Program Analysis for Adaptivity Analysis

Contents

1 `Labeled While` Language

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	$\sim$	::=	$< \mid \leq \mid ==$	
Label	l	$\in$	$\mathbb{N} \cup \{\mathtt{in}, \mathtt{ex}\}$	
Arithmetic Expression	a	::=	$n \in \mathbb{N}^{\infty} \mid x \mid a \oplus_a a \mid \mathtt{log}\ a \mid \mathtt{sign}\ a$	
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	::=	$n \mid \mathtt{true} \mid \mathtt{false} \mid [] \mid [v, \dots, v]$	
Query Expression	ψ	::=	$\alpha \mid a \mid \psi \oplus_a \psi \mid \chi[a]$	
Query Value	α	::=	$n \mid \chi[n] \mid \alpha \oplus_a \alpha \mid n \oplus_a \chi[n] \mid \chi[n] \oplus_a n$	
Labeled Command	c	::=	$[x \leftarrow e]^l \mid [x \leftarrow \mathtt{query}(\psi)]^l \mid \mathtt{while}\ [b]^l\ \mathtt{do}\ c$	
			$\mid c; c \mid \mathtt{if}\ ([b]^l, c, c) \mid [\mathtt{skip}]^l$	
Event	ϵ	::=	$(x, l, v, \bullet) \mid (x, l, v, \alpha)$	Assignment Event
			$\mid (b, l, v, \bullet)$	Testing Event

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
$\mathcal{E}^{\texttt{asn}}$	:	Set of Assignment Events
$\mathcal{E}^{\texttt{test}}$	:	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
$\mathcal{QD}$	:	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 `Labeled While` Language

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 1: Trace-based Operational Semantics for Language.

The trace based operational semantics rules are defined in Figure 1.
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 set of assigned variables and labeled variables for a labeled command c, defined in Definition 1 and 2.
$FV : e \rightarrow \mathcal{P}(\mathcal{VAR})$, 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 1 (Assigned Variables ($\mathbb{AV} : \mathcal{C} \rightarrow \mathcal{P}(\mathcal{VAR} \times \mathbb{N})$)).

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

Definition 2 (labelled Variables $\mathbb{LV}$)**.**

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

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

Definition 3 (Query Variables ($\mathbb{QV} : \mathcal{C} \to \mathcal{P}(\mathcal{LV})$))**.** *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.1.

Lemma 1.1 (Uniqueness of the Labeled Variables)**.** *For every program $c \in \mathcal{C}$ and every two labeled variables such that $x^i, y^j \in \mathbb{LV}(c)$, 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 4 (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 5 (Event Equivalence)**.** *Two events $\epsilon_1, \epsilon_2 \in \mathcal{E}$ are equivalent, denoted as $\epsilon_1 = \epsilon_2$ if and only if:*

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

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

2.2 Trace

Definition 6 (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 7. *(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:*

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

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

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

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

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

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

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.2, specifically the trace has the property that its length never decreases during the program execution.

Lemma 2.1 (Trace Non-Decreasing)**.** *For every program $c \in \mathcal{C}$ and traces $\tau, \tau' \in \mathcal{T}$, if $\langle c, \tau \rangle \to^* \langle \texttt{skip}, \tau' \rangle$, 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 4. So we have the following Corollary 2.0.1 with proof in Appendix A.3.

Corollary 2.0.1. *For every event and a trace $\tau \in \mathcal{T}$, if $\epsilon \in \tau$, then there exist another event $\epsilon' \in \mathcal{E}$ and traces $\tau_1, \tau_2 \in \mathcal{T}$ such that $\tau_1 {+\!+} [\epsilon'] {+\!+} \tau_2 = \tau$ with 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 Adapativity

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

3.1 Semantics-based Dependency Graph

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

Definition 8 (Semantics-based Dependency Graph)**.** *Given a program c, its* semantics-based dependency graph $\mathtt{G}_{\mathtt{trace}}(c) = (\mathtt{V}_{\mathtt{trace}}(c), \mathtt{E}_{\mathtt{trace}}(c), \mathtt{W}_{\mathtt{trace}}(c), \mathtt{Q}_{\mathtt{trace}}(c))$ *is defined as follows,*

$$
\begin{array}{lll}
\textit{Vertices} & \mathtt{V}_{\mathtt{trace}}(c) & := \left\{x^l \mid x^l \in \mathbb{LV}(c)\right\} \\
\textit{Directed Edges} & \mathtt{E}_{\mathtt{trace}}(c) & := \left\{(x^i, y^j) \mid x^i, y^j \in \mathbb{LV}(c) \land \mathsf{DEP}_{\mathsf{var}}(x^i, y^j, c)\right\} \\
\textit{Weights} & \mathtt{W}_{\mathtt{trace}}(c) & := \{(x^l, w) \mid w : \mathcal{T}_0(c) \to \mathbb{N} \land x^l \in \mathbb{LV}(c) \\
 & & \quad \land \forall \tau_0 \in \mathcal{T}_0(c), \tau' \in \mathcal{T} \,.\, \langle c, \tau_0 \rangle \to^* \langle \mathtt{skip}, \tau_0 {+\!+} \tau' \rangle \land w(\tau_0) = \mathtt{cnt}(\tau', l)\} \\
\textit{Query Annotations} & \mathtt{Q}_{\mathtt{trace}}(c) & := \left\{(x^l, n) \mid x^l \in \mathbb{LV}(c) \land n = 1 \Leftrightarrow x^l \in \mathbb{QV}(c) \land n = 0 \Leftrightarrow x^l \notin \mathbb{QV}(c)\right\}
\end{array},
$$

where ${+\!+} : \mathcal{T} \to \mathcal{T} \to \mathcal{T}$ *is the trace concatenation operator, which combines two traces, and* $\mathtt{cnt} : \mathcal{T} \to \mathbb{N} \to \mathbb{N}$ *is the counting operator, which counts the occurrence of of a labeled variable in the trace. All the definition details are in the appendix. A semantics-based dependency graph* $\mathtt{G}_{\mathtt{trace}}(c) = (\mathtt{V}_{\mathtt{trace}}(c), \mathtt{E}_{\mathtt{trace}}(c), \mathtt{W}_{\mathtt{trace}}(c), \mathtt{Q}_{\mathtt{trace}}(c))$ *is* well-formed *if and only if* $\{x^l \mid (x^l, w) \in \mathtt{W}_{\mathtt{trace}}(c)\} = \mathtt{V}_{\mathtt{trace}}(c)$.

There are four components in this graph.

1. The vertices $\mathtt{V}_{\mathtt{trace}}(c)$ of a program c are all its labeled variables, $\mathbb{LV}(c)$ which are statically collected.

2. $\mathtt{Q}_{\mathtt{trace}}(c)$ contains the *query annotation* for every vertex $x^l \in \mathtt{V}_{\mathtt{trace}}(c)$. It indicates whether x^l comes from a query request (1) or not (0) by checking if the labeled variable x^l of the vertex is in $\mathbb{QV}(c)$.

3. Edges in $\mathtt{E}_{\mathtt{trace}}(c)$ are built from the $\mathsf{DEP}_{\mathsf{var}}(x^i, y^j, c)$ relation between two labeled variables. This is the key definition in order to formalize the intuitive *may-dependency* relation between queries and the *adaptivity*. We present this formalization detail in Section 3.2 below.

4. The weight function in $\mathtt{W}_{\mathtt{trace}}(c)$ for each vertex, $w : \mathcal{T} \to \mathbb{N}$ maps from a starting trace $\tau_0 \in \mathcal{T}_0(c)$ to a natural number. For each vertex x^l, it tracks its visiting times (i.e., the evaluation times of the command with the label l) when the program c is evaluated from the initial trace τ_0 into `skip`, $\langle c, \tau_0 \rangle \to^* \langle \mathtt{skip}, \tau_0 {+\!+} \tau' \rangle$. The visiting times is computed by the counter operator $\mathtt{cnt}(\tau', l)$ by counting the occurrence of the label l in τ'. As an instance, in the semantics-based dependency graph of `twoRounds` in Figure 2(b), the weight, w_k of the vertex x^3 is a

function of type $\mathcal{T}_0(\texttt{twoRound(k)}) \to \mathbb{N}$. Given input τ_0, we execute the program under τ_0 as $\langle \texttt{twoRound(k)}, \tau_0 \rangle \to^* \langle \texttt{skip}, \tau_0 {+\!+} \tau' \rangle$. Then $w_k(\tau_0)$ outputs the occurrence time of the label 3 in τ'.

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

3.2 May-Dependency

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

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

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

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

The first case captures the data dependency. For instance, in a simple program $c_1 = [x \leftarrow \texttt{query}(\chi[2])]^1; [y \leftarrow \texttt{query}(\chi[3] + x)]^2$, we think $\texttt{query}(\chi[3]+x)$ (variable y^2) may depend on the query $\texttt{query}(\chi[2]))$ (variable x^1), because the equipped function of the former $\chi[3] + x$ may depend on the data stored in x assigned with the result of $\texttt{query}(\chi[2]))$. From our perspective, $\texttt{query}(\chi[1])$ is different from $\texttt{query}(\chi[2]))$.
The second case captures the control dependency. For instance, in the program $c_2 = [x \leftarrow \texttt{query}(\chi[1])]^1; \texttt{if}\ ([x > 2]^2, [y \leftarrow \texttt{query}(\chi[2])]^3, [\texttt{skip}]^4)$, we think the query $\texttt{query}(\chi[2])$ (or the labeled variable y^3) may depend on the query $\texttt{query}(\chi[1])$ (via the labeled variable x^1).

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

Definition 9 (Events Differ in Value ($\texttt{Diff}$))**.** *Two events $\epsilon_1, \epsilon_2 \in \mathcal{E}$ differ in their value, or query value, denoted as $\texttt{Diff}(\epsilon_1, \epsilon_2)$, if and only if:*

$$\pi_1(\epsilon_1) = \pi_1(\epsilon_2) \land \pi_2(\epsilon_1) = \pi_2(\epsilon_2) \tag{2a}$$
$$\land \big((\pi_3(\epsilon_1) \neq \pi_3(\epsilon_2) \land \pi_4(\epsilon_1) = \pi_4(\epsilon_2) = \bullet) \lor (\pi_4(\epsilon_1) \neq \bullet \land \pi_4(\epsilon_2) \neq \bullet \land \pi_4(\epsilon_1) \neq_q \pi_4(\epsilon_2))\big) \tag{2b}$$

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

$\pi_1(\epsilon_1) = \pi_1(\epsilon_2) \land \pi_2(\epsilon_1) = \pi_2(\epsilon_2)$ at Eq.2(a) requires that ϵ_1 and ϵ_2 have the same variable name and label. This guarantees that ϵ_1 and ϵ_2 are generated from the same labeled command. In Eq.2(b), two kinds of comparisons between the third and fourth element are for the non-query assignment and query request separately. For events generated from the non-query assignments (via checking

$\pi_4(\epsilon_1) =_q \pi_4(\epsilon_2) = \bullet$), we only compare their assigned values through $\pi_3(\epsilon_1) \neq \pi_3(\epsilon_2)$. But for these from query requests (via checking $\pi_4(\epsilon_1) \neq \bullet \land \pi_4(\epsilon_2) \neq \bullet$), we are comparing their query expressions by $\pi_4(\epsilon_1) \neq_q \pi_4(\epsilon_2)$ rather than the assigned value computed from the unknown database server. This matches the intuitive data dependency between queries, where one query is influenced by others as long as the query request is changed.

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

Definition 10 (Event May-Dependency). *An event 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 witness trace $\tau \in \mathcal{T}$, $\mathsf{DEP_e}(\epsilon_1, \epsilon_2, [\epsilon_1]{++}\tau{++}[\epsilon_2], c, D)$ if and only if*

$$\exists \tau_0, \tau_1, \tau' \in \mathcal{T}, \epsilon_1' \in \mathcal{E}^{\mathtt{asn}}, c_1, c_2 \in \mathcal{C} \,.\, \mathtt{Diff}(\epsilon_1, \epsilon_1') \land \tag{3a}$$

$$(\exists \epsilon_2' \in \mathcal{E} \,.\, \left(\begin{array}{ll} & \langle c, \tau_0 \rangle \to^* \langle c_1, \tau_1{++}[\epsilon_1] \rangle \to^* \langle c_2, \tau_1{++}[\epsilon_1]{++}\tau{++}[\epsilon_2] \rangle \\ \land & \langle c_1, \tau_1{++}[\epsilon_1'] \rangle \to^* \langle c_2, \tau_1{++}[\epsilon_1']{++}\tau'{++}[\epsilon_2'] \rangle \\ \land & \mathtt{Diff}(\epsilon_2, \epsilon_2') \land \mathtt{cnt}(\tau, \pi_2(\epsilon_2)) = \mathtt{cnt}(\tau', \pi_2(\epsilon_2')) \end{array}\right) \tag{3b}$$

$$\lor \left(\begin{array}{l} \exists \tau_3, \tau_3' \in \mathcal{T}, \epsilon_b \in \mathcal{E}^{\mathtt{test}} \,. \\ \quad \langle c, \tau_0 \rangle \to^* \langle c_1, \tau_1{++}[\epsilon_1] \rangle \to^* \langle c_2, \tau_1{++}[\epsilon_1]{++}\tau{++}[\epsilon_b]{++}\tau_3 \rangle \\ \quad \land \langle c_1, \tau_1{++}[\epsilon_1'] \rangle \to^* \langle c_2, \tau_1{++}[\epsilon_1']{++}\tau'{++}[(\neg\epsilon_b)]{++}\tau_3' \rangle \\ \quad \land \mathbb{TL}(\tau_3) \cap \mathbb{TL}(\tau_3') = \emptyset \land \mathtt{cnt}(\tau', \pi_2(\epsilon_b)) = \mathtt{cnt}(\tau, \pi_2(\epsilon_b)) \land \epsilon_2 \in \tau_3 \land \epsilon_2 \not\in \tau_3' \end{array}\right)), \tag{3c}$$

where $\mathbb{TL}(\tau) \subseteq \mathcal{L}$ is the set of the labels in all the events from trace τ and $\epsilon_2 \in \tau_3$ or $\epsilon_2 \notin \tau_3$ denotes that ϵ_2 belongs to τ_3 or not.

The first line in Eq. 3(a) requires that ϵ_1 comes from an assignment command and then modifies its assigned value via $\mathtt{Diff}(\epsilon_1, \epsilon_1')$.

Then, the following two parts in Eq 3(b) and (c) capture the intuitive value dependency and control dependency respectively. Both parts execute the program two times w.r.t. the different values in ϵ_1 (as line:1 in Eq 3(b) and line:2 in Eq 3(c)) and ϵ_1' (as line:2 in Eq 3(b) and line:3 in Eq 3(c)), but observe the difference in the newly generated traces in different ways (via 3rd line in Eq 3(b) and 4th line in Eq 3(c)). This idea is similar to the dependency definition from [1].

In Eq 3(b) line:2, if the newly generated trace, $\tau' + +[\epsilon_2']$ still contains ϵ_2 as ϵ_2', we check the difference on their value in line:3. If they only differ in their assigned values, i.e., $\mathtt{Diff}(\epsilon_2, \epsilon_2')$ and they are in the same loop iteration (via $\mathtt{cnt}(\tau, \pi_2(\epsilon_2)) = \mathtt{cnt}(\tau', \pi_2(\epsilon_2'))$), then we say there is a value *may-dependency* relation between ϵ_1 and ϵ_2.

The Eq 3(c) captures the control dependency through observing the disappearance ϵ_2 from newly generated traces, $\tau'{++}[(\neg\epsilon_b)]{++}\tau_3'$ in the second execution (line:3). $\epsilon_2 \in \tau_3 \land \epsilon_2 \not\in \tau_3'$ in Eq 3(c) line:4 specifies this disappearance. $\mathtt{cnt}(\tau', \pi_2(\epsilon_b)) = \mathtt{cnt}(\tau, \pi_2(\epsilon_b))$ is used to make sure the two executions are in the same loop iteration as well. Different from Eq 3(b) line:3, we use a testing event, ϵ_b here because $\mathtt{cnt}(\tau, \pi_2(\epsilon_2)) = \mathtt{cnt}(\tau', \pi_2(\epsilon_2'))$ cannot guarantee the disappearance if there are nested loops. This is correct because the control dependency can only be passed through the guard of if or while command, and this guard must be evaluated into two different values (ϵ_b and $\neg\epsilon_b$) in the two executions.

Then Considering all the assignment events newly generated during a program's executions, as long as there is one pair of events satisfying the *event may-dependency*, we say that the two labeled variables in the two assignment events satisfy the *variable may-dependency* relation below.

Definition 11 (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 and only if.*

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

From the definition, a labeled assigned variables $x_2^{l_2}$ may depend on another labeled assigned variable $x_1^{l_1}$ in a program c under the hidden database D, as long as there exist two assignment events ϵ_1 (for $x_1^{l_1}$) and ϵ_2 for $x_2^{l_2}$ satisfy the *event may-dependency* relation under a witness trace τ.

3.3 Trace-based Adaptivity

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

Definition 12 (Finite Walk (k)). *Given the semantics-based dependency graph* $\mathtt{G}_{\mathtt{trace}}(c) = (\mathtt{V}_{\mathtt{trace}}, \mathtt{E}_{\mathtt{trace}}, \mathtt{W}_{\mathtt{trace}}, \mathtt{Q}_{\mathtt{trace}})$ *of a program c, a* finite walk k *in* $\mathtt{G}_{\mathtt{trace}}(c)$ *is a function k. Given an input initial trace* $\tau_0 \in \mathcal{T}_0(c)$*,* $k(\tau_0)$ *is a sequence of edges* $(e_1 \dots e_{n-1})$ *for which there is a sequence of vertices* $(v_1, \dots, v_n)$ *such that:*

- $e_i = (v_i, v_{i+1}) \in \mathtt{E}_{\mathtt{trace}}$ *for every* $1 \leq i < n$.
- *every* $v_i \in \mathtt{V}_{\mathtt{trace}}$ *and* $(v_i, w_i) \in \mathtt{W}_{\mathtt{trace}}$*,* v_i *appears in* $(v_1, \dots, v_n)$ *at most* $w(\tau_0)$ *times.*

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

$\mathcal{WK}(\mathtt{G}_{\mathtt{trace}}(c))$ is the set of all the finite walks k in $\mathtt{G}_{\mathtt{trace}}(c)$, and $k_{v_1 \to v_2} \in \mathcal{WK}(\mathtt{G}_{\mathtt{trace}}(c))$ denotes the walk from vertex v_1 to v_2.

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

Definition 13 (Query Length of the Finite Walk($\mathtt{len}^{\mathtt{q}}$)). *Given the semantics-based dependency graph* $\mathtt{G}_{\mathtt{trace}}(c)$ *of a program c, and a* finite walk $k \in \mathcal{WK}(\mathtt{G}_{\mathtt{trace}}(c))$*. The query length of k,* $\mathtt{len}^{\mathtt{q}}(k)$ *is a function* $\mathcal{T}_0(c) \to \mathbb{N}$*, such that given an input initial trace* τ_0*,* $\mathtt{len}^{\mathtt{q}}(k)(\tau_0)$ *is the number of vertices which correspond to query variables in the vertex sequence,* $(v_1, \dots, v_n)$ *as follows,*

$$\mathtt{len}^{\mathtt{q}}(k)(\tau_0) = |\big(v \mid v \in (v_1, \dots, v_n) \land \mathtt{Q}(v) = 1\big)|.$$

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

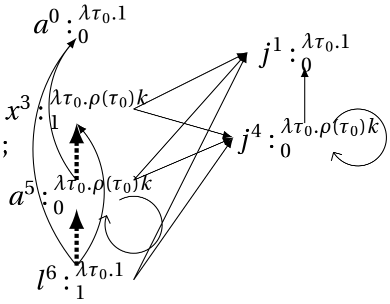
Definition 14 (Adaptivity of a Program). *Given a program c, its adaptivity* $A(c)$ *is function* $A(c) : \mathcal{T} \to \mathbb{N}$ *such that for an initial trace* $\tau_0 \in \mathcal{T}_0(c)$*,*

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

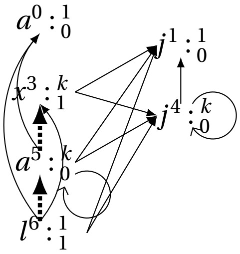
3.4 The Walk Through Example

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

(a)



(b)



(c)

Figure 2: (a) The program $\texttt{towRounds(k)}$, an example with two rounds of adaptivity (b) The corresponding semantics-based dependency graph (c) The estimated dependency graph from AdaptFun.

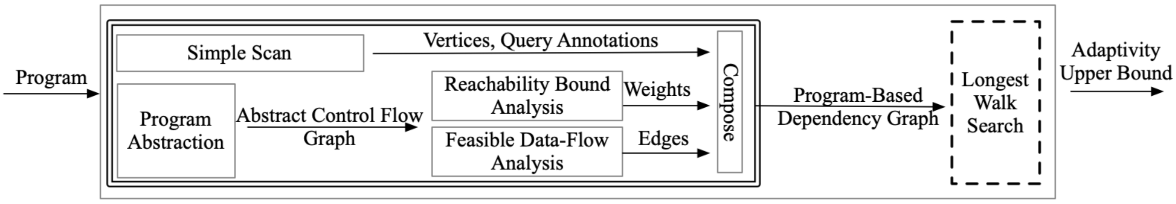


Figure 3: The overview of AdaptFun

4 The Adaptivity Analysis Algorithm - AdaptFun

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

4.1 A guide to AdaptFun

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

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

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

   (b) **Edge and Weight Estimation**
   This step estimates the edge and weight for a quantitative dependency graph. It combines the control, data-flow analysis algorithm and the loop bound inference algorithm. There are three computation steps in this algorithm.
   Abstract Control Flow Graph. In order to perform the dependency analysis and quantitative analysis, this step first generates an *abstract control flow graph* for a program in Section 4.3.1.
   Edges Estimation via Combined Flow Analysis. The step is presented in Section 4.3.2. It performs over the *abstract control flow graph*, which combines both control flow and data flow analysis. It estimates the *dependency relation* between each pair of the labeled variables in a program by considering both the control flow and data flow. Then it uses the estimated dependency relation to approximate the edge between each pair of vertices.
   Weights Estimation via Quantitative Analysis. This step is presented in Section 4.3.3. It performs over the same *abstract control flow graph* and computes the upper bound on the maximal visiting times of each labeled variable for a program. It estimates the reachability bound for every vertex over the *abstract control flow graph*, and this reachability bound is

used to estimate the maximal visiting times of each labeled variable in a program and the weight of the corresponding vertex.

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

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

4.2 Vertex and Query Annotation Estimations

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

$$\mathtt{V}_{\mathtt{est}}(c) \triangleq \left\{ x^l \in \mathcal{LV} \;\middle|\; x^l \in \mathbb{LV}(c) \right\}$$

where $\mathcal{A}_{\mathtt{in}}$ is the set of arithmetic expressions over $\mathbb{N}$ and program's input variables.

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

$$\mathtt{Q}_{\mathtt{est}}(c) = \left\{ (x^l, n) \in \mathcal{LV} \times \{0,1\} \;\middle|\; x^l \in \mathbb{LV}_c, n = 1 \iff x^l \in \mathbb{QV}_c \land n = 0 \iff x^l \not\in \mathbb{QV}_c. \right\}$$

4.3 Edge and Weight Estimation

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

4.3.1 Abstract Transition Graph

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

An *Abstract Transition Graph*, $\mathtt{absG}(c)$ for a program c is composed of a vertex set $\mathtt{absV}(c)$ and an edge set $\mathtt{absE}(c)$, $\mathtt{absG}(c) \triangleq (\mathtt{absV}(c), \mathtt{absE}(c))$.
Every vertex $l \in \mathtt{absV}(c)$ is the label of a labeled command in c, which is unique. We also call the

unique label as program point.
Each edge $(l \xrightarrow{dc} l') \in \texttt{absE}(c)$ is an abstract transition between two program points l, l'. There is an edge from l to l' if and only if the command with label l' can execute right after the execution of the command with label l. Each edge is annotated by a constraint $dc \in \mathcal{DC}^{\top}$, which is generated from the command with label l. This constraint describes the abstract execution of the command with l.

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

$$\texttt{absV}(c) = \mathbb{LV}(c) \cup \{\texttt{ex}\}$$

Abstract Control Flow Graph Edge Construction Each edge $(l \xrightarrow{dc} l') \in \texttt{absE}(c)$ is an abstract transition between two program points l, l'. There is an edge from l to l' if and only if the command with label l' can execute right after the execution of the command with label l. Each edge is annotated by a constraint $dc \in \mathcal{DC}^{\top}$ generated from the command with label l. This constraint describes the abstract execution of the command with l. This step shows how to generate the abstract transition graph $\texttt{absG}(c)$ of a program c through constructing its vertices and edges.
The vertices can be easily collected and the key point of the abstract transition graph for a program is constructing the edge set, $\texttt{absE}(c)$ for a program c. It relies on the control flow analysis and the program abstraction of each command. To make it easy to understand, it is an enriched control flow graph with an annotation on each edge. The edge set is constructed by a program abstraction method in three steps.
In the first step, **Constraint Computation** generates a constraint over the expression for every program's labeled command, which is used as the annotation of an edge.
In the second step, **Initial and Final State Computation** generates two sets for each command. The initial state is a set that contains the program point where this command starts executing, and the final state is a set that contains the constraint of this command and the continuation program points after the execution of this command.
In the third step, **Abstract Event Computation** generates a set of edges for the program. Each edge is a pair of initial and finial state.

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

Notations / Formal Definitions:

- Operator: $\texttt{absexpr} : \mathcal{A} \cup \mathcal{B} \to DC(\mathcal{VAR} \cup \mathcal{SMBCST}) \cup \mathcal{B} \cup \{\top\}$
- Constrains, $\mathcal{DC}^{\top}$ is composed of the *Difference Constraints* $DC(\mathcal{VAR} \cup \mathcal{SMBCST})$, the *Boolean Expressions* $\mathcal{B}$ and $\top$.
  - The difference constraints $DC(\mathcal{VAR} \cup \mathcal{SMBCST})$ is the set of all the inequality of form $x' \leq y + v$ or $x' \leq v$ where $x \in \mathcal{VAR}$, $y \in \mathcal{VAR}$ and $v \in \mathcal{SMBCST}$. The *Symbolic Constant* set $\mathcal{SMBCST} = \mathbb{N} \cup \mathcal{VAR}_{\texttt{in}} \cup \infty \cup Q_m$ is the set of natural numbers with ∞, the input variables, and a symbol Q_m representing the abstract value of a query request. An inequality

$x' \leq y + v$ describes that the value of x in the current state is at most the value of y in the previous state plus the symbolic constant v. An inequality $x' \leq v$ describes that the value of x in the current state is at most the value v. When a difference constrain shows up as an edge annotation, $l \xrightarrow{x' \leq y+v} l'$, it denotes that the value of variable x after executing the command at l is at most the value of variable y plus v before the execution, and $l \xrightarrow{x' \leq v} l'$ respectively denotes value of variable x after executing the command at l is at most the value of the symbolic constant v before the execution. For every expression in each of the label command, it is computed in three steps via program abstraction method adopted from the Section 6 in [6].

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

Computation Steps:

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

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

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

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

The symbolic expression set is a subset of arithmetic expressions over $\mathbb{N}$ with input variables, i.e., $\mathcal{A}_S \subseteq \mathcal{A}_{\texttt{in}}$.

Abstract Initial and Final State Computation This step computes two sets for each command. The initial state is a set that contains the program points before executing this command, which is computed by the standard initial state generation method from control flow analysis. The final state is a set that contains the constraint of this command and the program points after the execution of this command. This set is enriched from the standard control flow analysis.

Notations / Formal Definitions:

- The abstract initial state: $\texttt{absinit}(c) \in \mathcal{L}$.
- The abstract Final State: $\texttt{absfinal}(c) \in \mathcal{P}(\mathcal{L} \times \mathcal{DC}^{\top})$

Computation Steps:

- The *abstract initial state*, $\texttt{absinit}(c) \in \mathcal{P}(\mathcal{L})$ for a command c is the set of the initial program points. Each point in this set is a unique program label corresponds to the command before executing this command.
  Given a program c, its abstract initial state, $\texttt{absinit}(c)$ is computed as follows,

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

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

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

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

Notations / Formal Definitions:

- *Abstract Event*: $\overset{\alpha}{e} \in \mathcal{L} \times \mathcal{DC}^{\top} \times \mathcal{L}$
- *Abstract Event Computation*: $\texttt{abstrace} \in \mathcal{C} \to \mathcal{P}(\mathcal{L} \times \mathcal{DC}^{\top} \times \mathcal{L})$

Its type is defined as follows,

Definition 17 (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 $\mathcal{DC}^{\top}$.*

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

Computation Steps:
The set of the abstract events $\texttt{abstrace}(c)$ for a program c is computed as follows in Definition 18.

Definition 18 (Abstract Event Computation)**.** $\texttt{abstrace} \in \mathcal{C} \to \mathcal{P}(\mathcal{L} \times \mathcal{DC}^{\top} \times \mathcal{L})$

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

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

Notice $\texttt{abstrace}'([x := e]^{l})$, $\texttt{abstrace}'([x := \texttt{query}(\psi)]^{l})$ and $\texttt{abstrace}'([\texttt{skip}]^{l})$ are all empty set.

Theorem Guarantee: 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 $(l, \_, \_)$. Our soundness is presented below with the proof in Appendix D.1

Lemma 4.1 (Soundness of the Abstract Events)**.** *For every program c and an execution trace $\tau \in \mathcal{T}$ that is generated w.r.t. an initial trace $\tau_0 \in \mathcal{T}_0(c)$, there is an abstract event $\overset{\alpha}{\epsilon} = (l, \_, \_) \in \texttt{abstrace}(c)$ for every event $\epsilon \in \tau$ having the label l, i.e., $\epsilon = (\_, l, \_, \_)$.*

$$
\begin{array}{l}
\forall c \in \mathcal{C}, \tau_0 \in \mathcal{T}_0(c), \tau \in \mathcal{T}, \epsilon = (\_, l, \_, \_) \in \mathcal{E} \,.\, \langle c, \tau_0 \rangle \to^{*} \langle \texttt{skip}, \tau_0 {+\!\!+} \tau \rangle \land \epsilon \in \tau \\
\quad \Longrightarrow \exists \overset{\alpha}{\epsilon} = (l, \_, \_) \in (\mathcal{L} \times \mathcal{DC}^{\top} \times \mathcal{L}) \,.\, \overset{\alpha}{\epsilon} \in \texttt{abstrace}(c)
\end{array}
$$

For every program point l corresponding to an assignment command in a program c, there is a unique abstract event in the program's abstract events set $\overset{\alpha}{\epsilon} \in \texttt{abstrace}(c)$ of form $(l, \_, \_)$.

