Program Analysis for Adaptivity Analysis

In this appendix, we present the full details of the 2 languages: while language and the SSA language.

Contents

1 System Overview

In adaptive data analysis, a data analysis can depend on the results of previous analysis over the same data. This dependency may affect the *generalization properties of the data analysis*. To study this phenomenon in a formal way, we consider the *statistical query model*. In this model, a dataset D consisting of d attributes (columns) and n individuals' data (rows) can be accessed only through an

interface to which one can submit statistical queries. More precisely, suppose that the type of a row is R (as an example, a row with d binary attributes would have type $R = \{0,1\}^d$. Then, in the statistical query model one can access the dataset only by submitting a query to the interface, in the form of a function $p : D \to [0,1]$ where D represents dataset. The collected answer of the asked query is the average result of p on each row in the dataset D. For example, the result is the value $\frac{1}{n}\sum_{i=1}^{n} p(D_i)$ where D_i is the row of index i in D. While this model is rather simple, in fact it supports sufficient statistics one may be interested.

We are interested in the adaptivity of mechanisms in the model, which is straightforward supported by a high level language. In this language, queries are allowed to carry arguments to simulate the process of submitting a query to the interface in the model, for example, the expression $\texttt{query}(\psi)$ tells us the argument ψ is consumed to construct the query. To be precise, one submitted query who needs the average of answers of previous queries is expressed as $\texttt{query}(x)$, where the variable x stores the expected average results.This makes these mechanisms quite straightforward to express in the high level language. However, this convenience pays at the price that the adaptivity A of a mechanism P becomes quite tricky to estimate because the definition of dependency between two queries becomes vague in the high level language.

$$
\begin{aligned}
&x \leftarrow \texttt{query}(0);\\
&\texttt{if } (x_1 > 0)\\
&y \leftarrow \texttt{query}(x)
\end{aligned}
$$

The dependency between two query submissions is the essential of the adaptivity of a mechanism. To study the dependency, we first study its dual, independence between two queries, which is defined to be: one query $\texttt{query}(0)$ does not depend on another query $\texttt{query}(x)$ when the result of $\texttt{query}(0)$ remains the same regardless of the modification of the result of $\texttt{query}(x)$. Hence, it becomes hard to distinguish whether the variance of result of $\texttt{query}(0)$ comes from the control flow or the argument of queries. Since we know that the result of one query from a specific D may vary under different contexts in the high level language.

[[To resolve the dilemma, we translate any program(mechanism) into its counterpart in a low level language, which mimics the high level one except its only allowing atomic queries, – $\texttt{query}(0)$ –. That is to say, given a data base D, the result of the query from D becomes deterministic. We need to show the two programs P and P^* are observably equivalent over the translation. In this way, we can define the adaptivity of a program under this model only based on the control flow. To be specific, the adaptivity A of a program P is defined based on graphs, called dependency graph, which comes from the semantics of the low level program.]] The dependency graph is constructed using a trace of queries generated along with the semantics: The queries in the trace consists of the nodes in the graph while the edge represents dependency. If there is no dependency between two node(queries), there will be no edge. Intuitively, we want to give an approximation of the adaptivity by static analysis. To this end, we propose AdaptFun, which estimates an upper bound on the program.

[[The adaptivity A of arbitrary high level program c is defined to be the minimal of the adaptivity A of all the possible c via various valid translations. Being valid means the programs before and after the translation are observably equivalent. Naturally, following this definition, the upper bound estimated by AdaptFun is sound with respect to its low level adaptivity A, hence the high level one A.]]

Finally we extend the language to support the probabilistic program and extend the adaptivity definition accordingly.

The key component of the system is a program analysis tool, which provides an upper bound on the adaptivity of the program.

2 Labeled `While` Language

2.1 Syntax and Semantics

Arithmetic Operators	$\oplus_a$	::=	$+ \mid - \mid \times \mid \div$
Boolean Operators	$\oplus_b$	::=	$\vee \mid \wedge$
Relational Operators	$\sim$	::=	$< \mid \leq \mid ==$
Label	l	$\in$	$\mathbb{N}$
Arithmetic Expressions	a	::=	$n \mid x \mid a \oplus_a a \mid$
Boolean Expressions	b	::=	$\texttt{true} \mid \texttt{false} \mid \neg b \mid b \oplus_b b \mid a \sim a$
Value	v	::=	$n \mid \texttt{true} \mid \texttt{false} \mid [] \mid [v, \ldots, v]$
Expression	e	::=	$v \mid a \mid b \mid [e, \ldots, e]$
Query Value	α	::=	$n \mid \chi[n] \oplus_a \chi[n] \mid n \oplus_a \chi[n] \mid \chi[n] \oplus_a n$
Query Expression	ψ	::=	$\alpha \mid a \mid \psi \oplus_a \psi$
Labeled Command	c	::=	$[\texttt{skip}]^l \mid [x \leftarrow e]^l \mid [x \leftarrow \texttt{query}\ (\psi)]^l$
			$\mid \texttt{while}\ [b]^l, n\ \texttt{do}\ c \mid c;c \mid \texttt{if}\ ([b]^l, c, c)$
Memory	m	::=	$[] \mid (x^l \rightarrow v) :: m$
Annotated Variable	$\texttt{av}$	::=	(x, v, l, n)
Variable Trace	τ	::=	$[] \mid \texttt{av} :: \tau$
Variable Counter	$\texttt{vcnt}$	::=	$\mathcal{VAR} \rightarrow \mathbb{N}$

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

$\mathcal{VAR}$	:	Set of Variables
$\mathcal{VAL}$	:	Set of Values
$\mathcal{AV}$	:	Set of Annotated Variables
$\mathcal{M}$	:	Set of Memories
$\mathcal{DB}$	:	Set of Databases
$\mathcal{QD} = [-1, 1]$	:	Domain of Query Results

[[

Definition 1 (Assigned Variables (`aVar`)). *Given a program c, its assigned variables* $\texttt{aVar}_c$ *is a vector containing all variables newly assigned in the program preserving the order,* $\forall \mathbf{x} \in \texttt{aVar}, \mathbf{x} \in \mathcal{SVAR}$. *It is defined as follows:*

$$\texttt{aVar}_\mathbf{c} \triangleq \begin{cases} [\mathbf{x}] & \mathbf{c} = [\mathbf{x} \leftarrow \mathbf{e}]^{(l,w)} \\ [\mathbf{x}] & \mathbf{c} = [\mathbf{x} \leftarrow \texttt{query}\ (\boldsymbol{\psi})]^{(l,w)} \\ \texttt{aVar}_{\mathbf{c_1}} + + \texttt{aVar}_{\mathbf{c_2}} & \mathbf{c} = \mathbf{c_1}; \mathbf{c_2} \\ \texttt{aVar}_{\mathbf{c_1}} + + \texttt{aVar}_{\mathbf{c_2}} & \mathbf{c} = \texttt{if}\ ([b]^{(l,w)}, c_1, c_2) \\ \texttt{aVar}_{\mathbf{c'}} & \mathbf{c} = \texttt{while}\ [b]^{(l,w)}, n\ \texttt{do}\ c' \end{cases}$$

Definition 2 (Query Variables (`qVar`)). .
Given a program c, its query variables `qVar` *is a vector containing all variables newly assigned by a*

query in the programm, $\texttt{qVar} \subset \mathcal{VAR}$*. It is defined as follows:*

$$\texttt{qVar}_{\mathbf{c}} \triangleq \begin{cases} [] & \mathbf{c} = [\mathbf{x} \leftarrow \mathbf{e}]^{(l,w)} \\ [\mathbf{x}] & \mathbf{c} = [\mathbf{x} \leftarrow \texttt{query}\ (\boldsymbol{\psi})]^{(l,w)} \\ \texttt{qVar}_{\mathbf{c_1}} + + \texttt{qVar}_{\mathbf{c_2}} & \mathbf{c} = \mathbf{c_1}; \mathbf{c_2} \\ \texttt{qVar}_{\mathbf{c_1}} + + \texttt{qVar}_{\mathbf{c_2}} & \mathbf{c} = \texttt{if}\ ([\mathbf{b}]^{(l,w)}, c_1, c_2) \\ \texttt{qVar}_{\mathbf{c'}} & \mathbf{c} = \texttt{while}\ [b]^{(l,w)}, n\ \texttt{do}\ c' \end{cases}$$

We are abusing the notations and operators from list here. The notation [] **represents an empty vector and** $x :: A$ **represents add an element** x **to the head of the vector** A**. The concatenation operation between 2 vectors** A_1 **and** A_2**, i.e.,** $A_1 + + A_2$ **is mimic the standard list concatenation operations as follows:**

$$A_1 + + A_2 \triangleq \begin{cases} A_2 & A_1 = [] \\ x :: (A_1' + + A_2) & A_1 = x :: A_1' \end{cases} \tag{1}$$

We use index within parenthesis to denote the access to the element of corresponding location, $A(i)$ **denotes the element at location** i **in the vector** A **and** $M(i, j)$ **denotes the element at location** i**-th raw,** i**-th column in the matrix** M**.**
The variable counter $\texttt{vcnt}_c$ **maps every assigned variables** $\texttt{aVar}_c$ **to a natural number** $n \in \mathbb{N}$ **in a certain execution. This natural number** n **represents the visiting times of this variable in this certain execution.**
We use variable name x **within parenthesis to denote the access to the associated natural number of this variable in the variable counter** $\texttt{vcnt}_c$**,** $\texttt{vcnt}_c(x)$ **denote the visiting times of variable** x**.**

Definition 3 (Initial Variable Counter $\texttt{vcnt}_c^0$)**.** *Given a program* c *with its assigned variables* $\texttt{aVar}_c$ *of length* N*, its initial variable counter* $\texttt{vcnt}_c^0$ *maps all the variable to* 0*, i.e.:*

$$\texttt{vcnt}_c^0(x) = 0, x = \texttt{aVar}_c(i) \forall i = 1, \ldots, N$$

]]

2.2 Trace-based Operational Semantics

We evaluate programs in the `While` language by means of a trace-based operational semantics, to capture the dependency between queries. For distinguishing elements in the the trace, we add a label to commands in the `While` language as defined in the syntax. Each command is labeled with a label l, a natural number standing for the line of code where the command appears. Notice that we associate the label l to the conditional predicate b in the if statement, and to the guard b in the while statement. Some non-standard syntax is explained as follows:

A memory is standard, a map from labeled variables to values. Queries can be uniquely annotated as defined in $\mathcal{AQ}$, and the annotation (l, w) considers the location of the query by line number l and which iteration the query is at when it appears in a while statement, specified by w.

