texttt{Diff} \subset \mathbb{LV}$, $x^i \in (\mathbb{LV} \setminus \texttt{Diff})$ satisfies:
$\forall z^j \in \mathbb{LV} \setminus \texttt{Diff}, \tau, \tau' \in \mathcal{T}, v, v'\ .\ \rho(\tau)z = \rho(\tau')z \land \langle \tau, a \rangle \Downarrow_a v \land \langle \tau', a \rangle \Downarrow_a v' \land v = v'$ (1)
and $\forall z^j \in \mathbb{LV} \setminus (\texttt{Diff} \cup \{x^i\})\ .\ \exists \tau, \tau' \in \mathcal{T}, v, v'\ .\ \rho(\tau)z = \rho(\tau')z \land \langle \tau, a \rangle \Downarrow_a v \land \langle \tau', a \rangle \Downarrow_a v' \land v \neq v'$ (2),
Let $\tau, \tau' \in \mathcal{T}, v, v'$ be the two traces and values satisfies hypothesis (2).
To show: $x \in VAR(a) \land i = \iota(\tau)x$:
Assuming $x \notin VAR(a)$, we know from the Inversion Lemma D.1 of the arithmetic expression case,
$\forall z^j \in \mathbb{LV} \setminus \{x^i\}, \tau, \tau' \in \mathcal{T}, v, v'\ .\ \rho(\tau)z = \rho(\tau')z \land \langle \tau, a \rangle \Downarrow_a v \land \langle \tau', a \rangle \Downarrow_a v' \land v = v'$.
Then with the hypothesis (1), we know:
$\forall z^j \in \mathbb{LV} \setminus (\texttt{Diff} \cup \{x^i\}), \tau, \tau' \in \mathcal{T}, v, v'\ .\ \rho(\tau)z = \rho(\tau')z \land \langle \tau, a \rangle \Downarrow_a v \land \langle \tau', a \rangle \Downarrow_a v' \land v = v'$
This is contradicted to the hypothesis (2).
Then we know $x \in VAR(e)$.
Assuming $j = \iota(\tau)x \land i \neq j$, by hypothesis (2) where $\forall z^j \in \mathbb{LV} \setminus (\texttt{Diff} \cup \{x^i\})\ .\ \rho(\tau)z = \rho(\tau')z$, we know

$\rho(\tau)x = \rho(\tau')x$, i.e.,
$\forall z^j \in \mathbb{LV} \setminus (\mathtt{Diff})\ .\ \rho(\tau)z = \rho(\tau')z$.
Then we have $v' = v$ by hypothesis (1), which is contradicted to $v' \neq v$.
Then we know $i = \iota(\tau)x$.

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** ψ
This case is proved trivially in the same way as the case of the arithmetic expression. $\square$

Lemma D.4 (Event Inversion)**.** *For all* $c \in \mathcal{C}, \tau_0 \in \mathcal{T}, \epsilon \in \mathcal{E}$*such that* $\langle c, \tau_0 \rangle \to^* \langle \mathtt{skip}, \tau_0{+\!+}\tau_1 \rangle$*, and* $\epsilon \in_{\mathsf{e}} \tau_1$*, if*

- $\epsilon \in \mathcal{E}^{\mathtt{asn}}$*, then either*
  - *there exists* $\tau_1' \in \mathcal{T}, c' \in \mathcal{C}, e$ *such that*

$$\langle c, \tau_0 \rangle \to^* \langle [x \leftarrow e]^l; c', \tau_0{+\!+}\tau' \rangle \to^{\mathsf{assn}} \langle c', \tau_0{+\!+}\tau_1'{+\!+}[\epsilon] \rangle \to^* \langle \mathtt{skip}, \tau_0{+\!+}\tau_1 \rangle$$

  - *or there exists* $\tau_1' \in \mathcal{T}, c' \in \mathcal{C}, \psi$ *such that*

$$\langle c, \tau_0 \rangle \to^* \langle [x \leftarrow \mathtt{query}(\psi)]^l; c', \tau_0{+\!+}\tau_1' \rangle \to^{query} \langle c', \tau_0{+\!+}\tau_1'{+\!+}[\epsilon] \rangle \to^* \langle \mathtt{skip}, \tau_0{+\!+}\tau_1 \rangle$$

- $\epsilon \in \mathcal{E}^{\mathtt{test}}$ *then either*
  - *there exists* $\tau_1' \in \mathcal{T}, c', c_t, c_f, c'' \in \mathcal{C}, b$ *such that*

$$\langle c, \tau_0 \rangle \to^* \langle \mathtt{if}\ ([b]^l, c_t, c_f); c', \tau_0{+\!+}\tau_1' \rangle \to^{if\text{-}b} \langle c'', \tau_0{+\!+}\tau_1'{+\!+}[\epsilon] \rangle \to^* \langle \mathtt{skip}, \tau_0{+\!+}\tau_1 \rangle$$

  - *or there exists* $\tau_1' \in \mathcal{T}, c', c_w, c'' \in \mathcal{C}, b$ *such that*

$$\langle c, \tau_0 \rangle \to^* \langle \mathtt{while}\ ([b]^l, c_w); c', \tau_0{+\!+}\tau_1' \rangle \to^{while\text{-}b} \langle c'', \tau_0{+\!+}\tau_1'{+\!+}[\epsilon] \rangle \to^* \langle \mathtt{skip}, \tau_0{+\!+}\tau_1 \rangle$$

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 assn, we have $\langle [x \leftarrow a]^l, \tau \rangle \to \langle \mathtt{skip}, \tau{+\!+}[(x, l, v)] \rangle$.
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 $c' = \mathtt{skip}$ satisfying
$\langle c, \tau_0 \rangle \to^* \langle [x \leftarrow e]^l; c', \tau_0{+\!+}\tau' \rangle \to^{\mathsf{assn}} \langle c', \tau_0{+\!+}\tau_1'{+\!+}[\epsilon] \rangle \to^* \langle \mathtt{skip}, \tau_0{+\!+}\tau_1 \rangle$.
This case is proved.

case: $c = [x \leftarrow \mathtt{query}(\psi)]^l$
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: $\texttt{while}\,[b]^l\,\texttt{do}\,c$
If the rule applied to is while-t, we have:
$\langle\,\texttt{while}\,[b]^l\,\texttt{do}\,c_w, \tau\rangle \to \langle c_w;\,\texttt{while}\,[b]^l\,\texttt{do}\,c_w, \tau{+\!+}[(b,l,\texttt{true})]\rangle \xrightarrow{*} \langle \texttt{skip}, \tau{+\!+}\tau_1\rangle$,
$(b,l,\texttt{true}) \in \epsilon^{\texttt{test}}$ and $(b,l,\texttt{true}) \in \tau_1$.
Let $\tau' = []$, $c' = \texttt{skip}$ and $c'' = c_w;\,\texttt{while}\,[b]^l\,\texttt{do}\,c_w$, we know that they satisfy
$\langle c, \tau_0\rangle \to^* \langle\,\texttt{while}\,([b]^l, c_w); c', \tau_0{+\!+}\tau_1'\rangle \to^{while-b} \langle c'', \tau_0{+\!+}\tau_1'{+\!+}[\epsilon]\rangle \to^* \langle \texttt{skip}, \tau_0{+\!+}\tau_1\rangle$
This case is proved.
If the rule applied to is while-f, we have
$\langle\,\texttt{while}\,[b]^l\,\texttt{do}\,c_w, \tau\rangle \to^{\text{while-f}} \langle \texttt{skip}, \tau{+\!+}[((b,l,\texttt{false}))]\rangle$, $(b,l,\texttt{true}) \in \epsilon^{\texttt{test}}$, and $(b,l,\texttt{true}) \in \tau_1$.
Let $\tau' = []$, $c' = \texttt{skip}$ and $c'' = \texttt{skip}$, we know that they satisfy
$\langle c, \tau_0\rangle \to^* \langle\,\texttt{while}\,([b]^l, c_w); c', \tau_0{+\!+}\tau_1'\rangle \to^{\text{while-f}} \langle c'', \tau_0{+\!+}\tau_1'{+\!+}[(b,l,\texttt{false})]\rangle \to^* \langle \texttt{skip}, \tau_0{+\!+}\tau_1\rangle$
This case is proved.

case: $\texttt{if}\,([b]^l, c_t, c_f)$
This case is proved in the same way as **case:** $c = [x \leftarrow \texttt{query}(\psi)]^l$. □

Lemma D.5 (Reachable Varibale Inversion)**.** *For all $c \in \mathcal{C}\tau, \tau' \in \mathcal{T}$, if $\langle c, \tau\rangle \to^* \langle c', \tau'\rangle$, and for all $x^l \in \mathbb{LV}_c$ such that $\iota(\tau')x = l$, then $x^l \in \texttt{RD}(\texttt{absinit}(c), c)$.*

$$\forall c \in \mathcal{C}, \tau, \tau' \in \mathcal{T}\,.\,\langle c, \tau\rangle \to^* \langle c', \tau'\rangle \implies \forall x^l \in \mathbb{LV}_c\,.\,\iota(\tau')x = l \implies x^l \in \texttt{RD}(\texttt{absinit}(c), c)$$

Proof Summary: If a variable with the label which is the latest one in the trace, Then by the environment definition, the value associated to this labelled variable is read from the trace.
Then this labelled variable must be reachable at the point of $\texttt{entry}_{c'}$, i.e., $x^l \in \texttt{RD}(\texttt{absinit}(c), c)$.