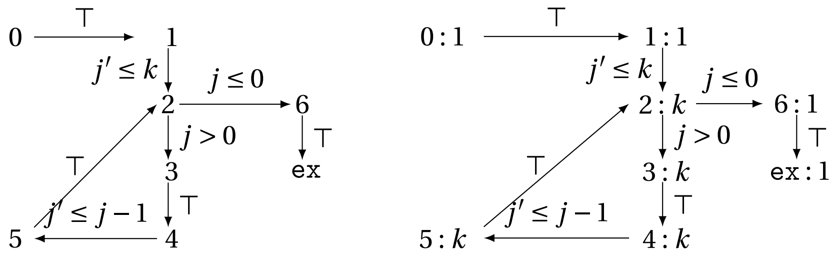
Lemma 4.2 (Uniqueness of the Abstract Events Computation)**.** *For every program c and an execution trace $\tau \in \mathcal{T}$ that is generated w.r.t. an initial trace $\tau_0 \in \mathcal{T}_0(c)$, there is a unique abstract event $\overset{\alpha}{\epsilon} = (l, \_, \_) \in \texttt{abstrace}(c)$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

$$
\begin{array}{l}
\forall c \in \mathcal{C}, \tau_0 \in \mathcal{T}_0(c), \tau \in \mathcal{T}, \epsilon = (\_, l, \_) \in \mathcal{E}^{\text{asn}} \,.\, \langle c, \tau_0 \rangle \to^{*} \langle \texttt{skip}, \tau_0 {+\!\!+} \tau \rangle \land \epsilon \in \tau \\
\quad \Longrightarrow \exists ! \overset{\alpha}{\epsilon} = (l, \_, \_) \in (\mathcal{L} \times \mathcal{DC}^{\top} \times \mathcal{L}) \,.\, \overset{\alpha}{\epsilon} \in \texttt{abstrace}(c)
\end{array}
$$

This lemma is proved in Appendix D.2.

$[a \leftarrow 0]^0; [j \leftarrow k]^1;$
$\texttt{while } [j > 0]^2 \texttt{ do}$
$\big([x \leftarrow \texttt{query}(\chi[j])]^3;$
$[j \leftarrow j-1]^4;$
$[a \leftarrow x + a]^5\big);$
$[l \leftarrow \texttt{query}(\chi[k] * a)]^6$

(a)



(b) (c)

Figure 4: (a) The same `towRounds(k)` program as Figure 2 (b) The abstract control flow graph for `towRounds(k)` (c) The abstract control flow graph with the reachability bound for `towRounds(k)`.

Edge Construction The edge for c's abstract transition graph is constructed simply by computing the program's abstract events set, $\texttt{abstrace}(c)$ as follows,

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

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

Definition 19 (Abstract Transition Graph)**.** *Given a program c, its* abstract transition graph $\texttt{absG}(c) = (\texttt{absV}(c), \texttt{absE}(c))$ *is computed as follows,*
$\texttt{absE}(c) = \{(l_1, dc, l_2) | (l_1, dc, l_2) \in \texttt{abstrace}(c)\}$,
$\texttt{absV}(c) = \mathbb{LV}(c) \cup \{\texttt{ex}\}$

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

4.3.2 Edge Estimation

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

1. The **Reaching Definition** analysis computes a set of labeled variables, $\texttt{RD}(l, c)$ for every label l in c over its abstract control flow graph, $\texttt{absG}(c)$. The computation performs the standard reaching definition analysis and working-list algorithm over the abstract control flow graph, $\texttt{absG}(c)$. $\texttt{RD}(l, c)$ contains all the labeled variables which are reachable at program point l.

2. The **Feasible Data-Flow** computation combines the $\mathtt{RD}(l, c)$, $\mathtt{absG}(c)$ and data flow analysis. It computes the *feasible data-flow* relation, $\mathtt{flowsTo}(x^i, y^j, c)$ for each pair of the c's labeled variables, $x^i, y^j \in \mathbb{LV}(c)$ in Definition 20. $\mathtt{flowsTo}(x^i, y^j, c)$ is a sound approximation of the *variable may-dependency* relation, $\mathsf{DEP}_{\mathsf{var}}(x^i, y^j, c)$ for every $x^i, y^j \in \mathbb{LV}(c)$. The formal proof is in the Appendix. We also discuss that the combined analysis gives more precise approximation on the *data may-dependency* than single analysis in Appendix.

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

The details are as follows.

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

1. The block, is either the command of the form of assignment, skip, or a test of the form of $[b]^l$, denoted by $\mathsf{blocks}(c)$ the set of all the blocks in program c, where $\mathsf{blocks} : \mathcal{C} \to \mathcal{P}(\mathcal{C} \cup [b]^l)$.

   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.

2. 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.

3. The operator gen: $blocks \to \mathcal{P}(\mathcal{VAR} \times \mathcal{L} \cup \{?\})$ generates the set of labelled variables generated by the block.

4. The operator $in(l)$, $out(l)$: $\mathcal{L} \to \mathcal{LV} \cup \{?\}$ for every block in program c is defined as follows,

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

   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 $\mathtt{RD}(l, c)$ to represent denote the stabilized result of $in(l)$ at label l in program c.

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

   (a) **initialize** in[l]=out[l]=∅; in[0] = ∅

   (b) **initialize** a work queue W, contains all the blocks in c

   (c) while $|W|$!= 0
   pop l in W
   old = out[l]

$in(l) = out(l')$ where $(l', \_, l) \in \texttt{absE}(c)$
$out(l) = \mathsf{gen}(b^l) \cup (in(l) - kill(b^l))$, where $b^l \in \mathsf{blk}(c)$
if (old != out(l)) $W = W \cup \{l' | (l, l') \in (l', \_, l) \in \texttt{absE}(c)\}$
end while

Feasible Data-Flow Computation This part presents the computation of the *feasible data-flow* relation between each pair of labeled variables in a program c, formally in Definition 20,

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

$$
\begin{array}{ll}
\texttt{flowsTo}(x^i, y^j, [y \leftarrow e]^l) & \triangleq (x^i, y^j) \in \{(x^i, y^l) | x \in \mathsf{FV}(e) \land x^i \in \texttt{RD}(l, [y \leftarrow e]^l)\} \\
\texttt{flowsTo}(x^i, y^j, [y \leftarrow \texttt{query}(\psi)]^l) & \triangleq (x^i, y^j) \in \{(x^i, y^l) | x \in \mathsf{FV}(\psi) \land x^i \in \texttt{RD}(l, [y \leftarrow \texttt{query}(\psi)]^l)\} \\
\texttt{flowsTo}(x^i, y^j, [\texttt{skip}]^l) & = \emptyset \\
\texttt{flowsTo}(x^i, y^j, \texttt{if}\ ([b]^l, c_1, c_2)) & \triangleq \texttt{flowsTo}(x^i, y^j, c_1) \lor \texttt{flowsTo}(x^i, y^j, c_2) \\
 & \lor (x^i, y^j) \in \left\{(x^i, y^j) | x \in \mathsf{FV}(b) \land x^i \in \texttt{RD}(l, \texttt{if}\ ([b]^l, c_1, c_2)) \land y^j \in \mathbb{LV}(c_1)\right\} \\
 & \lor (x^i, y^j) \in \left\{(x^i, y^j) | x \in \mathsf{FV}(b) \land x^i \in \texttt{RD}(l, \texttt{if}\ ([b]^l, c_1, c_2)) \land y^j \in \mathbb{LV}(c_2)\right\} \\
\texttt{flowsTo}(x^i, y^j, \texttt{while}\ [b]^l \ \mathsf{do}\ c_w) & \triangleq \texttt{flowsTo}(x^i, y^j, c_w) \lor \\
 & (x^i, y^j) \in \left\{(x^i, y^j) | x \in \mathsf{FV}(b) \land x^i \in \texttt{RD}(l, \texttt{while}\ [b]^l \ \mathsf{do}\ c_w) \land y^j \in \mathbb{LV}(c_w)\right\} \\
\texttt{flowsTo}(x^i, y^j, c_1; c_2) & \triangleq \texttt{flowsTo}(x^i, y^j, c_1) \lor \texttt{flowsTo}(x^i, y^j, c_2)
\end{array}
$$

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

Improvement Analysis. Combining the result of *reaching definition*, $\texttt{RD}(l, c)$ with the *abstract control flow graph*, $\texttt{absG}(c)$ with control flow analysis into the feasible data-flow generation improves the data-dependency relation approximation accuracy. For example, a program $[x = 0]^1; [x = 2]^2; [y = x + 1]^3$. 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.

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

$$
\begin{array}{rl}
\texttt{E}_{\texttt{est}}(c) \triangleq & \{(y^j, x^i) \mid y^j, x^i \in \texttt{V}_{\texttt{est}}(c) \land \exists n, z_1^{r_1}, \ldots, z_n^{r_n} \in \mathbb{LV}(c)\ . \\
 & \qquad n \geq 0 \land \texttt{flowsTo}(x^i, z_1^{r_1}, c) \land \cdots \land \texttt{flowsTo}(z_n^{r_n}, y^j, c)\}
\end{array}
$$

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

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

4.3.3 Weight Estimation

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

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

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

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

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

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

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

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

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

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

The computation steps are as follows,

1. **Variable Modifications** For each variable x in a program c, this step computes three edge sets, $\mathtt{inc}(c,x)$, $\mathtt{dec}(c,x)$, and $\mathtt{re}(c,x)$ for x. Every edge in a set corresponds to a transition in which x is increased, decreased or reset respectively.
$\mathtt{inc} : \mathcal{C} \to \mathcal{VAR} \to \mathcal{P}(\overset{\alpha}{\epsilon})$ is the set of the edges where the variable increase,
$\mathtt{inc}(c,x) = \{\overset{\alpha}{\epsilon} \mid \overset{\alpha}{\epsilon} = (l, l', x' \leq x + v) \land \overset{\alpha}{\epsilon} \in \mathtt{abstrace}(c)\}$
$\mathtt{dec} : \mathcal{VAR} \to \mathcal{P}(\overset{\alpha}{\epsilon})$ is the set of abstract events where the variable decrease,
$\mathtt{dec}(c,x) = \{\overset{\alpha}{\epsilon} \mid \overset{\alpha}{\epsilon} = (l, l', x' \leq x - v) \land \overset{\alpha}{\epsilon} \in \mathtt{abstrace}(c)\}$
$\mathtt{re} : \mathcal{C} \to \mathcal{VAR} \to \mathcal{P}(\overset{\alpha}{\epsilon})$ is the set of the abstract events where the variable is reset,
$\mathtt{re}(c,x) = \{\overset{\alpha}{\epsilon} \mid \overset{\alpha}{\epsilon} = (l, l', x' \leq y - v) \land x \neq y \land \overset{\alpha}{\epsilon} \in \mathtt{abstrace}(c)\}$
$\mathtt{rechain} : \mathcal{C} \to \mathcal{VAR} \to \mathcal{P}(\mathcal{P}(\overset{\alpha}{\epsilon}))$ is the set of the chain of abstract events where the variable is reset through the chain.
In addition to collect the edge set that x is reset on every edge in this set, i.e., compute the $\mathtt{re}(c,x)$, we also compute a set, $\mathtt{rechain}(c,x)$ contains sequences of edges for x based on the Definition.20 in [6]. In each sequence, $(e_0, \cdots, e_m) \in \mathtt{rechain}(c,x)$ a variable x_i is reset by another variable x_{i+1} on edge e_i and x_{i+1} is reset on edge e_{i+1} recursively for every $i = 0, \cdots, m-1$. x is reset on the first edge e_0 of every sequence in $\mathtt{rechain}(c,x)$. Rephrase: Each edge e_i in a sequence $(e_0, \cdots, e_m) \in \mathtt{rechain}(c,x)$ resets a variable x_i by another variable x_{i+1} such that x_{i+1} is reset on edge e_{i+1} recursively. The first edge e_0 of each sequence resets the variable x.
In the following steps, c is omitted in $\mathtt{inc}(x)$, $\mathtt{dec}(x)$ and $\mathtt{re}(x)$ for concise when the reference of a program c is clear in the context.

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

   Definition 21 (Local Bound Generatation)**.** *For every edge $\overset{\alpha}{\epsilon}$ in the transition graph* $\mathtt{absG}(c)$ *of a program c, its* local bound*,* $\mathtt{locb}(\overset{\alpha}{\epsilon}, c)$ *is the variable that decreases on this edge, computed as follows,*

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

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

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

3. **Local Bound Estimation** This step estimates the upper bound, $\mathtt{Vinvar}(x,c) \in \mathcal{A}_{\mathtt{in}}$ on the maximum value for each local bound $x \in \mathcal{VAR} \cup \mathcal{SMBCST}$.
For a program c, the *local bound* of an , $\mathtt{Vinvar}(\mathtt{locb}(\overset{\alpha}{\epsilon}, c)) \in \mathcal{A}_{\mathtt{in}}$ is the bound on the maximum value of the local bound assigned to the edge $\overset{\alpha}{\epsilon} \in \mathtt{absE}(c)$, formally in Definition 22 and 23.
In order to estimate the maximum value of $\mathtt{locb}(\overset{\alpha}{\epsilon}, c)$ assigned to edge $\overset{\alpha}{\epsilon} \in \mathtt{absE}(c)$, the bound

on the iteration times of each corresponding edge, $\mathtt{TB}(\overset{\alpha}{e}, c)$ is computed interactively in a path-insensitive manner.
$\mathtt{Vinvar} : (\mathcal{VAR} \cup \mathcal{SMBCST} \times \mathcal{C}) \to \mathcal{A}_{\mathtt{in}}$
$\mathtt{TB} : (\overset{\alpha}{e} \times \mathcal{C}) \to \mathcal{A}_{\mathtt{in}}$

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

$$\begin{array}{lll}
\mathtt{Vinvar}(x, c) & \triangleq x & x \in \mathcal{SMBCST} \\
\mathtt{Vinvar}(x, c) & \triangleq \mathtt{increase}(x, c) + \max(\{\mathtt{Vinvar}(y, c) + v \mid (l, x' \leq y + v, l') \in \mathtt{re}(x)\}) & x \notin \mathcal{SMBCST}
\end{array}$$

$$\mathtt{increase}(x, c) \triangleq \sum_{\overset{\alpha}{e} \in \mathtt{inc}(x)} \{\mathtt{TB}(\overset{\alpha}{e}, c) \times v \mid \overset{\alpha}{e} = (l, x' \leq x + v, l')\}$$

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

$$\begin{array}{lll}
\mathtt{TB}(\overset{\alpha}{e}, c) & \triangleq \mathtt{Vinvar}(\mathtt{locb}(\overset{\alpha}{e}, c), c) & \textit{if} \quad \mathtt{locb}(\overset{\alpha}{e}, c) \in \mathcal{SMBCST} \\
\mathtt{TB}(\overset{\alpha}{e}, c) & \triangleq \mathtt{increase}(x, c) + \displaystyle\sum_{\overset{\alpha'}{e} \in \mathtt{re}(x,c) \wedge \overset{\alpha'}{e} = (l, x \leq y + v, l')} \left(\mathtt{TB}(\overset{\alpha'}{e}, c) \times \left(\mathtt{Vinvar}(y, c) + v\right)\right) & \\
& \textit{if} \quad \mathtt{locb}(\overset{\alpha}{e}, c) = x \wedge x \notin \mathcal{SMBCST} & ,
\end{array}$$

Then we construct the set of reachability bound w for every program point l, as $\mathtt{absW}(c)$. For each pair $(l, w) \in \mathtt{absW}(c)$, $w = \sum\left\{\mathtt{TB}(\overset{\alpha}{e}, c) \middle| \overset{\alpha}{e} = (l, \_, \_)\right\}$.

Theorem Guarantee For a program c and an edge $\overset{\alpha}{e} \in \mathtt{absE}(c)$, $\mathtt{TB}(\overset{\alpha}{e})$ is a sound upper bound on the execution times of this transition by paper [6]. The soundness theorem is attached in Theorem 4.1 in Appendix D.

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

$$\forall c \in \mathcal{C}, l \in \mathbb{LV}(c), \tau_0 \in \mathcal{T}_0(c), \tau \in \mathcal{T}, v \in \mathbb{N} \,.\, \langle c, \tau_0 \rangle \to^* \langle \mathtt{skip}, \tau_0 {+\!+} \tau \rangle \wedge \langle \mathtt{TB}(\overset{\alpha}{e}, c), \tau_0 \rangle \Downarrow_a v \wedge \mathtt{cnt}(\tau, l) \leq v$$

Example We perform the symbolic reachability bound analysis on the abstract control flow graph in Figure 4(b) and compute the result in Figure 4(c). We would like to generate the closure of every edge, which is an equality relation between variables. Solving this closure gives us the reachability bound for this edge. With all the bound for all the edges in the abstract control flow graph, we can calculate the weight for every vertex in this graph. For example, we show the closure generated for the edge $4 \xrightarrow{j' \leq j-1} 5$, $\mathtt{TB}(4 \xrightarrow{j' \leq j-1} 5) = \mathtt{Vinvar}(j)$. The invariant for variable j, $\mathtt{Vinvar}(j)$ used here is $\mathtt{Vinvar}(j) = k * \mathtt{TB}(1 \xrightarrow{i' \leq k} 2)$, 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 $j' \leq k$ from edge $1 \xrightarrow{j' \leq k} 2$ and $j' \leq j - 1$ from $4 \xrightarrow{j' \leq j-1} 5$, 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 $4 \xrightarrow{j' \leq j-1} 5$, 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,$2 \xrightarrow{j \geq 0} 3$ and $2 \xrightarrow{j \leq 0} 6$. For the weight for vertex 2, we choose the max between the bound k from $2 \xrightarrow{j \geq 0} 3$ and 1 from $2 \xrightarrow{j \leq 0} 6$. 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 Because the vertices in the two graph share the same unique label, the line number. We use the reachability bound on each program label from $\mathtt{absW}(c)$ to estimate the maximal visiting times of each labeled variable. We show that the reachability bound on one vertex of $\mathtt{absW}(c)$ is also the upper bound for the corresponding vertex in the static analysis dependency graph, both vertices share the same unique line number.

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

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

We prove that this symbolic expression for $x^l \in \mathtt{V}_{\mathtt{est}}(c)$ is a sound upper bound of the weight for the same vertex x^l in Program's semantics-based dependency graph by Lemma B.3 in Appendix B, and Theorem D.2 in Appendix D.3.

Example Going back to the quantitative dependency graph for two-round example in Figure 5(c), which we aim to estimate. Every vertex from $\mathtt{V}_{\mathtt{est}}(c)$ 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 estimated dependency graph in Figrue 4(c) as k as well. The same way for the rest weights' computation.

4.4 Graph Construction

With the four components $\mathtt{V}_{\mathtt{est}}(c), \mathtt{E}_{\mathtt{est}}(c), \mathtt{W}_{\mathtt{est}}(c)$, and $\mathtt{Q}_{\mathtt{est}}(c)$ computed in each steps above, this step simply combine the four components into the quantitative dependency graph for program c as follows,

$$\mathtt{G}_{\mathtt{est}}(c) = (\mathtt{V}_{\mathtt{est}}(c), \mathtt{E}_{\mathtt{est}}(c), \mathtt{W}_{\mathtt{est}}(c), \mathtt{Q}_{\mathtt{est}}(c)).$$

We prove that this graph is a sound approximation of the program's semantics-based dependency graph by soundness of each component formally in Appendix.

This estimated graph estimated graph has a similar topology structure as the Semantics-based Dependency Graph. It has the same vertices but approximated edges and weights. This graph is a sound approximation of the quantitative dependency graph for a program c.

It is formally defined in Definition 24 as follows.

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

$$\mathtt{G}_{\mathtt{est}}(c) = (\mathtt{V}_{\mathtt{est}}(c), \mathtt{E}_{\mathtt{est}}(c), \mathtt{W}_{\mathtt{est}}(c), \mathtt{Q}_{\mathtt{est}}(c))$$

$$
\begin{array}{rrcl}
\textit{Vertices} & \mathtt{V_{est}} & := & \left\{x^l \in \mathcal{LV} \,\middle|\, x^l \in \mathbb{LV}_c\right\} \\
\textit{Directed Edges} & \mathtt{E_{est}} & := & \left\{(x_1^i, x_2^j) \in \mathcal{LV} \times \mathcal{LV} \,\middle|\, \begin{array}{l} x_1^i, x_2^j \in \mathtt{V} \land \exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c \,.\, n \geq 0 \land \\ \mathtt{flowsTo}(x^i, z_1^{r_1}, c) \land \cdots \land \mathtt{flowsTo}(z_n^{r_n}, y^j, c) \end{array} \right\} \\
\textit{Weights} & \mathtt{W_{est}} & := & \left\{(x^l, w) \in \mathcal{LV} \times \mathcal{A}_{in} \mid x^l \in \mathbb{LV}_c \land (l, w) \in \mathtt{absW}(c)\right\} \\
\textit{Query Annotation} & \mathtt{Q_{est}} & := & \left\{(x^l, n) \in \mathcal{LV} \times \{0,1\} \,\middle|\, x^l \in \mathbb{LV}_c, n = 1 \iff x^l \in \mathbb{QV}_c \land n = 0 \iff x^l \in \mathbb{QV}_c .\right\}
\end{array}
$$

The construction of the static 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 certain 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.

4.5 Adaptivity Upper Bound Computation

This phase computes the adaptivity upper bound for a program c.
Based on c's estimated dependency graph, $\mathtt{G_{est}}(c)$ approximated above, its adaptivity upper bound is estimated as the length of the longest finite walk over $\mathcal{WK}(\mathtt{G_{est}}(c))$ formally in Definition 27, and computed by Algorithm 1. $\mathcal{WK}(\mathtt{G_{est}}(c))$ represents the set of all finite walks on $\mathtt{G_{est}}(c)$. Different from the finite walk on $\mathtt{G_{trace}}(c)$, the $\kappa \in \mathcal{WK}(\mathtt{G_{est}}(c))$ doesn't rely on the initial trace. The occurrence time of every v_i in κ's vertices sequence is bound by an arithmetic expression w_i where $(v_i, w_i) \in \mathtt{W_{est}}(c)$ is v_i's estimated weight. Then its query length $\mathtt{len^q}(\kappa)$ and the estimated adaptivity $\mathtt{A_{est}}(c)$ are both arithmetic expression as well. They are formally defined as follows.

Definition 25 (Finite Walk on estimated dependency graph (κ))**.** .
Given a program c's estimated dependency graph $\mathtt{G_{est}}(c) = (\mathtt{V_{est}}(c), \mathtt{E_{est}}(c), \mathtt{W_{est}}(c), \mathtt{Q_{est}}(c))$, a finite walk *$k$ in $\mathtt{G_{trace}}(c)$ 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:*

- *$e_i = (v_i, v_{i+1}) \in \mathtt{E_{est}}(c)$ for every $1 \leq i < n$.*
- *every vertex $v_i \in \mathtt{V_{est}}(c)$, and $(v_i, w_i) \in \mathtt{W_{est}}(c)$, 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 $\mathcal{WK}(\mathtt{G_{est}}(c))$ represents the walks over the estimated dependency graph for c. Different from the walks on a program c's semantics based graph, $k \in \mathcal{WK}(\mathtt{G_{trace}}(c))$, $k \in \mathcal{WK}(\mathtt{G_{est}}(c))$ doesn't rely on initial trace. The occurrence times of every v_i in k's vertex sequence is bound by an arithmetic expression w_i where $(v_i, w_i) \in \mathtt{V_{est}}(c)$, is v_i's estimated weight. The length of a finite walk $k \in \mathcal{WK}(\mathtt{G_{est}}(c))$ is an arithmetic expression as well, i.e., $\mathtt{len}(k) \in \mathcal{A}_{in}$

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

Definition 26 (Query Length of the Finite Walk on estimated dependency graph ($\mathtt{len^q}$))**.** *Given a program c's semantics-based dependency graph $\mathtt{G_{est}}(c) = (\mathtt{V_{est}}(c), \mathtt{E_{est}}(c), \mathtt{W_{est}}(c), \mathtt{Q_{est}}(c))$, and a* finite walk *$k \in \mathcal{WK}(\mathtt{G_{est}}(c))$, The query length of k, $\mathtt{len^q}(k) \in \mathcal{A}_{in}$ is the number of vertices which correspond to query variables in the vertices sequence of this walk k $(v_1, \dots, v_n)$ as follows,*

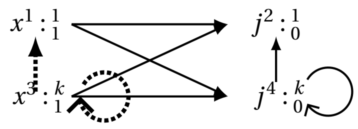
$$
\mathtt{len^q}(k) = \left|\left(v \mid v \in (v_1, \dots, v_n) \land v \in \mathtt{Q_{est}}(c)\right)\right|.
$$

Definition 27 (estimated Adaptivity)**.** *Given a program c and its estimated dependency graph $\mathtt{G_{est}}(c)$ the estimated adaptivity for c is*

$$
\mathtt{A_{est}}(c) \triangleq \max\left\{\mathtt{len^q}(k) \ \mid \ k \in \mathcal{WK}(\mathtt{G_{est}}(c))\right\}.
$$

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

(a)



(b)

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

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

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

$$\forall \tau_0 \in \mathcal{T}_0(c), v \in \mathbb{N}^\infty . \langle \mathtt{A_{est}}(c), \tau_0 \rangle \Downarrow_e v \implies A(c)(\tau_0) \leq c$$

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

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

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

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

Non-Termination Challenge: One naive walk finding method is to simply traverse on this graph and decrease the weight of every node by one after every visiting. However, this simple traversing strategy leads to non-termination dilemma for most programs which we are interested in. Because the weight of each vertex in a program's estimated dependency graph, which is an arithmetic expression containing input variables. In this sense, the simple traversing could never terminate when domain of the input variables isn't finite. However, it is very common that the domain of program's input variables is infinite such as natural number $\mathbb{N}$, real number $\mathbb{R}$, or etc. As the simple while loop example program in Figure 5 with k adaptivity rounds, the input variable k has domain $\mathbb{N}$. If we traverse on the estimated dependency graph, and decrease the weight of x^3 (the weight k is symbolic) by one after every visit, we will never terminate because we only know $k \in \mathbb{N}$.

To solve this non-termination challenge, we switch to another walk finding approach: finding the longest path in the estimated dependency graph via depth first search and then use this path as the estimated longest walk. Through a simple depth first search algorithm, we find the longest weighted path as the dotted arrow in Figure 5(c), $x^3 : {}^k_1 \to x^1 : {}^1_1$. Then, by summing up the weights on this path where the vertices have query annotations 1, depth first search algorithm gives the adaptivity bound k. This is a tight bound for this simple k adaptivity rounds example program.

Approximation Challenge: However, this naive approximation via depth first searching over-approximates the adaptivity rounds largely in many cases. It computes ∞ adaptivity upper bound for our `twoRounds` example program in Figure 2, which has only 2 adaptivity rounds. More specifically, the depth first searching finds the longest weighted path, $x^3 : {}^k_1 \to a^5 : {}^k_0 \to l^6 : {}^1_0$. Then, it computes the weighted length, $1 + k$. If we use this path to approximate the longest finite walk, and weight of each vertex as its visiting time, then we have a walk, $x^3 \to \cdots \to x^3 \to a^5 \to \cdots \to a^5 \to l^6$. However, this isn't a qualified walk by our Definition 12. Because l^6 has weight 1, it can only be visited as most once. In this sense, x^3 is only able to be visited at most once as well, because the only way to re-visit x^3 is through $l^6 \to a^5 \to x^3$. Contradictory, x^3 is visited k times in this approximated walk. As a result, the weighted length of this path is $1 + k$, which over approximates this two rounds example program's adaptivity rounds, which is supposed to be 2.

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

Algorithm 1 Adaptivity Computation Algorithm ($\mathsf{AdaptSearch}(\mathtt{c}, \mathtt{G}_{\mathtt{est}}(\mathtt{c}))$)

Require: The program c, Its estimated dependency graph: $\mathtt{G}_{\mathtt{est}}(c) = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{Q})$

init
  q: empty queue.
  `adapt` : the adaptivity of this graph initialize with 0.
Find all Strong Connected Components (SCC) in G: $\mathtt{SCC_1}, \cdots, \mathtt{SCC_n}, 0 \leq n \leq |\mathtt{V}|$,
for every SCC: $\mathtt{SCC_i}$, compute its Adaptivity $\mathtt{SCC_i}$:
  $\mathtt{adapt}_{\mathtt{scc}}[\mathtt{SCC_i}] = \mathsf{AdaptSearch}_{\mathsf{scc}}(\mathtt{c}, \mathtt{SCC_i})$;
for every $\mathtt{SCC_i}$:
  $q.append(\mathtt{SCC_i})$;
  $\mathtt{adapt}_{\mathtt{tmp}} = 0$;
  while q isn't empty:
    $\mathtt{s} = q.pop()$; #{take the top SCC from head of queue}
    $\mathtt{adapt}_{\mathtt{tmp_0}} = \mathtt{adapt}_{\mathtt{tmp}}$; #{record the adaptivity of last level}
    $\mathtt{SCC}_{\mathtt{max}}$; #{record the SCC with longest walk in this level}
    for every different SCC, $\mathtt{s}'$ connected by $\mathtt{s}$ by a directed edge from $\mathtt{s}$:
      if $(\mathtt{adapt}_{\mathtt{tmp}} < \mathtt{adapt}_{\mathtt{tmp_0}} + \mathtt{adapt}_{\mathtt{scc}}[\mathtt{s}'])$:
        $\mathtt{adapt}_{\mathtt{tmp}} = \mathtt{adapt}_{\mathtt{tmp_0}} + \mathtt{adapt}_{\mathtt{scc}}[\mathtt{s}']$;
        $\mathtt{SCC}_{\mathtt{max}} = \mathtt{s}'$; #{update the SCC with the longest walk in this level}
      $q.append(\mathtt{SCC}_{\mathtt{max}})$;
  $\mathtt{adapt} = \max(\mathtt{adapt}, \mathtt{adapt}_{\mathtt{tmp}})$;
return `adapt`.

The Adaptivity Computation Algorithm ($\mathsf{AdaptSearch}(c, \mathtt{G_{est}}(c))$) At Line:3, this algorithm first finds all the SCCs of $\mathtt{G_{est}}(c)$, $\mathtt{SCC_1}, \cdots, \mathtt{SCC_n}$ where $0 \leq n \leq |\mathtt{V}|$ by the standard Kosaraju's algorithm, where each $\mathtt{SCC_i} = (\mathtt{V}_i, \mathtt{E}_i, \mathtt{W}_i, \mathtt{Q}_i)$. Then, it computes the adaptivity local bound on every $\mathtt{SCC_i}$ in line:4-5 by $\mathsf{AdaptSearch_{SCC}}(\mathtt{c}, \mathtt{SCC_i})$. We guarantee the soundness of the adaptivity local bound on an SCC by Lemma E.1 with formal proof in Appendix E. The $\mathtt{G_{est}}(c)$ is then shrunk into a directed acyclic graph where $\mathtt{SCC_1}, \cdots, \mathtt{SCC_n}$ are all the vertices and the adaptivity local bounds are their weights. There is an edge $s_i \to s_j$ in this shrank graph, as long as we can find an edge $v_i \to v_j \in \mathtt{E_{est}}(c)$ such that $v_1 \in \mathtt{V}_i$, $v_j \in \mathtt{V}_j$ and $i \neq j$. Then, we use the standard breath first search strategy to find the longest weighted path on this DAG and return this length as the adaptivity upper bound.
We guarantee that the length of this longest weighted path is a sound computation of the adaptivity for program c and this longest weighted path is a sound computation of the finite walk having the longest query length on c's estimated dependency graph in Theorem E.1 in Appendix E. If a program c's estimated dependency graph $\mathtt{G_{est}}(c)$ is a DAG, then we prove that the adaptivity upper bound by Algorithm 1 is tight formally in Theorem F.1 in Appendix F.