A configuration, $\langle m, c, t, w \rangle$, contains four elements: a memory m, the command c to be evaluated, a starting trace t, a starting while map w. Most of the time, the while maps remains empty until the evaluation goes into while statements.
[[The annotated query $\texttt{av} = (\alpha, v, l, n)$ **is a tuple contains 4 elements.]]**

Definition 4 (Order of Annotated Variables)**.** .
Given 2 annotated queries $\mathtt{av}_1 = (x_1, v_1, l_1, n_1), \mathtt{av}_2 = (x_2, v_2, l_2, n_2)$ *:*

$$\mathtt{av}_1 <_{\mathtt{av}} \mathtt{av}_2 \triangleq \begin{cases} n_1 < n_2 & l_1 = l_2 \\ w_1 <_w w_2 & o.w. [WQ : hard : -() \end{cases}$$

$\mathtt{av}_1 \geq_{\mathtt{av}} \mathtt{av}_2$ *is defined vice versa.*

A variable trace τ is a list of annotated queries accumulated along the execution of the program. A trace can be regarded as the program history, where this history consists of all the queries asked by the analyst during the execution of the program. We collect the trace with a trace-based small-step operational semantics based on transitions of the program configuration $\langle m, c, \tau, \mathtt{vcnt}\rangle$, of form $\langle m, c, \tau, \mathtt{vcnt}\rangle \rightarrow \langle m', \mathtt{skip}, \tau', \mathtt{vcnt}'\rangle$. The evaluation rules for arithmetic and boolean expressions are standard. They have the form $\langle m, a\rangle \rightarrow_a a'$, evaluating an arithmetic expression a in the memory m, and similar for the boolean expressions $\langle m, b\rangle \rightarrow_b b'$, defined as follows:

$$\boxed{\langle m, a\rangle \rightarrow_a a' : Memory \times AExpr \Rightarrow AExpr}$$

$$\boxed{\langle m, b\rangle \rightarrow_b b' : Memory \times BExpr \Rightarrow BExpr}$$

Given the evaluation for the arithmetic and boolean expression, we defined the evaluation rules for query expression ψ correspondingly as follows:

$$\boxed{\langle m, \psi\rangle \rightarrow_q \psi' : Memory \times QExpr \rightarrow_q QExpr}$$

$$\frac{\langle m, n \oplus_a n\rangle \rightarrow_a n'}{\langle m, n \oplus_a n\rangle \rightarrow_q n'} \qquad \frac{\langle m, \psi\rangle \rightarrow_q \psi'}{\langle m, \psi \oplus_a \alpha\rangle \rightarrow_q \psi' \oplus_a \alpha} \qquad \frac{\langle m, \psi_2\rangle \rightarrow_q \psi_2'}{\langle m, \psi_1 \oplus_a \psi_2\rangle \rightarrow_q \psi_1 \oplus_a \psi_2'}$$

$$\frac{\langle m, a\rangle \rightarrow_a a'}{\langle m, \chi[a]\rangle \rightarrow_q \chi[a']} \qquad \frac{\langle m, a\rangle \rightarrow_a a'}{\langle m, a\rangle \rightarrow_q a'}$$

Given the evaluation rules for query expression, we can define its equivalence relation, as follows in Definition 17.

Definition 5 (Equivalence of Query)**.** . *Given a memory* m *and 2 query expressions* ψ_1*,* ψ_2 *s.t.,* $FV(\psi_1) \in \mathtt{dom}(m)$ *and* $FV(\psi_2) \in \mathtt{dom}(m)$*:*

$$\psi_1 =_q^m \psi_2 \triangleq \begin{cases} \mathtt{true} & \exists \alpha_1, \alpha_2. \ (\langle m, \psi_1\rangle \rightarrow_q \alpha_1 \wedge \langle m, \psi_2\rangle \rightarrow_q \alpha_2) \\ & \wedge (\forall r \in \mathcal{QD}. \exists v. \ s.t., \ \langle m, \alpha_1[r/\chi]\rangle \rightarrow_a v \wedge \langle m, \alpha_2[r/\chi]\rangle \rightarrow_a v) \\ \mathtt{false} & o.w. \end{cases}$$

, where $FV(\psi)$ *is the set of free variables in the query expression* ψ*.* $\psi_1 \neq_q^m \psi_2$ *is defined vice versa. We use* $=_q$ *and* $\neq_q$ *as the shorthands for* $=_q^{[]}$ *and* $\neq_q^{[]}$*.*

Then, we have the corresponding equivalence relation between 2 annotated queries defined in Definition 18: [[

Definition 6 (Equivalence of Annotated Variables)**.** *Given 2 annotated queries* $\mathtt{av}_1 = (x_1, v_1, l_1, n_1), \mathtt{av}_2 = (x_2, v_2, l_2, n_2)$ *:*

$$\mathtt{av}_1 =_{\mathtt{av}} \mathtt{av}_2 \triangleq (l_1 = l_2 \wedge w_1 =_w w_2 \wedge \alpha_1 =_q \alpha_2)$$

$\mathtt{av}_1 \neq_{\mathtt{av}} \mathtt{av}_2$ *is defined vice versa.*

]] [[**Given an annotated variable** $\mathtt{av}$ **and a trace** t**, the appending operation** $\mathtt{av} :: t$ **is the standard list appending operation, appends** $\mathtt{av}$ **to the head of trace** t**. The concatenation operation between 2 traces** t_1 **and** t_2**, i.e.,** $t_1 + + t_2$ **is the standard list concatenation operation as follows:**

$$t_1 + + t_2 \triangleq \begin{cases} t_2 & t_1 = [] \\ \mathtt{av} :: (t_1' + + t_2) & t_1 = \mathtt{av} :: t_1' \end{cases} \tag{2}$$

The subtraction operation between 2 traces t_1 **and** t_2**, i.e.,** $t_1 - t_2$ **is defined as follows:**

$$t_1 - t_2 \triangleq t_3 \ s.t., t_2 + + t_3 = t_1 \tag{3}$$

Given an annotated query $\mathtt{av}$**,** $\mathtt{av}$ **belongs to a trace** t**, i.e.,** $\mathtt{av} \in_{\mathtt{av}} t$ **are defined as follows:**

$$\mathtt{av} \in_{\mathtt{av}} t \triangleq \begin{cases} \mathtt{false} & t = [] \\ \mathtt{true} & t = \mathtt{av}' :: t' \quad \mathtt{av} =_{\mathtt{av}} \mathtt{av}' \\ \mathtt{av} \in t' & t = \mathtt{av}' :: t' \quad \mathtt{av} \neq_{\mathtt{av}} \mathtt{av}' \end{cases} \tag{4}$$

]] [[

Definition 7 (Equivalence of Program)**.** *Given 2 programs* c_1 *and* c_2*:*

$$c_1 =_c c_2 \triangleq \begin{cases} \mathtt{true} & c_1 = \mathtt{skip} \wedge c_2 = \mathtt{skip} \\ \forall m. \exists v. \langle m, e_1 \rangle \rightarrow_a^* v \wedge \langle m, e_1 \rangle \rightarrow_a^* v & c_1 = x \leftarrow e_1 \wedge c_2 = x \leftarrow e_2 \\ \psi_1 =_q \psi_2 & c_1 = x \leftarrow \mathtt{query}(\psi_1) \wedge c_1 = x \leftarrow \mathtt{query}(\psi_2) \\ c_1^f =_c c_2^f \wedge c_1^t =_c c_2^t & c_1 = \mathtt{if}\ (b, c_1^t, c_1^f) \wedge c_2 = \mathtt{if}\ (b, c_2^t, c_2^f) \\ c_1' =_c c_2' & c_1 = \mathtt{while}\ b\ \mathtt{do}\ c_1' \wedge c_2 = \mathtt{while}\ b\ \mathtt{do}\ c_2' \\ c_1^h =_c c_2^h \wedge c_1^t =_c c_2^t & c_1 = c_1^h; c_1^t \wedge c_2 = c_2^h; c_2^t \end{cases}$$

$c_1 \neq_c c_2$ *is defined vice versa.*

Given 2 programs c **and** c'**,** c' **is a sub-program of**c**, i.e.,** $c' \in_c c$ **is defined as:**

$$c' \in_c c \triangleq \exists c_1, c_2, c''. \ s.t., \ c =_c c_1; c'' ; c_2 \wedge c' =_c c'' \tag{5}$$

]]

The small-step transition states that a configuration $\langle m, c, t, w \rangle$ evaluates to another configuration with the trace and while map updated along with the evaluation of the command c to the normal form of the command $\mathtt{skip}$. We define rules of the trace-based operational semantics in Figure 1. The rule **query-e** evaluates the argument of a query request. When the argument is in normal form, this query will be answered. The rule **query-v** modifies the starting memory m to $m[\alpha/x]$ using the answer α of the query $\mathtt{query}(\alpha)$ from the mechanism, with the trace expanded by appending the query $\mathtt{query}(\alpha)$ with the current annotation (l, w). The rule for assignment is standard and the trace remains unchanged.The sequence rule keeps tracking the modification of the trace, and the evaluation rule for if conditional goes into one branch based on the result of the conditional predicate b. The rules for while modify the while map w. In the rule **ifw-true**, the while map w is updated by $w + l$ because the execution goes into another iteration when the condition $n > 0$ is satisfied. When n reaches 0, the loop exits and the while map w eliminates the label l of this while statement by $w - l$ in the rule **ifw-false**. With the operational semantics and relations between annotated queries, we restrict the well-formed trace w.r.t. the execution of a program c in Definition 23. [WQ: we define map update as follows, if we update the map m with value v at its key k, we denote $m[x \rightarrow v]$ or $m[v/x]$.]

[WQ: Evaluation context $E \quad ::= [\cdot] | x \leftarrow E | if\ E\ c_1, c_2 | x \leftarrow query(E) | while\ E\ do\ c | E; c | skip; E$

$$\frac{}{\langle m, E[e], \tau, \mathtt{vcnt} \rangle \rightarrow \langle m, E[e']\tau, \mathtt{vcnt} \rangle}$$

]