Proof. Take arbitrary $c \in \mathcal{C}, \tau, \tau' \in \mathcal{T}$ satisfying $\langle c, \tau\rangle \to^* \langle c', \tau'\rangle$, and an arbitrary $x^l \in \mathbb{LV}_c$ satisfying $\iota(\tau')x = l$.
By definition of ι, we know τ' 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 τ_b'.
Then, by the environment definition, we know: $\rho(\tau')x = v$, i.e., x^l is reachable at the point of $\texttt{absinit}(c)$.
By the $in(l)$ operator define in Section 4.3.2, we know x^l is in the $in(\texttt{absinit}(c)$ for prpgram c.
Since $\texttt{RD}(\texttt{absinit}(c), c)$ is a stabilized closure of $in(l)$ for c, we know $x^l \in \texttt{RD}(\texttt{absinit}(c), c)$.
This lemma is proved. □

Events and Dependency Inversions The Inversion Lemmas on *may-dependency* relation, trace and event.

Lemma D.6 (Diff Value Dependency Inversion)**.**

$$\begin{array}{l}\forall c \in \mathcal{C}, \tau_1, \tau_2 \in \mathcal{T}, x^i \in \mathbb{LV}(c)\,.\,\texttt{Diff}_{\texttt{seq}}(\tau_1, \tau_2, x^i) \neq \emptyset \land |\texttt{seq}(\tau_1, x^i)| = |\texttt{seq}(\tau_2, x^i)| \\ \quad \implies \exists \epsilon_1 = (x, i, \_, \_) \in \tau_1, \epsilon_2 = (x, i, \_, \_) \in \tau_2\,.\,\texttt{Diff}(\epsilon_1, \epsilon_2)\end{array}$$

Proof Summary, by unfolding the Difference Sequence Definition 14 and Value Sequence Definition 13.

Proof. Take arbitrary $c \in \mathcal{C}$, let $\tau_1, \tau_2 \in \mathcal{T}, x^i \in \mathbb{LV}(c)$ be the traces and variable satisfying $(\diamond)$ $\texttt{Diff}_{\texttt{seq}}(\tau_1, \tau_2, x^i) \neq \emptyset \land (\star)$ $|\texttt{seq}(\tau_1, x^i)| = |\texttt{seq}(\tau_2, x^i)|$.
Then by Definition 14 and the assumptions $(\diamond)$ and $(\star)$, we know $|\texttt{seq}(\tau_1, x^i)| = |\texttt{seq}(\tau_2, x^i)| \geq 1$.

By seq in Definition 13, we know there is at least an event $(x, i, \_, \_) \in \tau_1$ and $(x, i, \_, \_) \in \tau_2$ in order to have $|\mathtt{seq}(\tau_1, x^i)| = |\mathtt{seq}(\tau_2, x^i)| \geq 1$.
In order to show $\mathtt{Diff}(\epsilon_1,\epsilon_2)$, it is sufficient to a contradiction by assuming
(hc) $\forall \epsilon_1 = (x, i, \_, \_) \in \tau_1, \epsilon_2 = (x, i, \_, \_) \in \tau_2 \, . \, \neg\mathtt{Diff}(\epsilon_1,\epsilon_2)$
By seq in Definition 13 and (hc), we know every value v_1 in $\mathtt{seq}(\tau_1, x^i)$ and v_2 at the same index position of $\mathtt{seq}(\tau_2, x^i)$ are equivalent to each other.
Then we know $\mathtt{Diff}_{\mathtt{seq}}(\tau_1, \tau_2, x^i) \neq \emptyset$. This is contradicted to the hypothesis.
Then we know (hc) is false and this Lemma is proved. □

Lemma D.7 (Diff Control Dependency Inversion)**.**

$$\forall c \in \mathcal{C}, \tau_0 \in \mathcal{T}_0(c), \tau_1, \tau_2 \in \mathcal{T}, x^i, y^j \in \mathbb{LV}(c) \, . \, \mathsf{DEP}(x^i, y^j, \tau_1, \tau_2, \tau_0, c) \wedge |\mathtt{seq}(\tau_1, y^j)| \neq |\mathtt{seq}(\tau_2, y^j)|$$
$$\implies \exists \epsilon_b = (b, i, v, \bullet) \in \tau_1 \, . \, \neg\epsilon_b \in \tau_2 \wedge \forall z \in FV(b), \exists l \in \mathcal{L} \, . \, \mathtt{flowsTo}(z^l, y^j, c)$$

Proof Summary:
Proving by using the Inversion Lemmas D.1, D.2, D.4, and D.5, and the *May-Dependency* definition.

Proof. Take arbitrary $c \in \mathcal{C}$, let $\tau_0 \in \mathcal{T}_0(c), \tau_1, \tau_2 \in \mathcal{T}, x^i, y^j \in \mathbb{LV}(c)$ be the traces and variables satisfying $(\diamond)$ $\mathsf{DEP}(x^i, y^j, \tau_1, \tau_2, \tau_0, c) \wedge (\star)$ $|\mathtt{seq}(\tau_1, y^j)| \neq |\mathtt{seq}(\tau_2, y^j)|$.
To show $(c1)$ $\exists \epsilon_b = (b, i, v, \bullet) \in \tau_1 \, . \, \neg\epsilon_b \in \tau_2 \wedge (c2)$ $\forall z \in FV(b), \exists l \in \mathcal{L} \, . \, \mathtt{flowsTo}(z^l, y^j, c)$
Splitting the conjunction, there are two sub conclusions need to be proved:
Subproof of $(c1)$ $\exists \epsilon_b = (b, i, v, \bullet) \in \tau_1 \, . \, \neg\epsilon_b \in \tau_2$:
To show $(c1)$ it is sufficient to show a contradiction by assuming
(hc) $\neg(\exists \epsilon_b = (b, i, v, \bullet) \in \tau_1 \, . \, \neg\epsilon_b \in \tau_2)$.
By (hc), we know there are two cases
(1) $\forall \epsilon \in \tau_1 \, . \, \epsilon \in \mathcal{E}^{\mathrm{asn}}$.
(2) $\forall \epsilon_b = (b, i, v, \bullet) \in \tau_1 \, . \, \epsilon_b \notin \tau_2$.

sub-case: (1) $\forall \epsilon \in \tau_1 \, . \, \epsilon \in \mathcal{E}^{\mathrm{asn}}$
By Event Inversion Lemma D.4 on every $\epsilon \in \tau_1$, we know program c has the following form $c = [\mathtt{skip}]^*; [x_1 \leftarrow e_1/\mathrm{query}(\psi_1)]^{l_1}; [\mathtt{skip}]^*; \ldots; [\mathtt{skip}]^*$, i.e., c consists of only assignment and skip commands.
Then by the *May-Dependency* in Definition 16 and determinism of evaluation, we know $\forall \tau_0' \in \mathcal{T}_0(c)$ $\langle c, \tau_0 \rangle \to^* \langle [\mathtt{skip}]^l, \tau_0{+\!+}\tau_1 \rangle \wedge \langle c, \tau_0' \rangle \to^* \langle [\mathtt{skip}]^l, \tau_0'{+\!+}\tau_2 \rangle \implies |\mathtt{seq}(\tau_1, y^j)| = |\mathtt{seq}(\tau_2, y^j)|$.
This is contradicted to the condition $(\star)$ $|\mathtt{seq}(\tau_1, y^j)| \neq |\mathtt{seq}(\tau_2, y^j)|$.

sub-case: (2) $\forall \epsilon_b = (b, i, v, \bullet) \in \tau_1 \, . \, \neg\epsilon_b \notin \tau_2$
By Event Inversion Lemma D.4 on every $\epsilon \in \tau_1$,
Since $\neg\epsilon_b \notin \tau_2$, by the determinism of evaluation, we know the two executions are executing the same program.
In the same way by the *May-Dependency* in Definition 16 and we know $\forall \tau_0' \in \mathcal{T}_0(c)$ $\langle c, \tau_0 \rangle \to^* \langle [\mathtt{skip}]^l, \tau_0{+\!+}\tau_1 \rangle \wedge \langle c, \tau_0' \rangle \to^* \langle [\mathtt{skip}]^l, \tau_0'{+\!+}\tau_2 \rangle \implies |\mathtt{seq}(\tau_1, y^j)| = |\mathtt{seq}(\tau_2, y^j)|$.
This is contradicted to the condition $(\star)$ $|\mathtt{seq}(\tau_1, y^j)| \neq |\mathtt{seq}(\tau_2, y^j)|$.
Then we have (hc) is false, and $(c1)$ is proved.
Let $\epsilon_b = (b, i, v, \bullet) \in \tau_1$ be the testing event such that $\neg\epsilon_b \in \tau_2$.
By the $\mathbb{LV}(c)$ in Definition 5, we know $\forall z \in FV(b)$, either z is input variable, we have $z^{\mathrm{in}} \in \mathbb{LV}(c)$, or z is assigned in c, we have $\exists r \in \mathbb{N} \, . \, z^r \in \mathbb{LV}(c)$.
Let r be the label for every $z \in FV(b)$, we prove the second sub-conclusion $(c2)$ as follows.
Subproof of $\forall z \in FV(b), \exists l \in \mathcal{L} \, . \, \mathtt{flowsTo}(z^l, y^j, c)$