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

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

In this way, we resolve the **Approximation Challenge** without losing the soundness, formally in Appendix E. This step also guarantees the termination through a boolean list, `visited` in line:7 and line:13.

Algorithm 2 Adaptivity Computation Algorithm on An SCC ($\mathsf{AdaptSearch}_{\mathtt{scc}}(\mathtt{c}, \mathtt{SCC_i})$)

Require: The program c, An strong connected component of $\mathtt{G_{est}}(c)$: $\mathtt{SCC_i} = (\mathtt{V}_i, \mathtt{E}_i, \mathtt{W}_i, \mathtt{Q}_i)$

1: **init**

$\mathtt{r_{scc}}$: $\mathcal{A}_{\mathtt{in}}$, initialized 0, the Adaptivity of this SCC

2: **init**

$\mathtt{visited}$: $\{0, 1\}$ List,

#{length $|\mathtt{V}_i|$, initialize with 0 for every vertex, recording whether a vertex is visited.}

$\mathtt{r}$: $\mathcal{A}_{\mathtt{in}}$ List,

#{length $|\mathtt{V}_i|$, initialize with $\mathtt{Q}(v)$ for every vertex, recording the adaptivity reaching each vertex.}

$\mathtt{flowcapacity}$: $\mathcal{A}_{\mathtt{in}}$ List,

#{length $|\mathtt{V}|$, initialize with ∞ for every vertex, recording the minimum weight when the walk reaching that vertex, inside a cycle}

$\mathtt{querynum}$: INT List,

#{length $|\mathtt{V}|$, initialize with $\mathtt{Q}(v)$ for every vertex, recording the query numbers when the path reaching that vertex, inside a cycle}

3: **if** $|\mathtt{V}_i| = 1$ and $|\mathtt{E}_i| = 0$:

4: **return** $\mathtt{Q}(v)$

5: **def** $\mathtt{dfs}(\mathtt{G}, \mathtt{s}, \mathtt{visited})$:

6: **for** every vertex v connected by a directed edge from s:

7: **if** $\mathtt{visited}[v] = \mathtt{false}$:

8: $\mathtt{flowcapacity[v]} = \min(\mathtt{W_i(v)}, \mathtt{flowcapacity[s]})$;

9: $\mathtt{querynum[v]} = \mathtt{querynum[s]} + \mathtt{Q_i(v)}$;

10: $\mathtt{r[v]} = \max(\mathtt{r[v]}, \mathtt{flowcapacity[v]} \times \mathtt{querynum[v]})$;

11: $\mathtt{visited}[v] = 1$;

12: $\mathtt{dfs}(\mathtt{G}, \mathtt{v}, \mathtt{visited})$;

13: **else**: #{There is a cycle finished}

14: $\mathtt{r[v]} = \max(\mathtt{r[v]}, \mathtt{r[s]} + \min(\mathtt{W_i(v)}, \mathtt{flowcapacity[s]}) * (\mathtt{querynum[s]} + \mathtt{Q_i(v)}))$; #{update the length of the longest walk reaching this vertex on this cycle}

15: **return** $\mathtt{r[c]}$

16: **for** every vertex v in $\mathtt{V}_i$:

17: initialize the $\mathtt{visited}, \mathtt{r}, \mathtt{flowcapacity}, \mathtt{querynum}$ with the same value at line:2.

18: $\mathtt{r_{scc}} = \max(\mathtt{r_{scc}}, \mathtt{dfs}(\mathtt{SCC_i}, \mathtt{v}, \mathtt{visited}))$;

19: **return** $\mathtt{r_{scc}}$

Algorithm Detail Steps The detail steps of `dfs` from line: 2-15 in Algorithm 2 is described as follows.
Line:2 initialize parameters:

1. `flowcapacity` is a list of arithmetic expressions with length $|\mathtt{Q}_i(c)|$ and the initial value ∞ for every element. For every vertex, it records the minimum weight when the path traverses to this vertex.

2. `querynum` is a list of integer with length $|\mathtt{V}_i(c)|$ and the initial value $\mathtt{Q}_i(v)$ for every element. For every vertex, it records the total query numbers when the path traverses to this vertex.

3. The `visited` is initialized by 0 for every element and has length $|\mathtt{V}_{\mathtt{est}}(c)_i|$ as well. It is used to guarantee the termination during recursion.

4. `r` is a list of $\mathcal{A}_{\mathtt{in}}$ initialized with query annotation for each vertex. For each vertex, it maintains the longest query length when the recursion reaches it.

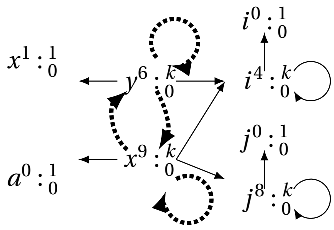
Line:7-12 updates the parameters and recursively traverses for every unvisited vertex heading out from v. In each recursion, Line:8 maintains the minimum weight for the `flowcapacity` and Line:9 updates the number of query vertices `querynum` so far when the traversing reaches v. Line:10 updates the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

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

Example The example program in Figure 6 illustrates how these special operations computes accurate and sound adaptivity for the program. AdaptSearch first find the SCC contains vertices y^6 and x^9, $\mathtt{SCC} = (\mathtt{V}, \mathtt{E})$ where $\mathtt{V} = \{y^6, x^9\}$ and $\mathtt{E} = \{(y^6, y^6), (x^9, x^9), (x^9, y^6), (y^6, x^9)\}$. Then $\mathsf{AdaptSearch}_{\mathtt{SCC}}(\mathtt{SCC}, \mathtt{nestedW})$ takes this SCC as input. When start from vertex y^6, it first finds the path $y^6 \to y^6$. By updating parameters through Line:10 and 14, it computes the longest query length for this path as k. As highlighted in Line:14, we do not update `querynum` and `flowcapacity` when we identify the simple cycle $y^6 \to y^6$. This improves the accuracy and still guarantees the soundness. Because in the following recursions, we continuously search for walks heading out from y^6, the `flowcapacity` of this simple cycle does not restrict the walks going out of this vertex that do not interleave with the cycle $y^6 \to y^6$. However, if we keep updating the minimum weight, then we restrict the visiting times of vertices on a walk by using the minimum weight of vertices that do not on this walk. This leads to the unsoundness in computing adpativity. Concretely, if we update the $\mathtt{flowcapacity}[y^6]$ as k after visiting y^6 the second time on this walk, and continuously visit x^9, then the $\mathtt{flowcapacity[k]}$ 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 of vertices on every searched walk will not be restricted by weights not on this walk, i.e., the soundness. Line: 15 returns the adaptivity heading out from its input vertex. Line:16-18 applies `dfs` on every vertex of this SCC and computes the adaptivity of this SCC by taking the maximum value.The soundness is formally guaranteed in Lemma E.1 in Appendix E.

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

(a)



(b)

Figure 6: (a) The nested while loop example, (b) The estimated dependency graph generated from AdaptFun.

Algorithm 3 Over-Approximated Adaptivity on SCC

Require: $G = (\texttt{V}, \texttt{E}, \texttt{W}, \texttt{Q})$ #{An Strong Connected Symbolic Weighted Directed Graph}

$\text{AdaptSearch}_{\texttt{scc-naive}}(\texttt{G})$**:**
init
$\texttt{r}_{\texttt{scc}}$: the Adaptivity of this SCC
for every vertex v in $\texttt{V}$:
  $r_{scc} + = \texttt{W}(v) * \texttt{Q}(v)$
return $r[c]$

Theorem 4.4 (Soundness of AdaptSearch)**.** *For every program c, given its* estimated dependency graph $\texttt{G}_{\texttt{est}}$,

$$\text{AdaptSearch}(\texttt{G}_{\texttt{est}}) \geq \texttt{A}_{\texttt{est}}(\texttt{G}_{\texttt{est}}).$$

5 Examples and Experimental Results

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

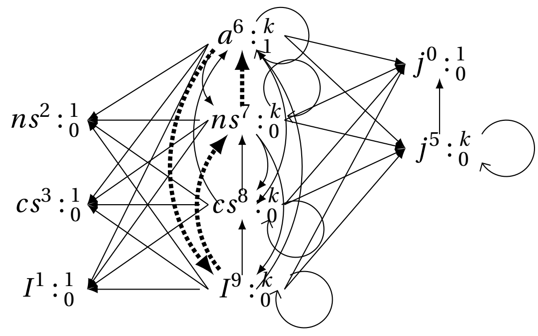
5.1 Examples

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

Different from `twoRounds(k)` *in Figure 2, the query request,* $\left[a \leftarrow \texttt{query}(I)\right]^6$ *in each loop iteration is not independent.* $\texttt{query}(I)$ *in each iteration depends on the tracking list I from all the previous iterations, and I is updated by all the query results in the previous iterations as well. In this sense, all these k queries are adaptively chosen according to our discussion in overview. The program-based dependency graph is presented in Figure 7(b). We omitted its execution-based dependency graph* $\texttt{G}_{\texttt{trace}}(\texttt{multipleRounds(k)})$ *because they have the same graph topology and only differ in weights. For each vertex v in* $\texttt{G}_{\texttt{est}}(\texttt{multipleRounds(k)})$ *in Figure 7(b), we use* w_v *to denote its weight function in* $\texttt{G}_{\texttt{trace}}(\texttt{multipleRounds(k)})$.

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

(a)

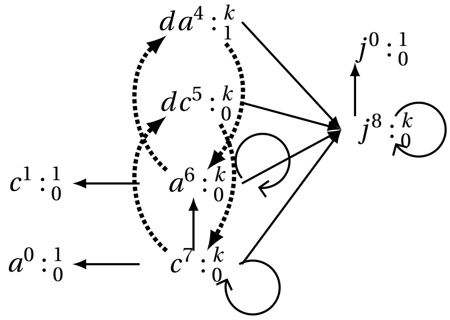


(b)

Figure 7: (a) The simplified multiple rounds example (b) The estimated dependency graph by AdaptFun

$\texttt{lR(k,r)} \triangleq$
$[a \leftarrow 0]^0; [c \leftarrow 0]^1; \left[j \leftarrow \texttt{k}\right]^2;$
$\texttt{while } \left[j > 0\right]^3 \texttt{ do}$
$\Big(\left[da \leftarrow \texttt{query}(-2 * (\chi[1] - (\chi[0] \times a + c)) \times (\chi[0]))\right]^4;$
$\left[dc \leftarrow \texttt{query}(-2 * (\chi[1] - (\chi[0] \times a + c)))\right]^5;$
$[a \leftarrow a - \texttt{r} * da]^6; [c \leftarrow c - \texttt{r} * dc]^7;$
$\left[j \leftarrow j-1\right]^8\Big);$

(a)



(b)

Figure 8: (a) The linear regression algorithm (b) The estimated dependency graph by AdaptFun

As the adaptivity definition in our formal adaptivity model in Definition 14, there is a finite walk along the dashed arrows, $a^6 \to I^9 \to ns^7 \to \cdots \to ns^7$, where the vertices a^6, I^9 and ns^7 are visited $w_{a^6}(\tau_0)$, $w_{I^9}(\tau_0)$ and $w_{ns^7}(\tau_0)$ times respectively with input τ_0. The vertex a^6 has query annotation 1, and it is visited $w_{a^6}(\tau_0)$ times. In this sense, the adaptivity of this program is $w_{a^6}(\tau_0)$ given input τ_0, i.e., $A(\texttt{multipleRounds(k)})(\tau_0) = w_{a^6}(\tau_0)$. Since $w_{a^6}(\tau_0)$ counts the execution times of command $\left[a \leftarrow \texttt{query}(I)\right]^6$;, this count is at most the loop iteration numbers, i.e., k's initial value, $\rho(\tau_0)k$ from the initial trace τ_0. Next, we show that AdaptFun provides a tight upper bound for this example by AdaptSearch($\texttt{multipleRounds(k)}$). *It first finds a path on $\texttt{G}_{\texttt{est}}(\texttt{multipleRounds(k)})$ $a^6 : {}^k_1 \to I^9 : {}^k_0 \to ns^7 : {}^k_0$ with three weighted vertices. Then* AdaptSearch *approximates this path to a walk $a^6 : {}^k_1 \to I^9 : {}^k_0 \to ns^7 : {}^k_0 \to a^6 : {}^k_1 \cdots$. In this walk, a^6, I^9, ns^7 are all visited k times respectively. So $\texttt{A}_{\texttt{est}}(\texttt{multipleRounds(k)}) = k$. We know for any initial trace τ_0, $\langle \tau_0, k \rangle \Downarrow_e \rho(\tau_0)$ So we guarantee $A(\texttt{multipleRounds(k)})(\tau_0) \leq \rho(\tau_0)$ for any τ_0 and k is a sound bound.*

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

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

$\mathtt{lR(k,r)}$ starts from initializing the linear model parameters and the counter variable in commands $0,1,2$, and then goes into the while iterations. In each iteration, it computes the differential value w.r.t. parameter a and c, through two query requests, $\mathtt{query}(-2*(\chi[1]-(\chi[0]\times a+c))\times(\chi[0]))$ and $\mathtt{query}(-2*(\chi[1]-(\chi[0]\times a+c)))$ in commands of label 4 and 5. Then, it uses these two differential values stored in variables da and dc to update the model parameters a and c. In this sense, the two query requests in each iteration depends on queries in the previous iterations and the intuitive adaptivity rounds is k. Its program-based dependency graph, $\mathtt{G_{est}}(\mathtt{lR(k,r)})$ is shown in Figure 12(b), and $\mathtt{G_{trace}}(\mathtt{lR(k,r)})$ is omitted for the same reason as Example 5.1. In Figure 12(b), we omit the edges which are constructed by the transition of `flowsTo` relation for concise, but these edges exist in $\mathtt{G_{trace}}(\mathtt{lR(k,r)})$ because they can be constructed directly by $\mathsf{DEP_{var}}$ relation. Given an input τ_0, there are multiple walks having the same longest query length in $\mathtt{G_{trace}}(\mathtt{lR(k,r)})$, such as the walks $c^7 \to dc^5 :\to c^7 \to \cdots \to dc^5$ and $a^6 \to da^4 \to a^6 \to \cdots \to da^4$ along the dotted arrows. The vertices c^7, dc^5, a^6, da^4 in the two walks are visited $w_{c^7}, w_{dc^5}, w_{a^6}, w_{da^4}$ respectively. Though the different weight functions count the execution times of the different corresponding command, the counts are expected to be the same number, i.e., the loop iterations. And the loop iterations are indeed the initial value of input k from initial trace τ_0. In this sense, $A(\mathtt{lR(k,r)})(\tau_0) = w_{dc^5}(\tau_0) = w_{da^4}(\tau_0)$. Similar to Example 5.1, AdaptFun estimates the tight adaptivity bound, k for this example.

Example 5.3 (Multiple Rounds Odds Algorithm)**.** *The AdaptFun comes across an over-approximation due to its path-insensitive nature. It occurs when the control flow can be decided in a particular way in front of conditional branches, while the static analysis fails to witness. As in Figure 9(a),* $\mathtt{multiRoundsO}(k)$ *is an example program with* $1+k$ *adaptivity rounds and two paths while loop. In each iteration, the query* $[y \leftarrow \mathtt{query}(\chi[x])]^5$ *and* $[p \leftarrow \mathtt{query}(\chi[x])]^6$ *are based on previous query results stored in x, which is similar to Example 5.1. The difference is that, only the query answer from* $[y \leftarrow \mathtt{query}(\chi[x])]^5$ *in the first branch is used in the query in command* 7, $\mathtt{query}(\chi[\ln(y)])$*, and the first branch is only executed in even iterations (*$i = 0,2,\cdots$*). From the Semantics-based dependency graph in Figure 9(b), the weight* $w_{y^5}(\tau_0)$ *for the vertex* y^5 *will count the precise evaluation times of* $[y \leftarrow \mathtt{query}(\chi[x])]^5$*, i.e., half of the iteration numbers. This number is expected to be half of the initial value of input k from* τ_0*. However, AdaptFun fails to realize that all the odd iterations only execute the first branch and only even iterations execute the second branch. So it considers both branches for every iteration when estimating the adaptivity. In this sense, the weight estimated for* y^5 *and* p^6 *are both k as in Figure 9(c). As a result, AdaptFun computes* $y^5 \to x^7 \to y^5 \to \cdots \to x^7$ *as the longest walk in Figure 9(c)*[1] *where each vertex is visited k times. In this sense, the estimated adaptivity is* $1+2*k$*, instead of* $1+k$*.*

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

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

[1] Again, we omit the edges which are constructed by the transition of `flowsTo` relation for concise, but these edges exist in $\mathtt{G_{trace}}(\mathtt{multiRoundsO(k)})$ because they can be constructed directly by $\mathsf{DEP_{var}}$ relation.

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

(a)

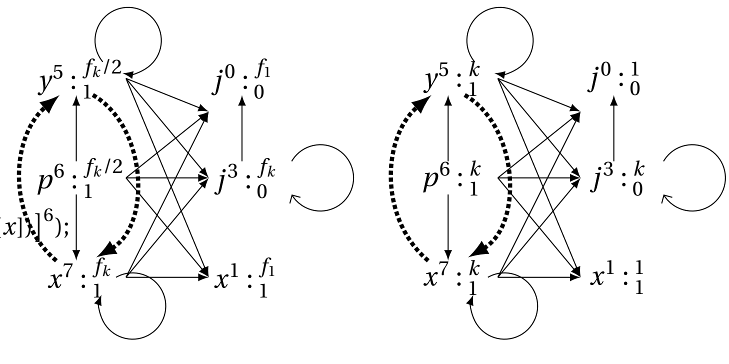


Figure 9: (a) The multiple rounds odd example (b) The semantics-based dependency graph (c) The estimated dependency graph from AdaptFun.

$\texttt{multiRoundsS(k)}$
$\left[j \leftarrow 0\right]^0; \left[z \leftarrow \texttt{query}(0)\right]^1; \left[p \leftarrow 0\right]^2;$
$\texttt{if}\ (\left[k = 0\right]^3, \left[y \leftarrow \texttt{query}(z)\right]^4, \left[\texttt{skip}\right]^5);$
$\texttt{while}\ \left[j \neq k\right]^6\ \texttt{do}$
$\Big(\left[p \leftarrow \texttt{query}(\chi[y] + p)\right]^7; \left[j \leftarrow j+1\right]^8$
$\texttt{if}\ (\left[j \neq k-2\right]^9, \left[p \leftarrow 0\right]^{10}, \left[\texttt{skip}\right]^{10})\Big);$

(a)

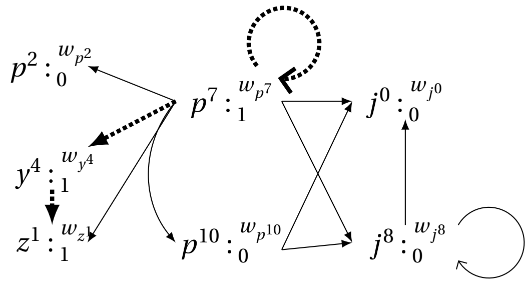


Figure 10: (a) The multi rounds single example (b) The semantics-based dependency graph.

query request. In this way, all the query answers stored in p *is erased and isn't used in the query request at next iteration, except the one at the* $(k-2)$*th iteration. In this sense, the intuitive* adaptivity *rounds for this example is* 2*. However, our adaptivity definition fails to realize that there is only dependency relation between* p^7 *and* p^7 *in one single iteration, but not in all the others. As shown in the semantics-based dependency graph in Figure 10(b), there is an edge from* p^7 *to itself representing the existence of* Variable May-Dependency *from* p^7 *on itself, and the visiting times of labeled variable* p^7 *is* $w(\tau_0)$*.* $w(\tau_0)$ *will count the execution times of command* $\left[p \leftarrow \texttt{query}(\chi[y] + p)\right]^7$ *during execution, which is expected to be equal to the loop iteration numbers, i.e., initial value of input* k *from the 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* $w_{p^7}(\tau_0)$*, as the dotted arrows. The adaptivity based on this walk is* $2 + w_{p^7}(\tau_0)$*, instead of* 2*. Though the* AdaptFun *is able to give us* $2 + k$*, as an accurate bound w.r.t this definition. x*

5.2 Implementation Results

We implemented AdaptFun as a tool which takes a labeled command as input and outputs two upper bounds on the program adaptivity and the number of query requests respectively. This implementation consists of an abstract control flow graph generation, edge estimation (as presented in Section 4.3.2), and weight estimation (as presented in Section 4.3.3) in Ocaml, and the adaptivity computation algorithm shown in Section 4.5 in Python. The OCaml program takes the labeled command as input and outputs the program-based dependency graph and the abstract transition graph, feeds into the python program and the python program provides the adaptivity upper bound and the query number as the final output.

We evaluated this implementation on 23 example programs with the evaluation results shown in Table 2. In this table, the first column is the name of each program. For each program c, the second column is its intuitive adaptivity rounds, the third column is the output of the AdaptFun implementation, which consists of two expressions. The first one is the upper bound for adaptivity and the second one is the upper bound for the total number of query requests in the program. And the last column is the performance evaluation w.r.t. the program size.

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

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

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

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

Overall for these examples, our system gives both the accurate adaptivity definition and estimated adaptivity upper bound through our formalization and analysis framework AdaptFun. The complete programs are defined below from Example 5.5 to Example H.14 in the Appendix H.

Table 1: Experimental results of AdaptFun implementation

Program c	*adaptivity*	AdaptFun		performance			
		AdaptSearch(c) (I $\mid$ II)	query# (I $\mid$ II)	lines	running time (second)		
					Ocaml	Weight	AdaptSearch
`twoRounds(k)`	2	$2 \mid -$	$k+1 \mid -$	8	0.0005	$0.0017 \mid 0.0002$	0.0003
`multiRounds(k)`	k	$k \mid \max(1,k)$	$k \mid -$	10	0.0012	$0.0017 \mid 0.0002$	0.0002
`lRGD(k,r)`	k	$k \mid \max(1,k)$	$2k \mid -$	10	0.0015	$0.0072 \mid 0.0002$	0.0002
`mROdd(k)`	$1+k$	$2+\max(1,2k) \mid -$	$1+3k \mid -$	10	0.0015	$0.0061 \mid 0.0002$	0.0002
`mRSingle(k)`	2	$1+\max(1,k) \mid -$	$1+k \mid 1+k$	9	0.0011	$0.0075 \mid 0.0002$	0.0002
`ifCD()`	3	$3 \mid 4$	$3 \mid 4$	5	0.0005	$0.0003 \mid 0.0001$	0.0001
`while(k)`	$1+k/2$	$1+\max(1,k/2) \mid -$	$1+k/2 \mid -$	7	0.0021	$0.0015 \mid 0.0001$	0.0001
`whileRV(k)`	$1+2k$	$1+2k \mid 1+\max(1,2k)$	$2+3k \mid -$	9	0.0016	$0.0056 \mid 0.0002$	0.0001
`whileVCD(k)`	$1+2Q_m$	$Q+\max(1,2Q_m) \mid -$	$2+2Q_m \mid -$	6	0.0016	$0.0007 \mid 0.0002$	0.0001
`whileMPVCD(k)`	$2+Q_m$	$2+Q_m \mid -$	$2+2Q_m \mid -$	9	0.0017	$0.0043 \mid 0.0002$	0.0001
`nestWhileVD(k)`	$2+k^2$	$3+k^2 \mid -$	$1+k+k^2 \mid -$	10	0.0018	$0.0126 \mid 0.0002$	0.0001
`nestWhileRV(k)`	$1+k+k^2$	$2+k+k^2 \mid -$	$2+k+k^2 \mid -$	10	0.0017	$0.0186 \mid 0.0002$	0.0001
`nestWhileMV(k)`	$1+2k$	$1+\max(1,2k) \mid -$	$1+k+k^2 \mid -$	10	0.0016	$0.0071 \mid 0.0002$	0.0001
`nestWhileMPRV(k)`	$1+k+k^2$	$3+k+k^2 \mid -$	$2+2k+k^2 \mid -$	10	0.019	$0.0999 \mid 0.0002$	0.0002
`whileM(k)`	$1+k$	$2+\max(1,2k) \mid -$	$1+3k \mid -$	9	0.0017	$0.0062 \mid 0.0002$	0.0001
`whileM2(k)`	$1+k$	$2+k \mid -$	$1+3k \mid -$	9	0.0017	$0.0062 \mid 0.0002$	0.0001
`nestWhileRC(k)`	$1+3k$	$1+3k \mid 2+3k+k^2$	$1+3k \mid 1+k+k^2$	11	0.019	$0.2669 \mid 0.0002$	0.0007
`mRComplete(k,N)`	k	$k \mid -$	$k \mid -$	27	0.0026	$85.9017 \mid 0.0003$	0.0004
`mRCompose(k)`	$2k$	$2k \mid -$	$2k \mid -$	46	0.0036	$5104 \mid 0.0003$	0.0013
`seqCompose(k)`	12	$12 \mid -$	$326 \mid -$	502	0.0426	$1.2743 \mid 0.0003$	0.0223
`tRCompose(k)`	2	$* \mid 2$	$* \mid 1+5k+2k^2$	42	0.0026	$* \mid 0.0003$	0.0005
`jumboS(k)`	$\max(20,8+k^2)$	$* \mid \max(20,6+k+k^2)$	$* \mid 44+k+k^2$	71	0.0035	$* \mid 0.0003$	0.0085
`jumbo(k)`	$\max(20,10+k+k^2)$	$* \mid \max(20,12+k+k^2)$	$* \mid 286+26k+10k^2$	502	0.0691	$* \mid 0.0009$	0.018
`big(k)`	$22+k+k*k$	$* \mid 28+k+k^2$	$* \mid 121+11k+4k^2$	214	0.0175	$* \mid 0.0004$	0.002

5.3 More Discussions on The Evaluated Examples

5.3.1 More on The Two Rounds Adaptive Data Analysis

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

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

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

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 $q_k(d) = d(k) \cdot \texttt{sign}\big(\sum_{i\in[k]} x(i) \cdot \ln \frac{1+a_i}{1-a_i}\big)$ where $x \in \{-1,+1\}^{k+1}$.
  {In the line above, $\texttt{sign}(y) = \begin{cases} +1 & \texttt{if } y \geq 0 \\ -1 & \texttt{otherwise} \end{cases}$.}
  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: $a_{k+1} \in [-1,+1]$

$$\texttt{multiRounds(k,c,N)} \triangleq$$

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

(a)

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

5.3.2 `mRComplete`

Algorithm 5 A multi-round analyst strategy for random data base [2]

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.

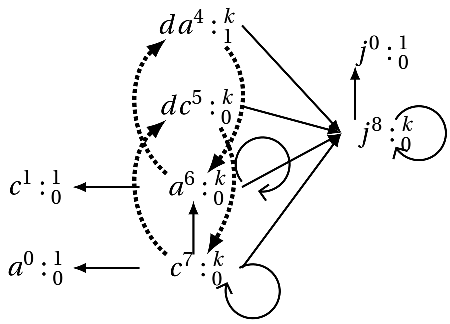
5.3.3 `lRGD`

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

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

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

(a)



(b)

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

the second column for the dependent variable y. $\texttt{lR(k,r)}$ is the program of this example written in our language model in Figure 12(a) with input variables k and r.

$\texttt{lR(k,r)}$ starts from initializing the linear model parameters and the counter variable in commands $0, 1, 2$, and then goes into the while iterations. In each iteration, it computes the differential value w.r.t. parameter a and c, through two query requests, $\texttt{query}(-2 * (\chi[1] - (\chi[0] \times a + c)) \times (\chi[0]))$ and $\texttt{query}(-2 * (\chi[1] - (\chi[0] \times a + c)))$ in commands of label 4 and 5. Then, it uses these two differential values stored in variables da and dc to update the model parameters a and c. In this sense, the two query requests in each iteration depends on queries in the previous iterations and the intuitive adaptivity rounds is k. Its program-based dependency graph, $\texttt{G}_{\texttt{est}}(\texttt{lR(k,r)})$ is shown in Figure 12(b), and $\texttt{G}_{\texttt{trace}}(\texttt{lR(k,r)})$ is omitted for the same reason as Example 5.1. In Figure 12(b), we omit the edges which are constructed by the transition of $\texttt{flowsTo}$ relation for concise, but these edges exist in $\texttt{G}_{\texttt{trace}}(\texttt{lR(k,r)})$ because they can be constructed directly by $\mathsf{DEP}_{\mathsf{var}}$ relation. Given an input τ_0, there are multiple walks having the same longest query length in $\texttt{G}_{\texttt{trace}}(\texttt{lR(k,r)})$, such as the walks $c^7 \to dc^5 :\to c^7 \to \cdots \to dc^5$ and $a^6 \to da^4 \to a^6 \to \cdots \to da^4$ along the dotted arrows. The vertices c^7, dc^5, a^6, da^4 in the two walks are visited $w_{c^7}, w_{dc^5}, w_{a^6}, w_{da^4}$ respectively. Though the different weight functions count the execution times of the different corresponding command, the counts are expected to be the same number, i.e., the loop iterations. And the loop iterations are indeed the initial value of input k from initial trace τ_0. In this sense, $A(\texttt{lR(k,r)})(\tau_0) = w_{dc^5}(\tau_0) = w_{da^4}(\tau_0)$. Similar to Example 5.1, AdaptFun estimates the tight adaptivity bound, k for this example.

Appendices

A Proofs of Lemmas for the Language Model

A.1 Proof of Lemma 1.1

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

A.2 Proof of Lemma 2.1

Proof. Taking arbitrary trace $\tau \in \mathcal{T}$, 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. $\square$

A.3 Proof of Lemma 2.0.1

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

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

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

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

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

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

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

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

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

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

B Proof of Theorem 4.2

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

$$\forall \tau_0 \in \mathcal{T}_0(c), v \in \mathbb{N}^\infty . \ \langle \mathtt{A_{est}}(c), \tau_0 \rangle \Downarrow_e v \implies A(c)(\tau_0) \leq c$$

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

Proof. Given a program c, we construct its
program-based graph $\mathtt{G_{est}}(c) = (\mathtt{V_{est}}, \mathtt{E_{est}}, \mathtt{W_{est}}, \mathtt{Q_{est}})$ by Definition 24
and trace-based graph $\mathtt{G_{trace}}(c) = (\mathtt{V_{trace}}, \mathtt{E_{trace}}, \mathtt{W_{trace}}, \mathtt{Q_{trace}})$ by Definition 8.