[JL:

$$\boxed{Memory \times Command \times VTrace \times VCounter \rightarrow Memory \times Command \times VTrace \times VCounter}$$

$$\boxed{\langle m, c, \tau, \texttt{vcnt} \rangle \rightarrow \langle m', c', \tau', \texttt{vcnt}' \rangle}$$

$$\frac{\langle m, e \rangle \rightarrow \langle m, e' \rangle}{\langle m, [x \leftarrow e]^l, \tau, \texttt{vcnt} \rangle \rightarrow \langle m, [x \leftarrow e']^l, \tau, \texttt{vcnt} \rangle} \textbf{ assn-e}$$

$$\frac{n = \texttt{vcnt}[x] + 1 \qquad \texttt{av} = (x, v, l, n)}{\langle m, [x \leftarrow v]^l, \tau, \texttt{vcnt} \rangle \rightarrow \langle m[v/x], [\texttt{skip}]^l, \tau + +[\texttt{av}], \texttt{vcnt}[x \rightarrow n] \rangle} \textbf{ assn-v}$$

$$\frac{}{\langle m, \texttt{while } [b]^l, n \texttt{ do } c, \tau, \texttt{vcnt} \rangle \rightarrow \langle m, c; \texttt{if } (b, c; \texttt{while } [b]^l, (n+1) \texttt{ do } c, \texttt{skip}), \tau, \texttt{vcnt} \rangle} \textbf{ while-b}$$

$$\frac{\langle m, \psi \rangle \rightarrow_q \psi'}{\langle m, [x \leftarrow \texttt{query}(\psi)]^l, \tau, \texttt{vcnt} \rangle \rightarrow \langle m, [x \leftarrow \texttt{query}(\psi')]^l, \tau, \texttt{vcnt} \rangle} \textbf{ query-e}$$

$$\frac{\texttt{query}(\alpha) = v \qquad n = \texttt{vcnt}[x] + 1 \qquad \texttt{av} = (x, \alpha, l, n)}{\langle m, [x \leftarrow \texttt{query}(\alpha)]^l, \tau, \texttt{vcnt} \rangle \rightarrow \langle m[v/x], \texttt{skip}, \tau + +[\texttt{av}], \texttt{vcnt}[x \rightarrow n] \rangle} \textbf{ query-v}$$

$$\frac{\langle m, c_1, \tau, \texttt{vcnt} \rangle \rightarrow \langle m', c_1', \tau', \texttt{vcnt}' \rangle}{\langle m, c_1; c_2, \tau, \texttt{vcnt} \rangle \rightarrow \langle m', c_1'; c_2, \tau', \texttt{vcnt}' \rangle} \textbf{ seq1} \qquad \frac{}{\langle m, [\texttt{skip}]^l; c_2, \tau, \texttt{vcnt} \rangle \rightarrow \langle m, c_2, \tau, \texttt{vcnt} \rangle} \textbf{ seq2}$$

$$\frac{\langle m, b \rangle \rightarrow_b b'}{\langle m, \texttt{if } ([b]^l, c_1, c_2), \tau, \texttt{vcnt} \rangle \rightarrow \langle m, \texttt{if } ([b']^l, c_1, c_2), \tau, \texttt{vcnt} \rangle} \textbf{ if-b}$$

$$\frac{}{\langle m, \texttt{if } ([\texttt{true}]^l, c_1, c_2), \tau, \texttt{vcnt} \rangle \rightarrow \langle m, c_1, \tau, \texttt{vcnt} \rangle} \textbf{ if-t}$$

$$\frac{}{\langle m, \texttt{if } ([\texttt{false}]^l, c_1, c_2), \tau, \texttt{vcnt} \rangle \rightarrow \langle m, c_2, \tau, \texttt{vcnt} \rangle} \textbf{ if-f}$$

]

Figure 1: Trace-based Operational Semantics of `While` Language.

2.3 Trace-based Adaptivity

We define adaptivity through a query-based dependency graph. In our model, an *analyst* asks a sequence of queries to the mechanism, and the analyst receives the answers to these queries from the mechanism. A query is adaptively chosen by the analyst when the choice of this query is affected by answers from previous queries. In this model, the adaptivity we are interested in is the length of the longest sequence of such adaptively chosen queries, among all the queries the data analyst asks to the mechanism. Also, when the analyst asks a query, the only information the analyst will have will be the answers to previous queries and the state of the program. It means that when we want to know if this query is adaptively chosen, we only need to check whether the choice of this query will be affected by changes of answers to previous queries. There are two possible situations that can affect the choice of a query, either the query argument directly uses the results of previous queries (data dependency), or the control flow of the program with respect to a query (whether to ask this query or not) depends on the results of previous queries (control flow dependency).

As a first step, we give a definition of when one query may depend on a previous query, which is supposed to consider both control dependency and data dependency. We first look at two possible candidates:

1. One query may depend on a previous query if and only if a change of the answer to the previous query may also change the result of the query.

2. One query may depend on a previous query if and only if a change of the answer to the previous query may also change the appearance of the query.

The first candidate works well by witnessing the result of one query according to the change of the answer of another query. We can easily find that the two queries have nothing to do with each other in a simple example $c = x \leftarrow \texttt{query}(\chi(1)); y \leftarrow \texttt{query}(\chi(2))$. This candidate definition works well with respect to data dependency. However, if fails to handle control dependency since it just monitors the changes to the answer of a query when the answer of previous queries returned change. The key point is that this query may also not be asked because of an analyst decision which depend on the answers of previous queries. An example of this situation is shown in program c_1 as follows.

$$c_1 = x \leftarrow \texttt{query}(\chi(1)); \texttt{ if } (x > 2, y \leftarrow \texttt{query}(\chi(2)), \texttt{skip})$$

We choose the second candidate, which performs well by witnessing the appearance of one query $\texttt{query}(\chi(2))$ upon the change of the result of one previous query $\texttt{query}(\chi(1))$ in c_1. It considers the control dependency, and at the same, does not miss the data dependency. In particular, the arguments of a query characterizes it. In this sense, if the data used in the arguments changes due to a different answer to a certain previous query, the appearance of the query may change as well. This situation is also captured by our definition. Let us look at another variant of program c, p_2, in which the queries equipped with functions using previously assigned variables storing answer of its previous query.

$$c_2 = x \leftarrow \texttt{query}(\chi(2)); y \leftarrow \texttt{query}(x + \chi(3))$$

As a reminder, in the `While` language, the query request is composed by two components: a symbol `query` representing a linear query type and the argument e, which represents the function specifying what the query asks. So we do think $\texttt{query}(\chi(1))$ is different from $\texttt{query}(\chi(2))$. Informally, we think $\texttt{query}(x + \chi(3))$ may depend on the query $\texttt{query}(\chi(2))$, because equipped function of the former $x + \chi(3)$ depend on the data assigned with $\texttt{query}(\chi(2))$. We can see the appearance definition catches

data dependency in such a way, since $\texttt{query}(x+\chi(2))$ will not be the same query if the value of x is changed.

We give a formal definition of variable may dependency based on the trace-based operational semantics as follows. [JL:

Definition 8 (Annotated Variables May Dependency)**.** .
One annotated variable $\texttt{av}_2 = (x_2, v_2, l_2, n_2)$ *may depend on another one* $\texttt{av}_1 = (x_1, v_1, l_1, n_1)$ *in a program c, with a starting memory m and hidden database D, denoted as* $\mathsf{DEP}_{\texttt{av}}(\texttt{av}_1, \texttt{av}_2, c, m, D)$ *is defined below.*

$$
\begin{array}{l}
\exists \mathbf{m}_1, \mathbf{m}_3, \tau_1, \tau_3, \mathbf{c}_2, v_1, (\alpha_1 \vee \mathbf{e}_1). \\
\left(\begin{array}{l}
\langle \mathbf{m}, \mathbf{c}, [], [] \rangle \rightarrow^* \langle \mathbf{m}_1, [\mathbf{x}_1 \leftarrow \texttt{query}(\alpha_1)(/\mathbf{e}_1)]^{l_1}; \mathbf{c}_1, \texttt{qt}_1, \tau_1, w_1 \rangle \rightarrow^{\textit{ssa-query-v (/ assn-v)}} \\
\langle \mathbf{m}_1[v_1/\mathbf{x}], c_2, \texttt{qt}_1', \tau_1 + +[\texttt{av}_1], w_1 \rangle \rightarrow^* \langle \mathbf{m}_3, \texttt{skip}, \texttt{qt}_3, \tau_3, w_3 \rangle \\
\bigwedge \left(\begin{array}{l}
\texttt{av}_2 \in (\tau_3' - (\tau_1 + +[\texttt{av}_1])) \\
\implies \exists v \in \mathcal{QD}, v \neq v_1, \mathbf{m}_3', \texttt{qt}_3', \tau_3', w_3'. \langle \mathbf{m}_1[v/\mathbf{x}], \mathbf{c}_2, \texttt{qt}_1', \tau_1 + +[\texttt{av}_1], w_1 \rangle \\
\quad \rightarrow^* (\langle \mathbf{m}_3', \texttt{skip}, \texttt{qt}_3', \tau_3', w_3' \rangle \\
\quad \wedge \texttt{av}_2 \notin (\tau_3' - (\tau_1 + +[\texttt{av}_1])))
\end{array} \right) \\
\bigwedge \left(\begin{array}{l}
\texttt{av}_2 \notin (\tau_3 - (\tau_1 + +[\texttt{av}_1])) \\
\implies \exists v \in \mathcal{QD}, v \neq v_1, \mathbf{m}_3', \texttt{qt}_3', \tau_3', w_3'. \langle \mathbf{m}_1[v/\mathbf{x}], \mathbf{c}_2, \texttt{qt}_1', \tau_1 + +[\texttt{av}_1], w_1 \rangle \\
\quad \rightarrow^* (\langle \mathbf{m}_3', \texttt{skip}, \texttt{qt}_3', \tau_3', w_3' \rangle \\
\quad \wedge \texttt{av}_2 \in (\tau_3' - (\tau_1 + +[\texttt{av}_1])))
\end{array} \right)
\end{array} \right)
\end{array}
$$

Definition 9 (Annotated Variables May Dependency – Version 2)**.** .
One annotated variable $\texttt{av}_2 = (x_2, v_2, l_2, n_2)$ *may depend on another one* $\texttt{av}_1 = (x_1, v_1, l_1, n_1)$*in a program* $\mathbf{c}$*, with a starting memory* $\mathbf{m}$ *and hidden database D, denoted as* $\mathsf{DEP}_{\texttt{av}}(\texttt{av}_1, \texttt{av}_2, c, m, D)$ *is defined below.*

$$
\begin{array}{l}
\exists \mathbf{m}, \mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3, \mathbf{m}_2', \mathbf{m}_3', \tau_1, \tau_2, \tau_2', t_1, t_2, t_2', \mathbf{c}_1, \mathbf{c}_2, v_1'. \\
\left(\begin{array}{l}
\langle \mathbf{m}, \mathbf{c}, [] \rangle \rightarrow^* \langle \mathbf{m}_1, [\mathbf{x}_1 \leftarrow \texttt{query}(\alpha_1)(/\mathbf{e}_1)]^{l_1}; \mathbf{c}_1, \tau_1, t_1 \rangle \\
\bigwedge \langle \mathbf{m}_1[v_1/\mathbf{x}_1], c_1, \tau_1 + +[\texttt{av}_1], t_1[\mathbf{x}_1] + + \rangle \rightarrow^* \langle \mathbf{m}_2, [/x_2 \leftarrow \texttt{query}(\alpha_2)(/\mathbf{e}_2)]^{l_2}; \mathbf{c}_2, \tau_2, t_2 \rangle \\
\quad \rightarrow^{\textit{ssa-query-v (/ assn-v)}} \langle \mathbf{m}_3, \mathbf{c}_2, \tau_2 + +[\texttt{av}_2], t_2[\mathbf{x}_2] + + \rangle \\
\bigwedge \langle \mathbf{m}_1[v_1'/\mathbf{x}_1], \mathbf{c}_1, \tau_1, t_1 \rangle \rightarrow^* \langle \mathbf{m}_2', \mathbf{c}_2, \tau_2', t_2' \rangle \\
\bigwedge \texttt{av}_2 \notin \tau_2'
\end{array} \right)
\end{array}
$$