By $(\diamond)$ and $(\star)$, we know there exists at least an event $\epsilon_y = (y, j, \_, \_)$ such that $\epsilon_y \in \tau_1$ or $\epsilon_y \in \tau_2$.
Without loss of generalization, we assume $\epsilon_y \in \tau_1$ (The case of $\epsilon_y \notin \tau_1$ and $\epsilon_y \in \tau_2$ will be proved as a symmetric case of this assumption).
Without loss of generalization, let ϵ_b be the testing event, and $\tau_1^1, \tau_1^2, \tau_1^3, \tau_2^1, \tau_2^2 \in \mathcal{T}$ be traces such that $\tau_1 = \tau_1^1 {+\!+} [\epsilon_b] {+\!+} \tau_1^2 {+\!+} [\epsilon_y] {+\!+} \tau_1^3$, and $\tau_2 = \tau_2^1 {+\!+} [\neg\epsilon_b] {+\!+} \tau_2^2$.
Then by Inversion Lemma D.4 on ϵ_y, and ϵ_b, we have the following instance of the first execution in Definition 16,

$$\begin{aligned}
&\langle c, \tau_0\rangle \to^* \langle\, \texttt{if}\ ([b]^{l_b}, c_t, c_f) / \texttt{while}\ [b]^{l_b}\ \texttt{do}\ c_w; c_3', \tau_0 {+\!+} \tau_1^1\rangle \\
&\quad \to^{\text{if-b / while-b}} \langle (c_t; c' / c_f; c') / (c_w;\ \texttt{while}\ [b]^{l_b}\ \texttt{do}\ c_w; c' / [\texttt{skip}]; c'), \tau_0 {+\!+} \tau_1^1 {+\!+} [\epsilon_b]\rangle \\
&\quad \to^* \langle [y \leftarrow e_2 / \texttt{query}(\psi_2)]^j; c'', \tau_0 {+\!+} \tau_1^1 [\epsilon_b] {+\!+} \tau_1^2\rangle \\
&\quad \to^{\text{assn/query}} \langle c'', \tau_0 {+\!+} \tau_1^1 [\epsilon_b] {+\!+} \tau_1^2 {+\!+} [\epsilon_y]\rangle \to^* \langle \texttt{skip}, \tau_0 {+\!+} \tau_1^1 [\epsilon_b] {+\!+} \tau_1^2 {+\!+} [\epsilon_y] {+\!+} \tau_1^3\rangle
\end{aligned} \tag{4}$$

, where $\texttt{if}\ ([b]^{l_b}, c_t, c_f) / \texttt{while}\ [b]^{l_b}\ \texttt{do}\ c_w$ is the conditional command of the assignment commands associated to the ϵ_b by applying Inversion Lemma D.4 on ϵ_b.
The notation $(c_t; c' / c_f; c') / (c_w;\ \texttt{while}\ [b]^{l_b}\ \texttt{do}\ c_w; c' / [\texttt{skip}]; c')$ represents:
In case of $\texttt{if}\ ([b]^{l_b}, c_t, c_f)$, if $\pi_3(\epsilon_b) = \texttt{true}$, we have the evaluation:

$$\langle\, \texttt{if}\ ([b]^{l_b}, c_t, c_f); c', \tau_1 {+\!+} [\epsilon_1] {+\!+} \tau\rangle \to^{\text{if-b}} \langle c_t; c' \tau_1 {+\!+} [\epsilon_1] {+\!+} \tau {+\!+} [\epsilon_b]\rangle$$

The same for case of $\texttt{if}\ ([b]^{l_b}, c_t, c_f)$ with $\pi_3(\epsilon_b) = \texttt{false}$, and case of $\texttt{while}\ [b]^{l_b}\ \texttt{do}\ c_w$ with $\pi_3(\epsilon_b) = \texttt{true}$ and $\pi_3(\epsilon_b) = \texttt{false}$.
By applying Inversion Lemma D.4 on ϵ_b, and the command label consistency, we also have the instance of second execution from Definition 16 as follows,

$$\begin{aligned}
&\langle c, \tau_0'\rangle \to^* \langle\, \texttt{if}\ ([b]^{l_b}, c_t, c_f) / \texttt{while}\ [b]^{l_b}\ \texttt{do}\ c_w; c_3', \tau_0' {+\!+} \tau_2^1\rangle \\
&\quad \to^{\text{if-b / while-b}} \langle (c_t; c' / c_f; c') / (c_w;\ \texttt{while}\ [b]^{l_b}\ \texttt{do}\ c_w; c' / [\texttt{skip}]; c'), \tau_0' {+\!+} \tau_2^1 {+\!+} [\neg\epsilon_b]\rangle \\
&\quad \to^* \langle [\texttt{skip}]^{l'} \tau_0' {+\!+} \tau_2^1 {+\!+} [\neg\epsilon_b] {+\!+} \tau_2^2\rangle
\end{aligned} \tag{5}$$

Then for every $z \in FV(b)$ with label r such that $z^r \in \mathbb{LV}(c)$, to show $\texttt{flowsTo}(z^r, y^j, c)$, by Definition 24, it is sufficient to show:
in the case of $\texttt{while}\ [b]^{l_b}\ \texttt{do}\ c_w$, $y^j \in \mathbb{LV}(c_w)$;
in the case of $\texttt{if}\ ([b]^{l_b}, c_t, c_f)$, $y^j \in \mathbb{LV}(c_t)$ or $y^j \in \mathbb{LV}(c_f)$.

sub-case: $\texttt{if}\ ([b]^{l_b}, c_t, c_f) \wedge \pi_3(\epsilon_b) = \texttt{true}$
In this case, we have the following execution instances for executions i n Equation 4 and 4.

$$\begin{aligned}
&\langle c, \tau_0\rangle \to^* \langle\, \texttt{if}\ ([b]^{l_b}, c_t, c_f); c', \tau_0 {+\!+} \tau_1^1\rangle \to^{\text{if-b}} \langle c_t; c', \tau_0 {+\!+} \tau_1^1 {+\!+} [\epsilon_b]\rangle \to^* \langle [y \leftarrow e_2 / \texttt{query}(\psi_2)]^j; c'', \tau_0 {+\!+} \tau_1^1 {+\!+} [\epsilon_b] {+\!+} \tau_1^2\rangle \\
&\quad \to^{\text{assn/query}} \langle c'', \tau_0 {+\!+} \tau_1^1 {+\!+} [\epsilon_b] {+\!+} \tau_1^2 {+\!+} [\epsilon_y]\rangle \to^* \langle \texttt{skip}, \tau_0 {+\!+} \tau_1^1 {+\!+} [\epsilon_b] {+\!+} \tau_1^2 {+\!+} [\epsilon_y] {+\!+} \tau_1^3\rangle \\
&\langle c, \tau_0'\rangle \to^* \langle\, \texttt{if}\ ([b]^{l_b}, c_t, c_f); c', \tau_0' {+\!+} \tau_2^1\rangle \to^{\text{if-b}} \langle c_f; c', \tau_0' {+\!+} \tau_2^1 {+\!+} [\neg\epsilon_b]\rangle \to^* \langle [\texttt{skip}]^{l'} \tau_0' {+\!+} \tau_2^1 {+\!+} [\neg\epsilon_b] {+\!+} \tau_2^2\rangle
\end{aligned}$$

Then, we know $\tau_1^2 {+\!+} [\epsilon_y] {+\!+} \tau_1^3$ corresponds to evaluation of $c_t; c'$ and τ_2^2 corresponds to evaluation of $c_f; c'$ and $\epsilon_y \notin \tau_2^2$.
If ϵ_y generated from evaluation of c_t, we know $y^j \in \mathbb{LV}(c_t)$ and this case is proved.
If ϵ_y generated from evaluation of c', since $\epsilon_y \notin \tau_2^2$ and τ_2^2 also contains evaluation of c'.
Then there must be another $\texttt{if}$ or $\texttt{while}$ command in c' such that ϵ_y is generated in the first execution but isn't evaluated in the second one.
In that case, there is another $\epsilon_b' \in \tau_1^2$ and $\neg\epsilon_b' \in \tau_2^2$ satisfying the same condition as ϵ_b.

Then we can always choose the ϵ_b be this ϵ_b' and choose $\tau_1^1, \tau_1^2, \tau_1^3, \tau_2^1, \tau_2^2 \in \mathcal{T}$ be traces such that $\tau_1 = \tau_1^1{+\!+}[\epsilon_b]{+\!+}\tau_1^2{+\!+}[\epsilon_y]{+\!+}\tau_1^3$, and $\tau_2 = \tau_2^1{+\!+}[\neg\epsilon_b]{+\!+}\tau_2^2$.
and $\mathbb{TL}(\tau_1^2{+\!+}[\epsilon_y]) \cap \mathbb{TL}(\tau_2^2) = \emptyset$.
Then, since $\mathbb{TL}(\tau_1^2{+\!+}[\epsilon_y]) \cap \mathbb{TL}(\tau_2^2) = \emptyset$, we know $\mathbb{TL}(\tau_1^2{+\!+}[\epsilon_y])$ doesn't contain any label of the program c'.
Then we know ϵ_y isn't generated from evaluating c', i.e., it is generated by only executing c_t.
Then we know $y^j \in \mathbb{LV}(c_t)$.
This case is proved.
The **sub-cases:** $\texttt{if}\ ([b]^{l_b}, c_t, c_f) \land \pi_3(\epsilon_b) = \texttt{false}$, $\texttt{while}\ [b]^{l_b}\ \texttt{do}\ c_w \land \pi_3(\epsilon_b) = \texttt{true}$, and $\texttt{while}\ [b]^{l_b}\ \texttt{do}\ c_w \land \pi_3(\epsilon_b) = \texttt{true}$ are proved in the same way.
This Lemma is proved. □

Lemma D.8 (While Loop Inversion). *For every $\tau, \tau' \in \mathcal{T}, c, c_1, c_2 \in \mathcal{C}$ if $\langle c, \tau\rangle \to^* \langle c_1; c_2, \tau'\rangle$ and $c_1 \in_c c_2$, then there must exist a* `while` *command in c_2 and c_1 must shows up in the body of that* `while` *command, i.e., $\exists l \in \mathbb{N}, b \in \mathcal{B}, c_w \in \mathcal{C}\ .\ (\texttt{while}\ [b]^l\ \texttt{do}\ c_w) \in_c c_2 \land c_1 \in_c c_w$.*

$$\begin{array}{l}\forall \tau, \tau' \in \mathcal{T}, c, c_1, c_2 \in \mathcal{C}\ .\\ \quad \langle c, \tau\rangle \to^* \langle c_1; c_2, \tau'\rangle \implies c_1 \in_c c_2 \implies \exists l \in \mathbb{N}, b \in \mathcal{B}, c_w \in \mathcal{C}\ .\ (\texttt{while}\ [b]^l\ \texttt{do}\ c_w) \in_c c_2 \land c_1 \in_c c_w\end{array}$$