The parameter (c) for the components in the two graphs are omitted for concise.
According to the Definition 27 and Definition 14, it is sufficient to show:

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

Then it is sufficient to show that:

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

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 B.1 and Lemma B.2, we know

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

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 $e_{pi} = (v_i, v_{i+1}) \in \mathtt{E}_{\mathtt{est}}$ 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

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

To show $k_p \in \mathtt{G}_{\mathtt{est}}(c)$, by Definition 12 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

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

Then, by Definition 25, we know the occurrence times for every $v_i \in (v_1, \cdots, v_n)$ is bound by the arithmetic expression w_i where $(v_i, w_i) \in \mathtt{W}_{\mathtt{est}}(c)$.
So we have $k_p \in \mathcal{WK}(\mathtt{G}_{\mathtt{est}})$ proved.
In order to show $n \geq \mathtt{len}^{\mathtt{q}}(k_t)(\tau)$, it is sufficient to show

$$\begin{array}{l}\forall v_i \in (v_1, \cdots, v_n), (v_i, w_i) \in \mathtt{W}_{\mathtt{est}}(c), (v_i, w_i') \in \mathtt{W}_{\mathtt{trace}}(c), n_i \in \mathbb{N} \,.\, \langle w_i, \tau\rangle \Downarrow_e n_i \\ \implies \sum_{\mathtt{Q}_{\mathtt{trace}}(c)(v_i)=1} w_i'(\tau) \leq \sum_{\mathtt{Q}_{\mathtt{est}}(c)(v_i)=1} n_i\end{array}$$

By Lemma B.4 and Definition 26, we know for every v_i, $\mathtt{Q}_{\mathtt{trace}}(c)(v_i) = \mathtt{Q}_{\mathtt{est}}(c)(v_i)$
Then by $(\star)$, we know $\sum_{\mathtt{Q}_{\mathtt{trace}}(c)(v_i)=1} w_i'(\tau) \leq \sum_{\mathtt{Q}_{\mathtt{est}}(c)(v_i)=1} n_i$.
Then we have $n \geq \mathtt{len}^{\mathtt{q}}(k_t)(\tau)$ proved.
This theorem is proved. □

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

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

$$\begin{array}{l}\forall c \in \mathcal{C}, v \in \mathcal{VAR} \times \mathbb{N} \,.\, \mathtt{G}_{\mathtt{est}}(c) = (\mathtt{V}_{\mathtt{est}}, \mathtt{E}_{\mathtt{est}}, \mathtt{W}_{\mathtt{est}}, \mathtt{Q}_{\mathtt{est}}) \wedge \mathtt{G}_{\mathtt{trace}}(c) = (\mathtt{V}_{\mathtt{trace}}, \mathtt{E}_{\mathtt{trace}}, \mathtt{W}_{\mathtt{trace}}, \mathtt{Q}_{\mathtt{trace}}) \\ \quad \implies v \in \mathtt{V}_{\mathtt{est}} \iff v \in \mathtt{V}_{\mathtt{trace}}\end{array}$$

Proof. Proof Summary: Proving by Definition 24 and Definition 8.
Taking arbitrary program c, by Definition 24 and Definition 8, we have
its program-based graph $\mathtt{G_{est}}(c) = (\mathtt{V_{est}}, \mathtt{E_{est}}, \mathtt{W_{est}}, \mathtt{Q_{est}})$
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 $\mathtt{V_{est}} = \mathbb{LV}_c$.
Then we know $\mathtt{V_{trace}} = \mathtt{V_{est}}$, i.e., for arbitrary $v \in \mathcal{VAR} \times \mathbb{N}$, $v \in \mathtt{V_{est}} \Longleftrightarrow v \in \mathtt{V_{trace}}$. □

Lemma B.2 (Mapping from Egdes of $\mathtt{G_{trace}}$ to $\mathtt{G_{est}}$). *Given a program c with its program-based graph $\mathtt{G_{est}}(c) = (\mathtt{V_{est}}, \mathtt{E_{est}}, \mathtt{W_{est}}, \mathtt{Q_{est}})$ 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 $e' = (v_1', v_2') \in \mathtt{E_{est}}$ with $v_1 = v_1' \wedge v_2 = v_2'$.*

$$
\begin{array}{l}
\forall c \in \mathcal{C} \,.\, \mathtt{G_{est}}(c) = (\mathtt{V_{est}}, \mathtt{E_{est}}, \mathtt{W_{est}}, \mathtt{Q_{est}}) \wedge \mathtt{G_{trace}}(c) = (\mathtt{V_{trace}}, \mathtt{E_{trace}}, \mathtt{W_{trace}}, \mathtt{Q_{trace}}) \\
\quad \Longrightarrow \forall e = (v_1, v_2) \in \mathtt{E_{trace}} \,.\, \exists e' \in \mathtt{E_{est}} \,.\, e' = (v_1, v_2)
\end{array}
$$

Proof. Proof Summary: Proving by Lemma B.1, Lemma C.1 Definition 24 and Definition 8
Taking arbitrary program c, by Definition 24 and Definition 8, we have
its program-based graph $\mathtt{G_{est}}(c) = (\mathtt{V_{est}}, \mathtt{E_{est}}, \mathtt{W_{est}}, \mathtt{Q_{est}})$
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 $(x^i, y^j) \in \mathtt{E_{est}}$.
By Lemma B.1, we know $x^i, y^j \in \mathtt{V_{est}}$.
By definition of $\mathtt{E_{trace}}$, we know $\mathsf{DEP}_{\mathsf{var}}(x^i, y^j, c)$.
By Theorem C.1, we know $\exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c \,.\, n \geq 0 \wedge \mathtt{flowsTo}(x^i, z_1^{r_1}, c) \wedge \cdots \wedge \mathtt{flowsTo}(z_n^{r_n}, y^j, c)$.
Then by definition of $\mathtt{E_{est}}$, we know $(x^i, y^j) \in \mathtt{E_{est}}$. This Lemma is proved. □

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

$$
\begin{array}{l}
\forall c \in \mathcal{C} \,.\, \mathtt{G_{est}}(c) = (\mathtt{V_{est}}, \mathtt{E_{est}}, \mathtt{W_{est}}, \mathtt{Q_{est}}) \wedge \mathtt{G_{trace}}(c) = (\mathtt{V_{trace}}, \mathtt{E_{trace}}, \mathtt{W_{trace}}, \mathtt{Q_{trace}}) \\
\quad \Longrightarrow \forall (x^l, w_t) \in \mathtt{W_{trace}}, (x^l, w_p) \in \mathtt{W_{est}}, \tau, \tau' \in \mathcal{T}, v \in \mathbb{N} \,.\, \langle w_p, \tau \rangle \Downarrow_e v \Longrightarrow w_t(\tau) \leq v
\end{array}
$$

Proof. Proof Summary: Proving by Definition 24, Definition 8 and relying on the soundness of Reachability Bound Analysis.
Taking arbitrary program c, by Definition 24 and Definition 8, we have
its program-based graph $\mathtt{G_{est}}(c) = (\mathtt{V_{est}}, \mathtt{E_{est}}, \mathtt{W_{est}}, \mathtt{Q_{est}})$
and trace-based graph $\mathtt{G_{trace}}(c) = (\mathtt{V_{trace}}, \mathtt{E_{trace}}, \mathtt{W_{trace}}, \mathtt{Q_{trace}})$.
Taking arbitrary $(x^l, w_t) \in \mathtt{W_{trace}}, (x^l, w_p) \in \mathtt{W_{est}}, \tau, \tau' \in \mathcal{T}$, 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 D.2, we know $\mathtt{cnt}(\tau', l) \leq v$
By definition 8, we know $w_t(\tau) = \mathtt{cnt}(\tau', l)$, then we have $w_t(\tau) \leq v$ and this is proved. □

Lemma B.4 (One-on-One Mapping for Query Vertices). *Given a program c with its program-based graph $\mathtt{G_{est}}(c) = (\mathtt{V_{est}}, \mathtt{E_{est}}, \mathtt{W_{est}}, \mathtt{Q_{est}})$ 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 $(x^i, n) \in \mathtt{Q_{est}}$.*

$$
\begin{array}{l}
\forall c \in \mathcal{C}, (x^i, n) \in \mathcal{VAR} \times \mathbb{N} \times \{0, 1\} \,. \\
\quad \mathtt{G_{est}}(c) = (\mathtt{V_{est}}, \mathtt{E_{est}}, \mathtt{W_{est}}, \mathtt{Q_{est}}) \wedge \mathtt{G_{trace}}(c) = (\mathtt{V_{trace}}, \mathtt{E_{trace}}, \mathtt{W_{trace}}, \mathtt{Q_{trace}}) \\
\quad \Longrightarrow (x^i, n) \in \mathtt{Q_{trace}} \Longleftrightarrow (x^i, n) \in \mathtt{Q_{est}}
\end{array}
$$

Proof. Proving by Definition 24, Definition 8.
Taking arbitrary program c, by Definition 24 and Definition 8, we have
its program-based graph $\mathtt{G_{est}}(c) = (\mathtt{V_{est}}, \mathtt{E_{est}}, \mathtt{W_{est}}, \mathtt{Q_{est}})$
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 $\mathtt{Q_{trace}} = \mathtt{Q_{est}}$, i.e., for arbitrary $(x^i, n) \in \mathcal{VAR} \times \mathbb{N} \times \{0, 1\}$, $(x^i, n) \in \mathtt{Q_{trace}} \Longleftrightarrow (x^i, n) \in \mathtt{Q_{est}}$.
This lemma is proved. ∎

C Soundness of Edge Estimation

C.1 Main Theorem

Theorem C.1 ($\mathsf{DEP}_{\mathsf{var}}$ implies $\mathtt{flowsTo}$)**.** *Given a program c, for all $x^i, y^j \in \mathbb{LV}_c$, if* $\mathsf{DEP}_{\mathsf{var}}(x^i, y^j, c)$*, then there exist $z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c$ with $n \geq 0$ such that* $\mathtt{flowsTo}(x^i, z_1^{r_1}, c) \land \cdots \land \mathtt{flowsTo}(z_n^{r_n}, y^j, c)$

$$
\begin{aligned}
&\forall x^i, y^j \in \mathbb{LV}_c . \mathsf{DEP}_{\mathsf{var}}(x^i, y^j, c) \\
&\quad \implies \Big(\exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c \ . \ n \geq 0 \land \mathtt{flowsTo}(x^i, z_1^{r_1}, c) \land \cdots \land \mathtt{flowsTo}(z_n^{r_n}, y^j, c) \Big)
\end{aligned}
$$

Proof Summary, for arbitrary two $x^i, y^j \in \mathbb{LV}_c$, with *Variable May-Dependency* relation, in order to show there exists a "flows-to chain" relation from the static analysis results from x^i to y^j, it is sufficient to show:
1. x^i is directly used in the expression of the assignment command associated to y^j, or a boolean expression of the guard for a `if` or `while` command with the assignment command associated to y^j showing up in the body of that command, we call it, x^i directly flows to y^j, i.e., $\mathtt{flowsTo}(x^i, y^j, c)$;
2. otherwise, there exists another labelled variable z^l with *variable May-Dependency* relation on x^i and z^l directly flows to y^j, where the *variable May-Dependency* relation between x^i and z^l implies a "sub flowsto-chain" from z^i to z^l, i.e.,
$\Big(\exists z^l \in \mathbb{LV}_c . \big(\mathsf{DEP}_{\mathsf{var}}(x^i, z^l, c) \implies \exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c \ . \ n \geq 0 \land \mathtt{flowsTo}(x^i, z_1^{r_1}, c) \land \cdots \land \mathtt{flowsTo}(z_n^{r_n}, z^l, c) \big) \land \mathtt{flowsTo}(z^l, y^j, c) \Big)$.
By definition of $\mathsf{DEP}_{\mathsf{var}}(x^i, y^j, c)$, let $D \in \mathcal{DB}$ be the dataset, and $\tau \in \mathcal{T}$, ϵ_x, ϵ_y be the trace and two events satisfying the definition, with $\pi_1(\epsilon_x) = x$ and $\pi_1(\epsilon_y) = y$,

$$
\begin{aligned}
&(\mathtt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c) \\
&\quad \lor (\exists \epsilon_z \in \tau' \ . \ \epsilon_z \in \mathcal{E}^{\mathsf{asn}} \land \mathsf{DEP}_{\mathsf{e}}(\epsilon_x, \epsilon_z, \tau[\epsilon_x : \epsilon_z], c, D) \\
&\qquad \implies \exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c \ . \ n \geq 0 \land \mathtt{flowsTo}(x^i, z_1^{r_1}, c) \land \cdots \land \mathtt{flowsTo}(z_n^{r_n}, \pi_1(\epsilon_z)^{\pi_2(\epsilon_z)}, c)) \\
&\quad \land \mathtt{flowsTo}(\pi_1(\epsilon_z)^{\pi_2(\epsilon_z)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)
\end{aligned}
$$

It is clearer to show it in two lemmas:
1. Existence of a middle event: in Lemma C.3.
2. The middle event with a sub-trace implies a "sub flowsto-chain", by induction on the trace τ

$$
\begin{aligned}
&\forall D \in \mathcal{DB}, c \in \mathcal{C}, \tau \in \mathcal{T} \ . \ \forall \epsilon_1, \epsilon_2 \in \mathcal{E} \ . \ \epsilon_1, \epsilon_2 \in \mathcal{E}^{\mathsf{asn}} \land \exists \tau' \in \mathcal{T} \ . \ \tau = [\epsilon_1] {++} \tau' {++} [\epsilon_2] \implies \mathsf{DEP}_{\mathsf{e}}(\epsilon_1, \epsilon_2, \tau, c, D) \\
&\quad \implies \exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c \ . \ n \geq 0 \land \mathtt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, z_1^{r_1}, c) \land \cdots \land \mathtt{flowsTo}(z_n^{r_n}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)
\end{aligned}
$$

with the induction hypothesis:

$$
\begin{aligned}
&\forall \epsilon_{ih1}, \epsilon_{ih2} \in \tau \ . \ \epsilon_{ih1}, \epsilon_{ih2} \in \mathcal{E}^{\mathsf{asn}} \land \exists \tau' \in \mathcal{T} \ . \ \tau[\epsilon_{ih1} : \epsilon_{ih2}] = [\epsilon_{ih1}] {++} \tau' {++} [\epsilon_{ih2}] \implies \mathsf{DEP}_{\mathsf{e}}(\epsilon_{ih1}, \epsilon_{ih2}, \tau[\epsilon_{ih1} : \epsilon_{ih2}], c, D) \\
&\quad \implies \exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c \ . \ n \geq 0 \land \mathtt{flowsTo}(\pi_1(\epsilon_{ih1})^{\pi_2(\epsilon_{ih1})}, z_1^{r_1}, c) \land \cdots \land \mathtt{flowsTo}(z_n^{r_n}, \pi_1(\epsilon_{ih2})^{\pi_2(\epsilon_{ih2})}, c)
\end{aligned}
$$

Proved in Theorem C.2 with the main proof logic:
(1). obtaining the existence of $\epsilon_z \in \mathcal{E}^{\mathsf{asn}}$ with dependency on ϵ_x, and a "direct flowsto" from ϵ_z to ϵ_y from step 1;
(2). from the dependency of the ϵ_z with ϵ_x on the subtrace, obtaining a "sub flowsto-chain" by induction hypothesis;
(3). composing the "sub flowsto-chain" from (2) with the "direct flowsto" from (1), and getting the conclusion of the complete "flowsto chain".

Proof. Taking arbitrary $x^i, y^j \in \mathbb{LV}_c$, by definition of $\mathsf{DEP}_{\mathsf{var}}(x^i, y^j, c)$, let $D \in \mathcal{DB}$ be the dataset, and $\tau \in \mathcal{T}$, ϵ_x,ϵ_y be the trace and two events satisfying the definition, with $\pi_1(\epsilon_x)^{\pi_2(\epsilon_x)} = x^i$ and $\pi_1(\epsilon_y)^{\pi_2(\epsilon_y)} = y^j$, it is sufficient to show:

$$
\begin{aligned}
&\mathsf{DEP}_{\mathsf{e}}(\epsilon_x, \epsilon_y, \tau, c, D)\\
&\quad \implies \exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c \,.\, n \geq 0 \land \mathtt{flowsTo}(x^i, z_1^{r_1}, c) \land \cdots \land \mathtt{flowsTo}(z_n^{r_n}, y^j, c)
\end{aligned}
$$

By Theorem C.2, we have this theorem proved. □

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

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

Definition 28 (Subtrace)**.** *Subtrace:* $[:] : \mathcal{T} \to \mathcal{E} \to \mathcal{E} \to \mathcal{T}$

$$
\tau[\epsilon_1 : \epsilon_2] \triangleq \begin{cases} \tau'[\epsilon_1 : \epsilon_2] & \tau = \epsilon :: \tau' \land \epsilon \neq_{\mathsf{e}} \epsilon_1 \\ \epsilon_1 :: \tau'[: \epsilon_2] & \tau = \epsilon :: \tau' \land \epsilon =_{\mathsf{e}} \epsilon_1 \\ [] & \tau = [] \end{cases}
$$

For any trace τ and two events $\epsilon_1, \epsilon_2 \in \mathcal{E}$, $\tau[\epsilon_1 : \epsilon_2]$ takes the subtrace of τ starting with ϵ_1 and ending with ϵ_2 including ϵ_1 and ϵ_2.
We use $\tau[: \epsilon_2]$ as the shorthand of subtrace starting from head and ending with ϵ_2, and similary for $\tau[\epsilon_1 :]$.

$$
\tau[: \epsilon] \triangleq \begin{cases} \epsilon' :: \tau'[: \epsilon] & \tau = \epsilon' :: \tau' \land \epsilon' \neq_{\mathsf{e}} \epsilon \\ \epsilon' & \tau = \epsilon' :: \tau' \land \epsilon' =_{\mathsf{e}} \epsilon \\ [] & \tau = [] \end{cases}
\qquad
\tau[\epsilon :] \triangleq \begin{cases} \tau'[\epsilon :] & \tau = \epsilon' :: \tau' \land \epsilon \neq_{\mathsf{e}} \epsilon' \\ \epsilon' :: \tau' & \tau = \epsilon' :: \tau' \land \epsilon =_{\mathsf{e}} \epsilon' \\ [] & \tau = [] \end{cases}
$$

Program Entry Point: $\mathtt{entry}_c : \text{Command} \to \mathbb{N}$

$$
\mathtt{entry}_c \triangleq \begin{cases} l & c = [\mathtt{skip}]^l \\ l & c = [x \leftarrow e_1]^l \\ l & c = \left[x \leftarrow \mathtt{query}(\psi_1)\right]^l \\ l & c_1 = \mathtt{if}\ ([b]^l, c_t, c_f) \\ l & c = \mathtt{while}\ [b]^l\ \mathtt{do}\ c' \\ \mathtt{entry}_{c1} & c = c1; c2 \end{cases}
$$

Theorem C.2 ($\mathsf{DEP}_{\mathsf{e}}$ implies `flowsTo`)**.** *For every $D \in \mathcal{DB}, c \in \mathcal{C}, \tau \in \mathcal{T} \,.\, \forall \epsilon_1, \epsilon_2 \in \mathcal{E} \,.\, \epsilon_1, \epsilon_2 \in \mathcal{E}^{\mathtt{asn}}$, if $\exists \tau' \in \mathcal{T} \,.\, \tau = [\epsilon_1]{++}\tau'{++}[\epsilon_2]$ and $\mathsf{DEP}_{\mathsf{e}}(\epsilon_1, \epsilon_2, \tau, c, D)$, then $z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c$ with $n \geq 0$ such that $\mathtt{flowsTo}(x^i, z_1^{r_1}, c) \land \cdots \land \mathtt{flowsTo}(z_n^{r_n}, y^j, c)$*

$$
\begin{aligned}
&\forall D \in \mathcal{DB}, c \in \mathcal{C}, \tau \in \mathcal{T} \,.\, \forall \epsilon_1, \epsilon_2 \in \mathcal{E} \,.\, \epsilon_1, \epsilon_2 \in \mathcal{E}^{\mathtt{asn}} \land \exists \tau' \in \mathcal{T} \,.\, \tau = [\epsilon_1]{++}\tau'{++}[\epsilon_2] \implies \mathsf{DEP}_{\mathsf{e}}(\epsilon_1, \epsilon_2, \tau, c, D)\\
&\quad \implies \exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c \,.\, n \geq 0 \land \mathtt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, z_1^{r_1}, c) \land \cdots \land \mathtt{flowsTo}(z_n^{r_n}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)
\end{aligned}
$$

Proof Summary: I. Vacuously True cases, where trace doesn't satisfy the hypothesis
II. Base case where $\tau = [\epsilon_1; \epsilon_2]$
III. inductive case where $\tau = [\epsilon_1, \cdots, \epsilon_2]$.
1. Existence of a middle event:
Proved by showing a contradiction, with detail in Lemma C.3.
2. The middle event with a sub-trace implies a "sub flowsto-chain", informally:

(1). obtaining the existence of $\epsilon_z \in \mathcal{E}^{\text{asn}}$ with dependency on ϵ_x, and a "direct flowsto" from ϵ_z to ϵ_y by Lemma C.3.
(2). from the dependency of the ϵ_z with ϵ_x on the subtrace, obtaining a "sub flowsto-chain" by induction hypothesis;
(3). composing the "sub flowsto-chain" from (2) with the "direct flowsto" from (1), and getting the conclusion of the complete "flowsto chain".

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

Case 1. $(\tau = [])$
Since for all $\epsilon_1, \epsilon_2 \in \mathcal{E}^{\text{asn}}$, $\nexists \tau' \in \mathcal{T}$,satisfies $[] = [\epsilon_1]{++}\tau'{++}[\epsilon_2]$, the theorem is vacuously true.

Case 2. $(\epsilon \in \mathcal{E}, \tau = [\epsilon])$
Since for all $\epsilon_1, \epsilon_2 \in \mathcal{E}^{\text{asn}}$, $\nexists \tau' \in \mathcal{T}$,satisfies $[] = [\epsilon_1]{++}\tau'{++}[\epsilon_2]$, the theorem is vacuously true.

Case 3. $(\epsilon'_1, \epsilon'_2 \in \mathcal{E}, \tau = [\epsilon'_1; \epsilon'_2])$
To show:

$$\begin{aligned}&\forall \epsilon_1, \epsilon_2 \in \mathcal{E}^{\text{asn}} . \exists \tau' \in \mathcal{T} . [\epsilon'_1; \epsilon'_2] = [\epsilon_1]{++}\tau'{++}[\epsilon_2]\\ &\quad\Longrightarrow \mathsf{DEP_e}(\epsilon_1, \epsilon_2, [\epsilon_1; \epsilon_2], c, D) \Longrightarrow \texttt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)\end{aligned}$$

Taking arbitrary $\epsilon_1, \epsilon_2 \in \mathcal{E}^{\text{asn}}$, by law of excluded middle, there are 2 cases:
$\epsilon_1 = \epsilon'_1 \wedge \epsilon_2 = \epsilon'_2$
$\neg(\epsilon_1 = \epsilon'_1 \wedge \epsilon_2 = \epsilon'_2)$
In case of $\neg(\epsilon_1 = \epsilon'_1 \wedge \epsilon_2 = \epsilon'_2)$, since $\nexists \tau' \in \mathcal{T}$,satisfies $[\epsilon'_1; \epsilon'_2] = [\epsilon_1]{++}\tau'{++}[\epsilon_2]$, the theorem is vacuously true.
In case of $\epsilon_1 = \epsilon'_1 \wedge \epsilon_2 = \epsilon'_2$, let $\tau' = []$, we know $\exists \tau' \in \mathcal{T}$ satisfying $[\epsilon_1; \epsilon_2] = [\epsilon_1]{++}\tau'{++}[\epsilon_2]$.
Then it is sufficient to show:

$$\mathsf{DEP_e}(\epsilon_1, \epsilon_2, [\epsilon_1; \epsilon_2], c, D) \Longrightarrow \texttt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)$$

By Lemma C.1, we have this case proved.

Case 4. $(\epsilon'_1, \epsilon'_2 \in \mathcal{E}, \tau_{ih} \in \mathcal{T}, \tau = [\epsilon'_1]{++}\tau_{ih}{++}[\epsilon'_2] \wedge \tau_{ih} \neq [])$
It is sufficient to show:

$$\begin{aligned}&\forall \epsilon_1, \epsilon_2 \in \mathcal{E} . \epsilon_1, \epsilon_2 \in \mathcal{E}^{\text{asn}} \wedge \exists \tau' \in \mathcal{T} . \tau = [\epsilon_1]{++}\tau'{++}[\epsilon_2] \Longrightarrow \mathsf{DEP_e}(\epsilon_1, \epsilon_2, [\epsilon'_1]{++}\tau_{ih}{++}[\epsilon'_2], c, D)\\ &\quad\Longrightarrow \exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c . n \geq 0 \wedge \texttt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, z_1^{r_1}, c) \wedge \cdots \wedge \texttt{flowsTo}(z_n^{r_n}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)\end{aligned}$$

with the induction hypothesis:

$$\begin{aligned}&\forall \epsilon_{ih1}, \epsilon_{ih2} \in \tau . \epsilon_{ih1}, \epsilon_{ih2} \in \mathcal{E}^{\text{asn}} \wedge \exists \tau' \in \mathcal{T} . \tau[\epsilon_{ih1} : \epsilon_{ih2}] = [\epsilon_{ih1}]{++}\tau'{++}[\epsilon_{ih2}] \Longrightarrow \mathsf{DEP_e}(\epsilon_{ih1}, \epsilon_{ih2}, \tau[\epsilon_{ih1} : \epsilon_{ih2}], c, D)\\ &\quad\Longrightarrow \exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c . n \geq 0 \wedge \texttt{flowsTo}(\pi_1(\epsilon_{ih1})^{\pi_2(\epsilon_{ih1})}, z_1^{r_1}, c) \wedge \cdots \wedge \texttt{flowsTo}(z_n^{r_n}, \pi_1(\epsilon_{ih2})^{\pi_2(\epsilon_{ih2})}, c)\end{aligned}$$

Taking arbitrary $\epsilon_1, \epsilon_2 \in \mathcal{E}^{\text{asn}}$, by law of excluded middle, there are 2 cases:
$\epsilon_1 = \epsilon'_1 \wedge \epsilon_2 = \epsilon'_2$
$\neg(\epsilon_1 = \epsilon'_1 \wedge \epsilon_2 = \epsilon'_2)$
In case of $\neg(\epsilon_1 = \epsilon'_1 \wedge \epsilon_2 = \epsilon'_2)$, since $\nexists \tau' \in \mathcal{T}$,satisfies $[\epsilon'_1]{++}\tau_{ih}{++}[\epsilon'_2] = [\epsilon_1]{++}\tau'{++}[\epsilon_2]$, the theorem is vacuously true.
In case of $\epsilon_1 = \epsilon'_1 \wedge \epsilon_2 = \epsilon'_2$, let $\tau' = \tau_{ih}$, we know $\exists \tau' \in \mathcal{T}$ satisfying $[\epsilon'_1]{++}\tau_{ih}{++}[\epsilon'_2] = [\epsilon_1]{++}\tau'{++}[\epsilon_2]$.
To show:

$$\begin{aligned}&\mathsf{DEP_e}(\epsilon_1, \epsilon_2, [\epsilon_1]{++}\tau_{ih}{++}[\epsilon_2], c, D)\\ &\quad\Longrightarrow \exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c . n \geq 0 \wedge \texttt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, z_1^{r_1}, c) \wedge \cdots \wedge \texttt{flowsTo}(z_n^{r_n}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)\end{aligned}$$

By Lemma C.3, we know:

$$
\begin{aligned}
&\texttt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)\\
&\qquad \vee \exists \epsilon \in \tau_{ih} . \mathsf{DEP_e}(\epsilon_1, \epsilon, \tau[\epsilon_1 : \epsilon], c, D) \wedge \texttt{flowsTo}(\pi_1(\epsilon)^{\pi_2(\epsilon)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)
\end{aligned}
$$

In first case, we have $\texttt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)$ proved directly.
In the second case, let ϵ_{ih} be this event, from the induction hypothesis, we know:

$$
\exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c . n \geq 0 \wedge \texttt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, z_1^{r_1}, c) \wedge \cdots \wedge \texttt{flowsTo}(z_n^{r_n}, \pi_1(\epsilon_{ih})^{\pi_2(\epsilon_{ih})}, c)
$$

Then we know:

$$
\begin{aligned}
&\exists n \in \mathbb{N}, z_1^{r_1}, \cdots, z_n^{r_n} \in \mathbb{LV}_c . n \geq 0 \wedge \texttt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, z_1^{r_1}, c) \wedge \cdots \wedge \texttt{flowsTo}(z_n^{r_n}, \pi_1(\epsilon_{ih})^{\pi_2(\epsilon_{ih})}, c))\\
&\quad \wedge \texttt{flowsTo}(\pi_1(\epsilon)^{\pi_2(\epsilon)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)
\end{aligned}
$$

This case is proved.

□

C.3 Inversion Lemmas and Helper Lemmas

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

Lemma C.1 (The One-Step Event Dependency Inversion)**.** *For every $c \in \mathcal{C}, D \in \mathcal{DB}$ and two assignment events $\epsilon_1, \epsilon_2 \in \mathcal{E}^{\text{asn}}$, if* $\mathsf{DEP_e}(\epsilon_1, \epsilon_2, [\epsilon_1; \epsilon_2], c, D)$*, then,* $\texttt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)$.

$$
\begin{aligned}
&\forall \epsilon_1, \epsilon_2 \in \mathcal{E}^{\text{asn}}, c \in \mathcal{C}, D \in \mathcal{DB} . \mathsf{DEP_e}(\epsilon_1, \epsilon_2, [\epsilon_1; \epsilon_2], c, D)\\
&\quad \implies \texttt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)
\end{aligned}
$$

Proof Summary:
1. case of (the labelled unique assignment command associated to the ϵ_2 is executed but the value assigned to the variable in this event is changed in second execution)
show x directly used by the assignment of the second event
2.(the labelled unique assignment command associated to the ϵ_2 isn't executed in second execution)
show x is directly used by the boolean expression for a conditional command and second event shows in the body of that conditional command

Proof. By the Definition 10 for $\mathsf{DEP_e}(\epsilon_1, \epsilon_2, [\epsilon_1; \epsilon_2], c, D)$, we know there are 2 cases:

case: 1
(the labelled unique assignment command associated to the ϵ_2 is executed but the value assigned to the variable in this event is changed in second execution).