Definition 10 (Variable May Dependency)**.** .
Given a program $\mathbf{c}$ *with its assigned variables* $\texttt{aVar}_{\mathbf{c}}$*, one variable* $\mathbf{x}_2 \in \texttt{aVar}_{\mathbf{c}}$ *may depend on another variable* $\mathbf{x}_1 \in \texttt{aVar}_{\mathbf{c}}$ *in* $\mathbf{c}$ *denoted as* $\mathsf{DEP}_{\texttt{var}}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{c})$ *is defined below.*

$$\exists v_1, v_2, n_1, n_2, m, D.\ \texttt{av}_1 = (x_1, v_1, l_1, n_1) \wedge \texttt{av}_2 = (x_2, v_2, l_2, n_2) \wedge \mathsf{DEP}_{\texttt{av}}(\texttt{av}_1, \texttt{av}_2, c, m, D)$$

Definition 11 (Execution Based Dependency Graph)**.** .
Given a program c, a database D, a starting memory m with its assigned variables $\texttt{aVar}_c$ *and initial variable counter* $\texttt{vcnt}_c^0$ *with its corresponding execution:* $\langle m, c, [], \texttt{vcnt}_c^0 \rangle \rightarrow^* \langle \mathbf{m}', \texttt{skip}, \tau, \texttt{vcnt} \rangle$*, the dependency graph* $\mathbf{G}_{\mathbf{trace}}(\mathbf{c}, \mathbf{m}, \mathbf{D}) = (\texttt{V}, \texttt{E}, \texttt{W}, \texttt{qF})$ *is defined as:*

$$
\begin{array}{rrcl}
\textit{Vertices} & \texttt{V} & := & \{x \in \mathcal{VAR} \mid x = \texttt{aVar}_c(i); i = 0, \ldots, |\texttt{aVar}_c|\} \\
\textit{Directed Edges} & \texttt{E} & := & \{(x, x') \in \mathcal{VAR} \times \mathcal{VAR} \mid \mathsf{DEP}_{\texttt{var}}(x, x', c); x = \texttt{aVar}_c(i); x' = \texttt{aVar}_c(j); i, j = 0, \ldots, |\texttt{aVar}_c|\} \\
\textit{Weights} & \texttt{W} & := & \{(x, n) \in \mathcal{VAR} \times \mathbb{N} \mid n = \texttt{vcnt}(x); x = \texttt{aVar}_c(i); i = 0, \ldots, |\texttt{aVar}_c|\} \\
\textit{Query Flags} & \texttt{qF} & := & \left\{ (x, n) \in \texttt{V} \times \{0, 1\} \;\middle|\; \left\{ \begin{array}{ll} n = 1 & x \in \texttt{qVar}_c \\ n = 0 & o.w. \end{array} \right\}; x = \texttt{aVar}_c(i); i = 0, \ldots, |\texttt{aVar}_c| \right\}
\end{array}
$$

Definition 12 (Finite Walk (k)). .
Given a labeled weighted graph $G = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{qF})$, a finite walk *$k$ in G 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:*

- *$e_i = (v_i, v_{i+1})$ for every $1 \le i < n$.*
- *every vertex $v \in \mathtt{V}$ appears in this vertices sequence $(v_1, \dots, v_n)$ of k at most $W(v)$ times.*

$(v_1, \dots, v_n)$ is the vertex sequence of this walk.
Length of this finite walk k is the number of vertices in its vertex sequence, i.e., $\mathtt{len}(k) = n$.

Given a labeled weighted graph $G = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{qF})$, we use $\mathcal{WALK}(G)$ to denote a set containing all finite walks k in G; and $k_{v_1 \to v_2} \in \mathcal{WALK}(G)$where $v_1, v_2 \in \mathtt{V}$ denotes the walk from vertex v_1 to v_2 .

Definition 13 (Length of Finite Walk w.r.t. Query ($\mathtt{len}_\mathtt{q}$)). .
Given a labeled weighted graph $G = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{qF})$ and a finite walk *k in G with its vertex sequence $(v_1, \dots, v_n)$, the length of k w.r.t query is defined as:*

$$\mathtt{len}_\mathtt{q}(k) = \mathtt{len}\big(v \mid v \in (v_1, \dots, v_n) \wedge \mathtt{F}(v) = 1\big)$$

, where $\big(v \mid v \in (v_1, \dots, v_n) \wedge \mathtt{F}(v) = 1\big)$ is a subsequence of k's vertex sequence.

Given a program c with a starting memory m and database D, we generate its program-based graph $\mathbf{G_{trace}}(\mathbf{c}, \mathbf{m}, \mathbf{D}) = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{qF})$. Then the adaptivity bound based on program analysis for $\mathbf{c}$ is the number of query vertices on a finite walk in $\mathbf{G_{prog}}(\mathbf{c})$. This finite walk satisfies:

- the number of query vertices on this walk is maximum
- the visiting times of each vertex v on this walk is bound by its weight $\mathtt{W}(v)$.

It is formally defined in 30.

Definition 14 (Adaptivity of A Program). .
Given a program $\mathbf{c}$ in SSA language, its adaptivity is defined for all possible starting SSA memory $\mathbf{m}$ and database D as follows:

$$A(c) = \max\left\{\mathtt{len}_\mathtt{q}(k) \mid \mathbf{m} \in \mathcal{SM}, D \in \mathcal{DB}, k \in \mathcal{WALK}(\mathbf{G_{trace}}(\mathbf{c}, \mathbf{m}, \mathbf{D})\right\}$$

]
We proved some useful properties for our language.
[[

Definition 15 (Well-formed Trace). *A trace t is well formed if and only if it preserves the following two properties:*

- (Uniqueness) $\forall \mathtt{av}_1, \mathtt{av}_2 \in_{\mathtt{av}} t.\ (\mathtt{av}_1 \neq_{\mathtt{av}} \mathtt{av}_2)$
- (Ordering) $\forall \mathtt{av}_1, \mathtt{av}_2 \in_{\mathtt{av}} t.\ (\mathtt{av}_1 <_{\mathtt{av}} \mathtt{av}_2) \Longleftrightarrow \exists t_1, t_2, t_3, \mathtt{av}'_1, \mathtt{av}'_2.\ s.t.,\ (\mathtt{av}_1 =_{\mathtt{av}} \mathtt{av}'_1) \wedge (\mathtt{av}_2 =_{\mathtt{av}} \mathtt{av}'_2) \wedge t_1 + +[\mathtt{av}'_1] + + t_2 + +[\mathtt{av}'_2] + + t_3 = t$

]]
[[

Theorem 2.1 (Variable Trace Generated from Operational Semantics is Well-formed)**.** .
Given a program c, with arbitrary starting memory m, trace τ and variable counter `vcnt` *if* $\langle m, c, \tau, \texttt{vcnt}\rangle \rightarrow^* \langle m', \texttt{skip}, \tau', \texttt{vcnt}'\rangle$*, then* $(\tau' - \tau)$ *is a well formed trace with respect to program c, m and w, denoted as* $m, c \vDash \tau' - \tau$*.*

Proof. Proof in File: "`thm_os_wf_trace.tex`". □

]] [[

Lemma 1 (While Map Remains Unchanged (Invariant))**.** *Given a program c with a starting memory m, trace t and while map w, s.t.,* $\langle m, c, t, w\rangle \rightarrow^* \langle m', \texttt{skip}, t', w'\rangle$ *and* $Labels(c) \cap Keys(w) = \emptyset$*, then*

$$w = w'$$

Proof of Lemma 8. Proof in File: "`lem_wunchange.tex`" ■

]] [[

Lemma 2 (Trace is Written Only)**.** *Given a program c with starting trace* t_1 *and* t_2*, for arbitrary starting memory m and while map w, if there exist evaluations*

$$\langle m, c, t_1, w\rangle \rightarrow^* \langle m_1', \texttt{skip}, t_1', w_1'\rangle$$

$$\langle m, c, t_2, w\rangle \rightarrow^* \langle m_2', \texttt{skip}, t_2', w_2'\rangle$$

then:

$$m_1' = m_2' \wedge w_1' = w_2'$$

Proof of Lemma 9. Proof in File: "`lem_twriteonly.tex`" ■

]] [[

Lemma 3 (Trace Uniqueness)**.** *Given a program c with a starting memory m, [WQ: a while map w,] for any starting trace* t_1 *and* t_2*, if there exist evaluations*

$$\langle m, c, t_1, w\rangle \rightarrow^* \langle m_1', \texttt{skip}, t_1', w_1'\rangle$$

$$\langle m, c, t_2, w\rangle \rightarrow^* \langle m_2', \texttt{skip}, t_2', w_2'\rangle$$

then:

$$t_1' - t_1 = t_2' - t_2$$

Proof of Lemma 10. Proof in File: "`lem_tunique.tex`" ■

]] [[

Corollary 2.1.1.

$$\texttt{av} \in_{\texttt{av}} t \implies \exists t_1, t_2, \texttt{av}'.\ s.t.,\ (\texttt{av} =_{\texttt{av}} \texttt{av}') \wedge t_1 +\!+ [\texttt{av}'] +\!+ t_2 = t$$

Proof. Proof in File: "`coro_aqintrace.tex`" ■

]]

Lemma 4 (Trace Non-Decreasing)**.** .
[JL: For any program c with a starting memory m, trace t and while map w:

$$\langle m, c, t, w\rangle \rightarrow \langle m, c', t', w'\rangle \Longrightarrow \exists\ t'',\ s.t.,\ t + +t'' = t'$$

]

Proof. Proof is obvious by induction on the operational semantic rules applied in the transition .
By induction on the operational semantic rules applied in the transition $\langle m, c, t, w\rangle \rightarrow \langle m, c', t', w'\rangle$, we have cases for each rule. By observation on the rules, the trace t remains unchanged in all the rules except the only one **query-v**. So, the rule **query-v** is the only interesting case to be discussed as following.

- **case:**

$$\frac{\texttt{query}(\alpha) = v}{\langle m, [x \leftarrow \texttt{query}(\alpha)]^{l}, t, w\rangle \rightarrow \langle m, \texttt{skip}, t + +[(\alpha, l, w)], w\rangle}\ \textbf{query-v}$$

  In this case, we have $c' = \texttt{skip}$, $t' = t + +[(\alpha, l, w)]$, $m' = m[v/x]$ and $w' = w$.
  Let $t'' = [(\alpha, l, w)]$, we have $t + +[(\alpha, l, w)] = t'$, i.e., $t + +t'' = t'$. This case is proved.

□

[[The following lemma describes a property of the trace-based dependency graph. For any program c with a database D and a starting memory m, the directed edges in its trace-based dependency graph can only be constructed from nodes representing smaller annotated queries to annotated queries of greater order. There doesn't exist backward edges with direction from greater annotated queries to smaller ones.]]

Lemma 5. *[Edges are Forwarding Only].*
Given a program c, a database D, a starting memory m and the corresponding trace-based dependency graph $G(c, D, m) = (\texttt{V}, \texttt{E})$, for any directed edge $(\texttt{av}', \texttt{av}) \in \texttt{E}$, this is not the case that:

$$\texttt{av}' \geq_{\texttt{av}} \texttt{av}$$

Proof. Proof in File: "edge_forward.tex". □

Lemma 6. *[Trace-based Dependency Graph is Directed Acyclic].*
Every trace-based dependency graph is a directed acyclic graph.