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

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

case: $c = [x \leftarrow e]^l$
By the evaluation rule assn, we have $\langle [x \leftarrow a]^l, \tau\rangle \to \langle \texttt{skip}, \tau{+\!+}[(x, l, v)]\rangle$.
Since there doesn't exist $c_1, c_2 \in \mathcal{C}$ satisfying $\texttt{skip} = c_1; c_2$, this theorem is vacuously true.

case: $c = [x \leftarrow \texttt{query}(\psi)]^l$
By the evaluation rule query, we have $\langle [x \leftarrow \texttt{query}(\psi)]^l, \tau\rangle \to \langle \texttt{skip}, \tau{+\!+}[(x, l, \alpha, v)]\rangle$.
Since there doesn't exist $c_1, c_2 \in \mathcal{C}$ satisfying $\texttt{skip} = c_1; c_2$, this theorem is vacuously true.

case: $c = \texttt{if}\ ([b]^l, c_1, c_2)$
By the evaluation rule query and if-f, and the label consistency, we know:
for all possible c_{t1} and c_{t2} such that c_t has the form $c_t = c_{t1}; c_{t2}$;
all possible c_{f1} and c_{f2} such that c_f has the form $c_f = c_{f1}; c_{f2}$,
$c_{t1} \notin c_{t1}$ and $c_{f1} \notin c_{f2}$.
Then this theorem is vacuously true.

case: $c = c_{s1}; c_{s2}$
By label consistency, we know for every $c_1' \in_c c_{s1}$, $c_1' \notin c_{s2}$.
Then by the induction hypothesis on c_{s1} and c_{s2} separately, we have this case proved.

case: $\texttt{while}\ [b]^l\ \texttt{do}\ c$
By rule while-t, we have:

$$\langle\, \texttt{while}\ [b]^l\ \texttt{do}\ c_w, \tau\rangle \to \langle c_w; \texttt{while}\ [b]^l\ \texttt{do}\ c_w, \texttt{skip}), \tau{+\!+}[\epsilon]\rangle$$

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 $c_2 = c_{w2}; \texttt{while}\ [b]^l\ \texttt{do}\ c_w, \texttt{skip})$ and $c_1 \in_c c_2$.
And we also have the existence of $l = l_b, b$ and c_w, and $\texttt{while}\ [b]^l\ \texttt{do}\ c_w \in_c c_2$ and $c_1 \in c_w$.
If c_w isn't a sequence command, let $c_1 = c_w$, then we have $c_2 = \texttt{while}\ [b]^l\ \texttt{do}\ c_w, \texttt{skip})$ and $c_1 \in_c c_2$.

And we also have the existence of $l = l_b, b$ and c_w, and $\texttt{while}\ [b]^l\ \texttt{do}\ c_w \in_c c_2$ and $c_1 \in c_w$.
This case is proved.
By the evaluation rule while-f, we have $\langle\ \texttt{while}\ [b]^l,\ \texttt{do}\ c_w, \tau\rangle \to \langle[\texttt{skip}]^l, \tau{+}{+}[(b, l, \texttt{false})]\rangle$.
Since there doesn't exist $c_1, c_2 \in \mathcal{C}$ satisfying $\texttt{skip} = c_1; c_2$, this theorem is vacuously true. $\square$

Lemma D.9 (Only `skip` Command doesn't Produce Event)**.** . *For all trace $\tau \in \mathcal{T}$, and $c, c' \in \mathcal{C}$, $\langle c, \tau\rangle \to \langle c', \tau\rangle$ if and only if $c = [\texttt{skip}]; c'$.*

$$\forall \tau \in \mathcal{T}, c, c' \in \mathcal{C}\ .\ \langle c, \tau\rangle \to \langle c', \tau\rangle \Leftrightarrow c = [\texttt{skip}]; c'$$

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

E Soundness of Reachability Bounds Estimation

Theorem E.1 (Soundness of the Reachability Bounds Estimation)**.** *Given a program c with its program-based dependency graph* $\mathtt{G_{prog}}(c) = (\mathtt{V_{prog}}, \mathtt{E_{prog}})$*, we have:*

$$\begin{array}{l}\forall c \in \mathcal{C} \;.\; \mathtt{G_{prog}}(c) = (\mathtt{V_{prog}}, \mathtt{E_{prog}}) \land \mathtt{G_{trace}}(c) = (\mathtt{V_{trace}}, \mathtt{E_{trace}}) \\ \quad \implies \forall (x^l, w_t) \in \mathtt{V_{trace}}, (x^l, w_p) \in \mathtt{V_{prog}}, \tau_0 \in \mathcal{T}_0(c), \tau' \in \mathcal{T}, v \in \mathbb{N} \;. \\ \quad \langle c, \tau_0 \rangle \to^* \langle \mathtt{skip}, \tau_0 +\!\!+ \tau' \rangle \land \langle w^p, \tau_0 \rangle \Downarrow_e v \implies w_t(\tau) \leq v\end{array}$$

Proof. Taking an arbitrary a program c with its program-based dependency graph $\mathtt{G_{prog}}(c) = (\mathtt{V_{prog}}, \mathtt{E_{prog}})$, and an arbitrary pair of labeled variable and weights $(x^l, w) \in \mathtt{V_{prog}}$, and arbitrary $\tau, \tau' \in \mathbb{T}$, $v \in \mathbb{N}$ satisfying
$\langle c, \tau \rangle \to^* \langle \mathtt{skip}, \tau +\!\!+ \tau' \rangle \land \langle \tau, w \rangle \Downarrow_e v$
By Definition of $\mathtt{V_{prog}}$ in $\mathtt{G_{prog}}(c)$, we know $w = \mathtt{absW}(l) = \max\{\mathtt{Tclosure}(\overset{\alpha}{\epsilon}) | \overset{\alpha}{\epsilon} = (l, \_, \_)\}$.
By Lemma E.1, there exists an abstract event in $\mathtt{abstrace}(c)$ of form $(\overset{\alpha}{\epsilon}) = (l, \_, \_)$, corresponding to the assignment command associated to labeled variable x^l.
Let $(\overset{\alpha}{\epsilon}) = (l, dc, l') \in \mathtt{abstrace}(c)$ be this event for some dc and l' such that $(\overset{\alpha}{\epsilon}) = (l, dc, l') \in \mathtt{abstrace}(c)$, by the last step of phase 2, we know $\mathtt{W_{prog}}(x^l) \triangleq \mathtt{Tclosure}(\overset{\alpha}{\epsilon})$. Then, it is sufficient to show:

$$\forall v \in \mathbb{N} \;.\; \langle \mathtt{Tclosure}(\overset{\alpha}{\epsilon}), \tau \rangle \Downarrow_e \mathtt{cnt}(\tau', l) \leq v \mathtt{Tclosure}(\overset{\alpha}{\epsilon})$$

By definition of $\mathtt{Tclosure}(\overset{\alpha}{\epsilon})$:

$$\begin{array}{ll}\mathtt{locb}(\overset{\alpha}{\epsilon}) & \mathtt{locb}(\overset{\alpha}{\epsilon}) \in \mathcal{SMBCST} \\ Incr(\mathtt{locb}(\overset{\alpha}{\epsilon})) + \sum\{\mathtt{Tclosure}(\overset{\alpha'}{\epsilon}) \times \max(\mathtt{Vinvar}(a) + c, 0) | (\overset{\alpha'}{\epsilon}, a, c) \in \mathtt{re}(\mathtt{locb}(\overset{\alpha}{\epsilon}))\} & \mathtt{locb}(\overset{\alpha}{\epsilon}) \notin \mathcal{SMBCST}\end{array}$$

case: $\mathtt{locb}(\overset{\alpha}{\epsilon}) \in \mathcal{SMBCST}$

Proved by the soundness of Local bound in Lemma E.2.

case: $\mathtt{locb}(\overset{\alpha}{\epsilon}) \notin \mathcal{SMBCST}$
To show:

$$\begin{array}{l}\max\{\mathtt{cnt}(\tau')l \mid \forall \tau \in \mathcal{T} \;.\; \langle c, \tau \rangle \to^* \langle \mathtt{skip}, \tau +\!\!+ \tau' \rangle\} \\ \leq Incr(\mathtt{locb}(\overset{\alpha}{\epsilon})) + \sum\{\mathtt{Tclosure}(\overset{\alpha'}{\epsilon}) \times \max(\mathtt{Vinvar}(a) + c, 0) | (\overset{\alpha'}{\epsilon}, a, c) \in \mathtt{re}(\mathtt{locb}(\overset{\alpha}{\epsilon}))\}\end{array}$$

Taking an arbitrary initial trace $\tau_0 \in \mathcal{T}$, executing c with τ_0, let τ be the trace after evaluation, i.e., $\langle c, \tau_0 \rangle \to^* \langle \mathtt{skip}, \tau \rangle$, it is sufficient to show:

$$\mathtt{cnt}(\tau')l \leq Incr(\mathtt{locb}(\overset{\alpha}{\epsilon})) + \sum\{\mathtt{Tclosure}(\overset{\alpha'}{\epsilon}) \times \max(\mathtt{Vinvar}(a) + c, 0) | (\overset{\alpha'}{\epsilon}, a, c) \in \mathtt{re}(\mathtt{locb}(\overset{\alpha}{\epsilon}))\}$$

By the soundness of the (1) Transition Bound and (2) Variable Bound Invariant in [2] Theorem 1, This case is proved. □