Proof of the Basecase: Case 1. We have the following by the definition $\mathsf{DEP_e}(\epsilon_1, \epsilon_2, [\epsilon_1; \epsilon_2], c, D)$ for case 1:

$$
\exists \tau_0, \tau_1, \tau' \in \mathcal{T}, \epsilon_1' \in \mathcal{E}^{\text{asn}}, \epsilon_2' \in \mathcal{E}, c_1, c_2 \in \mathcal{C} . \texttt{Diff}(\epsilon_1, \epsilon_1') \wedge \left(\begin{array}{ll} & \langle c, \tau_0 \rangle \to^* \langle c_1, \tau_1 +\!\!+ [\epsilon_1] \rangle \to^* \langle c_2, \tau_1 +\!\!+ [\epsilon_1; \epsilon_2] \rangle \\ \wedge & \langle c_1, \tau_1 +\!\!+ [\epsilon_1'] \rangle \to^* \langle c_2, \tau_1 +\!\!+ [\epsilon_1'] +\!\!+ \tau' +\!\!+ [\epsilon_2'] \rangle \\ \wedge & \texttt{Diff}(\epsilon_2, \epsilon_2') \wedge \texttt{cnt}(\tau)\ \pi_2(\epsilon_2) = \texttt{cnt}(\tau')\ \pi_2(\epsilon_2) \end{array} \right) \tag{4}
$$

Let $\tau_0, \tau_1, \tau' \in \mathcal{T}, \epsilon'_2 \in \mathcal{E}, \epsilon'_1 \in \mathcal{E}^{\text{asn}}, c_1, c_2$ be the traces, events and commands satisfying the executions, by Inversion Lemma C.7 on ϵ_1, ϵ_2, we have the following instance of the first execution in Eq. 4,

$$\begin{aligned}\langle c, \tau_0 \rangle &\to^* \langle [x_1 \leftarrow e_1/\texttt{query}(\psi_1)]^{\pi_2(\epsilon_1)}; c_1, \tau_1 \rangle \to^{\textsf{assn/query}} \langle c_1, \tau_1{}_{++}[\epsilon_1] \rangle \quad {}^2 \\ &\to^* \langle [x_2 \leftarrow e_2/\texttt{query}(\psi_2)]^{l_2}; c_2, \tau_1{}_{++}[\epsilon_1] \rangle \to^{\textsf{assn/query}} \langle c_2, \tau_1{}_{++}[\epsilon_1; \epsilon_2] \rangle\end{aligned} \tag{5}$$

, where $x_1 = \pi_1(\epsilon_1)$, $l_1 = \pi_2(\epsilon_1)$, $x_2 = \pi_1(\epsilon_2)$, $l_2 = \pi_2(\epsilon_2)$, and e_1/ψ_1, e_2/ψ_2 are the expressions of the assignment commands associated to the ϵ_1 and ϵ_2 from Lemma C.7.
By $\texttt{Diff}(\epsilon_2, \epsilon'_2)$ and the command label consistency, we also have the instance of second execution in Eq. 4 as follows:

$$\langle c_1, \tau_1{}_{++}[\epsilon'_1] \rangle \to^* \langle [x_2 \leftarrow e_2/\texttt{query}(\psi_2)]^{l_2}; c_2, \tau_1{}_{++}[\epsilon'_1] \cdot \tau'_2 \rangle \to^{\textsf{assn/query}} \langle c_2, \tau_1{}_{++}[\epsilon'_1] \cdot \tau'_2{}_{++}[\epsilon'_2] \rangle \tag{6}$$

From Eq. 4, we also have

$$\texttt{cnt}(\tau') l_2 = \texttt{cnt}([]) l_2 = 0 \tag{7}$$

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

$$c_1 =_c [\texttt{skip}]^*; [x_2 \leftarrow e_2/\texttt{query}(\psi_2)]^{l_2}; c_2{}^3$$

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

Subproof. There are two possibilities by the law of excluded middle:
$[x_2 \leftarrow e_2/\texttt{query}(\psi_2)]^{l_2} \in_c c_2$
or $[x_2 \leftarrow e_2/\texttt{query}(\psi_2)]^{l_2} \notin_c c_2$.

1. $[x_2 \leftarrow e_2/\texttt{query}(\psi_2)]^{l_2} \notin_c c_2$
   In this case, we have the following execution instance: [4]

$$\langle c_1, \tau_1{}_{++}[\epsilon'_1] \rangle \to^{\textsf{skip}^*} \langle [x_2 \leftarrow e_2/\texttt{query}(\psi_2)]^{l_2}; c_2, \tau_1{}_{++}[\epsilon'_1] \rangle \to^{\textsf{assn/query}} \langle c_2, \tau_0{}_{++}\tau_1{}_{++}[\epsilon'_1; \epsilon'_2] \rangle$$

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

$$\langle c_1, \tau_1{}_{++}[\epsilon'_1] \rangle \to^{\textsf{skip}^*} \langle [x_2 \leftarrow e_2/\texttt{query}(\psi_2)]^{l_2}; c_2, \tau_1{}_{++}[\epsilon'_1] \rangle \to^{\textsf{assn/query}} \langle c_2, \tau_1{}_{++}[\epsilon'_1]{}_{++}[\epsilon'_2] \rangle$$

$$\begin{aligned}\langle c_1, \tau_1{}_{++}[\epsilon'_1] \rangle &\to^{\textsf{skip}^*} \langle [x_2 \leftarrow e_2/\texttt{query}(\psi_2)]^{l_2}; c_2, \tau_1{}_{++}[\epsilon'_1] \rangle \to^{\textsf{assn/query}} \langle c_2, \tau_1{}_{++}[\epsilon'_1]{}_{++}[(x_2, l_2, v'_2)] \rangle \\ &\to^* \langle [x_2 \leftarrow e_2/\texttt{query}(\psi_2)]^{l_2}; c_2, \tau_1{}_{++}[\epsilon'_1]{}_{++}[(x_2, l_2, v'_2)]{}_{++}\tau_3 \rangle \\ &\to^{\textsf{assn/query}} \langle c_2, \tau_1{}_{++}[\epsilon'_1]{}_{++}[(x_2, l_2, v'_2)]{}_{++}\tau_3{}_{++}[\epsilon'_2] \rangle \\ &\qquad \cdots\end{aligned}$$

   , where each execution instance iterates the conditional command $(\texttt{while}\, [b_w]^l_w\, \texttt{do}\, c_w)$ in c_2, $0, 1$ or more times.
   For each execution instance, we have the corresponding instance of τ' as follows:

[2] $x \leftarrow e/\texttt{query}(\psi)$ denotes variable x is assigned by either an expression e or query $\texttt{query}(\psi)$
[3] $([\texttt{skip}];)^*$ denotes a sequence command only composed of $[\texttt{skip}]$ commands.
[4] $\to^{\textsf{skip}^*}$ denotes the rule applied on every evaluation step of this execution is the skip rule.

$\tau' = []$
$\tau' = [(x_2, l_2, v_2')]_{++}\tau_3$
$\cdots$
By Eq. 7 where $\mathtt{cnt}(\tau')l_2 = 0$, we know only the first execution instance with 0 iteration of `while` command in c_2 satisfies this restriction, i.e., $\tau' = []$.

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

$$\langle c_1, \tau_{1++}[\epsilon_1']\rangle \rightarrow^{\mathsf{skip}^*} \langle [x_2 \leftarrow e_2/\mathtt{query}(\psi_2)]^{l_2}; c_2, \tau_{1++}[\epsilon_1']\rangle \rightarrow^{\mathsf{assn/query}} \langle c_2, \tau_{1++}[\epsilon_1']_{++}[\epsilon_2']\rangle$$

$\blacksquare$

Then we know by the environment definition, ρ obtains different values only for variable x_1 from trace $\tau_{1++}[\epsilon_1]$ and $\tau_{1++}[\epsilon_1']$, i.e.,

$$\forall z^r \in \mathbb{LV}_c \setminus \{x_1^{l_1}\}, \rho(\tau_{1++}[\epsilon_1])(z) = \rho(\tau_{1++}[\epsilon_1'])(z)$$

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

$$x_1 \in VAR(e_2/\psi_2)$$

Since $\iota(\tau_{1++}[\epsilon_1])x_1 = l_1$, by Inversion Lemma C.8 we know $x_1^{l_1} \in \mathtt{RD}(l_2, c)$.
By `flowsTo` definition, we have:

$$\mathtt{flowsTo}(x_1^{l_1}, x_2^{l_2}, c)$$

i.e.,

$$\mathtt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)$$

This case is proved. $\blacksquare$

case: 2
(the labelled unique assignment command associated to the ϵ_2 isn't executed in second execution).
Proof Summary:
1. Let ϵ_b be the testing event, in the same way of case 1, we get: $\pi_1(\epsilon_1) \in VAR(\pi_1(\epsilon_b)) \land \pi_1(\epsilon_1)^{l_1} \in \mathtt{RD}(l_b, c)$
2. By Lemma C.2, we know: $\forall z \in VAR(\pi_1(\epsilon_b))\ .\ \exists i \in \mathbb{N}\ .\ \mathtt{flowsTo}(z^i, \pi_1(\epsilon)^{\pi_2(\epsilon)}, c)$
3. By `flowsTo` definition we have: $\mathtt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)$

Proof of the Basecase: Case 2. We have the following by the definition $\mathsf{DEP}_{\mathsf{e}}(\epsilon_1, \epsilon_2, [\epsilon_1; \epsilon_2], c, D)$ of case 2:

$$\begin{aligned}
&\exists \tau_0, \tau_1, \tau', \tau_3, \tau_3' \in \mathcal{T}, \epsilon_1' \in \mathcal{E}^{\mathtt{asn}}, c_1, c_2 \in \mathcal{C}, \epsilon_b \in \mathcal{E}^{\mathtt{test}}\ . \\
&\quad \mathtt{Diff}(\epsilon_1, \epsilon_1') \land \Big(\langle c, \tau_0\rangle \rightarrow^* \langle c_1, \tau_{1++}[\epsilon_1]\rangle \rightarrow^* \langle c_2, \tau_{1++}[\epsilon_1; \epsilon_b]_{++}\tau_3\rangle \\
&\quad \bigwedge \langle c_1, \tau_{1++}[\epsilon_1']\rangle \rightarrow^* \langle c_2, \tau_{1++}[\epsilon_1']_{++}\tau'_{++}[(\neg\epsilon_b)]_{++}\tau_3'\rangle \\
&\quad \bigwedge \mathbb{TL}_{\tau_3} \cap \mathbb{TL}_{\tau_3'} = \emptyset \land \mathtt{cnt}(\tau')\ \pi_2(\epsilon_b) = \mathtt{cnt}(\tau)\ \pi_2(\epsilon_b) \land \epsilon_2 \in_{\mathsf{e}} \tau_3 \land \epsilon_2 \notin_{\mathsf{e}} \tau_3'\Big)
\end{aligned} \tag{8}$$

Let $\tau_0, \tau_1, \tau', \tau_3, \tau_3' \in \mathcal{T}, \epsilon_2' \in \mathcal{E}, \epsilon_1' \in \mathcal{E}^{\mathtt{asn}}, \epsilon_b, c_1, c_2$ be the traces, events and commands satisfying the executions, by Inversion Lemma C.7 on ϵ_1, ϵ_2, and ϵ_b, we have the following instance of the first execution in Eq. 8,

$$
\begin{aligned}
&\langle c, \tau_0 \rangle \to^* \langle [x_1 \leftarrow e_1 / \mathtt{query}(\psi_1)]^{l_1}; c_1, \tau_1 \rangle \to^{assn/query} \langle c_1, \tau_1 +\!\!+ [\epsilon_1] \rangle \\
&\quad \to^* \langle\, \mathtt{if}\ ([b]^{l_b}, c_t, c_f) / \mathtt{while}\ [b]^{l_b}\ \mathtt{do}\ c_w; c_3', \tau_1 +\!\!+ [\epsilon_1] \rangle \\
&\quad \to^{\text{if-b / while-b}} \langle (c_t; c_3' / c_f; c_3') / (c_3' / c_w;\ \mathtt{while}\ [b]^{l_b}\ \mathtt{do}\ c_w; c_3'), \tau_1 +\!\!+ [\epsilon_1; \epsilon_b] \rangle \\
&\quad \to^* \langle c_3, \tau_1 +\!\!+ [\epsilon_1; \epsilon_b] +\!\!+ \tau_3 \rangle
\end{aligned}
\tag{9}
$$

, where $x_1 = \pi_1(\epsilon_1)$, $l_1 = \pi_2(\epsilon_1)$, and $\mathtt{if}\ ([b]^{l_b}, c_t, c_f) / \mathtt{while}\ [b]^{l_b}\ \mathtt{do}\ c_w$ is the conditional command of the assignment commands associated to the ϵ_b from Inversion Lemma C.7 of testing event.
By the command label consistency, we also have the instance of second execution in Eq. 8 as follows:

$$
\begin{aligned}
&\langle c, \tau_0 \rangle \to^* \langle [x_1 \leftarrow e_1 / \mathtt{query}(\psi_1)]^{l_1}; c_1, \tau_1 \rangle \to^{assn/query} \langle c_1, \tau_1 +\!\!+ [\epsilon_1] \rangle \\
&\quad \to^* \langle\, \mathtt{if}\ ([b]^{l_b}, c_t, c_f) / \mathtt{while}\ [b]^{l_b}\ \mathtt{do}\ c_w; c_3', \tau_1 +\!\!+ [\epsilon_1] +\!\!+ \tau' \rangle \\
&\quad \to^{\text{if-b / while-b}} \langle (c_f; c_3' / c_t; c_3') / (c_w;\ \mathtt{while}\ [b]^{l_b}\ \mathtt{do}\ c_w; c_3' / c_3'), \tau_1 +\!\!+ [\epsilon_1] +\!\!+ \tau' +\!\!+ [\neg\epsilon_b] \rangle \\
&\quad \to^* \langle c_3, \tau_1 +\!\!+ [\epsilon_1] +\!\!+ \tau' +\!\!+ [\neg\epsilon_b] +\!\!+ \tau_3' \rangle
\end{aligned}
\tag{10}
$$

From Eq. 8, we also have $\mathtt{cnt}(\tau')\, l_b = \mathtt{cnt}([])\, l_b = 0$.
By the same proof steps from case 1 in Subproof C.3, we have

$$x_1 \in VAR(b) \wedge x_1^{l_1} \in \mathtt{RD}(l_b, c)$$

By Lemma C.2, we also know:

$$\forall z \in VAR(\pi_1(\epsilon_b)) \,.\, \exists i \in \mathbb{N} \,.\, \mathtt{flowsTo}(z^i, \pi_1(\epsilon)^{\pi_2(\epsilon)}, c)$$

Then by $\mathtt{flowsTo}$ definition, we have $\mathtt{flowsTo}(x_1^{l_1}, x_2^{l_2}, c)$ i.e.,

$$\mathtt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)$$

This case is proved. ■

□

Lemma C.2 (Control Dependency Inversion)**.** *For every $c \in \mathcal{C}$, $D \in \mathcal{DB}, \tau \in \mathcal{T}$ and two assignment events $\epsilon_1, \epsilon_2 \in \mathcal{E}^{\mathsf{asn}}$, if they are in the second case of the* Event May-Dependency *relation from Definition. 10,* $\mathsf{DEP_e}(\epsilon, \epsilon, c, \tau, D)$ *as Eq. 11, then for all $z \in VAR(\pi_1(\epsilon_b))$ there exists a label $i \in \mathbb{N}$ such that* $\mathtt{flowsTo}(z^i, \pi_1(\epsilon)^{\pi_2(\epsilon)}, c)$

$$
\begin{aligned}
&\forall D \in \mathcal{DB}, c \in \mathcal{C}, \tau \in \mathcal{T}, \epsilon_1, \epsilon_2 \in \mathcal{E}^{\mathsf{asn}} \,. \\
&\exists \tau_0, \tau_1, \tau', \tau_3, \tau_3' \in \mathcal{T}, \epsilon_1' \in \mathcal{E}^{\mathsf{asn}}, c_1, c_2 \in \mathcal{C}, \epsilon_b \in \mathcal{E}^{\mathtt{test}}, \tau_{ih} \in \mathcal{T} \,.\, \tau = [\epsilon_1] +\!\!+ \tau_{ih} +\!\!+ [\epsilon_2] \\
&\quad \Longrightarrow \langle c, \tau_0 \rangle \to^* \langle c_1, \tau_1 +\!\!+ [\epsilon_1] \rangle \to^* \langle c_2, \tau_1 +\!\!+ [\epsilon_1] +\!\!+ \tau +\!\!+ [\epsilon_b] +\!\!+ \tau_3 \rangle \\
,&\quad\quad \wedge \langle c_1, \tau_1 +\!\!+ [\epsilon_1'] \rangle \to^* \langle c_2, \tau_1 +\!\!+ [\epsilon_1'] +\!\!+ \tau' +\!\!+ [(\neg\epsilon_b)] +\!\!+ \tau_3' \rangle \\
&\quad\quad \wedge \mathbb{TL}_{\tau_3} \cap \mathbb{TL}_{\tau_3'} = \emptyset \wedge \mathtt{cnt}(\tau')\ \pi_2(\epsilon_b) = \mathtt{cnt}(\tau)\ \pi_2(\epsilon_b) \wedge \epsilon_2 \in_{\mathsf{e}} \tau_3 \wedge \epsilon_2 \notin_{\mathsf{e}} \tau_3' \\
&\quad \Longrightarrow \forall z \in VAR(\pi_1(\epsilon_b)) \,.\, \exists l \in \mathbb{N} \,.\, \mathtt{flowsTo}(z^l, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)
\end{aligned}
\tag{11}
$$

Proof Summary:
Proving by using the Inversion Lemmas C.5, C.6, C.7, and C.8, and the *Event May-Dependency* definition of the second case.

Proof. Take arbitrary $D \in \mathcal{DB}, c \in \mathcal{C}, \tau \in \mathcal{T}, \epsilon_1, \epsilon_2 \in \mathcal{E}^{\text{asn}}$, let $\tau_0, \tau_1, \tau', \tau_3, \tau'_3 \in \mathcal{T}, \epsilon'_2 \in \mathcal{E}, \epsilon'_1 \in \mathcal{E}^{\text{asn}}, \epsilon_b, c_1, c_2$ be the traces, events and commands satisfying the executions, by Inversion Lemma C.7 on ϵ_2, and ϵ_b, we have the following instance of the first execution in Eq. 11,

$$\begin{aligned}&\langle c, \tau_0\rangle \to^* \langle\, \texttt{if}\ ([b]^{l_b}, c_t, c_f) / \texttt{while}\ [b]^{l_b}\ \texttt{do}\ c_w; c'_3, \tau_1{++}[\epsilon_1]{++}\tau\rangle\\ &\quad \to^{\text{if-b / while-b}} \langle (c_t; c'_3 / c_f; c'_3) / (c_w; \texttt{while}\ [b]^{l_b}\ \texttt{do}\ c_w; c'_3 / [\texttt{skip}]; c'_3), \tau_1{++}[\epsilon_1]{++}\tau{++}[\epsilon_b]\rangle\\ &\quad \to^* \langle [x_2 \leftarrow e_2 / \texttt{query}(\psi_2)]^{l_2}; c'_{3b}, \tau_1{++}[\epsilon_1]{++}\tau{++}[\epsilon_b]{++}\tau_{3a}\rangle\\ &\quad \to^{\text{assn/query}} \langle c'_{3b}, \tau_1{++}[\epsilon_1]{++}\tau{++}[\epsilon_b]{++}\tau_{3a}{++}[\epsilon_2]\rangle \to^* \langle c_3, \tau_1{++}[\epsilon_1]{++}\tau{++}[\epsilon_b]{++}\tau_{3a}{++}[\epsilon_2]{++}\tau_{3b}\rangle\end{aligned} \tag{12}$$

, where $\tau_3 = \tau_{3a}{++}[\epsilon_2]{++}\tau_{3b}$, $x_2 = \pi_1(\epsilon_2)$, $l_2 = \pi_2(\epsilon_2)$, and $\texttt{if}\ ([b]^{l_b}, c_t, c_f) / \texttt{while}\ [b]^{l_b}\ \texttt{do}\ c_w$ is the conditional command of the assignment commands associated to the ϵ_b from Inversion Lemma C.7 of testing event.

The notation $(c_t; c'_3 / c_f; c'_3) / (c_w; \texttt{while}\ [b]^{l_b}\ \texttt{do}\ c_w; c'_3 / [\texttt{skip}]; c'_3)$ represents:

In case of $\texttt{if}\ ([b]^{l_b}, c_t, c_f)$, if $\pi_3(\epsilon_b) = \texttt{true}$, we have the evaluation:

$$\langle\, \texttt{if}\ ([b]^{l_b}, c_t, c_f); c'_3, \tau_1{++}[\epsilon_1]{++}\tau\rangle \to^{\text{if-b}} \langle c_t; c'_3 \tau_1{++}[\epsilon_1]{++}\tau{++}[\epsilon_b]\rangle$$

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 the command label consistency, we also have the instance of second execution as follows:

$$\begin{aligned}&\langle c, \tau_0\rangle \to^* \langle\, \texttt{if}\ ([b]^{l_b}, c_t, c_f) / \texttt{while}\ [b]^{l_b}\ \texttt{do}\ c_w; c'_3, \tau_1{++}[\epsilon_1]{++}\tau'\rangle\\ &\quad \to^{\text{if-b / while-b}} \langle (c_f; c'_3 / c_t; c'_3) / ([\texttt{skip}]; c'_3 / c_w; \texttt{while}\ [b]^{l_b}\ \texttt{do}\ c_w; c'_3), \tau_1{++}[\epsilon_1]{++}\tau'{++}[\neg\epsilon_b]\rangle\\ &\quad \to^* \langle c_3, \tau_1{++}[\epsilon_1]{++}\tau'{++}[\neg\epsilon_b]{++}\tau'_3\rangle\end{aligned} \tag{13}$$

By the label consistency, and $\mathbb{TL}_{\tau_3} \cap \mathbb{TL}_{\tau'_3} = \emptyset$, we know τ_3 and τ'_3 doesn't contain any event of evaluating the commands in c'_3. Otherwise, $\mathbb{TL}_{\tau_3} \cap \mathbb{TL}_{\tau'_3} \neq \emptyset$, which is a contradiction.

Since $\tau_3 = \tau_{3a}{++}[\epsilon_2]{++}\tau_{3b}$, we know ϵ_2 doesn't comes from evaluating of c'_3, i.e.,:

In the case of $\texttt{if}\ ([b]^{l_b}, c_t, c_f)$, ϵ_2 comes from the evaluation of c_t or c_f, i.e., $[x_2 \leftarrow e_2 / \texttt{query}(\psi_2)]^{l_2} \in_c c_t$ or c_f;

and in the case of $\texttt{while}\ [b]^{l_b}\ \texttt{do}\ c_w$, ϵ_2 comes from the evaluation of c_w, i.e., $[x_2 \leftarrow e_2 / \texttt{query}(\psi_2)]^{l_2} \in_c c_w$.

In both of the two cases, we know $\forall z \in VAR(\pi_1(\epsilon_b))$ there is a label $l \in \mathbb{N}$ for this variable, and by the $\texttt{flowsTo}$ definition, $\texttt{flowsTo}(z^l, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)$.

This lemma is proved. □

Lemma C.3 (The Multiple-Steps Event Dependency Inversion). *For every $D \in \mathcal{DB}, c \in \mathcal{C}, \tau \in \mathcal{T}$, and two assignment events $\epsilon_1, \epsilon_2 \in \mathcal{E}^{\text{asn}}$, if the trace $trace$ has the form $\tau = [\epsilon_1]{++}\tau'{++}[\epsilon_2]$ with $\tau' \in \mathcal{T}$, and $\mathsf{DEP_e}(\epsilon_1, \epsilon_2, \tau, c, D)$ then $\texttt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)$, or otherwise there exists $\epsilon \in \tau'$ such that $\big(\mathsf{DEP_e}(\epsilon_1, \epsilon, \tau[\epsilon_1 : \epsilon], c, D) \land \texttt{flowsTo}(\pi_1(\epsilon)^{\pi_2(\epsilon)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)\big)$.*

$$\begin{aligned}&\forall D \in \mathcal{DB}, c \in \mathcal{C}, \tau \in \mathcal{T}\ .\ \forall \epsilon_1, \epsilon_2 \in \mathcal{E}^{\text{asn}}\ .\ \exists \tau' \in \mathcal{T}\ .\ \tau = [\epsilon_1]{++}\tau'{++}[\epsilon_2] \implies \mathsf{DEP_e}(\epsilon_1, \epsilon_2, \tau, c, D)\\ &\quad \implies \texttt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)\\ &\qquad \lor \exists \epsilon \in \tau'\ .\ \big(\mathsf{DEP_e}(\epsilon_1, \epsilon, \tau[\epsilon_1 : \epsilon], c, D) \land \texttt{flowsTo}(\pi_1(\epsilon)^{\pi_2(\epsilon)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)\big)\end{aligned}$$

Proof Summary:

Proving by using Lemma C.4, Lemma C.2, and the Inversion Lemmas C.5, C.6, C.7, and C.8 and showing a contradiction.

Proof. Taking arbitrary $D \in \mathcal{DB}, c \in \mathcal{C}, \tau \in \mathcal{T}$ and two events $\epsilon_1, \epsilon_2 \in \mathcal{E}^{\mathsf{asn}}$, where τ has the form $\tau = [\epsilon_1] ++ \tau' ++ [\epsilon_2]$ for some $\tau' \in \mathcal{T}$ and $\mathsf{DEP_e}(\epsilon_1, \epsilon_2, \tau, c, D)$
Assume

$$\neg \mathtt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c) \text{ (1)} \\ \wedge \forall \epsilon \in \tau' . \left(\neg \mathsf{DEP_e}(\epsilon_1, \epsilon, \tau[\epsilon_1 : \epsilon], c, D) \vee \neg \mathtt{flowsTo}(\pi_1(\epsilon)^{\pi_2(\epsilon)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)\right) \text{ (2)}$$

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

$$\mathtt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)$$

, which is contradict to (1).
This Lemma is proved. □

Lemma C.4 (Independent Events Doesn't Block `flowsTo`). *For every $D \in \mathcal{DB}, c \in \mathcal{C}, \tau \in \mathcal{T}$, one assignment events $\epsilon_1 \in \mathcal{E}^{\mathsf{asn}}$, and another event $\epsilon_2 \in \mathcal{E}$, if the trace τ has the form $\tau = [\epsilon_1] ++ \tau' ++ [\epsilon_2]$ with $\tau' \in \mathcal{T}$, and $\mathsf{DEP_e}(\epsilon_1, \epsilon_2, \tau, c, D)$, then the following two conclusions hold when ϵ_2 is an assignment event and a testing event respectively.*

- *If $\epsilon_2 \in \mathcal{E}^{\mathsf{asn}}$, then for every $\epsilon \in \tau'$, if it either doesn't have the* Event May-Dependency *relation on ϵ_1, or $\pi_1(\epsilon)^{\pi_2(\epsilon)}$ doesn't have the* `flowsTo` *relation with $\pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}$, then the labelled variable $\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}$ directly flows to the other one $\pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}$, i.e.,* $\mathtt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)$.

$$\forall D \in \mathcal{DB}, c \in \mathcal{C}, \tau \in \mathcal{T} . \forall \epsilon_1, \epsilon_2 \in \mathcal{E}^{\mathsf{asn}} . \exists \tau' \in \mathcal{T} . \tau = [\epsilon_1] ++ \tau' ++ [\epsilon_2] \implies \mathsf{DEP_e}(\epsilon_1, \epsilon_2, \tau, c, D) \\ \implies \left(\forall \epsilon \in \tau' . \neg \mathsf{DEP_e}(\epsilon_1, \epsilon, \tau[\epsilon_1 : \epsilon], c, D) \vee \neg \mathtt{flowsTo}(\pi_1(\epsilon)^{\pi_2(\epsilon)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)\right) \\ \implies \mathtt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, \pi_1(\epsilon_2)^{\pi_2(\epsilon_2)}, c)$$

- *If $\epsilon_2 \in \mathcal{E}^{\mathsf{test}}$, then for every $\epsilon \in \tau'$, if it either doesn't have the* Event May-Dependency *relations on ϵ_1, or $\pi_1(\epsilon) \notin VAR(\pi_1(\epsilon_2))$, then $\pi_1(\epsilon_1) \in VAR(\pi_1(\epsilon_2))$, and $\pi_2(\epsilon_1) = \iota(\tau)$*

$$\forall D \in \mathcal{DB}, c \in \mathcal{C}, \tau \in \mathcal{T} . \forall \epsilon_1, \in \mathcal{E}^{\mathsf{asn}}, \epsilon_2 \in \mathcal{E}^{\mathsf{test}} . \exists \tau' \in \mathcal{T} . \tau = [\epsilon_1] ++ \tau' ++ [\epsilon_2] \implies \mathsf{DEP_e}(\epsilon_1, \epsilon_2, \tau, c, D) \\ \implies \left(\forall \epsilon \in \tau' . \neg \mathsf{DEP_e}(\epsilon_1, \epsilon, \tau[\epsilon_1 : \epsilon], c, D) \vee \pi_1(\epsilon) \notin VAR(\pi_1(\epsilon_2))\right) \\ \implies \pi_1(\epsilon_1) \in VAR(\pi_1(\epsilon_2)) \wedge \pi_2(\epsilon_1) = \iota(\tau)$$

Proof. Taking arbitrary $D \in \mathcal{DB}, c \in \mathcal{C}$, and an assignment events $\epsilon_1 \in \mathcal{E}^{\mathsf{asn}}$ and another event $\epsilon_2 \in \mathcal{E}$. Without loss of generalization, taking arbitrary trace has the form $\tau = [\epsilon_1; \cdots; \epsilon_2]$ for arbitrary $\tau_2 \in \mathcal{T}$, then we know $\exists \tau' \in \mathcal{T} . \tau = [\epsilon_1] ++ \tau' ++ [\epsilon_2]$, let τ_2 be this τ'.

case: $\epsilon_2 \in \mathcal{E}^{\mathsf{asn}}$
By the definition of $\mathsf{DEP_e}(\epsilon_1, \epsilon_2, \tau, c, D)$, taking $\epsilon_1', \epsilon_2' \in \mathcal{E}^{\mathsf{asn}}, \tau_2' \in \mathcal{T}, c_1, c_2 \in \mathcal{C}$ as the events, traces and commands satisfying the definition, we have following two executions:

$$\langle c, \tau_0 \rangle \to^* \langle c_1, \tau_1 ++ [\epsilon_1] \rangle \to^* \langle c_2, \tau_1 ++ [\epsilon_1] ++ \tau_2 ++ [\epsilon_2] \rangle \\ \langle c_1, \tau_1 ++ [\epsilon_1'] \rangle \to^* \langle c_2, \tau_1 ++ [\epsilon_1'] ++ \tau_2' ++ [\epsilon_2'] \rangle$$

By inversion Lemma. C.7 on ϵ_2 and ϵ_2' in the two executions and $\mathtt{Diff}(\epsilon_2, \epsilon_2)$, we have the following two execution instances:

$$\langle c_1, \tau_1 ++ [\epsilon_1] \rangle \to^* \langle [\pi_1(\epsilon_2) \leftarrow e_2 / \mathtt{query}(\psi_2)]^{\pi_2(\epsilon_2)}; c_2, \tau_1 ++ [\epsilon_1] ++ \tau_2 \rangle \to^{\textsf{asn / query}} \langle c_2, \tau_1 ++ [\epsilon_1] ++ \tau_2 ++ [\epsilon_2] \rangle$$

$$\langle c_1, \tau_1 ++ [\epsilon_1'] \rangle \to^* \langle [\pi_1(\epsilon_2) \leftarrow e_2 / \mathtt{query}(\psi_2)]^{\pi_2(\epsilon_2)}; c_2, \tau_1 ++ [\epsilon_1'] ++ \tau_2' \rangle \to^{\textsf{asn / query}} \langle c_2, \tau_1 ++ [\epsilon_1'] ++ \tau_2' ++ [\epsilon_2'] \rangle$$

, where e_2/ψ_2 is the expression of the assignment command associated to the ϵ_2 and ϵ_2' by the Inversion Lemma. C.7.
Taking arbitrary $\epsilon_z \in \tau_2$, we know $\neg\mathsf{DEP}_{\mathsf{e}}(\epsilon_1, \epsilon, \tau[\epsilon_1 : \epsilon_z], c, D) \vee \pi_1(\epsilon_z) \notin VAR(e_2/\psi_2)$.
In case of $\neg\mathsf{DEP}_{\mathsf{e}}(\epsilon_1, \epsilon, \tau[\epsilon_1 : \epsilon_z], c, D)$, by Definition 10, we know $\epsilon_z \in \tau_2'$ and

$$\rho(\tau_1{+\!+}\tau[\epsilon_1 : \epsilon_z])\pi_1(\epsilon_z) = \rho(\tau_1{+\!+}\tau[\epsilon_1' : \epsilon_z])\pi_1(\epsilon_z)$$