Proof. Proof is obvious based on the Lemma 12. □

Lemma 7 (Adaptivity is Bounded)**.** .
Given the program c with a certain database D and starting memory m, the $A(c)$ w.r.t. the D and m is bounded, i.e.,:

$$\langle m, c, [], []\rangle \rightarrow^{*} \langle m', \texttt{skip}, t', w'\rangle \Longrightarrow A_{D,m}(c) \leq |t'|$$

Proof. Proof is obvious based on the Lemma 13. □

3 Labeled SSA Language

3.1 The Limit of `While` Language

we see the power of the labelled loop language to achieve the adaptivity semantically, from its being capable to express many adaptive data analysis algorithm, allowing the construction of the query-based dependency graph using traces from the execution, and so on. However, it is not powerful enough to reach the adaptivity syntactically. The main difficulty is its implicit control flow which raises extra complexity to figure out where some variables used come from. We use $\texttt{query}(0), \texttt{query}(1)$ to represent linear queries.

$$
c_1 = \begin{array}{l} [x \leftarrow \texttt{query}(0)]^1; \\ \texttt{if}\ [(x<0)]^2 \\ \texttt{then}\ [x \leftarrow \texttt{query}(1)]^3 \\ \texttt{else}\ [\texttt{skip}]^4; \\ [y \leftarrow \texttt{query}(x+\chi(3))]^5 \end{array}
\quad
c_2 = \begin{array}{l} [x \leftarrow \texttt{query}(0)]^1; \\ \texttt{if}\ [(x<0)]^2 \\ \texttt{then}\ [x \leftarrow \texttt{query}(1)]^3 \\ \texttt{else}\ [x \leftarrow \texttt{query}(2)]^4; \\ [y \leftarrow \texttt{query}(x+\chi(3))]^5 \end{array}
\quad
c_3 = \begin{array}{l} [x \leftarrow \texttt{query}(0)]^1; \\ \texttt{if}\ [(x<0)]^2 \\ \texttt{then}\ [z \leftarrow \texttt{query}(1)]^3 \\ \texttt{else}\ [\texttt{skip}]^4; \\ [y \leftarrow \texttt{query}(x+\chi(3)]^5 \end{array}
$$

In these three examples, the variable x at line 5 is implicit. In program c_1, it refers to the either x at line 1, or x at line 3, which means the result of query request $\texttt{query}(x+\chi(3))$ assigned to the variable y may depend on $\texttt{query}(0)$(bound to x at line 1) or $\texttt{query}(1)$(x at line 3). When we have a look at the other two programs c_2 and c_3, it is another talk. We think $\texttt{query}(x+\chi(3))$ may depend on either $\texttt{query}(1)$(x at line 3) or $\texttt{query}(x+\chi(3))$($x$ at line 4) in c_2, while $\texttt{query}(x+\chi(3))$ only depends on $\texttt{query}(0)$ at line 1 in program c_3. These three examples are structural similar in loop language, however, the dependency between variables are quite dissimilar. We consider variables here because query request is also bound to variables. To solve this dilemma, we move to single static assignment as follows.

$$
c_1^{ssa} = \begin{array}{l} [\mathbf{x}_1 \leftarrow \texttt{query}(0)]^1; \\ \texttt{if}\ [(\mathbf{x_1}<0)]^2 \\ ([], [\mathbf{x_3}, \mathbf{x_1}, \mathbf{x_2}], []) \\ \texttt{then}\ [\mathbf{x_2} \leftarrow \texttt{query}(1)]^3 \\ \texttt{else}\ [\texttt{skip}]^4; \\ [\mathbf{y_1} \leftarrow \texttt{query}(\mathbf{x_3}+\chi(3))]^5 \end{array}
\quad
c_2^{ssa} = \begin{array}{l} [\mathbf{x_1} \leftarrow \texttt{query}(0)]^1; \\ \texttt{if}\ [(\mathbf{x_1}<0)]^2, \\ ([\mathbf{x_4}, \mathbf{x_2}, \mathbf{x_3}], [], []) \\ \texttt{then}\ [\mathbf{x_2} \leftarrow \texttt{query}(1)]^3 \\ \texttt{else}\ [\mathbf{x_3} \leftarrow \texttt{query}(2)]^4; \\ [\mathbf{y_1} \leftarrow \texttt{query}(\mathbf{x_4}+\chi(3))]^5 \end{array}
\quad
c_3^{ssa} = \begin{array}{l} [\mathbf{x_1} \leftarrow \texttt{query}(0)]^1; \\ \texttt{if}\ [(\mathbf{x_1}<0)]^2 \\ ([], [], []) \\ \texttt{then}\ [\mathbf{z_1} \leftarrow \texttt{query}(1)]^3 \\ \texttt{else}\ [\texttt{skip}]^4; \\ [\mathbf{y_1} \leftarrow \texttt{query}(\mathbf{x_1}+\chi(3))]^5 \end{array}
$$

To distinguish between the loop language and in ssa form, we denote the ssa variable $\mathbf{x_1}$ in bold. As we can see, the data flow becomes explicit in ssa form and the analysis on the dependency between variables in the program becomes much clear now. Considering this advantage, we aim to estimate the adaptivity through an analysis on program in ssa form.

3.2 SSA form Language

Arithmetic Operators	$\oplus_a$	::=	$+ \mid - \mid \times \mid \div$
Boolean Operators	$\oplus_b$	::=	$\vee \mid \wedge$
Relational Operators	$\sim$	::=	$< \mid \leq \mid ==$
Label	l	:=	$\mathbb{N}$
SSA Arithmetic Expression	$\mathbf{a}$	::=	$n \mid \mathbf{x} \mid \mathbf{a} \oplus_a \mathbf{a}$
SSA Boolean Expression	$\mathbf{b}$	::=	$\texttt{true} \mid \texttt{false} \mid \neg\mathbf{b} \mid \mathbf{b} \oplus_b \mathbf{b} \mid \mathbf{a} \sim \mathbf{a}$
SSA Query Expression	$\boldsymbol{\psi}$	::=	$\alpha \mid \mathbf{a} \mid \psi \oplus_a \psi$
Query Value	α	::=	$n \mid \chi[n] \mid \chi[n] \oplus_a \chi[n] \mid n \oplus_a \chi[n] \mid \chi[n] \oplus_a n$
Value	v	::=	$n \mid \texttt{true} \mid \texttt{false} \mid [] \mid [v, \ldots, v]$
SSA Expression	$\mathbf{e}$	::=	$v \mid \mathbf{a} \mid \mathbf{b} \mid [e, \ldots, e]$
Labeled SSA Command	$\mathbf{c}$	::=	$[\mathbf{x} \leftarrow \mathbf{e}]^l \mid [\mathbf{x} \leftarrow \texttt{query}(\boldsymbol{\psi})]^l \mid \texttt{ifvar}(\bar{\mathbf{x}}, \bar{\mathbf{x}}')$
			$\texttt{while } [\mathbf{b}]^l, n, [\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}] \texttt{ do } \mathbf{c}$
			$\mid \mathbf{c}; \mathbf{c} \mid [\texttt{ if } (\mathbf{b}, [\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}], \mathbf{c}, \mathbf{c})]^l \mid [\texttt{skip}]^l$
Event	ϵ	::=	$(\mathbf{x}, v, l, n) \mid (\mathbf{x}, \alpha, l, n) \mid (\mathbf{b}, v, l, n)$
Trace	τ	::=	$[] \mid \epsilon :: \tau$
Environment	θ	::=	$\tau \rightarrow v$
Variable Counter	$\texttt{vcnt}$	::=	$\mathcal{SVAR} \rightarrow \mathbb{N}$

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

$\mathcal{SVAR}$	:	Set of Variables
$\mathcal{VAL}$	:	Set of Values
$\mathcal{E}$	:	Set of Events
$\mathcal{SM}$	:	Set of SSA Memories
$\mathcal{DB}$	:	Set of Databases
$\mathcal{QD} = [-1, 1]$	:	Domain of Query Results

Consistences to the `While` Language Each command is labeled with a label l, a natural number standing for the line of code where the command appears. Notice that we associate the label l to the conditional predicate b in the if statement, and to the guard b in the while statement.
A memory is standard, a map from labeled variables to values.
The variable counter `vcnt` maps every variable to a natural number $n \in \mathbb{N}$ in a certain execution of program $\mathbf{c}$. This natural number n represents the visiting times of this variable in this certain execution. We use variable name x within parenthesis to denote the access to the associated natural number of this variable in the variable counter $\texttt{vcnt}_c$, $\texttt{vcnt}_c(x)$ denote the visiting times of variable x.
The annotated variable / event $\texttt{av} = (x, l, n, v)$ or $\texttt{av} = (x, l, n, \alpha)$ is a quaternary tuple contains 4 elements. α is a query value representing the corresponding query request $x \leftarrow \texttt{query}(\alpha)$ during the execution of the program.
A variable trace τ is a list of annotated queries accumulated along the execution of the program. A trace can be regarded as the program history, where this history consists of all the queries asked by the analyst during the execution of the program. **[[Given an annotated variable `av` and a trace t, the appending operation $\texttt{av} :: t$ is the standard list appending operation, appends `av` to the head of trace t. The concatenation operation between 2 traces t_1 and t_2, i.e., $t_1 + + t_2$ is the standard list concatenation operation as follows:**

$$t_1 + + t_2 \triangleq \begin{cases} t_2 & t_1 = [] \\ \texttt{av} :: (t_1' + + t_2) & t_1 = \texttt{av} :: t_1' \end{cases} \tag{6}$$

The subtraction operation between 2 traces t_1 and t_2, i.e., $t_1 - t_2$ is defined as follows:

$$t_1 - t_2 \triangleq t_3 \ s.t., t_2 + + t_3 = t_1 \tag{7}$$

]] A configuration, $\langle \mathbf{m}, \mathbf{c}, \boldsymbol{\tau}, \texttt{vcnt} \rangle$, contains four elements: a SSA memory $\mathbf{m}$, the command $\mathbf{c}$ to be evaluated, a trace t and variable `vcnt`.
We collect the trace with a trace-based small-step operational semantics based on transitions of the program configuration $\langle m, c, \tau, \texttt{vcnt} \rangle$, of form $\langle m, c, \tau, \texttt{vcnt} \rangle \rightarrow \langle m', \texttt{skip}, \tau', \texttt{vcnt}' \rangle$.

Differences in SSA form Language We use $\mathbf{a}$ to express arithmetic expressions which now contains ssa variable $\mathbf{x} \in \mathcal{SVAR}$, and the boolean expression as $\mathbf{b}$. The ssa expression can be either $\mathbf{a}$ and $\mathbf{b}$. We also have the ssa variables annotated in a similar way as the annotated queries in the while language. The labeled commands $\mathbf{c}$ are now in the ssa form. In the assignment command $[\mathbf{x} \leftarrow \mathbf{e}]^l$ and query request command $[\mathbf{x} \leftarrow \texttt{query}(\boldsymbol{\psi})]^l$, the expression $\mathbf{e}$ and query expression $\boldsymbol{\psi}$ is now in their corresponding ssa forms.