Lemma E.1 (Soundness of the Abstract Execution Trace)**.** *Given a program c, we have:*

$$\begin{array}{l}\forall \tau_0, \tau \in \mathcal{T}, \epsilon = (\_, l, \_) \in \mathcal{E} \;.\; \langle c, \tau_0 \rangle \to^* \langle \mathtt{skip}, \tau_0 +\!\!+ \tau \rangle \land \epsilon \in \tau \\ \quad \implies \exists \overset{\alpha}{\epsilon} = (l, \_, \_) \in (\mathcal{L} \times \mathcal{DC}^{\top} \times \mathcal{L}) \;.\; \overset{\alpha}{\epsilon} \in \mathtt{abstrace}(c)\end{array}$$

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

$$\begin{aligned}&\forall \tau \in \mathcal{T} \,.\, \langle c, \tau_0 \rangle \to^* \langle \texttt{skip}, \tau_0 {++} \tau \rangle \land \epsilon \in \tau \\ &\quad \Longrightarrow \exists \overset{\alpha}{\epsilon} = (l, \_, \_) \in (\mathcal{L} \times \mathcal{DC}^\top \times \mathcal{L}) \,.\, \overset{\alpha}{\epsilon} \in \texttt{abstrace}(c)\end{aligned}$$

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

case: $c = [x \leftarrow e]^{l'}$

By the evaluation rule $\textsf{assn}$, we have $\langle [x \leftarrow a]^{l'}, \tau \rangle \to \langle \texttt{skip}, \tau {++} [(x, l', v)] \rangle$, for some $v \in \mathbb{N}$ and $\tau = [(x, l', v)]$.
There are 2 cases, where $l' = l$ and $l' \neq l$.
In case of $l' \neq l$, we know $\epsilon \not\in_{\mathsf{e}} \tau$, then this Lemma is vacuously true.
In case of $l' = l$, by the abstract Execution Trace computation, we know $\texttt{abstrace}(c) = \texttt{abstrace}'([x := e]^l; [\texttt{skip}]^{l_e}) = \{(l, \texttt{absexpr}(e), l_e)\}$
Then we have $\overset{\alpha}{\epsilon} = (l, \texttt{absexpr}(e), l_e)$ and $\overset{\alpha}{\epsilon} \in \texttt{abstrace}(c)$.
This case is proved.

case: $c = [x \leftarrow \texttt{query}(\psi)]^{l'}$

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

case: $\texttt{while}\, [b]^{l_w}\, \texttt{do}\, c$

If the rule applied to is $\textsf{while-t}$, we have
$\langle \texttt{while}\, [b]^{l_w}\, \texttt{do}\, c_w, \tau \rangle \to \langle c_w; \texttt{while}\, [b]^{l_w}\, \texttt{do}\, c_w, \tau_0 {++} [(b, l, \texttt{true})] \rangle$.
Let $\tau_w \in \mathcal{T}$ satisfying following execution:
$\langle c_w, \tau_0 {++} [(b, l_w, \texttt{true})] \rangle \xrightarrow{*} \langle \texttt{skip}, \tau_0 {++} [(b, l_w, \texttt{true})] {++} \tau_w \rangle$
Then we have the following execution:
$\langle \texttt{while}\, [b]^{l_w}\, \texttt{do}\, c_w, \tau \rangle \to \langle c_w; \texttt{while}\, [b]^{l_w}\, \texttt{do}\, c_w, \tau_0 {++} [(b, l_w, \texttt{true})] \rangle \xrightarrow{*} \langle \texttt{while}\, [b]^{l_w}\, \texttt{do}\, c_w, \tau_0 {++} [(b, l_w, \texttt{true})] {++} \tau_w \rangle \xrightarrow{*}$
$\langle \texttt{skip}, \tau_0 {++} [(b, l_w, \texttt{true})] {++} \tau_w {++} \tau_1 \rangle$ for some $\tau_1 \in \mathcal{T}$ and $\tau = [(b, l_w, \texttt{true})] {++} \tau_w {++} \tau_1$.
Then we have 3 cases: (1) $\epsilon =_{\mathsf{e}} (b, l_w, \texttt{true})$, (2) $\epsilon \in \tau_w$ or (3) $\epsilon \in \tau_1$.
In case of (1). $\epsilon =_{\mathsf{e}} (b, l_w, \texttt{true})$, since $\texttt{abstrace}(c) = \texttt{abstrace}'(c; [\texttt{skip}]^{l_e}) = \{(l, \top, \texttt{init}(c_w))\} \cup \cdots$, we have $\overset{\alpha}{\epsilon} = (l, \top, \texttt{init}(c_w))$ and this case is proved.
In case of (2). $\epsilon \in \tau_w$, by induction hypothesis on c_w with the execution $\langle c_w, \tau_0 {++} [(b, l_w, \texttt{true})] \rangle \xrightarrow{*} \langle \texttt{skip}, \tau_0 {++} [(b, l_w, \texttt{true})] {++} \tau_w \rangle$ and trace τ_w, we know there is an abstract event of the form $\overset{\alpha'}{\epsilon} = (l, \_, \_) \in \texttt{abstrace}(c_w)$ where $\texttt{abstrace}(c_w) = \texttt{abstrace}'(c_w; [\texttt{skip}]^{l_e})$.
Let $\overset{\alpha'}{\epsilon} = (l, dc, l')$ for some dc and l' such that $\overset{\alpha}{\epsilon} \in \texttt{abstrace}(c)$.
By definition of $\texttt{abstrace}'$, we have $\texttt{abstrace}'(c_w; [\texttt{skip}]^{l_e}) = \texttt{abstrace}'(c_w) \cup \{(l', dc, l_e) | (l', dc) \in \texttt{absfinal}(c_w)\}$.
There are 2 subcases: (2.1) $\overset{\alpha'}{\epsilon} \in \texttt{abstrace}'(c_w)$ or (2.2) $\overset{\alpha'}{\epsilon} \in \{(l', dc, l_e) | (l', dc) \in \texttt{absfinal}(c_w)\}$.

sub-case: (2.1)

Since $\texttt{abstrace}(c) = \texttt{abstrace}'(c_w) \cup \{(l', dc, l_w) | (l', dc) \in \texttt{absfinal}(c_w)\} \cup \cdots$, we know the abstract event $\overset{\alpha'}{\epsilon} \in \texttt{abstrace}(c)$.
This case is proved.

sub-case: (2.2) $\overset{\alpha'}{\epsilon} \in \{(l', dc, l_e) | (l', dc) \in \texttt{absfinal}(c_w)\}$

In this case, we know $(l, dc) \in \texttt{absfinal}(c_w)$.
Since $\texttt{abstrace}(c) = \texttt{abstrace}'(c_w) \cup \{(l', dc, l_w) | (l', dc) \in \texttt{absfinal}(c_w)\} \cup \cdots$, we know $(l, dc, l_w) \in \{(l', dc, l_w) | (l', dc) \in \texttt{absfinal}(c_w)\}$, i.e., the abstract event $(l, dc, l_w) \in \texttt{abstrace}(c)$ and (l, dc, l_w) has the form $(l, \_, \_)$.
This case is proved.

In case of (3). $\epsilon \in \tau_1$, we know either $\epsilon = (b, l_w, \_)$, or $\epsilon \in \tau'_w$ where $\tau'_w \in \mathcal{T}$ is the trace of executing c_w in an iteration.
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
$\langle \texttt{while}\ [b]^{l_w}\ \texttt{do}\ c_w, \tau_0\rangle \to^{\textsf{while-f}} \langle \texttt{skip}, \tau_0 +\!\!+ [(b, l_w, \texttt{false})]\rangle$, In this case, we have $\tau = [(b, l_w, \texttt{false})]$ and $\epsilon = (b, l_w, \texttt{false})$ (o.w., $\epsilon \not\in_e \tau$ and this lemma is vacuously true) with $l = l_w$.
By the abstract execution trace computation, $\texttt{abstrace}(c) = \{(l, \top, \texttt{init}(c_w))\} \cup \cdots$, we have $\overset{\alpha}{\epsilon} = (l, \top, \texttt{init}(c_w))$ and $\overset{\alpha}{\epsilon} \in \texttt{abstrace}(c)$.
This case is proved.

case: $\texttt{if}\ ([b]^l, c_t, c_f)$
This case is proved in the same way as **case:** $c = \texttt{while}\ [b]^l\ \texttt{do}\ c$.

case: $c = c_{s1}; c_{s2}$
By the induction hypothesis on c_{s1} and c_{s2} separately, and the same step as case (2). of **case:** $c = \texttt{while}\ [b]^l\ \texttt{do}\ c$, we have this case proved. □

Lemma E.2 (Soundness of the Local Bound)**.** *Given a program c, we have:*

$$\forall \overset{\alpha}{\epsilon} = (l, dc, l')\ .\ \max\left\{\texttt{cnt}(\tau')l \,\middle|\, \forall \tau \in \mathcal{T}\ .\ \langle c, \tau\rangle \to^* \langle \texttt{skip}, \tau +\!\!+ \tau'\rangle\right\} \leq \texttt{locb}(\overset{\alpha}{\epsilon})$$

Proof.

sub-case: $l \notin SCC(\texttt{absG}(c))$
In this case, we know variable x^l isn't involved in the body of any $\texttt{while}$ command.
Taking an arbitrary $\tau_0 \in \mathcal{T}$, let $\tau \in \mathcal{T}$ be of resulting trace of executing c with τ, i.e., $\langle c, \tau_0\rangle \to^* \langle \texttt{skip}, \tau\rangle$, we know the assignment command at line l associated with the abstract event $\overset{\alpha}{\epsilon}$ will be executed at most once, i.e.,: $\texttt{cnt}(\tau)l \leq 1$
By $\texttt{locb}$ definition, we know $\texttt{locb}(\overset{\alpha}{\epsilon}) = 1$.
This case is proved.

sub-case: $l \in SCC(\texttt{absG}(c)) \land \overset{\alpha}{\epsilon} \in \texttt{dec}(x)$
in this case, we know $\texttt{locb}(\overset{\alpha}{\epsilon}) \triangleq x$.

sub-case: $l \in SCC(\texttt{absG}(c)) \land \overset{\alpha}{\epsilon} \notin \bigcup_{x \in VAR} \texttt{dec}(x) \land \overset{\alpha}{\epsilon} \notin SCC(\texttt{absG}(c)/\texttt{dec}(x))$
in this case, we know $\texttt{locb}(\overset{\alpha}{\epsilon}) \triangleq x$.
In the two cases above, the soundness is discussed in [2] Section 4 of Paragraph *Discussion on Soundness* in Page 25. □

For every labeled variable in program c, $x^l \in \mathbb{LV}_c$, there is a unique abstract event in program's abstract execution trace $\overset{\alpha}{\epsilon} \in \texttt{abstrace}(c)$ of form $(l, \_, \_)$.