In case of $\pi_1(\epsilon_z) \notin VAR(e_2/\psi_2)$, by Inversion Lemma C.5 of arithmetic and query expression cases, we know:

$$\forall x^i \in \mathbb{LV}, \tau, \tau' \in \mathcal{T}, v, v' . \left(\forall z^j \in \mathbb{LV}/\{\pi_1(\epsilon_z)^{\pi_2(\epsilon_z)}\} . \rho(\tau)z = \rho(\tau')z\right) \wedge \langle \tau, e_2/\psi_2\rangle \Downarrow_a v \wedge \langle \tau', e_2\rangle \Downarrow_a v' \implies v = v'$$

$$\forall x^i \in \mathbb{LV}, \tau, \tau' \in \mathcal{T}, \alpha, \alpha' . \left(\forall z^j \in \mathbb{LV}/\{\pi_1(\epsilon_z)^{\pi_2(\epsilon_z)}\} . \rho(\tau)z = \rho(\tau')z\right) \wedge \langle \tau, \psi_2\rangle \Downarrow_q \alpha \wedge \langle \tau', \psi_2\rangle \Downarrow_q \alpha' \implies \alpha =_q \alpha'$$

for e_2 or ψ_2 respectively.
Let $\mathtt{use}_{\tau_2}$ a subset of the events in τ_2, satisfying:

$$\forall \epsilon \in \mathcal{E}^{\mathtt{asn}} . \epsilon \in \mathtt{use}_{\tau_2} \Longleftrightarrow \epsilon \in \tau_2 \wedge \pi_1(\epsilon) \in VAR(e_2/\psi_2)$$

Then we also know for every $\epsilon_z \in \mathtt{use}_{\tau_2}$, $\neg\mathsf{DEP}_{\mathsf{e}}(\epsilon_1, \epsilon_z, \tau[\epsilon_1 : \epsilon_z], c, D)$, i.e.,:

$$\forall z^l \in \mathbb{LV} \setminus \left((\mathbb{LV}_{\tau_2} \setminus \mathbb{LV}_{\mathtt{use}_{\tau_2}}) \cup \{\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}\}\right) . \rho(\tau_1{+\!+}[\epsilon_1]{+\!+}\tau_2)z = \rho(\tau_1{+\!+}[\epsilon_1']{+\!+}\tau_2')z \ (1)$$

and
$\forall z^l \in \mathbb{LV} \setminus (\mathbb{LV}_{\tau_2} \setminus \mathbb{LV}_{\mathtt{use}_{\tau_2}}), \tau, \tau' \in \mathcal{T}, v, v' . \rho(\tau)z = \rho(\tau')z \wedge \langle \tau, e_2\rangle \Downarrow_a v \wedge \langle \tau', e_2\rangle \Downarrow_a v' \implies v = v' \ (2a)$;
$\forall z^l \in \mathbb{LV} \setminus (\mathbb{LV}_{\tau_2} \setminus \mathbb{LV}_{\mathtt{use}_{\tau_2}}), \tau, \tau' \in \mathcal{T}, \alpha, \alpha' . \rho(\tau)z = \rho(\tau')z \wedge \langle \tau, \psi_2\rangle \Downarrow_q \alpha \wedge \langle \tau', \psi_2\rangle \Downarrow_q \alpha' \implies \alpha =_q \alpha' \ (2q)$,
where $\mathbb{LV}_{\tau_2}$ and $\mathbb{LV}_{\mathtt{use}_{\tau_2}}$ are the sets of labelled variables of every event in τ_2 and $\mathtt{use}_{\tau_2}$ respectively .
Since $\mathtt{Diff}(\epsilon_2, \epsilon_2')$, we also know:

$$\langle \tau_1{+\!+}[\epsilon_1]{+\!+}\tau_2, e_2\rangle \Downarrow_a \pi_3(\epsilon_2) \wedge \langle \tau_1{+\!+}[\epsilon_1']{+\!+}\tau_2', e_2\rangle \Downarrow_a \pi_3(\epsilon_2') \wedge \pi_3(\epsilon_2) \neq \pi_3(\epsilon_2')$$

We know ϵ_1 is the only cause of the difference in ϵ_y and ϵ_y' when evaluating $[\pi_1(\epsilon_2) \leftarrow e_2/\mathtt{query}(\psi_2)\ ]^{\pi_2(\epsilon_2)}$.
By inversion Lemma. C.6 of arithmetic and query expression cases, given the two traces $\tau_1{+\!+}[\epsilon_1']{+\!+}\tau_2'$ and $\tau_1{+\!+}[\epsilon_1']{+\!+}\tau_2'$ satisfying this lemma by (1), $(2a)$ and $(2q)$, we know

$$\pi_1(\epsilon_1) \in VAR(e_2/\psi_2) \wedge \pi_2(\epsilon_1) = \iota(\tau_1{+\!+}[\epsilon_1]{+\!+}\tau_2)\pi_1(\epsilon_1)$$

By $\mathtt{flowsTo}$ definition:

$$\mathtt{flowsTo}(\pi_1(\epsilon_1)^{\pi_2(\epsilon_1)}, \pi_1(\epsilon_y)^{\pi_2(\epsilon_y)}, c)$$

This case is proved.

case: $\epsilon_2 \in \mathcal{E}^{\mathtt{test}}$

By the definition of $\mathsf{DEP}_{\mathsf{e}}(\epsilon_1, \epsilon_2, \tau, c, D)$, taking $\epsilon_1' \in \mathcal{E}^{\mathtt{asn}}, \tau_2' \in \mathcal{T}, c_1, c_2 \in \mathcal{C}$ and $\epsilon_2' \in \mathcal{E}^{\mathtt{test}}$ as the events, traces and commands satisfying the definition, we have following two executions:

$$\langle c, \tau_0\rangle \to^* \langle c_1, \tau_1{+\!+}[\epsilon_1]\rangle \to^* \langle c_2, \tau_1{+\!+}[\epsilon_1]{+\!+}\tau_2{+\!+}[\epsilon_2]\rangle$$
$$\langle c_1, \tau_1{+\!+}[\epsilon_1']\rangle \to^* \langle c_2, \tau_1{+\!+}[\epsilon_1']{+\!+}\tau_2'{+\!+}[\epsilon_2']\rangle$$

Taking arbitrary $\epsilon_z \in \tau_2$, we know $\neg\mathsf{DEP_e}(\epsilon_1, \epsilon, \tau[\epsilon_1 : \epsilon_z], c, D) \vee \pi_1(\epsilon_z) \notin VAR(e_2/\psi_2)$.
Then by the same proof in **case:** $\epsilon_2 \in \mathcal{E}^{\mathtt{asn}}$, and applying the Inversion Lemma C.5 and C.6 of the boolean expression case, we have:

$$\pi_1(\epsilon_1) \in VAR(\pi_1(\epsilon_2)) \wedge \pi_2(\epsilon_1) = \iota(\tau)$$

This case is proved. $\square$

Lemma C.5 (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. $\square$

Lemma C.6 (Expression Inversion Generalization). *For all subset of the labelled variables $\mathtt{Diff} \subset \mathbb{LV}$, and $x^i \in (\mathbb{LV} \setminus \mathtt{Diff})$, and an expression e, if*

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

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

- *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 $x \in VAR(b) \land i = \iota(\tau)x$*

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

- *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 $x \in VAR(\psi) \land i = \iota(\tau)x$.*

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

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 C.5 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.,
$\forall z^j \in \mathbb{LV} \setminus (\texttt{Diff})\ .\ \rho(\tau)z = \rho(\tau')z$.
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. □

Lemma C.7 (Event Inversion)**.** *For all $c \in \mathcal{C}, \tau_0 \in \mathcal{T}, \epsilon \in \mathcal{E}$ such that $\langle c, \tau_0 \rangle \to^* \langle \texttt{skip}, \tau_0 {+\!+} \tau_1 \rangle$, and $\epsilon \in_{\mathsf{e}} \tau_1$, if*

- *$\epsilon \in \mathcal{E}^{\texttt{asn}}$, then either*
  - *there exists $\tau'_1 \in \mathcal{T}, c' \in \mathcal{C}, e$ such that*

$$\langle c, \tau_0 \rangle \to^* \langle [x \leftarrow e]^l; c', \tau_0 {+\!+} \tau' \rangle \to^{\mathsf{assn}} \langle c', \tau_0 {+\!+} \tau'_1 {+\!+} [\epsilon] \rangle \to^* \langle \texttt{skip}, \tau_0 {+\!+} \tau_1 \rangle$$

  - *or there exists $\tau'_1 \in \mathcal{T}, c' \in \mathcal{C}, \psi$ such that*

$$\langle c, \tau_0 \rangle \to^* \langle [x \leftarrow \texttt{query}(\psi)]^l; c', \tau_0 {+\!+} \tau'_1 \rangle \to^{query} \langle c', \tau_0 {+\!+} \tau'_1 {+\!+} [\epsilon] \rangle \to^* \langle \texttt{skip}, \tau_0 {+\!+} \tau_1 \rangle$$

- *$\epsilon \in \mathcal{E}^{\texttt{test}}$ then either*
  - *there exists $\tau'_1 \in \mathcal{T}, c', c_t, c_f, c'' \in \mathcal{C}, b$ such that*

$$\langle c, \tau_0 \rangle \to^* \langle\, \texttt{if}\ ([b]^l, c_t, c_f); c', \tau_0 {+\!+} \tau'_1 \rangle \to^{if-b} \langle c'', \tau_0 {+\!+} \tau'_1 {+\!+} [\epsilon] \rangle \to^* \langle \texttt{skip}, \tau_0 {+\!+} \tau_1 \rangle$$

  - *or there exists $\tau'_1 \in \mathcal{T}, c', c_w, c'' \in \mathcal{C}, b$ such that*

$$\langle c, \tau_0 \rangle \to^* \langle\, \texttt{while}\ ([b]^l, c_w); c', \tau_0 {+\!+} \tau'_1 \rangle \to^{while-b} \langle c'', \tau_0 {+\!+} \tau'_1 {+\!+} [\epsilon] \rangle \to^* \langle \texttt{skip}, \tau_0 {+\!+} \tau_1 \rangle$$

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 $\langle [x \leftarrow a]^l, \tau \rangle \to \langle \texttt{skip}, \tau {+\!+} [(x, l, v)] \rangle$.
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 $c' = \texttt{skip}$ satisfying
$\langle c, \tau_0 \rangle \to^* \langle [x \leftarrow e]^l; c', \tau_0 {+\!+} \tau' \rangle \to^{\mathsf{assn}} \langle c', \tau_0 {+\!+} \tau'_1 {+\!+} [\epsilon] \rangle \to^* \langle \texttt{skip}, \tau_0 {+\!+} \tau_1 \rangle$.
This case is proved.

case: $c = [x \leftarrow \texttt{query}(\psi)]^l$
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 $\mathsf{while\text{-}t}$, 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
$\langle \texttt{while}\ [b]^l\ \texttt{do}\ c_w, \tau\rangle \to^{\textsf{while-f}} \langle \texttt{skip}, \tau{+\!+}[((b,l,\texttt{false}))]\rangle$, $(b,l,\texttt{true}) \in \epsilon^{\texttt{test}}$, and $(b,l,\texttt{true}) \in \tau_1$.
Let $\tau' = []$, $c' = \texttt{skip}$ and $c'' = \texttt{skip}$, we know that they satisfy
$\langle c, \tau_0\rangle \to^* \langle \texttt{while}\ ([b]^l, c_w); c', \tau_0{+\!+}\tau'_1\rangle \to^{\textsf{while-f}} \langle c'', \tau_0{+\!+}\tau'_1{+\!+}[(b,l,\texttt{false})]\rangle \to^* \langle \texttt{skip}, \tau_0{+\!+}\tau_1\rangle$
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 C.8 (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. □

Lemma C.9 (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., $\exists l \in \mathbb{N}, b \in \mathcal{B}, c_w \in \mathcal{C}\ .\ (\texttt{while}\ [b]^l\ \texttt{do}\ c_w) \in_c c_2 \wedge c_1 \in_c c_w$.*

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

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. □

Lemma C.10 (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. □

Lemma C.11. *(Event Dependency Transitivity) For every $D \in \mathcal{DB}, c \in \mathcal{C}, \tau \in \mathcal{T}$, and $\epsilon_1, \epsilon_2, \epsilon_3 \in \mathcal{E}^{\mathsf{asn}}, \tau_{12}, \tau_{23} \in \mathcal{T}$, if $\mathsf{DEP_e}(\epsilon_1, \epsilon_2, \tau_{12}, c, D)$ and $\mathsf{DEP_e}(\epsilon_2, \epsilon_3, \tau_{23}, c, D)$, then $\mathsf{DEP_e}(\epsilon_1, \epsilon_3, \tau_{12}{+\!+}\tau_{23}, c, D)$.*

$$\begin{aligned}&\forall D \in \mathcal{DB}, c \in \mathcal{C}, \epsilon_1, \epsilon_2, \epsilon_3 \in \mathcal{E}^{\mathsf{asn}}, \tau_{12}, \tau_{23} \in \mathcal{T}\,.\,\mathsf{DEP_e}(\epsilon_1, \epsilon_2, \tau_{12}, c, D) \land \mathsf{DEP_e}(\epsilon_2, \epsilon_3, \tau_{23}, c, D)\\&\quad\implies \mathsf{DEP_e}(\epsilon_1, \epsilon_3, \tau_{12}{+\!+}\tau_{23}, c, D)\end{aligned}$$

Lemma C.12 (*Variable May-Dependency* Transitivity). *For every $c \in \mathcal{C}, x^i, y^j, z^l \in \mathbb{LV}_c$, if $\mathsf{DEP_{var}}(x^i, y^j, c)$ and $\mathsf{DEP_{var}}(y^j, z^l, c)$, then $\mathsf{DEP_{var}}(x^i, z^l, c)$.*

$$\forall c \in \mathcal{C}, x^i, y^j, z^l \in \mathbb{LV}_c\,.\,\mathsf{DEP_{var}}(x^i, y^j, c) \land \mathsf{DEP_{var}}(y^j, z^l, c) \implies \mathsf{DEP_{var}}(x^i, z^l, c)$$

D Soundness of The Weight Estimation

D.1 Proof of Lemma 4.1

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

$$\forall c \in \mathcal{C}, \tau_0 \in \mathcal{T}_0(c), \tau \in \mathcal{T}, \epsilon = (\_, l, \_) \in \mathcal{E} \,.\, \langle c, \tau_0 \rangle \to^* \langle \texttt{skip}, \tau_0 +\!\!+ \tau \rangle \land \epsilon \in \tau$$
$$\implies \exists \overset{\alpha}{\epsilon} = (l, \_, \_) \in (\mathcal{L} \times \mathcal{DC}^\top \times \mathcal{L}) \,.\, \overset{\alpha}{\epsilon} \in \texttt{abstrace}(c)$$

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

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

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

case: $c = [x \leftarrow e]^{l'}$
By the evaluation rule assn, we have $\langle [x \leftarrow a]^{l'}, \tau \rangle \to \langle \texttt{skip}, \tau +\!\!+ [(x, l', v)] \rangle$, 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 $\mathsf{while\text{-}t}$, 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., $\epsilon \notin_e \tau$ 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. □

D.2 Proof of Lemma 4.2

Lemma (Uniqueness of the Abstract Events Computation). *For every program c and an execution trace $\tau \in \mathcal{T}$ that is generated w.r.t. an initial trace $\tau_0 \in \mathcal{T}_0(c)$, there is a unique abstract event $\overset{\alpha}{\epsilon} = (l, \_, \_) \in \texttt{abstrace}(c)$ for every assignment event $\epsilon \in \mathcal{E}^{\mathrm{asn}}$ in the execution trace having the label l, i.e., $\epsilon = (\_, l, \_, \_)$ and $\epsilon \in \tau$.*

$$\begin{array}{l} \forall c \in \mathcal{C}, \tau_0 \in \mathcal{T}_0(c), \tau \in \mathcal{T}, \epsilon = (\_, l, \_) \in \mathcal{E}^{\mathrm{asn}} \,.\, \langle c, \tau_0 \rangle \to^* \langle \texttt{skip}, \tau_0 +\!\!+ \tau \rangle \land \epsilon \in \tau \\ \quad \Longrightarrow \exists! \overset{\alpha}{\epsilon} = (l, \_, \_) \in (\mathcal{L} \times \mathcal{DC}^{\top} \times \mathcal{L}) \,.\, \overset{\alpha}{\epsilon} \in \texttt{abstrace}(c) \end{array}$$

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

D.3 Soundness of Weight Estimation, Theorem 4.1

Preliminary Theorem from paper [6].

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

$$\forall c \in \mathcal{C}, l \in \mathbb{LV}(c), \tau_0 \in \mathcal{T}_0(c), \tau \in \mathcal{T}, v \in \mathbb{N} \,.\, \langle c, \tau_0 \rangle \to^* \langle \texttt{skip}, \tau_0 +\!\!+ \tau \rangle \land \langle \texttt{TB}(\overset{\alpha}{\epsilon}, c), \tau_0 \rangle \Downarrow_a v \land \texttt{cnt}(\tau, l) \leq v$$

Theorem D.2 (Soundness of the Weight Estimation)**.** *Given a program c with its program-based dependency graph* $\mathtt{G_{est}} = (\mathtt{V_{est}}, \mathtt{E_{est}}, \mathtt{W_{est}}, \mathtt{Q_{est}})$, *we have:*

$$\forall (x^l, w) \in \mathtt{W_{est}}, \tau, \tau' \in \mathbb{T}, v \in \mathbb{N} \,.\, \langle c, \tau \rangle \to^* \langle \mathtt{skip}, \tau +\!\!+ \tau' \rangle \wedge \langle \tau, w \rangle \Downarrow_e v \wedge \mathtt{cnt}(\tau', l) \leq v$$

Proof. Taking an arbitrary a program c with its program-based dependency graph $\mathtt{G_{est}} = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{Q})$, and an arbitrary pair of labeled variable and weights $(x^l, w) \in \mathtt{W_{est}}$, and arbitrary $\tau, \tau' \in \mathbb{T}, v \in \mathbb{N}$ satisfying $\langle c, \tau \rangle \to^* \langle \mathtt{skip}, \tau +\!\!+ \tau' \rangle \wedge \langle \tau, w \rangle \Downarrow_e v$

By Definition of $\mathtt{W_{est}}$ in $\mathtt{G_{est}}(c)$, we know $w = \mathtt{absW}(l) = \max\{\mathtt{TB}(\overset{\alpha}{\epsilon}) | \overset{\alpha}{\epsilon} = (l, \_, \_)\}$.
By Lemma 4.1, 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 $\mathtt{W_{est}}(x^l) \triangleq \mathtt{TB}(\overset{\alpha}{\epsilon})$. Then, it is sufficient to show:

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

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

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

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

Proved by the soundness of Local bound in Lemma D.1.

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

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

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:

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

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

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

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

Proof.

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

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

sub-case: $l \in SCC(\mathtt{absG}(c)) \wedge \overset{\alpha}{\epsilon} \notin \bigcup_{x \in VAR} \mathtt{dec}(x) \wedge \overset{\alpha}{\epsilon} \notin SCC(\mathtt{absG}(c)/\mathtt{dec}(x))$
in this case, we know $\mathtt{locb}(\overset{\alpha}{\epsilon}) \triangleq x$.
In the two cases above, the soundness is discussed in [6] Section 4 of Paragraph *Discussion on Soundness* in Page 25. □

E Soundness of Adaptivity Computation Algorithm

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

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

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

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

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

By line:3 of $\mathsf{AdaptSearch}(\mathtt{G_{est}})$ algorithm defined in Algorithm **??**, let $\mathtt{G^{SCC}}_1, \cdots, \mathtt{G^{SCC}}_n$ be all the strong connected components on $\mathtt{G_{est}}$ with $0 \leq n \leq |\mathtt{V}|$, where each $\mathtt{G^{SCC}}_{\mathtt{i}} = (\mathtt{V}_i, \mathtt{E}_i, \mathtt{W}_i, \mathtt{Q}_i)$,
By line:5-6 in Algorithm **??**, let $\mathtt{adapt_{scc}}[\mathtt{G^{SCC}}_{\mathtt{i}}]$ 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 E.1, we know

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

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

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

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 E.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 $\mathsf{AdaptSearch}(\mathtt{G_{est}})$. This case is proved. □

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

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

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) $\mathtt{querynum}(p^i_{x,x}) * \mathtt{flowcapacity}(p^i_{x,x}) \geq \mathtt{len}(s | s \in (s_1, \cdots, x) \wedge \mathtt{Q}(s) = 1) = \mathtt{len^q}(k)$,
(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, $\max\{\mathtt{querynum}(p^i_{x,x}) * \mathtt{flowcapacity}(p^i_{x,x}) | x \in \mathtt{G^{SCC}} \wedge p^i_{x,x} \in \mathcal{PATH}(p_{x,x})\} \geq \max\{\mathtt{len^q}(k^i_{x,x}) | x \in \mathtt{G^{SCC}} \wedge k^i_{x,x} \in \mathcal{WK}(k_{x,x})\}$
(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,$\max\{\mathtt{len^q}(k^i_{x,x}) | x \in \mathtt{G^{SCC}} \wedge k^i_{x,x} \in \mathcal{WK}(k_{x,x})\} \geq \max\{\mathtt{len^q}(k^i_{x,y}) | x, y \in \mathtt{G^{SCC}} \wedge k^i_{x,y} \in \mathcal{WK}(k_{x,y})\}$
i.e., $\max\{\mathtt{len^q}(k^i_{x,x}) | x \in \mathtt{G^{SCC}} \wedge k^i_{x,x} \in \mathcal{WK}(k_{x,x})\} \geq \max\{\mathtt{len^q}(k) | k \in \mathcal{WK}(\mathtt{G^{SCC}})\} = \mathtt{A_{est}}(\mathtt{G^{SCC}})$.
(8) We also know $\mathsf{AdaptSearch}_{scc}(\mathtt{G^{SCC}}) = \max\{\mathtt{querynum}(p^i_{x,x}) * \mathtt{flowcapacity}(p^i_{x,x}) | x \in \mathtt{G^{SCC}} \wedge p^i_{x,x} \in \mathcal{PATH}(p_{x,x})\}$ by the $\mathsf{AdaptSearch}_{scc}$ algorithm.
Then we have $\mathsf{AdaptSearch}_{scc}(\mathtt{G^{SCC}}) \geq \mathtt{A_{est}}(\mathtt{G^{SCC}})$

Proof. Taking arbitrary program $c \in \mathcal{C}$, let $\mathtt{G_{est}}(c) = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{Q})$ 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 $\mathtt{G_{est}}$.
There are 2 cases:

case: $\mathtt{G^{SCC}}$ contains no edge and only 1 vertex v, i.e., $|\mathtt{E}| = 0 \wedge |\mathtt{V}| = 1$
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: $\mathtt{G}^{\mathtt{SCC}}$ **contains at least 1 edge and at least 1 vertex** v**, i.e.,** $1 \leq |\mathtt{E}| \wedge 1 \leq |\mathtt{V}|$
Taking arbitrary walk $k_{x,y} \in \mathcal{WK}(\mathtt{G}^{\mathtt{SCC}})$, with vertices sequence $(x, s_1, \cdots, y)$, it is sufficient to show:

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

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. □

F Conditional Completeness of Adaptivity Computation Algorithm

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

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

proof Summary:
1. for every two vertices x, y with a walk $k_{x,y}$ from x to y on $\mathtt{G_{est}}$,
2 since $\mathtt{G_{est}}$ is acyclic directed, then this walk corresponds to a path $p_{x,y}$ where every vertex is visited exactly once.
3. the query length is 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 $\mathtt{G_{est}}(c) = (\mathtt{V_{est}}, \mathtt{E_{est}}, \mathtt{W_{est}}, \mathtt{Q_{est}})$ be its program based dependency graph.
Let the walk $k_{max} \in \mathcal{WK}(\mathtt{G_{est}}(c))$ be the finite walk with the longest query length, and the vertices sequence $(s_1, \cdots, s_n)$, it is sufficient to show:

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

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

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

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

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

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:

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

By line:3 in Algorithm **??**, 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 $\mathtt{Q}_i = \{(s_i, \mathtt{Q_{est}}(s_i))\}$.
So we know $n = m$.

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

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

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 $\mathsf{AdaptSearch}(\mathtt{G}_{\mathtt{est}}(c))$, 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 $\mathtt{len^q}(k'_{max}) = adapt_{p'_{max}}$ and $\mathtt{len^q}(k'_{max}) > \mathtt{len^q}(k_{max})$.
Then there is a contradiction that k'_{max} is the walk with the longest query length rather than k_{max}.
Then, we have (2) proved. $\square$

G The Detail Evaluation Table

Table 2: Experimental results of AdaptFun implementation

Program c		True Value	AdaptFun (I ∣ II)	performance			
				lines	running time (second)		
					Ocaml	Weight	AdaptSearch
`twoRounds(k)`	A	2	$2 \mid 2$	8	0.0005	$0.0017 \mid 0.0002$	0.0003
	query #	$k+1$	$k+1 \mid k+1$				
`multiRounds(k)`	A	k	$k \mid \max(1,k)$	10	0.0012	$0.0017 \mid 0.0002$	0.0002
	query #	k	$k \mid -$				
`lRGD(k,r)`	A	k	$k \mid \max(1,k)$	10	0.0015	$0.0072 \mid 0.0002$	0.0002
	query #	$2k$	$2k \mid -$				
`mROdd(k)`	A	$1+k$	$2+\max(1,2k) \mid -$	10	0.0015	$0.0061 \mid 0.0002$	0.0002
	query #	$1+2k$	$1+3k \mid -$				
`mRSingle(k)`	A	2	$1+\max(1,k) \mid -$	9	0.0011	$0.0075 \mid 0.0002$	0.0002
	query #	$1+k$	$1+k \mid 1+k$				
`ifCD()`	A	3	$3 \mid 3$	5	0.0005	$0.0003 \mid 0.0001$	0.0001
	query #	3	$3 \mid 4$				
`while(k)`	A	$1+k/2$	$1+\max(1,k/2) \mid -$	7	0.0021	$0.0015 \mid 0.0001$	0.0001
	query #	$1+k/2$	$1+k/2 \mid -$				
`whileM(k)`	A	$1+k$	$2+\max(1,2k) \mid -$	9	0.0017	$0.0062 \mid 0.0002$	0.0001
	query #	$1+2k$	$1+3k \mid -$				
`whileRV(k)`	A	$1+2k$	$1+2k \mid 1+\max(1,2k)$	9	0.0016	$0.0056 \mid 0.0002$	0.0001
	query #	$2+3k$	$2+3k \mid -$				
`whileVCD(k)`	A	$1+2Q$	$Q_m+\max(1,2Q_m) \mid -$	6	0.0016	$0.0007 \mid 0.0002$	0.0001
	query #	$2+2Q_m$	$2+2Q_m \mid -$				
`whileMPVCD(k)`	A	$2+Q_m$	$2+2Q_m \mid -$	9	0.0017	$0.0043 \mid 0.0002$	0.0001
	query #	$2+Q_m$	$2+2Q_m \mid -$				
`nestWhileRC(k)`	A	$1+3k$	$1+3k \mid 2+3k+k^2$	11	0.019	$0.2669 \mid 0.0002$	0.0007
	query #	$1+3k$	$1+3k \mid 1+k+k^2$				
`nestWhileVD(k)`	A	$2+k^2$	$3+k^2 \mid -$	10	0.0018	$0.0126 \mid 0.0002$	0.0001
	query #	$1+k+k^2$	$1+k+k^2 \mid -$				
`nestWhileRV(k)`	A	$1+k+k^2$	$2+k+k^2 \mid -$	10	0.0017	$0.0186 \mid 0.0002$	0.0001
	query #	$2+k+k^2$	$2+k+k^2 \mid -$				
`nestWhileMV(k)`	A	$1+2k$	$1+\max(1,2k) \mid -$	10	0.0016	$0.0071 \mid 0.0002$	0.0001
	query #	$1+k+k^2$	$1+k+k^2 \mid -$				
`nestWhileMPRV(k)`	A	$1+k+k^2$	$3+k+k^2 \mid -$	10	0.019	$0.0999 \mid 0.0002$	0.0002
	query #	$2+k+k^2$	$2+2k+k^2 \mid -$				
`mRComplete(k)`	A	k	$k \mid -$	27	0.0026	$85.9017 \mid 0.0003$	0.0004
	query #	k	$k \mid -$				
`mRCompose(k)`	A	$2k$	$2k \mid -$	46	0.0036	$5104 \mid 0.0003$	0.0013
	query #	$2k$	$2k \mid -$				
`seqCompose(k)`	A	12	$12 \mid -$	502	0.0426	$1.2743 \mid 0.0003$	0.0223
	query #	326	$326 \mid -$				
`tRCompose(k)`	A	2	$* \mid 2$	42	0.0026	$* \mid 0.0003$	0.0005
	query #	$1+5k+2k^2$	$* \mid 1+5k+2k^2$				
`jumboS(k)`	A	$\max(20,8+k^2)$	$* \mid \max(20,6+k+k^2)$	71	0.0035	$* \mid 0.0003$	0.0085
	query #	$37+k+k^2$	$* \mid 44+k+k^2$				
`jumbo(k)`	A	$\max(20,10+k+k^2)$	$* \mid \max(20,12+k+k^2)$	502	0.0691	$* \mid 0.0009$	0.018
	query #	$270+22k+10k^2$	$* \mid 286+26k+10k^2$				
`big(k)`	A	$22+k+k*k$	$* \mid 28+k+k^2$	214	0.0175	$* \mid 0.0004$	0.002
	query #	$110+10k+4k^2$	$* \mid 121+11k+4k^2$				

H The Programs and Codes of The Evaluated Examples in Table 2

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

Example H.1 (The Complete Gradient Decent Optimization Algorithm)**.** *This example is the gradient decent algorithm example is a generalization of the linear regression on a higher degree data relation.*

It uses gradient decent algorithm to minimize the mean square loss function for a two-degree relation $y = a_1 \times x_1^2 + a_2 \times x_2 + c$ on the dataset of two feature columns and one indicator column.

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

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

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

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

Evaluation Result: $\mathtt{A}_{\mathtt{est}}(\texttt{seq}()) = 4$

Example H.3 (Sequence with Query Dependency between Related Variables)**.** *This example algorithm contains a sequence of four query commands. Each of them depends on one or more of the previous queries. The longest dependency depth, i.e., the adaptivity is expectation to be* 4.

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

Evaluation Result: $\mathtt{A}_{\mathtt{est}}(\texttt{seqMultiVar}()) = 4$

Example H.4 (If with Data-Value Dependency Separated)**.** *This example algorithm contains a* `if` *command and a query requests in each branch. Only the query in the first branch depend on the query in the command* 0*, and the variable in the guard is not assigned by a query request. The longest dependency depth, i.e., the adaptivity is expectation to be* 3.