The if command now contains the extra part $([\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}], [\bar{\mathbf{y}}, \bar{\mathbf{y}_1}, \bar{\mathbf{y}_2}], [\bar{\mathbf{z}}, \bar{\mathbf{z}_1}, \bar{\mathbf{z}_2}])$, which helps to track the dependency of new assigned variables in both branches($[\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}]$), then branch $[\bar{\mathbf{y}}, \bar{\mathbf{y}_1}, \bar{\mathbf{y}_2}]$, and else branch $[\bar{\mathbf{z}}, \bar{\mathbf{z}_1}, \bar{\mathbf{z}_2}]$. The $\bar{\mathbf{x}}$ is a list of ssa variables, in which every element $\mathbf{x}$ may depends on the corresponding element $\mathbf{x_1}$ from $\bar{\mathbf{x}_1}$ collected in the then branch or the corresponding element $\mathbf{x_2}$ from $\bar{\mathbf{x}_2}$ collected in the else branch. Every tuple $(\mathbf{x}, \mathbf{x_1}, \mathbf{x_2})$ from $[\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}]$ can be understood as $\mathbf{x} = \phi(\mathbf{x_1}, \mathbf{x_2})$ in the normal ssa form. The previous example c_2^s can be used for reference. The second part $[\bar{\mathbf{y}}, \bar{\mathbf{y}_1}, \bar{\mathbf{y}_2}]$ focuses on the then branch. The list of ssa variables y_1 stores the assigned ssa variables before the if command, whose non-ssa version (variables in the while language) will be modified only in the then branch. We can look at program c_1 as a reference,in which x at line 1 may be modified only in the then branch at line 3. The list $\bar{\mathbf{y}_2}$ tracks the ssa variables assigned only in the then branch. If the variables are assigned in both branches such as in the program c_2, they goes into $[\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}]$. Then we think every ssa variable in $\bar{\mathbf{y}}$ may come from the corresponding variable $\mathbf{y_1}$ in $\bar{\mathbf{y}_1}$ before the if command or $\mathbf{y_2}$ in $\bar{\mathbf{y}_2}$ in the then branch. In this sense, we can also regard every tuple $(\mathbf{y}, \mathbf{y_1}, \mathbf{y_2})$ from $[\bar{\mathbf{y}}, \bar{\mathbf{y}_1}, \bar{\mathbf{y}_2}]$ as $\mathbf{y} = \phi(\mathbf{y_1}, \mathbf{y_2})$. The rest part $[\bar{\mathbf{z}}, \bar{\mathbf{z}_1}, \bar{\mathbf{z}_2}]$ focus on the else branch and can be understood similarly.

The while command also has similar part $[\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}]$, focusing on the while body. The new command $\texttt{ifvar}(\bar{\mathbf{x}}, \bar{\mathbf{x}}')$ does not have explicit label because it is only used for evaluation internally, we will discuss more about it when in the small-step operational semantics for SSA language.
The SSA memory $\mathbf{m}$ is a map from SSA variables $\mathbf{x}$ to values.
The others remain the same in SSA form language as in the `While` language.

3.3 Trace-based Operational Semantics for SSA Language

The small-step transition states that a configuration $\langle m, c, \tau, \texttt{vcnt} \rangle$ evaluates to another configuration with the trace and while map updated along with the evaluation of the command c to the normal form of the command `skip`. We define rules of the trace-based operational semantics in Figure 1. The rule **query-e** evaluates the argument of a query request. When the argument is in normal form, this query will be answered. The rule **query-v** modifies the starting memory m to $m[\alpha/x]$ using the answer α of the query $\texttt{query}(\alpha)$ from the mechanism, with the trace expanded by appending the query $\texttt{query}(\alpha)$ with the current annotation (l, w). The rule for assignment is standard and the trace remains unchanged.The sequence rule keeps tracking the modification of the trace, and the evaluation rule for if conditional goes into one branch based on the result of the conditional predicate b. The rules for

while modify the while map w. In the rule **ifw-true**, the while map w is updated by $w + l$ because the execution goes into another iteration when the condition $n > 0$ is satisfied. When n reaches 0, the loop exits and the while map w eliminates the label l of this while statement by $w - l$ in the rule **ifw-false**. With the operational semantics and relations between annotated queries, we restrict the well-formed trace w.r.t. the execution of a program c in Definition 23. When switching to the SSA language, we show that we are still able to achieve what we can get in Section 2. The operational semantics of the SSA language mimics its counterpart, of the form $\langle \mathbf{m}, \mathbf{c}, t, w\rangle \rightarrow \langle \mathbf{m}', \texttt{skip}, t', w'\rangle$. The SSA memory $\mathbf{m}$ is a map from SSA variable $\mathbf{x}$ to values. It still uses a trace to track the query requests during the execution, starting from an SSA configuration with an SSA memory $\mathbf{m}$ and a program in its SSA form $\mathbf{c}$, which allows a similar construction of the query-based dependency graph in the SSA language as in the `While` language. We show the evaluation rules in Figure 2. The command $\texttt{ifvar}(\bar{x}, \bar{x}')$ stores the variable map during the run time, which is a map from ssa variable $\mathbf{x}$ to variable $\mathbf{x}_i$. This map is designed for `if` command, when the variable may comes from two branches and this command records which branch the variable comes from. The key idea underneath the operational semantics is to have the trace and the execution path being constructed in a similar way as in the loop language. Take the query request as an example, the argument $\mathbf{e}$ which contains ssa variables will be evaluated to a value v first before the request is sent to the database in rule **ssa-query-arg**. The trace expands in the rule **ssa-query** likewise in the loop language. The query q, a primitive symbol representing the abstract query in both the ssa language and the loop language, makes no difference in the two languages. Since we add the extra part $[\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}], [\bar{\mathbf{y}}, \bar{\mathbf{y}_1}, \bar{\mathbf{y}_2}], [\bar{\mathbf{z}}, \bar{\mathbf{z}_1}, \bar{\mathbf{z}_2}]$ in the if command compared to its counterpart in the while language, the rules relevant to the if condition (**ssa-if-t** and **ssa-if-f**) use the command $\texttt{ifvar}(\bar{\mathbf{x}}, \bar{\mathbf{x}}')$ to update the ssa memory $\mathbf{m}$ with the mapping from the new generated variable $\mathbf{x}$ in $\bar{\mathbf{x}}$ to the appropriate value $\mathbf{m}(\mathbf{x}')$ where $\mathbf{x}'$ is the corresponding variable w.r.t $\mathbf{x}$ in $\bar{\mathbf{x}}'$. The rule **ssa-**`ifvar` reflects the usage of $\texttt{ifvar}(\bar{\mathbf{x}}, \bar{\mathbf{x}}')$. It is easier to understand the usage of $\texttt{ifvar}(\bar{\mathbf{x}}, \bar{\mathbf{x}}')$ in the rule **ssa-if-t** when we think about how ssa works: in the ssa form, when a variable to be used may come from two sources (e.g. $\mathbf{x_1}$ and $\mathbf{x_2}$ in the rule), it generates a new variable $\mathbf{x}$, assigning it with $\phi(\mathbf{x_1}, \mathbf{x_2})$, and replaces the variable to be used with newly assigned $\mathbf{x}$. We know that in the future program after the if command, only the variables $\bar{\mathbf{x}}$ will be available instead of $\bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}$ from two branches. For the evaluation of the program after the if command, we need to tell the memory the exact value of the newly generated variable $\mathbf{x}$, which is the value stored in $\mathbf{x_1}$ when the conditional $\mathbf{b}$ is true, or the value in $\mathbf{x}_2$ when $\mathbf{b}$ is false. To this end, the internal command $\texttt{ifvar}(\bar{\mathbf{x}}, \bar{\mathbf{x}}')$ plays its role. For the if rule, w e need to instantiate the variables from $\bar{\mathbf{x}}$ whose values come from two branches, $\bar{\mathbf{y}}$ whose values from then branch or assignment before the if command, and $\bar{\mathbf{z}}$ whose values from else branch or before the if command. Correspondingly, we need to have three ifvar commands.

The evaluation of while depends on the while iteration counter $\mathbf{n}$ and the guard $\mathbf{b}$. When $\mathbf{b}$ is evaluated to `true`, the while is still executing, and all the variables $\mathbf{x}$ in $\bar{\mathbf{x}}$ of the loop body $\mathbf{c}$ are replaced as the corresponding variables in $\bar{\mathbf{x_1}}$ in the first iteration($n = 0$), or $\bar{\mathbf{x_2}}$ in other iterations($n > 0$). The while turns to an exit when $\mathbf{n} > 0$, and the memory $\mathbf{m}$ updates the mapping of variables in $\bar{\mathbf{x}}$ with $\bar{\mathbf{x_1}}$ if the guard b evaluates to `false`, which means the while body is not executed once. When the while enters the exit after executing the body a few times(n), the variables in $\bar{\mathbf{x}}$ is instantiated with the value from the body $\mathbf{m}(\bar{\mathbf{x_2}})$.

3.4 Event and Trace

[[Events preserve the order of execution. The order relation is defined in Definition 16.

Definition 16 (Order of Annotated Variables / Events)**.** .

[JL:

$$\boxed{Memory \times Command \times VTrace \times VCounter \to Memory \times Command \times VTrace \times VCounter}$$

$$\boxed{\langle \mathbf{m}, \mathbf{c}, \boldsymbol{\tau}, \mathtt{vcnt} \rangle \to \langle \mathbf{m}', \mathbf{c}', \boldsymbol{\tau}', \mathtt{vcnt}' \rangle}$$

$$\frac{\langle \mathbf{m}, \mathbf{e} \rangle \to \mathbf{e}'}{\langle \mathbf{m}, [\mathbf{x} \leftarrow \mathbf{e}]^{\mathbf{l}}, \boldsymbol{\tau}, \mathtt{vcnt} \rangle \to \langle \mathbf{m}, [\mathbf{x} \leftarrow \mathbf{e}']^{\mathbf{l}}, \boldsymbol{\tau}, \mathtt{vcnt} \rangle} \textbf{ assn-e}$$

$$\frac{\mathtt{vcnt}'(\mathbf{x}) = \mathtt{vcnt}(\mathbf{x}) + 1 \qquad \mathtt{av} = (\mathbf{x}, l, \mathtt{vcnt}'(\mathbf{x}), v)}{\langle \mathbf{m}, [\mathbf{x} \leftarrow v]^{l}, \tau, \mathtt{vcnt} \rangle \to \langle \mathbf{m}[v/\mathbf{x}], [\mathtt{skip}]^{l}, \tau + +[\mathtt{av}], \mathtt{vcnt}' \rangle} \textbf{ assn-v}$$