Lemma E.3 (Uniqueness of the Abstract Execution Trace)**.** *Given a program c, we have:*

$$\begin{gathered}\forall \tau_0, \tau \in \mathcal{T}, \epsilon = (\_, l, \_, \_) \in \mathcal{E}^{\text{asn}}\ .\ \langle c, \tau_0\rangle \to^* \langle \texttt{skip}, \tau_0 +\!\!+ \tau\rangle \land \epsilon \in \tau \\ \Longrightarrow \exists! \overset{\alpha}{\epsilon} = (l, \_, \_) \in (\mathcal{L} \times \mathcal{DC}^\top \times \mathcal{L})\ .\ \overset{\alpha}{\epsilon} \in \texttt{abstrace}(c)\end{gathered}$$

Proof. This is proved trivially by induction on the program c. □

F Soundness of Edge Weight Estimation

Theorem F.1 (Soundness of the Edge Weights Estimation)**.** *Given a program c with its program-based dependency graph* $\mathtt{G_{prog}}(c) = (\mathtt{V_{prog}}, \mathtt{E_{prog}})$*, we have:*

$$\begin{array}{l}\forall c \in \mathcal{C} \,.\, \mathtt{G_{prog}}(c) = (\mathtt{V_{prog}}, \mathtt{E_{prog}}) \land \mathtt{G_{trace}}(c) = (\mathtt{V_{trace}}, \mathtt{E_{trace}}) \\ \quad \Longrightarrow \forall (v_1, w^p, v_2) \in \mathtt{E_{trace}}, (v_1, w^t, v_2) \in \mathtt{E_{prog}}, \tau_0 \in \mathcal{T}_0(c), \tau' \in \mathcal{T}, v \in \mathbb{N} \,. \\ \quad \langle c, \tau_0 \rangle \to^* \langle \mathtt{skip}, \tau{+\!+}\tau' \rangle \land \langle w^p, \tau_0 \rangle \Downarrow_e v \implies w_t(\tau) \le v\end{array}$$

Proof. Taking an arbitrary a program c with its program-based dependency graph $\mathtt{G_{prog}}(c) = (\mathtt{V_{prog}}, \mathtt{E_{prog}})$, and an arbitrary pair of labeled variable and weights $(x^l, w) \in \mathtt{V_{prog}}$, and arbitrary $\tau, \tau' \in \mathbb{T}, v \in \mathbb{N}$ satisfying
$\langle c, \tau \rangle \to^* \langle \mathtt{skip}, \tau{+\!+}\tau' \rangle \land \langle \tau, w \rangle \Downarrow_e v$
By Definition of $\mathtt{W_{prog}}$ in $\mathtt{G_{prog}}(c)$, we know $w = \mathtt{absW}(l) = \max\{\mathtt{Tclosure}(\overset{\alpha}{\epsilon}) | \overset{\alpha}{\epsilon} = (l, \_, \_)\}$.
By Lemma E.1, there exists an abstract event in $\mathtt{abstrace}(c)$ of form $(\overset{\alpha}{\epsilon}) = (l, \_, \_)$, corresponding to the assignment command associated to labeled variable x^l.
Let $(\overset{\alpha}{\epsilon}) = (l, dc, l') \in \mathtt{abstrace}(c)$ be this event for some dc and l' such that $(\overset{\alpha}{\epsilon}) = (l, dc, l') \in \mathtt{abstrace}(c)$, by the last step of phase 2, we know $\mathtt{W_{prog}}(x^l) \triangleq \mathtt{Tclosure}(\overset{\alpha}{\epsilon})$. Then, it is sufficient to show:

$$\forall v \in \mathbb{N} \,.\, \langle \mathtt{Tclosure}(\overset{\alpha}{\epsilon}), \tau \rangle \Downarrow_e \mathtt{cnt}(\tau', l) \le v \mathtt{Tclosure}(\overset{\alpha}{\epsilon})$$

By definition of $\mathtt{Tclosure}(\overset{\alpha}{\epsilon})$:

$$\begin{array}{ll}\mathtt{locb}(\overset{\alpha}{\epsilon}) & \mathtt{locb}(\overset{\alpha}{\epsilon}) \in \mathcal{SMBCST} \\ Incr(\mathtt{locb}(\overset{\alpha}{\epsilon})) + \sum\{\mathtt{Tclosure}(\overset{\alpha'}{\epsilon}) \times \max(\mathtt{Vinvar}(a) + c, 0) | (\overset{\alpha'}{\epsilon}, a, c) \in \mathtt{re}(\mathtt{locb}(\overset{\alpha}{\epsilon}))\} & \mathtt{locb}(\overset{\alpha}{\epsilon}) \notin \mathcal{SMBCST}\end{array}$$

case: $\mathtt{locb}(\overset{\alpha}{\epsilon}) \in \mathcal{SMBCST}$

Proved by the soundness of Local bound in Lemma E.2.

case: $\mathtt{locb}(\overset{\alpha}{\epsilon}) \notin \mathcal{SMBCST}$
To show:

$$\begin{array}{l}\max\{\mathtt{cnt}(\tau')l \mid \forall \tau \in \mathcal{T} \,.\, \langle c, \tau \rangle \to^* \langle \mathtt{skip}, \tau{+\!+}\tau' \rangle\} \\ \le Incr(\mathtt{locb}(\overset{\alpha}{\epsilon})) + \sum\{\mathtt{Tclosure}(\overset{\alpha'}{\epsilon}) \times \max(\mathtt{Vinvar}(a) + c, 0) | (\overset{\alpha'}{\epsilon}, a, c) \in \mathtt{re}(\mathtt{locb}(\overset{\alpha}{\epsilon}))\}\end{array}$$

Taking an arbitrary initial trace $\tau_0 \in \mathcal{T}$, executing c with τ_0, let τ be the trace after evaluation, i.e., $\langle c, \tau_0 \rangle \to^* \langle \mathtt{skip}, \tau \rangle$, it is sufficient to show:

$$\mathtt{cnt}(\tau')l \le Incr(\mathtt{locb}(\overset{\alpha}{\epsilon})) + \sum\{\mathtt{Tclosure}(\overset{\alpha'}{\epsilon}) \times \max(\mathtt{Vinvar}(a) + c, 0) | (\overset{\alpha'}{\epsilon}, a, c) \in \mathtt{re}(\mathtt{locb}(\overset{\alpha}{\epsilon}))\}$$

By the soundness of the (1) Transition Bound and (2) Variable Bound Invariant in [2] Theorem 1, This case is proved. □

G Soundness of Adaptivity Computation Algorithm

Theorem G.1 (Soundness of $\mathsf{AdaptSearch}$)**.** *For every program c, given its* Program-Based Dependency Graph $\mathtt{G_{prog}}$,

$$\mathsf{AdaptSearch}(\mathtt{G_{prog}}) \geq A_{\mathtt{prog}}(\mathtt{G_{prog}}).$$

proof Summary:
1. for every two vertices x, y with a walk $k_{x,y}$ from x to y on $\mathtt{G_{prog}}$,
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 G.1, $\mathtt{len^q}$ of this walk is bound by the longest walk of this SCC.
2.3 The output of $\mathsf{AdaptSearch}(\mathtt{G_{prog}})$ 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 G.1, $\mathtt{len^q}$ 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 $\mathsf{AdaptSearch}(\mathtt{G_{prog}})$ is greater than sum up the $\mathtt{len^q}$ of longest walk in every SCC that each sub-walk belongs to.
4. Then we have $\mathsf{AdaptSearch}(\mathtt{G_{prog}}(c)) \geq A_{\mathtt{prog}}(c)$.