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

Evaluation Result: $\mathtt{A}_{\mathtt{est}}(\texttt{ifVD}()) = 3$

Example H.5 (If with Data-Control Dependency Overlapped)**.** *This example algorithm contains a* `if` *command and a query requests in each branch. The variable in the guard is assigned by a query* *request in command* 1*. The two queries in the branches depend on the second query in command* 1 *but not depend on the query in the command* 0*. Even though the variable* x *isn't used in the query expression in the query* 3 *and* 4*, there are still dependency relation because* x *is in the guard. The longest dependency depth, i.e., the adaptivity is expectation to be* 3*.*

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

Evaluation Result: $\mathtt{A_{est}}(\mathtt{ifCD}()) = 3$

Example H.6 (While with Nested Query Dependency)**.** *This example algorithm contains a simple while loop. There is one query requests in the loop body at command* 3*. In each iteration, the query request depend on the query result from previous iteration. The longest dependency depth, i.e., the adaptivity is expectation to be* k*.*

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

The Evaluation Result: $\mathtt{A_{est}}(\mathtt{whileRec}(k)) = 1 + k$

Example H.7 (While with Multi-Path Query Dependency)**.** *This example algorithm contains a simple while loop and a* `if` *command in the loop body. Each branch has a query request (in the commands* 5 *and* 6*) depend on the query at command* 1 *and the query at command* 7*. Among the* $\frac{k}{2}$ *iterations, result from previous iteration. The longest dependency depth, i.e., the adaptivity is expectation to be* $1 + 2 * \lfloor \frac{k}{2} \rfloor$.

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

The Evaluation Result: $\mathtt{A_{est}}(\mathtt{whileM}(k)) = 1 + 2 * \lfloor \frac{k}{2} \rfloor$

Example H.8 (While with Query Dependency through Related Variables)**.** *This example algorithm contains a simple while loop and a sequence of three query requests in the loop body. In each iteration, every query request depend on one or more query results from previous iteration. The longest dependency depth, i.e., the adaptivity is expectation to be* $1 + 2 * k$*.*

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

The Evaluation Result: $\mathtt{A_{est}}(\mathtt{whileRV}(k)) = 1 + 2 * k$

Example H.9 (While with Query Dependency trhough Control Flow and Data Flow)**.** *This example algorithm contains a simple while loop and a sequence of three query requests in the loop body. The variable in the guard is assigned by a query request in command* 0*. In each iteration, the query at*

3 *depends on either the query at line* 1*, and the query result at line* 4 *from the previous iteration. In each iteration, the query at* 4 *depends on either the query at line* 0 *and the query at line* 3 *in the same iteration. The longest dependency depth, i.e., the adaptivity is expectation to be* $1+2*k$*.*

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

The Evaluation Result: $\mathtt{A_{est}}(\texttt{whileVCD}(k)) = 1+2*k$

Example H.10 (While with Multiple Path Query Dependency Dependency)**.** *This example algorithm contains a simple while loop and a* `if` *command in the loop body. Each branch has a query request (in the commands* 5 *and* 6*) depend on either the query at command* 1 *or the query at command* 7*. The longest dependency depth, i.e., the adaptivity is expectation to be* $2+k$*.*

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

The Evaluation Result: $\mathtt{A_{est}}(\texttt{whileMPVCD}(k)) = 2+k$

Example H.11 (Nested While with Nested Query Dependency)**.** *This example algorithm contains two nested while loops. The query in the outer loop at line* 5 *depends on either the query at line* 1 *or the query results at line* 8 *from the previous iteration of the inner loop. The longest dependency depth, i.e., the adaptivity is expectation to be* $2+k^2$*.*

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

The Evaluation Result: $\mathtt{A_{est}}(\texttt{nestWhileVD}(k)) = 2+k^2$

Example H.12 (Nested While with Query Dependency through Related Variables)**.** *This example algorithm contains two nested while loops, one query in the outer loop, and one query in the inner loop. The query in the outer loop at line* 8 *depends on only the query result at line* 7 *from the last iteration of the inner loop. However, the query at line* 7 *depends on either the query at line* 1 *the query results at line* 8 *from the previous iteration. The longest dependency depth, i.e., the adaptivity is expectation to be* $1+2*k$*.*

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

The Evaluation Result: $\mathtt{A_{est}}(\texttt{nestWhileRV}(k)) = 1+2*k$

Example H.13 (Nested While with Nest Query Dependency and Related Variable Accross Outer and Inner Loop). *This example algorithm contains two nested while loops, one query in the outer loop, and one query in the inner loop as well. The two queries depend on both the query results assigned to themselves in previous iteration. The longest dependency depth, i.e., the adaptivity is expectation to be* $1+k+k^2$.

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

The Evaluation Result: $\mathtt{A_{est}}(\texttt{nestWhileMR}(k)) = 1+k+k^2$
Reachability Bound The Evaluation Result:
*weight for Variable: j of label 6 is: 0 + 0 + 1 * k * k*
*weight for Variable: y of label 7 is: 0 + 0 + 1 * k * k*
*weight for Variable: j of label 4 is: 0 + 1 * k*
*weight for Variable: i of label 3 is: 0 + 1 * k*
*weight for Variable: x of label 8 is: 0 + 1 * k*
weight for Variable: x of label 1 is: 1
weight for Variable: i of label 0 is: 1

Example H.14 (Nested While with MultiplePath and Nested Recursive Multiple Variable Data-Value Dependency Across Outer and Inner Loop). *We then show a more complex example with nested while command and nested data-flow across the outer and inner while loop through multiple variables. This example also contains the if command with data dependency occurred through the if guard. The longest dependency depth, i.e., the adaptivity is expectation to be* $1+k+k^2$.

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

The Evaluation Result: $\mathtt{A_{est}}(\texttt{nestWhileMPRV}(k)) = 1+k+k^2$
Reachability Bound The Evaluation Result:
weight for Variable: j of label 10 is: 0 + 0 + 1 * k * k
weight for Variable: x of label 11 is: 0 + 0 + 1 * k * k
weight for Variable: y of label 7 is: 0 + 1 * k
weight for Variable: y of label 8 is: 0 + 1 * k
weight for Variable: j of label 5 is: 0 + 1 * k
weight for Variable: i of label 4 is: 0 + 1 * k
weight for Variable: y of label 2 is: 1
weight for Variable: x of label 1 is: 1
weight for Variable: i of label 0 is: 1

H.2 The Programs for Examples from line:16 - 20 in Table.2

Example H.15 (`mRCompose`). *The composed multiple rounds program:*

```
[ j <- N ] 0 ;
[ l <- 0 ] 1 ;
[ cs <- -1 ] 2 ;
[ ns <- -1 ] 3 ;
while [  < (0, j) ] 4 do {
[ j <- - ( j, 1 ) ] 5;
[ cs <- + ( cs, 0 ) ] 6;
[ ns <- + ( ns, 0 ) ] 7 };
[w <- k] 8;
while [< (0, w)] 9 do {
    [ w <- - ( w, 1 ) ] 10;
    [ p <- c ] 11;
    [ q <- c ] 12;
    [ a <- query ( l ) ] 13 ;
    [ i <- N ] 14;
    while [ < (0, i) ] 15 do {
        [ i <- - (i, 1) ] 16;
        [ csi <- + (csi, * (- (a, p), - (q, p))) ] 17;
        if [ > (i , l) ] 18
            then { [ nsi <- + (nsi, * (- (a, p), - (q, p)) ) ] 19 }
            else { [ nsi <- nsi ] 20 }
            };
    [ i2 <- N ] 21;
    while [ < (0, i2) ] 22 do {
        [ i2 <- - (i2, 1) ] 23;
        if [ > (nsi , l) ] 24
            then { [ l <- + ( l , i2 ) ] 25 }
            else { [ l <- l ] 26 }
        }
 };
 [w <- k] 27;
 while [< (0, w)] 28 do {
    [ w <- - ( w, 1 ) ] 29;
    [ p <- c ] 30;
    [ q <- c ] 31;
    [ a <- query ( l ) ] 32 ;
    [ i <- N ] 33;
    while [ < (0, i) ] 34 do {
        [ i <- - (i, 1) ] 35;
        [ csi <- + (csi, * (- (a, p), - (q, p))) ] 36;
        if [ > (i , l) ] 37
            then { [ nsi <- + (nsi, * (- (a, p), - (q, p)) ) ] 38 }
            else { [ nsi <- nsi ] 39 }
            };
    [ i2 <- N ] 40;
    while [ < (0, i2) ] 41 do {
        [ i2 <- - (i2, 1) ] 42;
        if [ > (nsi , l) ] 43
            then { [ l <- + ( l , i2 ) ] 44 }
            else { [ l <- l ] 45 }
        }
 }
```

Example H.16 (`tRCompose`). *The composed two rounds program:*

```
[ i <- k ] 0 ;
[ l <- 0 ] 1 ;
```

```
while [ > (i, 0) ] 2 do {
[ i <- - ( i, 1 ) ] 3 ;
[ a <- query ( - (k, i) ) ] 4 ;
[ l <- + ( l, a ) ] 5
 } ;
[ y <- query ( l ) ] 6;
[ j <- k ] 7 ;
[ l <- 0 ] 8 ;
while [ > (j, 0) ] 9 do {
[ j <- - ( j, 1 ) ] 10 ;
[ a <- query ( - (k, j) ) ] 11 ;
[ l <- + ( l, a ) ] 12;
[ i <- k ] 13 ;
[ l <- 0 ] 14 ;
while [ > (i, 0) ] 15 do {
[ i <- - ( i, 1 ) ] 16 ;
[ a <- query ( - (k, i) ) ] 17 ;
[ l <- + ( l, a ) ] 18
 }
};
[ i <- k ] 19 ;
[ l <- 0 ] 20 ;
while [ > (i, 0) ] 21 do {
[ i <- - ( i, 1 ) ] 22 ;
[ a <- query ( - (k, i) ) ] 23 ;
[ l <- + ( l, a ) ] 24
 };
 [ i <- k ] 25 ;
[ l <- 0 ] 26 ;
while [ > (i, 0) ] 27 do {
[ i <- - ( i, 1 ) ] 28 ;
[ a <- query ( - (k, i) ) ] 29 ;
[ l <- + ( l, a ) ] 30
 };
 while [ > (j, 0) ] 31 do {
[ j <- - ( j, 1 ) ] 32 ;
[ a <- query ( - (k, j) ) ] 33 ;
[ l <- + ( l, a ) ] 34;
[ i <- k ] 35 ;
[ l <- 0 ] 36 ;
while [ > (i, 0) ] 37 do {
[ i <- - ( i, 1 ) ] 38 ;
[ a <- query ( - (k, i) ) ] 39 ;
[ l <- + ( l, a ) ] 40
 }
};
[ y <- query ( l ) ] 41
```

Example H.17 (`seqCompose`). *The composed two rounds program:*

```
[ x <- query ( 0 ) ] 0 ;
[ y <- query ( x ) ] 1 ;
[ z <- query ( y ) ] 2 ;
[ a <- + ( x, 0 ) ] 3 ;
[ b <- + ( a, z ) ] 4 ;
[ c <- + ( a, b ) ] 5 ;
[ w <- query ( a ) ] 6 ;
```

```
[ x <- query ( b ) ] 7 ;
[ y <- query ( c ) ] 8 ;
[ z <- query ( z ) ] 9 ;
[ w <- query ( z ) ] 10 ;
[ d <- + ( x, w ) ] 11 ;
[ e <- + ( c, z ) ] 12 ;
[ f <- + ( a, b ) ] 13 ;
[ x <- query ( 0 ) ] 14 ;
[ y <- query ( x ) ] 15 ;
[ z <- query ( y ) ] 16 ;
[ x <- query ( z ) ] 17;
[ g <- + ( f, w ) ] 18 ;
[ h <- + ( c, x ) ] 19 ;
[ i <- + ( w, e ) ] 20 ;
[ z <- query ( x ) ] 21 ;
if [ > (x , 0) ] 22
then { [ y <- query ( 0 ) ] 23 }
else { [ w <- query ( 0 ) ] 24 };
[ x <- - (y, w) ] 25 ;
[ j <- 5 ] 26 ;
[ x <- query ( chi : x : ) ] 27 ;
 [ y <- query ( chi : x :) ] 28 ;
[ j <- - ( j, 1 )  ] 29 ;
if [ < (j , 5) ] 30
then { [ y <- query ( chi : x :) ] 31 }
else { [ w <- query ( chi : x : ) ] 32 } ;
[ x <- query ( chi : y : ) ] 33;
[ y <- query ( x ) ] 34 ;
[ z <- query ( + (x, y) ) ] 35 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 36;
[ z <- query ( w ) ] 37 ;
[ g <- + ( f, z ) ] 38 ;
[ h <- + ( c, x ) ] 39 ;
[ i <- + ( w, g ) ] 40 ;
[ z <- query ( x ) ] 41 ;
[ e <- + ( c, z ) ] 42 ;
[ f <- + ( a, i ) ] 43 ;
[ x <- query ( 0 ) ] 44 ;
[ y <- query ( x ) ] 45 ;
[ z <- query ( y ) ] 46 ;
[ z <- query ( z ) ] 47;
[ x <- / (k, 2 ) ] 48 ;
if [ > (x , 0) ] 49
then { [ y <- query ( z ) ] 50 }
else { [ y <- query ( 0 ) ] 51 } ;
[ i <- k ] 52 ;
[ x <- query ( y ) ] 53 ;
[ y <- query ( x ) ] 54 ;
 [  w <- - ( w, 1 ) ] 55 ;
[ i <- - (i, 1) ] 56 ;
[ y <- query ( + (z, y) ) ] 57 ;
[ j <- k ] 58 ;
 [ y <- query ( chi : x :) ] 59;
[ j <- - (j, 1) ] 60 ;
[ x <- query ( + ( x, y) ) ] 61;
 [ j <- N ] 62 ;
[ l <- x ] 63 ;
```

```
[ cs <- -1 ] 64 ;
[ ns <- -1 ] 65 ;
 [  y <- query ( chi : x :) ] 66 ;
[ j <- - ( j, 1 ) ] 67;
[ cs <- + ( cs, 0 ) ] 68 ;
[ ns <- + ( ns, 0 ) ] 69 ;
[w <- k] 70;
 [w <- - ( w, 1 )] 71 ;
    [ w <- - ( w, 1 ) ] 72 ;
    [ p <- c ] 73;
    [ q <- c ] 74;
    [ a <- query ( l ) ] 75 ;
    [ i <- N ] 76;
     [ w <- - ( w, 1 ) ] 77 ;
        [ i <- - (i, 1) ] 78;
        [ csi <- + (csi, * (- (a, p), - (q, p))) ] 79;
        if [ > (i , I) ] 80
            then { [ nsi <- + (nsi, * (- (a, p), - (q, p)) ) ] 81 }
            else { [ nsi <- nsi ] 82 };
    [ i2 <- N ] 83;
     [ y <- query ( chi : x :) ] 84 ;
        [ i2 <- - (i2, 1) ] 85;
        if [ > (ns , I) ] 86
            then { [ l <- + ( l , i2 ) ] 87 }
            else { [ l <- l ] 88 };
[ x <- query ( cs ) ] 89 ;
[ y <- query ( x ) ] 90 ;
[ z <- query ( ns ) ] 91 ;
[ w <- query ( z ) ] 92 ;
[ a <- x ] 93 ;
[ c <- z ] 94 ;
[ j <- k ] 95 ;
 [  y <- query ( chi : x :) ] 96 ;
[ da <- query ( * ( a , c ) ) ] 97 ;
[ dc <- query ( * ( a , c ) ) ] 98 ;
[ a <- - (a, da) ] 99 ;
[ c <- - (c, dc) ] 100 ;
[ j <- - (j, 1 ) ] 101 ;
[ x <- query ( 0 ) ] 102 ;
[ y <- query ( cs ) ] 103 ;
[ z <- query ( c ) ] 104 ;
[ w <- query ( z ) ] 105 ;
[ x <- query ( 0 ) ] 106 ;
[ y <- query ( x ) ] 107 ;
[ z <- query ( y ) ] 108 ;
[ w <- query ( z ) ] 109 ;
[ x <- query ( 0 ) ] 110 ;
[ y <- query ( x ) ] 111 ;
[ z <- query ( + (x, y) ) ] 112 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 113 ;
[ z <- query ( w ) ] 114 ;
[ x <- query ( e ) ] 115 ;
if [ > (x , 0) ] 116
then { [ y <- query ( 0 ) ] 117 }
else { [ w <- query ( 0 ) ] 118 } ;
[ a <- x ] 119 ;
[ c <- z ] 120 ;
```

```
[ j <- k ] 121 ;
[ cs <- + ( cs, 0 ) ] 122 ;
[ ns <- + ( ns, 0 ) ] 123 ;
 [ i <- k ] 124 ;
[ x <- query ( 0 ) ] 125 ;
[ y <- query ( cs ) ] 126 ;
 [ w <- - ( w, 1 ) ] 127 ;
[ i <- - (i, 1) ] 128 ;
[ j <- k ] 129 ;
if [ > (x , 0) ] 130
then { [ y <- query ( + ( chi : x : , chi : y : )  ) ] 131 }
else { [ y <- query ( chi : x : ) ] 132 } ;
 [  y <- query ( chi : x :) ] 133 ;
[ j <- - (j, 1) ] 134 ;
[ x <- query ( + ( x, y) ) ] 135;
[ x <- query ( z ) ] 136 ;
[ y <- query ( cs ) ] 137 ;
[ z <- query ( c ) ] 138 ;
[ w <- query ( z ) ] 139 ;
[ y <- query ( x ) ] 140 ;
[ z <- query ( + (x, y) ) ] 141 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 142 ;
[ z <- query ( 0 ) ] 143 ;
if [ > (x , 0) ] 144
then { [ y <- query ( x ) ] 145 }
else { [ w <- query ( z ) ] 146 } ;
[ y <- query ( cs ) ] 147 ;
[ z <- query ( c ) ] 148 ;
[ w <- query ( z ) ] 149 ;
[ x <- query ( w ) ] 150 ;
[ y <- query ( x ) ] 151 ;
[ z <- query ( y ) ] 152 ;
[ w <- query ( z ) ] 153 ;
[ x <- query ( 0 ) ] 154 ;
[ y <- query ( x ) ] 155 ;
[ z <- query ( + (x, y) ) ] 156 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 157 ;
[ z <- query ( 0 ) ] 158 ;
[ x <- query ( w ) ] 159 ;
[ i <- k ] 160 ;
[ x <- query ( chi : cs : ) ] 161 ;
 [  w <- - ( w, 1 ) ] 162 ;
[ i <- - (i, 1) ] 163 ;
[ j <- k ] 164 ;
[ y <- query ( chi : x : ) ] 165 ;
 [ y <- query ( chi : x :)] 166;
 [ j <- - (j, 1) ] 167 ;
[ x <- query ( chi : x : ) ] 168;
 [ x <- query ( cs ) ] 169 ;
[ y <- query ( x ) ] 170 ;
[ z <- query ( ns ) ] 171 ;
[ w <- query ( z ) ] 172 ;
[ a <- x ] 173 ;
[ c <- z ] 174 ;
[ j <- k ] 175 ;
 [ y <- query ( chi : x :) ] 176 ;
[ da <- query ( * ( a , c ) ) ] 177 ;
```

```
[ dc <- query ( * ( a , c ) ) ] 178 ;
[ a <- - (a, da) ] 179 ;
[ c <- - (c, dc) ] 180 ;
[ j <- - (j, 1 ) ] 181 ;
[ x <- query ( 0 ) ] 182 ;
[ y <- query ( cs ) ] 183 ;
[ z <- query ( c ) ] 184 ;
[ w <- query ( z ) ] 185 ;
[ x <- query ( 0 ) ] 186 ;
[ y <- query ( x ) ] 187 ;
[ z <- query ( y ) ] 188 ;
[ w <- query ( z ) ] 189 ;
[ x <- query ( 0 ) ] 190 ;
[ y <- query ( x ) ] 191 ;
[ z <- query ( + (x, y) ) ] 192 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 193 ;
[ b <- + ( a, z ) ] 194 ;
[ c <- + ( a, b ) ] 195 ;
[ w <- query ( a ) ] 196 ;
[ x <- query ( b ) ] 197 ;
[ y <- query ( c ) ] 198 ;
[ z <- query ( z ) ] 199 ;
[ w <- query ( z ) ] 200 ;
[ j <- - (w, 1 ) ] 201 ;
[ x <- query ( 0 ) ] 202 ;
[ y <- query ( cs ) ] 203 ;
[ z <- query ( c ) ] 204 ;
[ w <- query ( z ) ] 205 ;
[ x <- query ( 0 ) ] 206 ;
[ y <- query ( x ) ] 207 ;
[ z <- query ( y ) ] 208 ;
[ w <- query ( z ) ] 209 ;
[ x <- query ( 0 ) ] 210 ;
[ d <- + ( x, w ) ] 211 ;
[ e <- + ( c, z ) ] 212 ;
[ f <- + ( a, b ) ] 213 ;
[ x <- query ( w ) ] 214 ;
[ y <- query ( x ) ] 215 ;
[ z <- query ( y ) ] 216 ;
[ x <- query ( z ) ] 217;
[ g <- + ( f, w ) ] 218 ;
[ h <- + ( c, x ) ] 219 ;
[ i <- + ( w, e ) ] 220 ;
[ z <- query ( x ) ] 221 ;
[ cs <- + ( cs, 0 ) ] 222 ;
[ ns <- + ( ns, 0 ) ] 223 ;
 [ i <- k ] 224 ;
[ x <- query ( z ) ] 225 ;
[ y <- query ( cs ) ] 226 ;
 [ w <- - ( w, 1 ) ] 227 ;
[ i <- - (i, 1) ] 228 ;
[ j <- k ] 229 ;
if [ > (x , 0) ] 230
then { [ y <- query ( + ( chi : x : , chi : y : )  ) ] 231 }
else { [ y <- query ( chi : x : ) ] 232 } ;
 [  y <- query ( chi : x :) ] 233;
[ j <- - (j, 1) ] 234 ;
```

```
[ x <- query ( + ( x, y) ) ] 235;
[ x <- query ( z ) ] 236 ;
[ y <- query ( cs ) ] 237 ;
[ z <- query ( c ) ] 238 ;
[ w <- query ( z ) ] 239 ;
[ y <- query ( x ) ] 240 ;
[ z <- query ( + (x, y) ) ] 241 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 242 ;
[ z <- query ( 0 ) ] 243 ;
if [ > (x , 0) ] 244
then { [ y <- query ( x ) ] 245 }
else { [ w <- query ( z ) ] 246 } ;
[ y <- query ( cs ) ] 247 ;
[ z <- query ( c ) ] 248 ;
[ w <- query ( z ) ] 249 ;
[ x <- query ( w ) ] 250 ;
[ y <- query ( x ) ] 251 ;
[ z <- query ( w ) ] 252 ;
[ w <- query ( z ) ] 253 ;
[ x <- query ( 0 ) ] 254 ;
[ y <- query ( x ) ] 255 ;
[ z <- query ( + (x, y) ) ] 256 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 257 ;
[ z <- query ( 0 ) ] 258 ;
[ x <- query ( w ) ] 259 ;
[ i <- k ] 260 ;
[ x <- query ( chi : cs : ) ] 261 ;
 [ w <- - ( w, 1 )] 262 ;
[ i <- - (i, 1) ] 263 ;
[ j <- k ] 264 ;
[ y <- query ( chi : x : ) ] 265 ;
 [ w <- - ( w, 1 ) ] 266;
[ j <- - (j, 1) ] 267 ;
[ x <- query ( chi : x : ) ] 268;
 [ x <- query ( cs ) ] 269 ;
[ y <- query ( x ) ] 270 ;
[ z <- query ( ns ) ] 271 ;
[ w <- query ( z ) ] 272 ;
[ a <- x ] 273 ;
[ c <- z ] 274 ;
[ j <- k ] 275 ;
 [  y <- query ( chi : x :) ] 276 ;
[ da <- query ( * ( a , c ) ) ] 277 ;
[ dc <- query ( * ( a , c ) ) ] 278 ;
[ a <- - (a, da) ] 279 ;
[ c <- - (c, dc) ] 280 ;
[ j <- - (j, 1 ) ] 281 ;
[ x <- query ( a ) ] 282 ;
[ y <- query ( cs ) ] 283 ;
[ z <- query ( c ) ] 284 ;
[ w <- query ( z ) ] 285 ;
[ x <- query ( w ) ] 286 ;
[ y <- query ( x ) ] 287 ;
[ z <- query ( y ) ] 288 ;
[ w <- query ( z ) ] 289 ;
[ x <- query ( 0 ) ] 290 ;
[ y <- query ( x ) ] 291 ;
```

```
[ z <- query ( + (x, y) ) ] 292 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 293 ;
[ z <- query ( 0 ) ] 294 ;
[ x <- query ( w ) ] 295 ;
if [ > (x , 0) ] 296
then { [ y <- query ( 0 ) ] 297 }
else { [ w <- query ( 0 ) ] 298 } ;
[ a <- x ] 299 ;
[ c <- z ] 300 ;
[ j <- k ] 301 ;
[ cs <- + ( cs, 0 ) ] 302 ;
[ ns <- + ( ns, 0 ) ] 303 ;
 [ i <- k ] 304 ;
[ x <- query ( 0 ) ] 305 ;
[ y <- query ( cs ) ] 306 ;
 [  w <- - ( w, 1 ) ] 307 ;
[ i <- - (i, 1) ] 308 ;
[ j <- k ] 309 ;
if [ > (x , 0) ] 310
then { [ y <- query ( + ( chi : x : , chi : y : )  ) ] 311 }
else { [ y <- query ( chi : x : ) ] 312 } ;
 [ y <- query ( chi : x :) ] 313;
 [ j <- - (j, 1) ] 314 ;
[ x <- query ( + ( x, y) ) ] 315;
[ x <- query ( c ) ] 316 ;
[ y <- query ( cs ) ] 317 ;
[ z <- query ( c ) ] 318 ;
[ w <- query ( z ) ] 319 ;
[ y <- query ( x ) ] 320 ;
[ z <- query ( + (x, y) ) ] 321 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 322 ;
[ z <- query ( 0 ) ] 323 ;
if [ > (x , 0) ] 324
then { [ y <- query ( w ) ] 325 }
else { [ w <- query ( z ) ] 326 } ;
[ y <- query ( cs ) ] 327 ;
[ z <- query ( c ) ] 328 ;
[ w <- query ( z ) ] 329 ;
[ x <- query ( 0 ) ] 330 ;
[ y <- query ( x ) ] 331 ;
[ z <- query ( y ) ] 332 ;
[ w <- query ( z ) ] 333 ;
[ x <- query ( y ) ] 334 ;
[ y <- query ( x ) ] 335 ;
[ z <- query ( + (x, y) ) ] 336 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 337 ;
[ z <- query ( 0 ) ] 338 ;
[ x <- query ( w ) ] 339 ;
[ i <- k ] 340 ;
[ x <- query ( chi : cs : ) ] 341 ;
 [  w <- - ( w, 1 ) ] 342 ;
[ i <- - (i, 1) ] 343 ;
[ j <- k ] 344 ;
[ y <- query ( chi : x : ) ] 345 ;
 [ y <- query ( chi : x :)] 346;
 [ j <- - (j, 1) ] 347 ;
[ x <- query ( chi : x : ) ] 348;
```

```
 [ x <- query ( cs ) ] 349 ;
[ y <- query ( x ) ] 350 ;
[ z <- query ( ns ) ] 351 ;
[ w <- query ( z ) ] 352 ;
[ a <- x ] 353 ;
[ c <- z ] 354 ;
[ j <- k ] 355 ;
 [  y <- query ( chi : x :) ] 356 ;
[ da <- query ( * ( a , c ) ) ] 357 ;
[ dc <- query ( * ( a , c ) ) ] 358 ;
[ a <- - (a, da) ] 359 ;
[ c <- - (c, dc) ] 360 ;
[ j <- - (j, 1 ) ] 361 ;
[ x <- query ( 0 ) ] 362 ;
[ y <- query ( cs ) ] 363 ;
[ z <- query ( c ) ] 364 ;
[ w <- query ( z ) ] 365 ;
[ x <- query ( 0 ) ] 366 ;
[ y <- query ( x ) ] 367 ;
[ z <- query ( y ) ] 368 ;
[ w <- query ( z ) ] 369 ;
[ x <- query ( 0 ) ] 370 ;
[ y <- query ( x ) ] 371 ;
[ z <- query ( + (x, y) ) ] 372 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 373 ;
[ z <- query ( 0 ) ] 374 ;
[ x <- query ( w ) ] 375 ;
if [ > (x , 0) ] 376
then { [ y <- query ( x ) ] 377 }
else { [ w <- query ( 0 ) ] 378 } ;
[ a <- x ] 379 ;
[ c <- z ] 380 ;
[ j <- k ] 381 ;
[ cs <- + ( cs, 0 ) ] 382 ;
[ ns <- + ( ns, 0 ) ] 383 ;
 [ i <- k ] 384 ;
[ x <- query ( 0 ) ] 385 ;
[ y <- query ( cs ) ] 386 ;
 [ w <- - ( w, 1 ) ] 387 ;
[ i <- - (i, 1) ] 388 ;
[ j <- k ] 389 ;
if [ > (x , 0) ] 390
then { [ y <- query ( + ( chi : x : , chi : y : )  ) ] 391 }
else { [ y <- query ( chi : x : ) ] 392 } ;
 [ y <- query ( chi : x :) ] 393;
[ j <- - (j, 1) ] 394 ;
[ x <- query ( + ( x, y) ) ] 395;
[ x <- query ( 0 ) ] 396 ;
[ y <- query ( cs ) ] 397 ;
[ z <- query ( c ) ] 398 ;
[ w <- query ( z ) ] 399 ;
[ y <- query ( x ) ] 400 ;
[ z <- query ( + (x, y) ) ] 401 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 402 ;
[ z <- query ( 0 ) ] 403 ;
if [ > (x , 0) ] 404
then { [ y <- query ( w ) ] 405 }
```

```
else { [ w <- query ( z ) ] 406 } ;
[ y <- query ( cs ) ] 407 ;
[ z <- query ( c ) ] 408 ;
[ w <- query ( z ) ] 409 ;
[ x <- query ( 0 ) ] 410 ;
if [ > (w , 0) ] 411
then { [ y <- w ] 412 }
else { [ w <- query ( z ) ] 413 } ;
[ x <- query ( w ) ] 414 ;
[ y <- query ( x ) ] 415 ;
[ z <- query ( y ) ] 416 ;
[ x <- query ( z ) ] 417;
[ g <- + ( f, w ) ] 418 ;
[ h <- + ( c, x ) ] 419 ;
[ i <- + ( w, e ) ] 420 ;
[ z <- query ( x ) ] 421 ;
[ cs <- + ( cs, 0 ) ] 422 ;
[ ns <- + ( ns, 0 ) ] 423 ;
 [ i <- k ] 424 ;
[ x <- query ( z ) ] 425 ;
[ y <- query ( cs ) ] 426 ;
 [ w <- - ( w, 1 ) ] 427 ;
[ i <- - (i, 1) ] 428 ;
[ j <- k ] 429 ;
if [ > (x , 0) ] 430
then { [ y <- query ( + ( chi : x : , chi : y : )  ) ] 431 }
else { [ y <- query ( chi : x : ) ] 432 } ;
 [  y <- query ( chi : x :) ] 433;
 [ j <- - (j, 1) ] 434 ;
[ x <- query ( + ( x, y) ) ] 435 ;
[ x <- query ( z ) ] 436 ;
[ y <- query ( cs ) ] 437 ;
[ z <- query ( c ) ] 438 ;
[ w <- query ( z ) ] 439 ;
[ y <- query ( w ) ] 440 ;
[ z <- query ( + (x, y) ) ] 441 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 442 ;
[ z <- query ( 0 ) ] 443 ;
if [ > (x , 0) ] 444
then { [ y <- query ( x ) ] 445 }
else { [ w <- query ( z ) ] 446 } ;
[ y <- query ( cs ) ] 447 ;
[ z <- query ( c ) ] 448 ;
[ w <- query ( z ) ] 449 ;
[ x <- query ( w ) ] 450 ;
[ y <- query ( x ) ] 451 ;
[ z <- query ( w ) ] 452 ;
[ w <- query ( z ) ] 453 ;
[ x <- query ( 0 ) ] 454 ;
[ y <- query ( x ) ] 455 ;
[ z <- query ( + (x, y) ) ] 456 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 457 ;
[ z <- query ( 0 ) ] 458 ;
[ x <- query ( w ) ] 459 ;
[ i <- k ] 460 ;
[ x <- query ( chi : cs : ) ] 461 ;
[ i <- - (i, 1)  ] 462;
```

```
[ i <- - (i, 1) ] 463 ;
[ j <- k ] 464 ;
[ y <- query ( chi : x : ) ] 465 ;
[ j <- - (j, 1) ] 466;
[ j <- - (j, 1) ] 467 ;
[ x <- query ( chi : x : ) ] 468;
 [ x <- query ( cs ) ] 469 ;
[ y <- query ( x ) ] 470 ;
[ z <- query ( ns ) ] 471 ;
[ w <- query ( z ) ] 472 ;
[ a <- x ] 473 ;
[ c <- z ] 474 ;
[ j <- k ] 475 ;
[  da <- query ( * ( a , c ) )  ] 476;
[ da <- query ( * ( a , c ) ) ] 477 ;
[ dc <- query ( * ( a , c ) ) ] 478 ;
[ a <- - (a, da) ] 479 ;
[ c <- - (c, dc) ] 480 ;
[ j <- - (j, 1 ) ] 481 ;
[ x <- query ( a ) ] 482 ;
[ y <- query ( cs ) ] 483 ;
[ z <- query ( c ) ] 484 ;
[ w <- query ( z ) ] 485 ;
[ x <- query ( w ) ] 486 ;
[ y <- query ( x ) ] 487 ;
[ z <- query ( y ) ] 488 ;
[ w <- query ( z ) ] 489 ;
[ x <- query ( 0 ) ] 490 ;
[ y <- query ( x ) ] 491 ;
[ z <- query ( + (x, y) ) ] 492 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 493 ;
[ z <- query ( 0 ) ] 494 ;
[ x <- query ( w ) ] 495 ;
if [ > (x , 0) ] 496
then { [ y <- query ( 0 ) ] 497 }
else { [ w <- query ( 0 ) ] 498 } ;
[ a <- x ] 499 ;
[ c <- z ] 500 ;
[ r <- query ( c ) ] 501
```