$$\frac{}{\begin{array}{c}\langle \mathbf{m}, \mathtt{while}\ [\mathbf{b}]^{\mathbf{l}}, \mathbf{n}, [\bar{\mathbf{x}}', \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}]\ \mathtt{do}\ \mathbf{c}, \boldsymbol{\tau}, \mathtt{vcnt} \rangle \\ \to \langle \mathbf{m}, \mathtt{if}_{\mathbf{w}}(\mathbf{b}, \mathbf{n}, [\bar{\mathbf{x}_i}/\bar{\mathbf{x}'}], [\bar{\mathbf{x}}', \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}], \mathbf{c}[\bar{\mathbf{x}_i}/\bar{\mathbf{x}'}]; \mathtt{while}\ [\mathbf{b}]^{\mathbf{l}}, (\mathbf{n}+1), [\bar{\mathbf{x}}', \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}]\ \mathtt{do}\ \mathbf{c}, \mathtt{skip}), \boldsymbol{\tau}, \mathtt{vcnt} \rangle\end{array}} \textbf{ ssa-while-b}$$

$$\frac{\langle \mathbf{m}, \mathbf{b} \rangle \to \mathbf{b}'}{\langle \mathbf{m}, \mathtt{if}_{\mathbf{w}}(\mathbf{b}, [\bar{\mathbf{x}}', \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}], \mathbf{n}, \mathbf{c}_1, \mathbf{c}_2), \tau, \mathtt{vcnt} \rangle \to \langle \mathbf{m}, \mathtt{if}_{\mathbf{w}}(\mathbf{b}', [\bar{\mathbf{x}}', \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}], \mathbf{n}, \mathbf{c}_1, \mathbf{c}_2), \tau, \mathtt{vcnt} \rangle} \textbf{ ssa-ifw-b}$$

$$\frac{}{\begin{array}{c}\langle \mathbf{m}, \mathtt{if}_{\mathbf{w}}(\mathtt{true}, [\bar{\mathbf{x}}', \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}], \mathbf{n}, \mathbf{c}; \mathtt{while}\ [\mathbf{b}]^{\mathbf{l}}, \mathbf{n}, [\bar{\mathbf{x}}', \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}]\ \mathtt{do}\ \mathbf{c}, \mathtt{skip}), \boldsymbol{\tau}, \mathtt{vcnt} \rangle \\ \to \langle \mathbf{m}, \mathbf{c}; \mathtt{while}\ [\mathbf{b}]^{\mathbf{l}}, \mathbf{n}, [\bar{\mathbf{x}}', \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}]\ \mathtt{do}\ \mathbf{c}, \tau, \mathtt{vcnt} \rangle\end{array}} \textbf{ ssa-ifw-true}$$

$$\frac{n = 0 \to i = 1 \qquad n > 0 \to i = 2}{\begin{array}{c}\langle \mathbf{m}, \mathtt{if}_{\mathbf{w}}(\mathtt{false}, [\bar{\mathbf{x}}', \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}], \mathbf{n}, \mathbf{c}; \mathtt{while}\ [\mathbf{b}]^{\mathbf{l}}, \mathbf{n}, [\bar{\mathbf{x}}', \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}]\ \mathtt{do}\ \mathbf{c}, \mathtt{skip}), \boldsymbol{\tau}, \mathtt{vcnt} \rangle \\ \to \langle \mathbf{m}, \mathtt{skip}; \mathtt{ifvar}(\bar{\mathbf{x}'}, \bar{\mathbf{x}_i}), \tau, \mathtt{vcnt} \rangle\end{array}} \textbf{ ssa-ifw-false}$$

$$\frac{\langle \mathbf{m}, \boldsymbol{\psi} \rangle \to_q \psi'}{\langle \mathbf{m}, [\mathbf{x} \leftarrow \mathtt{query}(\boldsymbol{\psi})]^{\mathbf{l}}, \boldsymbol{\tau}, \mathtt{vcnt} \rangle \to \langle \mathbf{m}, [\mathbf{x} \leftarrow \mathtt{query}(\boldsymbol{\psi}')]^{\mathbf{l}}, \boldsymbol{\tau}, \mathtt{vcnt} \rangle} \textbf{ ssa-query-e}$$

$$\frac{\mathtt{query}(\alpha) = v \qquad \mathtt{vcnt}'[\mathbf{x}] = \mathtt{vcnt}[\mathbf{x}] + 1 \qquad \mathtt{av} = (\mathbf{x}, \alpha, l, \mathtt{vcnt}'[\mathbf{x}])}{\langle \mathbf{m}, [\mathbf{x} \leftarrow \mathtt{query}(\boldsymbol{\alpha})]^{\mathbf{l}}, \boldsymbol{\tau}, \mathtt{vcnt} \rangle \to \langle \mathbf{m}[\mathbf{v}/\mathbf{x}], \mathtt{skip}, \boldsymbol{\tau} + +[\mathtt{av}], \mathtt{vcnt}' \rangle} \textbf{ ssa-query-v}$$

$$\frac{\langle \mathbf{m}, \mathbf{c}_1, \boldsymbol{\tau}, \mathtt{vcnt} \rangle \to \langle \mathbf{m}', \mathbf{c}_1', \boldsymbol{\tau}', \mathtt{vcnt}' \rangle}{\langle \mathbf{m}, \mathbf{c}_1; \mathbf{c}_2, \boldsymbol{\tau}, \mathtt{vcnt} \rangle \to \langle \mathbf{m}', \mathbf{c}_1'; \mathbf{c}_2, \boldsymbol{\tau}', \mathtt{vcnt}' \rangle} \textbf{ ssa-seq1} \qquad \frac{}{\langle \mathbf{m}, [\mathtt{skip}]^{\mathbf{l}}; \mathbf{c}_2, \boldsymbol{\tau}, \mathtt{vcnt} \rangle \to \langle \mathbf{m}, \mathbf{c}_2, \boldsymbol{\tau}, \mathtt{vcnt} \rangle} \textbf{ ssa-seq2}$$

$$\frac{\langle \mathbf{m}, \mathbf{b} \rangle \to_b \mathbf{b}'}{\begin{array}{c}\langle \mathbf{m}, \mathtt{if}\ ([\mathbf{b}]^{\mathbf{l}}, [\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}], [\bar{\mathbf{y}}, \bar{\mathbf{y}_1}, \bar{\mathbf{y}_2}], [\bar{\mathbf{z}}, \bar{\mathbf{z}_1}, \bar{\mathbf{z}_2}], \mathbf{c}_1, \mathbf{c}_2), \boldsymbol{\tau}, \mathtt{vcnt} \rangle \\ \to \langle \mathbf{m}, [\,\mathtt{if}\ ([\mathbf{b}']^{\mathbf{l}}, [\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}], [\bar{\mathbf{y}}, \bar{\mathbf{y}_1}, \bar{\mathbf{y}_2}], [\bar{\mathbf{z}}, \bar{\mathbf{z}_1}, \bar{\mathbf{z}_2}], \mathbf{c}_1, \mathbf{c}_2)]^{\mathbf{l}}, \boldsymbol{\tau}, \mathtt{vcnt} \rangle\end{array}} \textbf{ ssa-if-b}$$

$$\frac{}{\begin{array}{c}\langle \mathbf{m}, \mathtt{if}\ ([\mathtt{true}]^{\mathbf{l}}, [\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}], [\bar{\mathbf{y}}, \bar{\mathbf{y}_1}, \bar{\mathbf{y}_2}], [\bar{\mathbf{z}}, \bar{\mathbf{z}_1}, \bar{\mathbf{z}_2}], \mathbf{c}_1, \mathbf{c}_2), \boldsymbol{\tau}, \mathtt{vcnt} \rangle \\ \to \langle \mathbf{m}, \mathbf{c}_1; \mathtt{ifvar}(\bar{\mathbf{x}}, \bar{\mathbf{x}_2}); \mathtt{ifvar}(\bar{\mathbf{y}}, \bar{\mathbf{y}_1}); \mathtt{ifvar}(\bar{\mathbf{z}}, \bar{\mathbf{z}_2}), \boldsymbol{\tau}, \mathtt{vcnt} \rangle\end{array}} \textbf{ ssa-if-t}$$

$$\frac{}{\begin{array}{c}\langle \mathbf{m}, \mathtt{if}\ ([\mathtt{false}]^{\mathbf{l}}, [\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}], [\bar{\mathbf{y}}, \bar{\mathbf{y}_1}, \bar{\mathbf{y}_2}], [\bar{\mathbf{z}}, \bar{\mathbf{z}_1}, \bar{\mathbf{z}_2}], \mathbf{c}_1, \mathbf{c}_2), \boldsymbol{\tau}, \mathtt{vcnt} \rangle \\ \to \langle \mathbf{m}, \mathbf{c}_2; \mathtt{ifvar}(\bar{\mathbf{x}}, \bar{\mathbf{x}_2}); \mathtt{ifvar}(\bar{\mathbf{y}}, \bar{\mathbf{y}_1}); \mathtt{ifvar}(\bar{\mathbf{z}}, \bar{\mathbf{z}_2}), \boldsymbol{\tau}, \mathtt{vcnt} \rangle\end{array}} \textbf{ ssa-if-f}$$

$$\frac{}{\langle \mathbf{m}, \mathtt{ifvar}(\bar{\mathbf{x}}, \bar{\mathbf{x}'}), \boldsymbol{\tau}, \mathtt{vcnt} \rangle \to \langle (\bar{\mathbf{x}} \to \mathbf{m}(\bar{\mathbf{x}'})) :: \mathbf{m}, \mathtt{skip}, \boldsymbol{\tau}, \mathtt{vcnt} \rangle} \textbf{ ssa-}\mathtt{ifvar}$$

]

Given 2 annotated queries $\mathtt{av}_1 = (x_1, v_1, l_1, n_1), \mathtt{av}_2 = (x_2, v_2, l_2, n_2)$ *:*

$$\mathtt{av}_1 <_{\mathtt{av}} \mathtt{av}_2 \triangleq \begin{cases} n_1 < n_2 & l_1 = l_2 \\ w_1 <_w w_2 & o.w. \end{cases}$$

$\mathtt{av}_1 \geq_{\mathtt{av}} \mathtt{av}_2$ *is defined vice versa.*

]] Given the evaluation rules for query expression, we define its equivalence relation in Definition 17.