Proof. Taking arbitrary program $c \in \mathcal{C}$, let $\mathtt{G_{prog}}(c) = (\mathtt{V_{prog}}, \mathtt{E_{prog}}, \mathtt{W_{prog}}, \mathtt{Q_{prog}})$ be its program based dependency graph.
Taking an arbitrary walk $k_{x,y} \in \mathcal{WK}(\mathtt{G_{prog}})$, with vertices sequence $(x, s_1, \cdots, y)$, it is sufficient to show:

$$\mathtt{len^q}(k_{x,y}) = \mathtt{len}(s | s \in (x, s_1, \cdots, y) \land \mathtt{Q}(s) = 1) \leq \mathsf{AdaptSearch}(\mathtt{G_{prog}}(c))$$

By line:3 of $\mathsf{AdaptSearch}(\mathtt{G_{prog}})$ algorithm defined in Algorithm 1, let $\mathtt{G^{SCC}}_1, \cdots, \mathtt{G^{SCC}}_n$ be all the strong connected components on $\mathtt{G_{prog}}$ with $0 \leq n \leq |\mathtt{V}|$, where each $\mathtt{G^{SCC}}_i = (\mathtt{V}_i, \mathtt{E}_i, \mathtt{W}_i, \mathtt{Q}_i)$,
By line:5-6 in Algorithm 1, let $\mathtt{adapt_{scc}}[\mathtt{G^{SCC}}_i]$ be the result of $\mathsf{AdaptSearch_{scc}}(\mathtt{G^{SCC}}_i)$ for each $\mathtt{G^{SCC}}_i$.
There are 2 cases:

case: x, y on the same SCC
Let $\mathtt{G^{SCC}}$ be this SCC where vertices x and y on, by Lemma G.1, we know

$$\mathtt{len^q}(k_{x,y}) \leq \max\{\mathtt{len^q}(k) | k \in \mathcal{WK}(\mathtt{G^{SCC}})\} \leq \mathsf{AdaptSearch}_{scc}(\mathtt{G^{SCC}})$$

By line:15 and line:18 in $\mathsf{AdaptSearch}(\mathtt{G_{prog}})$ algorithm in Algorithm 1, let $\mathtt{adapt}$ be the output value, we know $\mathsf{AdaptSearch}(\mathtt{G_{prog}}(c)) = \mathtt{adapt} \geq \mathtt{adapt_{tmp}} \geq \mathtt{adapt_{scc}}(\mathtt{SSC})$.
i.e.,

$$\mathtt{len^q}(k_{x,y}) \leq \mathsf{AdaptSearch}(\mathtt{G_{prog}}(c))$$

This case is proved.

case: x, y on different SSC
Let $\mathtt{G^{SCC}}_x, \mathtt{G^{SCC}}_1, \cdots, \mathtt{G^{SCC}}_m, \mathtt{G^{SCC}}_y, 0 \leq m$ 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:
$k_x = (x, s_1, \cdots, s_{scc_x})$;

$k_1 = (s_{scc_x}, \cdots, s_{scc_1})$;
$\cdots$
$k_y = (s_{scc_m}, \cdots, s_y)$;
where $k_x \in \mathcal{WK}(\mathtt{G^{SCC}}_x), \cdots, k_y \in \mathcal{WK}(\mathtt{G^{SCC}}_y)$.
By Lemma G.1, we know for each walk k_i:

$$\mathtt{len^q}(k_i) \leq \max\{\mathtt{len^q}(k_i) | k_i \in \mathcal{WK}(\mathtt{G^{SCC}}_i)\} \leq \mathsf{AdaptSearch}_{scc}(\mathtt{G^{SCC}}_i) = \mathtt{adapt_{scc}}[\mathtt{G^{SCC}}_\mathtt{i}]$$

Then we have:

$$\mathtt{len^q}(k_{x,y}) = \mathtt{len^q}(k_x) + \mathtt{len^q}(k_1) + \cdots + \mathtt{len^q}(k_y) \leq \mathtt{adapt_{scc}}[\mathtt{G^{SCC}}_1] + \mathtt{adapt_{scc}}[\mathtt{G^{SCC}}_1] + \cdots + \mathtt{adapt_{scc}}[\mathtt{G^{SCC}}_\mathtt{y}] \leq \mathtt{adapt}$$

, where $\mathtt{adapt}$ is the output of $\mathsf{AdaptSearch}(\mathtt{G_{prog}})$. This case is proved. $\square$

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

$$\forall c \in \mathcal{C}, \mathtt{G^{SCC}} \in \mathcal{G} \,.\, \mathtt{G^{SCC}} \subseteq_{\mathtt{graph}} \mathtt{G_{prog}}(c) \implies \max\{\mathtt{len^q}(k) | k \in \mathcal{WK}(\mathtt{G^{SCC}})\} \leq \mathsf{AdaptSearch}_{scc}(\mathtt{G^{SCC}})$$

ProofSummary:
(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 $\mathtt{G^{SCC}}$.
(2) For every path $p^i_{x,x} = (x, v_1, \cdots, x) \in \mathcal{PATH}(p_{x,x})$, $\mathtt{flowcapacity}(p^i_{x,x})$ is the maximum visiting times for every $v \in (x, v_1, \cdots, x)$, $\mathtt{visit}(s)(s_1, \cdots, x)) \leq \mathtt{flowcapacity}(p^i_{x,x})$;
(3) $\mathtt{querynum}(p^i_{x,x}) * \mathtt{flowcapacity}(p^i_{x,x}) \geq \mathtt{len}(s | s \in (s_1, \cdots, x) \land \mathtt{Q}(s) = 1) = \mathtt{len^q}(k)$,
(4) Then, the $\max\limits_{p^i_{x,x} \in \mathcal{PATH}(p_{x,x})} \geq \max\{\mathtt{len^q}(k_{x,x}) | k_{x,x} \in \mathcal{WK}(k_{x,x})\}$
(5) Then, $\max\{\mathtt{querynum}(p^i_{x,x}) * \mathtt{flowcapacity}(p^i_{x,x}) | x \in \mathtt{G^{SCC}} \land p^i_{x,x} \in \mathcal{PATH}(p_{x,x})\} \geq \max\{\mathtt{len^q}(k^i_{x,x}) | x \in \mathtt{G^{SCC}} \land k^i_{x,x} \in \mathcal{WK}(k_{x,x})\}$
(6) We also know by the property of SCC, $\forall x, y \in \mathtt{G^{SCC}}$, let $k_{x,y}$ be arbitrary walk on $\mathtt{G^{SCC}}$, $\mathtt{len^q}(k_{x,y}) \leq \max\{\mathtt{len^q}(k^i_{x,x}) | k^i_{x,x} \in \mathcal{WK}(k_{x,x})\}$.
(7) Then,$\max\{\mathtt{len^q}(k^i_{x,x}) | x \in \mathtt{G^{SCC}} \land k^i_{x,x} \in \mathcal{WK}(k_{x,x})\} \geq \max\{\mathtt{len^q}(k^i_{x,y}) | x, y \in \mathtt{G^{SCC}} \land k^i_{x,y} \in \mathcal{WK}(k_{x,y})\}$
i.e., $\max\{\mathtt{len^q}(k^i_{x,x}) | x \in \mathtt{G^{SCC}} \land k^i_{x,x} \in \mathcal{WK}(k_{x,x})\} \geq \max\{\mathtt{len^q}(k) | k \in \mathcal{WK}(\mathtt{G^{SCC}})\} = A_{\mathtt{prog}}(\mathtt{G^{SCC}})$.
(8) We also know $\mathsf{AdaptSearch}_{scc}(\mathtt{G^{SCC}}) = \max\{\mathtt{querynum}(p^i_{x,x}) * \mathtt{flowcapacity}(p^i_{x,x}) | x \in \mathtt{G^{SCC}} \land p^i_{x,x} \in \mathcal{PATH}(p_{x,x})\}$ by the $\mathsf{AdaptSearch}_{scc}$ algorithm.
Then we have $\mathsf{AdaptSearch}_{scc}(\mathtt{G^{SCC}}) \geq A_{\mathtt{prog}}(\mathtt{G^{SCC}})$

Proof. Taking arbitrary program $c \in \mathcal{C}$, let $\mathtt{G_{prog}}(c) = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{Q})$ be its program based dependency graph and $\mathtt{G^{SCC}} = (\mathtt{V_{scc}}, \mathtt{E_{scc}}, \mathtt{W_{scc}}, \mathtt{Q_{scc}})$ be an arbitrary sub SCC graph of $\mathtt{G_{prog}}$.
There are 2 cases:

case: $\mathtt{G^{SCC}}$ contains no edge and only 1 vertex v, i.e., $|\mathtt{E}| = 0 \land |\mathtt{V}| = 1$
In this case there is no walk in this graph, i.e., $\mathcal{WK}(\mathtt{G^{SCC}}) = \emptyset$.
The adaptivity is $\mathtt{Q}(v)$.
This case is proved.

case: $\mathtt{G^{SCC}}$ contains at least 1 edge and at least 1 vertex v, i.e., $1 \leq |\mathtt{E}| \land 1 \leq |\mathtt{V}|$
Taking arbitrary walk $k_{x,y} \in \mathcal{WK}(\mathtt{G^{SCC}})$, with vertices sequence $(x, s_1, \cdots, y)$, it is sufficient to show:

$$\mathtt{len^q}(k_{x,y}) = \mathtt{len}(s | s \in (x, s_1, \cdots, y) \land \mathtt{Q}(s) = 1) \leq \mathsf{AdaptSearch}_{scc}(\mathtt{G^{SCC}})$$

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

$$\mathtt{len}(s | s \in (x, s_1, \cdots, y) \wedge \mathtt{Q}(s) = 1) \leq \mathtt{dfs}(\mathtt{G}^{\mathtt{SCC}}, \mathtt{x}, \mathtt{visited}).$$

Let $\{v_1, \cdots, x\}$ be the set of all the distinct vertices of $k_{x,y}$'s vertices sequence $(x, s_1, \cdots, y)$, and $(v_1, \cdots, x)$ be a subsequence containing all the vertices in $\{x, v_1, \cdots, y\}$.
By the definition of walk, there is a path $p_{x,y}$ from x to y with this vertices sequence: $(x, v_1, \cdots, y)$.
By line:13 of the $\mathtt{dfs}(\mathtt{G}^{\mathtt{SCC}}, \mathtt{x}, \mathtt{visited})$ in Algorithm 2,
we know $\mathtt{dfs}(\mathtt{G}^{\mathtt{SCC}}, \mathtt{x}, \mathtt{visited}) = r[x]$ and $r[x] = \max\{\mathtt{flowcapacity}(p) \times \mathtt{querynum}(p) | p \in \mathcal{PATH}_{x,x}(\mathtt{G}^{\mathtt{SCC}})\}$, where $\mathcal{PATH}_{x,x}(\mathtt{G}^{\mathtt{SCC}})$ is a subset of $\mathcal{PATH}_{x,x}(\mathtt{G}^{\mathtt{SCC}})$, in which every path starts from x and goes back to x.
By the property of strong connected graph, we know in this case $\mathcal{PATH}_{x,x}(\mathtt{G}^{\mathtt{SCC}}) \neq \emptyset$ and there are 2 cases, $x = y$ and $x \neq y$.