Example H.18 (`jumboS`). *The composed program with nested loops.*

```
[ x <- query ( 0 ) ] 0 ;
[ y <- query ( x ) ] 1 ;
[ z <- query ( y ) ] 2 ;
[ a <- + ( x, 0 ) ] 3 ;
[ b <- + ( a, z ) ] 4 ;
[ c <- + ( a, b ) ] 5 ;
[ w <- query ( a ) ] 6 ;
[ x <- query ( b ) ] 7 ;
[ y <- query ( c ) ] 8 ;
[ z <- query ( z ) ] 9 ;
[ w <- query ( z ) ] 10 ;
[ d <- + ( x, w ) ] 11 ;
[ e <- + ( c, z ) ] 12 ;
[ f <- + ( a, b ) ] 13 ;
[ x <- query ( 0 ) ] 14 ;
```

```
[ y <- query ( x ) ] 15 ;
[ z <- query ( y ) ] 16 ;
[ x <- query ( z ) ] 17;
[ g <- + ( f, w ) ] 18 ;
[ h <- + ( c, x ) ] 19 ;
[ i <- + ( w, e ) ] 20 ;
[ z <- query ( x ) ] 21 ;
if [ > (x , 0) ] 22
then { [ y <- query ( 0 ) ] 23 }
else { [ w <- query ( 0 ) ] 24 };
[ x <- - (y, w) ] 25 ;
[ j <- 5 ] 26 ;
[ x <- query ( chi : x : ) ] 27 ;
while [  > (j, 0) ] 28 do {
[ j <- - ( j, 1 )  ] 29 ;
if [ < (j , 5) ] 30
then { [ y <- query ( chi : x :) ] 31 }
else { [ w <- query ( chi : x : ) ] 32 } ;
[ x <- query ( chi : y : ) ] 33
};
[ y <- query ( x ) ] 34 ;
[ z <- query ( + (x, y) ) ] 35 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 36;
[ z <- query ( w ) ] 37 ;
[ g <- + ( f, z ) ] 38 ;
[ h <- + ( c, x ) ] 39 ;
[ i <- + ( w, g ) ] 40 ;
[ z <- query ( x ) ] 41 ;
[ e <- + ( c, z ) ] 42 ;
[ f <- + ( a, i ) ] 43 ;
[ x <- query ( 0 ) ] 44 ;
[ y <- query ( x ) ] 45 ;
[ z <- query ( y ) ] 46 ;
[ z <- query ( z ) ] 47;
[ x <- / (k, 2 ) ] 48 ;
if [ > (x , 0) ] 49
then { [ y <- query ( z ) ] 50 }
else { [ y <- query ( 0 ) ] 51 } ;
[ i <- k ] 52 ;
[ x <- query ( y ) ] 53 ;
[ y <- query ( x ) ] 54 ;
while [  > (i , 0) ] 55 do {
[ i <- - (i, 1) ] 56 ;
[ y <- query ( + (z, y) ) ] 57 ;
[ j <- k ] 58 ;
while [  > (j , 0) ] 59 do
{ [ j <- - (j, 1) ] 60 ;
[ x <- query ( + ( x, y) ) ] 61 }
 } ;
 [ j <- N ] 62 ;
[ l <- x ] 63 ;
[ cs <- -1 ] 64 ;
[ ns <- -1 ] 65 ;
while [  < (0, j) ] 66 do {
[ j <- - ( j, 1 ) ] 67;
[ cs <- + ( cs, 0 ) ] 68 ;
[ ns <- + ( ns, 0 ) ] 69 };
```

```
[w <- k] 70
```

Example H.19 (`jumbo`). *The composed program with multiple paths nested loops.*

```
[ x <- query ( 0 ) ] 0 ;
[ y <- query ( x ) ] 1 ;
[ z <- query ( y ) ] 2 ;
[ a <- + ( x, 0 ) ] 3 ;
[ b <- + ( a, z ) ] 4 ;
[ c <- + ( a, b ) ] 5 ;
[ w <- query ( a ) ] 6 ;
[ x <- query ( b ) ] 7 ;
[ y <- query ( c ) ] 8 ;
[ z <- query ( z ) ] 9 ;
[ w <- query ( z ) ] 10 ;
[ d <- + ( x, w ) ] 11 ;
[ e <- + ( c, z ) ] 12 ;
[ f <- + ( a, b ) ] 13 ;
[ x <- query ( 0 ) ] 14 ;
[ y <- query ( x ) ] 15 ;
[ z <- query ( y ) ] 16 ;
[ x <- query ( z ) ] 17;
[ g <- + ( f, w ) ] 18 ;
[ h <- + ( c, x ) ] 19 ;
[ i <- + ( w, e ) ] 20 ;
[ z <- query ( x ) ] 21 ;
if [ > (x , 0) ] 22
then { [ y <- query ( 0 ) ] 23 }
else { [ w <- query ( 0 ) ] 24 };
[ x <- - (y, w) ] 25 ;
[ j <- 5 ] 26 ;
[ x <- query ( chi : x : ) ] 27 ;
while [  > (j, 0) ] 28 do {
[ j <- - ( j, 1 )  ] 29 ;
if [ < (j , 5) ] 30
then { [ y <- query ( chi : x :) ] 31 }
else { [ w <- query ( chi : x : ) ] 32 } ;
[ x <- query ( chi : y : ) ] 33
};
[ y <- query ( x ) ] 34 ;
[ z <- query ( + (x, y) ) ] 35 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 36;
[ z <- query ( w ) ] 37 ;
[ g <- + ( f, z ) ] 38 ;
[ h <- + ( c, x ) ] 39 ;
[ i <- + ( w, g ) ] 40 ;
[ z <- query ( x ) ] 41 ;
[ e <- + ( c, z ) ] 42 ;
[ f <- + ( a, i ) ] 43 ;
[ x <- query ( 0 ) ] 44 ;
[ y <- query ( x ) ] 45 ;
[ z <- query ( y ) ] 46 ;
[ z <- query ( z ) ] 47;
[ x <- / (k, 2 ) ] 48 ;
if [ > (x , 0) ] 49
then { [ y <- query ( z ) ] 50 }
else { [ y <- query ( 0 ) ] 51 } ;
```

```
[ i <- k ] 52 ;
[ x <- query ( y ) ] 53 ;
[ y <- query ( x ) ] 54 ;
while [  > (i , 0) ] 55 do {
[ i <- - (i, 1) ] 56 ;
[ y <- query ( + (z, y) ) ] 57 ;
[ j <- k ] 58 ;
while [  > (j , 0) ] 59 do
{ [ j <- - (j, 1) ] 60 ;
[ x <- query ( + ( x, y) ) ] 61 }
 } ;
 [ j <- N ] 62 ;
[ l <- x ] 63 ;
[ cs <- -1 ] 64 ;
[ ns <- -1 ] 65 ;
while [  < (0, j) ] 66 do {
[ j <- - ( j, 1 ) ] 67;
[ cs <- + ( cs, 0 ) ] 68 ;
[ ns <- + ( ns, 0 ) ] 69 };
[w <- k] 70;
while [< (0, w)] 71 do {
    [ w <- - ( w, 1 ) ] 72 ;
    [ p <- c ] 73;
    [ q <- c ] 74;
    [ a <- query ( l ) ] 75 ;
    [ i <- N ] 76;
    while [ < (0, i) ] 77 do {
        [ i <- - (i, 1) ] 78;
        [ csi <- + (csi, * (- (a, p), - (q, p))) ] 79;
        if [ > (i , I) ] 80
            then { [ nsi <- + (nsi, * (- (a, p), - (q, p)) ) ] 81 }
            else { [ nsi <- nsi ] 82 }
            };
    [ i2 <- N ] 83;
    while [ < (0, i2) ] 84 do {
        [ i2 <- - (i2, 1) ] 85;
        if [ > (ns , I) ] 86
            then { [ l <- + ( l , i2 ) ] 87 }
            else { [ l <- l ] 88 }
        }
 };
[ x <- query ( cs ) ] 89 ;
[ y <- query ( x ) ] 90 ;
[ z <- query ( ns ) ] 91 ;
[ w <- query ( z ) ] 92 ;
[ a <- x ] 93 ;
[ c <- z ] 94 ;
[ j <- k ] 95 ;
while [  < (0, j) ] 96 do {
[ da <- query ( * ( a , c ) ) ] 97 ;
[ dc <- query ( * ( a , c ) ) ] 98 ;
[ a <- - (a, da) ] 99 ;
[ c <- - (c, dc) ] 100 ;
[ j <- - (j, 1 ) ] 101
 };
[ x <- query ( 0 ) ] 102 ;
[ y <- query ( cs ) ] 103 ;
```

```
[ z <- query ( c ) ] 104 ;
[ w <- query ( z ) ] 105 ;
[ x <- query ( 0 ) ] 106 ;
[ y <- query ( x ) ] 107 ;
[ z <- query ( y ) ] 108 ;
[ w <- query ( z ) ] 109 ;
[ x <- query ( 0 ) ] 110 ;
[ y <- query ( x ) ] 111 ;
[ z <- query ( + (x, y) ) ] 112 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 113 ;
[ z <- query ( w ) ] 114 ;
[ x <- query ( e ) ] 115 ;
if [ > (x , 0) ] 116
then { [ y <- query ( 0 ) ] 117 }
else { [ w <- query ( 0 ) ] 118 } ;
[ a <- x ] 119 ;
[ c <- z ] 120 ;
[ j <- k ] 121 ;
[ cs <- + ( cs, 0 ) ] 122 ;
[ ns <- + ( ns, 0 ) ] 123 ;
 [ i <- k ] 124 ;
[ x <- query ( 0 ) ] 125 ;
[ y <- query ( cs ) ] 126 ;
while [  > (i , 0) ] 127 do {
[ i <- - (i, 1) ] 128 ;
[ j <- k ] 129 ;
if [ > (x , 0) ] 130
then { [ y <- query ( + ( chi : x : , chi : y : )  ) ] 131 }
else { [ y <- query ( chi : x : ) ] 132 } ;
while [  > (j , 0) ] 133 do
{ [ j <- - (j, 1) ] 134 ;
[ x <- query ( + ( x, y) ) ] 135 }
 };
[ x <- query ( z ) ] 136 ;
[ y <- query ( cs ) ] 137 ;
[ z <- query ( c ) ] 138 ;
[ w <- query ( z ) ] 139 ;
[ y <- query ( x ) ] 140 ;
[ z <- query ( + (x, y) ) ] 141 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 142 ;
[ z <- query ( 0 ) ] 143 ;
if [ > (x , 0) ] 144
then { [ y <- query ( x ) ] 145 }
else { [ w <- query ( z ) ] 146 } ;
[ y <- query ( cs ) ] 147 ;
[ z <- query ( c ) ] 148 ;
[ w <- query ( z ) ] 149 ;
[ x <- query ( w ) ] 150 ;
[ y <- query ( x ) ] 151 ;
[ z <- query ( y ) ] 152 ;
[ w <- query ( z ) ] 153 ;
[ x <- query ( 0 ) ] 154 ;
[ y <- query ( x ) ] 155 ;
[ z <- query ( + (x, y) ) ] 156 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 157 ;
[ z <- query ( 0 ) ] 158 ;
[ x <- query ( w ) ] 159 ;
```

```
[ i <- k ] 160 ;
[ x <- query ( chi : cs : ) ] 161 ;
while [  > (i , 0) ] 162 do {
[ i <- - (i, 1) ] 163 ;
[ j <- k ] 164 ;
[ y <- query ( chi : x : ) ] 165 ;
while [  > (j , 0) ] 166 do
{ [ j <- - (j, 1) ] 167 ;
[ x <- query ( chi : x : ) ] 168 }
 };
 [ x <- query ( cs ) ] 169 ;
[ y <- query ( x ) ] 170 ;
[ z <- query ( ns ) ] 171 ;
[ w <- query ( z ) ] 172 ;
[ a <- x ] 173 ;
[ c <- z ] 174 ;
[ j <- k ] 175 ;
while [  < (0, j) ] 176 do {
[ da <- query ( * ( a , c ) ) ] 177 ;
[ dc <- query ( * ( a , c ) ) ] 178 ;
[ a <- - (a, da) ] 179 ;
[ c <- - (c, dc) ] 180 ;
[ j <- - (j, 1 ) ] 181
 };
[ x <- query ( 0 ) ] 182 ;
[ y <- query ( cs ) ] 183 ;
[ z <- query ( c ) ] 184 ;
[ w <- query ( z ) ] 185 ;
[ x <- query ( 0 ) ] 186 ;
[ y <- query ( x ) ] 187 ;
[ z <- query ( y ) ] 188 ;
[ w <- query ( z ) ] 189 ;
[ x <- query ( 0 ) ] 190 ;
[ y <- query ( x ) ] 191 ;
[ z <- query ( + (x, y) ) ] 192 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 193 ;
[ b <- + ( a, z ) ] 194 ;
[ c <- + ( a, b ) ] 195 ;
[ w <- query ( a ) ] 196 ;
[ x <- query ( b ) ] 197 ;
[ y <- query ( c ) ] 198 ;
[ z <- query ( z ) ] 199 ;
[ w <- query ( z ) ] 200 ;
[ j <- - (w, 1 ) ] 201 ;
[ x <- query ( 0 ) ] 202 ;
[ y <- query ( cs ) ] 203 ;
[ z <- query ( c ) ] 204 ;
[ w <- query ( z ) ] 205 ;
[ x <- query ( 0 ) ] 206 ;
[ y <- query ( x ) ] 207 ;
[ z <- query ( y ) ] 208 ;
[ w <- query ( z ) ] 209 ;
[ x <- query ( 0 ) ] 210 ;
[ d <- + ( x, w ) ] 211 ;
[ e <- + ( c, z ) ] 212 ;
[ f <- + ( a, b ) ] 213 ;
[ x <- query ( w ) ] 214 ;
```

```
[ y <- query ( x ) ] 215 ;
[ z <- query ( y ) ] 216 ;
[ x <- query ( z ) ] 217;
[ g <- + ( f, w ) ] 218 ;
[ h <- + ( c, x ) ] 219 ;
[ i <- + ( w, e ) ] 220 ;
[ z <- query ( x ) ] 221 ;
[ cs <- + ( cs, 0 ) ] 222 ;
[ ns <- + ( ns, 0 ) ] 223 ;
 [ i <- k ] 224 ;
[ x <- query ( z ) ] 225 ;
[ y <- query ( cs ) ] 226 ;
while [  > (i , 0) ] 227 do {
[ i <- - (i, 1) ] 228 ;
[ j <- k ] 229 ;
if [ > (x , 0) ] 230
then { [ y <- query ( + ( chi : x : , chi : y : )  ) ] 231 }
else { [ y <- query ( chi : x : ) ] 232 } ;
while [  > (j , 0) ] 233 do
{ [ j <- - (j, 1) ] 234 ;
[ x <- query ( + ( x, y) ) ] 235 }
 };
[ x <- query ( z ) ] 236 ;
[ y <- query ( cs ) ] 237 ;
[ z <- query ( c ) ] 238 ;
[ w <- query ( z ) ] 239 ;
[ y <- query ( x ) ] 240 ;
[ z <- query ( + (x, y) ) ] 241 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 242 ;
[ z <- query ( 0 ) ] 243 ;
if [ > (x , 0) ] 244
then { [ y <- query ( x ) ] 245 }
else { [ w <- query ( z ) ] 246 } ;
[ y <- query ( cs ) ] 247 ;
[ z <- query ( c ) ] 248 ;
[ w <- query ( z ) ] 249 ;
[ x <- query ( w ) ] 250 ;
[ y <- query ( x ) ] 251 ;
[ z <- query ( w ) ] 252 ;
[ w <- query ( z ) ] 253 ;
[ x <- query ( 0 ) ] 254 ;
[ y <- query ( x ) ] 255 ;
[ z <- query ( + (x, y) ) ] 256 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 257 ;
[ z <- query ( 0 ) ] 258 ;
[ x <- query ( w ) ] 259 ;
[ i <- k ] 260 ;
[ x <- query ( chi : cs : ) ] 261 ;
while [  > (i , 0) ] 262 do {
[ i <- - (i, 1) ] 263 ;
[ j <- k ] 264 ;
[ y <- query ( chi : x : ) ] 265 ;
while [  > (j , 0) ] 266 do
{ [ j <- - (j, 1) ] 267 ;
[ x <- query ( chi : x : ) ] 268 }
 };
 [ x <- query ( cs ) ] 269 ;
```

```
[ y <- query ( x ) ] 270 ;
[ z <- query ( ns ) ] 271 ;
[ w <- query ( z ) ] 272 ;
[ a <- x ] 273 ;
[ c <- z ] 274 ;
[ j <- k ] 275 ;
while [  < (0, j) ] 276 do {
[ da <- query ( * ( a , c ) ) ] 277 ;
[ dc <- query ( * ( a , c ) ) ] 278 ;
[ a <- - (a, da) ] 279 ;
[ c <- - (c, dc) ] 280 ;
[ j <- - (j, 1 ) ] 281
 };
[ x <- query ( a ) ] 282 ;
[ y <- query ( cs ) ] 283 ;
[ z <- query ( c ) ] 284 ;
[ w <- query ( z ) ] 285 ;
[ x <- query ( w ) ] 286 ;
[ y <- query ( x ) ] 287 ;
[ z <- query ( y ) ] 288 ;
[ w <- query ( z ) ] 289 ;
[ x <- query ( 0 ) ] 290 ;
[ y <- query ( x ) ] 291 ;
[ z <- query ( + (x, y) ) ] 292 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 293 ;
[ z <- query ( 0 ) ] 294 ;
[ x <- query ( w ) ] 295 ;
if [ > (x , 0) ] 296
then { [ y <- query ( 0 ) ] 297 }
else { [ w <- query ( 0 ) ] 298 } ;
[ a <- x ] 299 ;
[ c <- z ] 300 ;
[ j <- k ] 301 ;
[ cs <- + ( cs, 0 ) ] 302 ;
[ ns <- + ( ns, 0 ) ] 303 ;
 [ i <- k ] 304 ;
[ x <- query ( 0 ) ] 305 ;
[ y <- query ( cs ) ] 306 ;
while [  > (i , 0) ] 307 do {
[ i <- - (i, 1) ] 308 ;
[ j <- k ] 309 ;
if [ > (x , 0) ] 310
then { [ y <- query ( + ( chi : x : , chi : y : )  ) ] 311 }
else { [ y <- query ( chi : x : ) ] 312 } ;
while [  > (j , 0) ] 313 do
{ [ j <- - (j, 1) ] 314 ;
[ x <- query ( + ( x, y) ) ] 315 }
 };
[ x <- query ( c ) ] 316 ;
[ y <- query ( cs ) ] 317 ;
[ z <- query ( c ) ] 318 ;
[ w <- query ( z ) ] 319 ;
[ y <- query ( x ) ] 320 ;
[ z <- query ( + (x, y) ) ] 321 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 322 ;
[ z <- query ( 0 ) ] 323 ;
if [ > (x , 0) ] 324
```

```
then { [ y <- query ( w ) ] 325 }
else { [ w <- query ( z ) ] 326 } ;
[ y <- query ( cs ) ] 327 ;
[ z <- query ( c ) ] 328 ;
[ w <- query ( z ) ] 329 ;
[ x <- query ( 0 ) ] 330 ;
[ y <- query ( x ) ] 331 ;
[ z <- query ( y ) ] 332 ;
[ w <- query ( z ) ] 333 ;
[ x <- query ( y ) ] 334 ;
[ y <- query ( x ) ] 335 ;
[ z <- query ( + (x, y) ) ] 336 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 337 ;
[ z <- query ( 0 ) ] 338 ;
[ x <- query ( w ) ] 339 ;
[ i <- k ] 340 ;
[ x <- query ( chi : cs : ) ] 341 ;
while [  > (i , 0) ] 342 do {
[ i <- - (i, 1) ] 343 ;
[ j <- k ] 344 ;
[ y <- query ( chi : x : ) ] 345 ;
while [  > (j , 0) ] 346 do
{ [ j <- - (j, 1) ] 347 ;
[ x <- query ( chi : x : ) ] 348 }
 };
 [ x <- query ( cs ) ] 349 ;
[ y <- query ( x ) ] 350 ;
[ z <- query ( ns ) ] 351 ;
[ w <- query ( z ) ] 352 ;
[ a <- x ] 353 ;
[ c <- z ] 354 ;
[ j <- k ] 355 ;
while [  < (0, j) ] 356 do {
[ da <- query ( * ( a , c ) ) ] 357 ;
[ dc <- query ( * ( a , c ) ) ] 358 ;
[ a <- - (a, da) ] 359 ;
[ c <- - (c, dc) ] 360 ;
[ j <- - (j, 1 ) ] 361
 };
[ x <- query ( 0 ) ] 362 ;
[ y <- query ( cs ) ] 363 ;
[ z <- query ( c ) ] 364 ;
[ w <- query ( z ) ] 365 ;
[ x <- query ( 0 ) ] 366 ;
[ y <- query ( x ) ] 367 ;
[ z <- query ( y ) ] 368 ;
[ w <- query ( z ) ] 369 ;
[ x <- query ( 0 ) ] 370 ;
[ y <- query ( x ) ] 371 ;
[ z <- query ( + (x, y) ) ] 372 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 373 ;
[ z <- query ( 0 ) ] 374 ;
[ x <- query ( w ) ] 375 ;
if [ > (x , 0) ] 376
then { [ y <- query ( x ) ] 377 }
else { [ w <- query ( 0 ) ] 378 } ;
[ a <- x ] 379 ;
```

```
[ c <- z ] 380 ;
[ j <- k ] 381 ;
[ cs <- + ( cs, 0 ) ] 382 ;
[ ns <- + ( ns, 0 ) ] 383 ;
 [ i <- k ] 384 ;
[ x <- query ( 0 ) ] 385 ;
[ y <- query ( cs ) ] 386 ;
while [  > (i , 0) ] 387 do {
[ i <- - (i, 1) ] 388 ;
[ j <- k ] 389 ;
if [ > (x , 0) ] 390
then { [ y <- query ( + ( chi : x : , chi : y : )  ) ] 391 }
else { [ y <- query ( chi : x : ) ] 392 } ;
while [  > (j , 0) ] 393 do
{ [ j <- - (j, 1) ] 394 ;
[ x <- query ( + ( x, y) ) ] 395 }
 };
[ x <- query ( 0 ) ] 396 ;
[ y <- query ( cs ) ] 397 ;
[ z <- query ( c ) ] 398 ;
[ w <- query ( z ) ] 399 ;
[ y <- query ( x ) ] 400 ;
[ z <- query ( + (x, y) ) ] 401 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 402 ;
[ z <- query ( 0 ) ] 403 ;
if [ > (x , 0) ] 404
then { [ y <- query ( w ) ] 405 }
else { [ w <- query ( z ) ] 406 } ;
[ y <- query ( cs ) ] 407 ;
[ z <- query ( c ) ] 408 ;
[ w <- query ( z ) ] 409 ;
[ x <- query ( 0 ) ] 410 ;
if [ > (w , 0) ] 411
then { [ y <- w ] 412 }
else { [ w <- query ( z ) ] 413 } ;
[ x <- query ( w ) ] 414 ;
[ y <- query ( x ) ] 415 ;
[ z <- query ( y ) ] 416 ;
[ x <- query ( z ) ] 417;
[ g <- + ( f, w ) ] 418 ;
[ h <- + ( c, x ) ] 419 ;
[ i <- + ( w, e ) ] 420 ;
[ z <- query ( x ) ] 421 ;
[ cs <- + ( cs, 0 ) ] 422 ;
[ ns <- + ( ns, 0 ) ] 423 ;
 [ i <- k ] 424 ;
[ x <- query ( z ) ] 425 ;
[ y <- query ( cs ) ] 426 ;
while [  > (i , 0) ] 427 do {
[ i <- - (i, 1) ] 428 ;
[ j <- k ] 429 ;
if [ > (x , 0) ] 430
then { [ y <- query ( + ( chi : x : , chi : y : )  ) ] 431 }
else { [ y <- query ( chi : x : ) ] 432 } ;
while [  > (j , 0) ] 433 do
{ [ j <- - (j, 1) ] 434 ;
[ x <- query ( + ( x, y) ) ] 435 }
```

```
 };
[ x <- query ( z ) ] 436 ;
[ y <- query ( cs ) ] 437 ;
[ z <- query ( c ) ] 438 ;
[ w <- query ( z ) ] 439 ;
[ y <- query ( w ) ] 440 ;
[ z <- query ( + (x, y) ) ] 441 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 442 ;
[ z <- query ( 0 ) ] 443 ;
if [ > (x , 0) ] 444
then { [ y <- query ( x ) ] 445 }
else { [ w <- query ( z ) ] 446 } ;
[ y <- query ( cs ) ] 447 ;
[ z <- query ( c ) ] 448 ;
[ w <- query ( z ) ] 449 ;
[ x <- query ( w ) ] 450 ;
[ y <- query ( x ) ] 451 ;
[ z <- query ( w ) ] 452 ;
[ w <- query ( z ) ] 453 ;
[ x <- query ( 0 ) ] 454 ;
[ y <- query ( x ) ] 455 ;
[ z <- query ( + (x, y) ) ] 456 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 457 ;
[ z <- query ( 0 ) ] 458 ;
[ x <- query ( w ) ] 459 ;
[ i <- k ] 460 ;
[ x <- query ( chi : cs : ) ] 461 ;
while [  > (i , 0) ] 462 do {
[ i <- - (i, 1) ] 463 ;
[ j <- k ] 464 ;
[ y <- query ( chi : x : ) ] 465 ;
while [  > (j , 0) ] 466 do
{ [ j <- - (j, 1) ] 467 ;
[ x <- query ( chi : x : ) ] 468 }
 };
 [ x <- query ( cs ) ] 469 ;
[ y <- query ( x ) ] 470 ;
[ z <- query ( ns ) ] 471 ;
[ w <- query ( z ) ] 472 ;
[ a <- x ] 473 ;
[ c <- z ] 474 ;
[ j <- k ] 475 ;
while [  < (0, j) ] 476 do {
[ da <- query ( * ( a , c ) ) ] 477 ;
[ dc <- query ( * ( a , c ) ) ] 478 ;
[ a <- - (a, da) ] 479 ;
[ c <- - (c, dc) ] 480 ;
[ j <- - (j, 1 ) ] 481
 };
[ x <- query ( a ) ] 482 ;
[ y <- query ( cs ) ] 483 ;
[ z <- query ( c ) ] 484 ;
[ w <- query ( z ) ] 485 ;
[ x <- query ( w ) ] 486 ;
[ y <- query ( x ) ] 487 ;
[ z <- query ( y ) ] 488 ;
[ w <- query ( z ) ] 489 ;
```

```
[ x <- query ( 0 ) ] 490 ;
[ y <- query ( x ) ] 491 ;
[ z <- query ( + (x, y) ) ] 492 ;
[ w <- query ( * (chi : y : , chi : z :) ) ] 493 ;
[ z <- query ( 0 ) ] 494 ;
[ x <- query ( w ) ] 495 ;
if [ > (x , 0) ] 496
then { [ y <- query ( 0 ) ] 497 }
else { [ w <- query ( 0 ) ] 498 } ;
[ a <- x ] 499 ;
[ c <- z ] 500 ;
[ r <- query ( c ) ] 501
```

References

[1] Patrick Cousot. Abstract semantic dependency. In Bor-Yuh Evan Chang, editor, *Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, volume 11822 of *Lecture Notes in Computer Science*, pages 389–410. Springer, 2019.

[2] Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Roth. Generalization in adaptive data analysis and holdout reuse. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 2350–2358, 2015.

[3] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 292–304. ACM, 2010.

[4] Ryan Rogers, Aaron Roth, Adam D. Smith, Nathan Srebro, Om Thakkar, and Blake E. Woodworth. Guaranteed validity for empirical approaches to adaptive data analysis. In Silvia Chiappa and Roberto Calandra, editors, *The 23rd International Conference on Artificial Intelligence and Statistics, AISTATS 2020, 26-28 August 2020, Online [Palermo, Sicily, Italy]*, volume 108 of *Proceedings of Machine Learning Research*, pages 2830–2840. PMLR, 2020.

[5] Moritz Sinn, Florian Zuleger, and Helmut Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 745–761. Springer, 2014.

[6] 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.

[7] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In Eran Yahav, editor, *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, volume 6887 of *Lecture Notes in Computer Science*, pages 280–297. Springer, 2011.