case: $x = y$
In this case, we know $p_{x,y} \in p \in \mathcal{PATH}_{x,x}(\mathtt{G}^{\mathtt{SCC}})$, then it is sufficient to show:

$$\mathtt{len}(s | s \in (x, s_1, \cdots, y) \wedge \mathtt{Q}(s) = 1) \leq \mathtt{flowcapacity}(p_{x,y}) \times \mathtt{querynum}(p_{x,y})$$

By line:7 and line:13 in Algorithm 2, we know $\mathtt{flowcapacity}(p_{x,y})$ is the maximum visiting times for every $v \in (x, v_1, \cdots, y)$,
we know for every s in the vertices sequence of walk $k_{x,y}$, $\mathtt{visit}(s)(x, s_1, \cdots, y) \leq \mathtt{flowcapacity}(p_{x,y})$
Also by line:8 and line:13 in Algorithm 2, we know $\mathtt{querynum}(p_{x,y})$ is the number of vertices with $\mathtt{Q}$ equal to 1,
Then we know
$\mathtt{len}(s | s \in (x, s_1, \cdots, y) \wedge \mathtt{Q}(s) = 1) \leq \mathtt{flowcapacity}(p_{x,y}) \times \mathtt{querynum}(p_{x,y})$
This case is proved.

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 $(y, v'_1, \cdots, x)$, we have a path $p_{x,x}$, which is the path $p_{x,y}$ concatenated by path $p_{y,x}$ with vertices sequence $(x, v_1, \cdots, y, v'_1, \cdots, v'_m, x)$, where $m \geq 0$.
Then in this case, it is sufficient to show:

$$\mathtt{len}(s | s \in (x, s_1, \cdots, y) \wedge \mathtt{Q}(s) = 1) \leq \mathtt{flowcapacity}(p_{x,x}) \times \mathtt{querynum}(p_{x,x})$$

Since $\mathtt{flowcapacity}(p_{x,y} + p_{y,x})$ is the maximum visiting times for every $v \in (x, v_1, \cdots, y, v'_1, \cdots, x)$,
By line:7 in Algorithm 2, we know $\mathtt{flowcapacity}(p_{x,y})$ is the maximum visiting times for every $v \in (x, v_1, \cdots, y)$,
we know for every s in the vertices sequence of walk $k_{x,y}$, $\mathtt{visit}(s)(x, s_1, \cdots, y) \leq \mathtt{flowcapacity}(p_{x,y})$
Also by line:8 in Algorithm 2, we know $\mathtt{querynum}(p_{x,y})$ is the number of vertices with $\mathtt{Q}$ equal to 1,
Then we know
$\mathtt{len}(s | s \in (x, s_1, \cdots, y) \wedge \mathtt{Q}(s) = 1) \leq \mathtt{flowcapacity}(p_{x,y}) \times \mathtt{querynum}(p_{x,y}) = r[y]$
By line:13, we also know $r[x] = \max(r[x], r[v'_m] + \mathtt{flowcapacity}(p_{x,x}) \times \mathtt{querynum}(p_{x,x})$, and $r[y] \leq r[w'_m]$ then we know $r[y] \leq r[x]$, i.e., $\mathtt{len}(s | s \in (x, s_1, \cdots, y) \wedge \mathtt{Q}(s) = 1) \leq r[x]$
This case is proved. □

H Conditional Completeness of Adaptivity Computation Algorithm

Theorem H.1 (Conditional Completeness of $\mathsf{AdaptSearch}$). *For every program c, given its* Program-Based Dependency Graph $\mathtt{G_{prog}}$, *if* $\mathtt{G_{prog}}(c)$ *is acyclic directed, then*

$$\mathsf{AdaptSearch}(\mathtt{G_{prog}}) = A_{\mathtt{prog}}(\mathtt{G_{prog}}).$$

proof Summary:
1. for every two vertices x, y with a walk $k_{x,y}$ from x to y on $\mathtt{G_{prog}}$,
2 since $\mathtt{G_{prog}}$ 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 sum of the query annotation.
From Algorithm 2, every vertex is a SCC with only one vertex and zeor edge, its adaptivity is exactly its query annotation.
=> $\mathtt{len^q}(k_{x,y}) = \sum_{v_i \in ssc_i} \mathtt{Adapt[scc_i]}$
This is proved.

Proof. Taking arbitrary program $c \in \mathcal{C}$, let $\mathtt{G_{prog}}(c) = (\mathtt{V_{prog}}, \mathtt{E_{prog}}, \mathtt{W_{prog}}, \mathtt{Q_{prog}})$ be its program based dependency graph.
Let the walk $k_{max} \in \mathcal{WK}(\mathtt{G_{prog}}(c))$ be the finite walk with the longest query length, and the vertices sequence $(s_1, \cdots, s_n)$, it is sufficient to show:

$$\mathtt{len^q}(k_{max}) = \mathtt{len}(s | s \in (s_1, \cdots, s_n) \wedge \mathtt{Q_{prog}}(s) = 1) = \mathsf{AdaptSearch}(\mathtt{G_{prog}}(c))$$

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

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

By line: 13, through searching on all the vertices connected on $\mathtt{G_{prog}}(c)$ from the starting node s_i, we know that $\mathsf{AdaptSearch}(\mathtt{G_{prog}}(c))$ finds this path $p_{max} = (s_1, \cdots, s_n)$.
Then, it is sufficient to show

$$\mathtt{adapt_{p_{max}}} = \sum_{s_i \in (s_1, \cdots, s_n)} \mathtt{Q_{prog}}(s_i).$$

By line: 15, let $\mathtt{G^{SCC}}_1, \cdots, \mathtt{G^{SCC}}_m$ be all the SCC, where each vertex in $(s_1, \cdots, s_n)$ belongs to, it is sufficient to show:

$$\sum_{\mathtt{G^{SCC}}_i \in (\mathtt{G^{SCC}}_1, \cdots, \mathtt{G^{SCC}}_m)} \mathtt{adapt_{scc}}[\mathtt{G^{SCC}}_\mathtt{i}] = \sum_{s_i \in (s_1, \cdots, s_n)} \mathtt{Q_{prog}}(s_i).$$

By line:3 in Algorithm 1, let $\mathtt{G^{SCC}}_\mathtt{i} = (\mathtt{V}_i, \mathtt{E}_i, \mathtt{W}_i, \mathtt{Q}_i)$ for $\mathtt{G^{SCC}}_i \in (\mathtt{G^{SCC}}_1, \cdots, \mathtt{G^{SCC}}_m)$ be the SCC found by the standard Algorithm.,
Then, by the property of ACG, we know every $\mathtt{G^{SCC}}_i$ is a single vertex v_i without edge and $\mathtt{Q}_i$ is the query annotation of v_i, i.e., $\mathtt{V}_i = \{s_i\}$ and $\mathtt{Q}_i = \{(s_i, \mathtt{Q_{prog}}(s_i))\}$.
So we know $n = m$.

Also by Algorithm 2 line: 4-5, we know $\mathtt{adapt}_{\mathtt{scc}}[\mathtt{G}^{\mathtt{SCC}}{}_{\mathtt{i}}] = \mathtt{Q}_{\mathtt{prog}}(s_i)$.
Then we can conclude:

$$\sum_{\mathtt{G}^{\mathtt{SCC}}{}_i \in (\mathtt{G}^{\mathtt{SCC}}{}_1, \cdots, \mathtt{G}^{\mathtt{SCC}}{}_m)} \mathtt{adapt}_{\mathtt{scc}}[\mathtt{G}^{\mathtt{SCC}}{}_{\mathtt{i}}] = \sum_{\mathtt{G}^{\mathtt{SCC}}{}_i \in (\mathtt{G}^{\mathtt{SCC}}{}_1, \cdots, \mathtt{G}^{\mathtt{SCC}}{}_n)} \mathtt{Q}_{\mathtt{prog}}(s_i) = \sum_{s_i \in (s_1, \cdots, s_n)} \mathtt{Q}_{\mathtt{prog}}(s_i).$$

So we have (1). "the existence" proved. In order to show p_{max} is the longest path found and $\mathtt{adapt}_{\mathtt{p_{max}}}$ is returned by $\mathsf{AdaptSearch}(\mathtt{G}_{\mathtt{prog}}(c))$, by line: 18, it is sufficient to show $\mathtt{adapt} = \mathtt{adapt}_{\mathtt{p_{max}}}$.
It is sufficient to show a contradiction if $\mathtt{adapt} \neq \mathtt{adapt}_{\mathtt{p_{max}}}$ in following two cases:

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

case: $\mathtt{adapt} > \mathtt{adapt}_{\mathtt{p_{max}}}$
, Let p'_{max} be the path such that $\mathtt{adapt} = \mathtt{adapt}_{\mathtt{p'_{max}}} > \mathtt{adapt}_{\mathtt{p_{max}}}$ with vertices sequence $(s'_1, \cdots, s'_n)$.
Then we know p'_{max} corresponds to a walk k'_{max} with the same vertices sequence.
Then by the same proof above, we know $\mathtt{len}^{\mathtt{q}}(k'_{max}) = adapt_{p'_{max}}$ and $\mathtt{len}^{\mathtt{q}}(k'_{max}) > \mathtt{len}^{\mathtt{q}}(k_{max})$.
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. □

References

[1] Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Leon Roth. Preserving statistical validity in adaptive data analysis. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 117–126, 2015.

[2] Moritz Sinn, Florian Zuleger, and Helmut Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *Journal of automated reasoning*, 59(1):3–45, 2017.