Definition 17 (Equivalence of Query)**.** . *Given a memory* m *and 2 query expressions* ψ_1, ψ_2 *s.t.,* $FV(\psi_1) \in \mathtt{dom}(m)$ *and* $FV(\psi_2) \in \mathtt{dom}(m)$*:*

$$\psi_1 =_q^m \psi_2 \triangleq \begin{cases} \mathtt{true} & \exists \alpha_1, \alpha_2. \ (\langle m, \psi_1 \rangle \rightarrow_q \alpha_1 \wedge \langle m, \psi_2 \rangle \rightarrow_q \alpha_2) \\ & \wedge (\forall r \in \mathbb{QD}. \exists v. \ s.t., \ \langle m, \alpha_1[r/\chi] \rangle \rightarrow_a v \wedge \langle m, \alpha_2[r/\chi] \rangle \rightarrow_a v) \\ \mathtt{false} & o.w. \end{cases}$$

, where $FV(\psi)$ *is the set of free variables in the query expression* ψ*.* $\psi_1 \neq_q^m \psi_2$ *is defined vice versa. We use* $=_q$ *and* $\neq_q$ *as the shorthands for* $=_q^{[]}$ *and* $\neq_q^{[]}$*.*

Then, we have the corresponding equivalence relation between 2 annotated queries defined in Definition 18: [[

Definition 18 (Equivalence of Annotated Variables / Events)**.** *Given 2 annotated queries* $\mathtt{av}_1 = (x_1, v_1, l_1, n_1), \mathtt{av}_2 = (x_2, v_2, l_2, n_2)$ *:*

$$\mathtt{av}_1 =_{\mathtt{av}} \mathtt{av}_2 \triangleq (l_1 = l_2 \wedge w_1 =_w w_2 \wedge \alpha_1 =_q \alpha_2)$$

$\mathtt{av}_1 \neq_{\mathtt{av}} \mathtt{av}_2$ *is defined vice versa.*

]] [[**Given an annotated variable** $\mathtt{av}$ **and a trace** t**, the appending operation** $\mathtt{av} :: t$ **is the standard list appending operation, appends** $\mathtt{av}$ **to the head of trace** t**. The concatenation operation between 2 traces** t_1 **and** t_2**, i.e.,** $t_1 + + t_2$ **is the standard list concatenation operation as follows:**

$$t_1 + + t_2 \triangleq \begin{cases} t_2 & t_1 = [] \\ \mathtt{av} :: (t_1' + + t_2) & t_1 = \mathtt{av} :: t_1' \end{cases} \tag{8}$$

The subtraction operation between 2 traces t_1 **and** t_2**, i.e.,** $t_1 - t_2$ **is defined as follows:**

$$t_1 - t_2 \triangleq t_3 \ s.t., t_2 + + t_3 = t_1 \tag{9}$$

Given an annotated query $\mathtt{av}$**,** $\mathtt{av}$ **belongs to a trace** t**, i.e.,** $\mathtt{av} \in_{\mathtt{av}} t$ **are defined as follows:**

$$\mathtt{av} \in_{\mathtt{av}} t \triangleq \begin{cases} \mathtt{false} & t = [] \\ \mathtt{true} & t = \mathtt{av}' :: t' \quad \mathtt{av} =_{\mathtt{av}} \mathtt{av}' \\ \mathtt{av} \in t' & t = \mathtt{av}' :: t' \quad \mathtt{av} \neq_{\mathtt{av}} \mathtt{av}' \end{cases} \tag{10}$$

]] [[

Definition 19 (Equivalence of Program)**.** *Given 2 programs* c_1 *and* c_2*:*

$$c_1 =_c c_2 \triangleq \begin{cases} \mathtt{true} & c_1 = \mathtt{skip} \wedge c_2 = \mathtt{skip} \\ \forall m. \exists v. \ \langle m, e_1 \rangle \rightarrow_a^* v \wedge \langle m, e_1 \rangle \rightarrow_a^* v & c_1 = x \leftarrow e_1 \wedge c_2 = x \leftarrow e_2 \\ \psi_1 =_q \psi_2 & c_1 = x \leftarrow \mathtt{query}(\psi_1) \wedge c_1 = x \leftarrow \mathtt{query}(\psi_2) \\ c_1^f =_c c_2^f \wedge c_1^t =_c c_2^t & c_1 = \mathtt{if}\ (b, c_1^t, c_1^f) \wedge c_2 = \mathtt{if}\ (b, c_2^t, c_2^f) \\ c_1' =_c c_2' & c_1 = \mathtt{while}\ b\ \mathtt{do}\ c_1' \wedge c_2 = \mathtt{while}\ b\ \mathtt{do}\ c_2' \\ c_1^h =_c c_2^h \wedge c_1^t =_c c_2^t & c_1 = c_1^h; c_1^t \wedge c_2 = c_2^h; c_2^t \end{cases}$$

$c_1 \neq_c c_2$ *is defined vice versa.*

Given 2 programs c and c', c' is a sub-program of$c$, i.e., $c' \in_c c$ is defined as:

$$c' \in_c c \triangleq \exists c_1, c_2, c''.\ s.t.,\ c =_c c_1; c''; c_2 \wedge c' =_c c'' \tag{11}$$

]] [[

Definition 20 (Assigned Variables ($\texttt{aVar}$)). *Given a program* $\mathbf{c}$*, its assigned variables* $\texttt{aVar}$ *is a vector containing all variables newly assigned in the program preserving the order. It is defined as follows:*

$$\texttt{aVar}_{\mathbf{c}} \triangleq \begin{cases} [\mathbf{x}] & \mathbf{c} = [\mathbf{x} \leftarrow \mathbf{e}]^{(l,w)} \\ [\mathbf{x}] & \mathbf{c} = [\mathbf{x} \leftarrow \texttt{query}\ (\boldsymbol{\psi})]^{(l,w)} \\ \texttt{aVar}_{\mathbf{c_1}} + +\texttt{aVar}_{\mathbf{c_2}} & \mathbf{c} = \mathbf{c_1}; \mathbf{c_2} \\ \texttt{aVar}_{\mathbf{c_1}} + +\texttt{aVar}_{\mathbf{c_2}} + +[\bar{\mathbf{x}}, \bar{\mathbf{y}}, \bar{\mathbf{z}}] & \mathbf{c} = \texttt{if}\ ([\mathbf{b}]^{(l,w)}, [\bar{\mathbf{x}}, \bar{\mathbf{x}_2}, \bar{\mathbf{x}_2}], [\bar{\mathbf{y}}, \bar{\mathbf{y}_2}, \bar{\mathbf{y}_3}], [\bar{\mathbf{z}}, \bar{\mathbf{z}_2}, \bar{\mathbf{z}_3}], \mathbf{c_1}, \mathbf{c_2}) \\ \texttt{aVar}_{\mathbf{c'}} + +[\bar{\mathbf{x}}] & \mathbf{c} = \texttt{while}\ ([\mathbf{b}]^{(l,w)}, [\bar{\mathbf{x}}, \bar{\mathbf{x}_2}, \bar{\mathbf{x}_2}], \mathbf{c'}) \end{cases}$$

Definition 21 (Query Variables ($\texttt{qVar}$)). .
Given a program c, its query variables $\texttt{qVar}$ *is a vector containing all variables newly assigned by a query in the programm,* $\texttt{qVar} \subset \mathcal{VAR}$*. It is defined as follows:*

$$\texttt{qVar}_{\mathbf{c}} \triangleq \begin{cases} [] & \mathbf{c} = [\mathbf{x} \leftarrow \mathbf{e}]^{(l,w)} \\ [\mathbf{x}] & \mathbf{c} = [\mathbf{x} \leftarrow \texttt{query}\ (\boldsymbol{\psi})]^{(l,w)} \\ \texttt{aVar}_{\mathbf{c_1}} + +\texttt{aVar}_{\mathbf{c_2}} & \mathbf{c} = \mathbf{c_1}; \mathbf{c_2} \\ \texttt{aVar}_{\mathbf{c_1}} + +\texttt{aVar}_{\mathbf{c_2}} + +[\bar{\mathbf{x}}, \bar{\mathbf{y}}, \bar{\mathbf{z}}] & \mathbf{c} = \texttt{if}\ ([\mathbf{b}]^{(l,w)}, [\bar{\mathbf{x}}, \bar{\mathbf{x}_2}, \bar{\mathbf{x}_2}], [\bar{\mathbf{y}}, \bar{\mathbf{y}_2}, \bar{\mathbf{y}_3}], [\bar{\mathbf{z}}, \bar{\mathbf{z}_2}, \bar{\mathbf{z}_3}], \mathbf{c_1}, \mathbf{c_2}) \\ \texttt{aVar}_{\mathbf{c'}} + +[\bar{\mathbf{x}}] & \mathbf{c} = \texttt{while}\ ([\mathbf{b}]^{(l,w)}, [\bar{\mathbf{x}}, \bar{\mathbf{x}_2}, \bar{\mathbf{x}_2}], \mathbf{c'}) \end{cases}$$

We are abusing the notations and operators from list here. The notation $[]$ represents an empty vector and $x :: A$ represents add an element x to the head of the vector A. The concatenation operation between 2 vectors A_1 and A_2, i.e., $A_1 + +A_2$ is mimic the standard list concatenation operations as follows:

$$A_1 + +A_2 \triangleq \begin{cases} A_2 & A_1 = [] \\ x :: (A_1' + +A_2) & A_1 = x :: A_1' \end{cases} \tag{12}$$

We use index within parenthesis to denote the access to the element of corresponding location, $A(i)$ denotes the element at location i in the vector A and $M(i, j)$ denotes the element at location i-th raw, i-th column in the matrix M.

Definition 22 (Initial Variable Counter $\texttt{vcnt}_c^0$). *Given a program c with its assigned variables* $\texttt{aVar}_c$ *of length N, its initial variable counter* $\texttt{vcnt}_c^0$ *maps all the variable to* 0*, i.e.:*

$$\texttt{vcnt}_c^0(x) = 0, x = \texttt{aVar}_c(i) \forall i = 1, \ldots, N$$

We define some properties and prove lemmas for trace and event w.r.t. the operational semantics as follows. [[

Definition 23 (Well-formed Trace). *A trace t is well formed if and only if it preserves the following two properties:*

- (Uniqueness) $\forall \texttt{av}_1, \texttt{av}_2 \in_{\texttt{av}} t.\ (\texttt{av}_1 \neq_{\texttt{av}} \texttt{av}_2)$
- (Ordering) $\forall \texttt{av}_1, \texttt{av}_2 \in_{\texttt{av}} t.\ (\texttt{av}_1 <_{\texttt{av}} \texttt{av}_2) \Longleftrightarrow \exists t_1, t_2, t_3, \texttt{av}_1', \texttt{av}_2'.\ s.t.,\ (\texttt{av}_1 =_{\texttt{av}} \texttt{av}_1') \wedge (\texttt{av}_2 =_{\texttt{av}} \texttt{av}_2') \wedge t_1 + +[\texttt{av}_1'] + +t_2 + +[\texttt{av}_2'] + +t_3 = t$

]]
[[

Theorem 3.1 (Variable Trace Generated from Operational Semantics is Well-formed)**.** .
Given a program c, with arbitrary starting memory m, trace τ and variable counter `vcnt` *if* $\langle m, c, \tau, \texttt{vcnt}\rangle \rightarrow^* \langle m', \texttt{skip}, \tau', \texttt{vcnt}'\rangle$*, then* $(\tau' - \tau)$ *is a well formed trace with respect to program c, m and w, denoted as* $m, c \vDash \tau' - \tau$*.*

Proof. Proof in File: "`thm_os_wf_trace.tex`". □

]] [[

Lemma 8 (While Map Remains Unchanged (Invariant))**.** *Given a program c with a starting memory m, trace t and while map w, s.t.,* $\langle m, c, t, w\rangle \rightarrow^* \langle m', \texttt{skip}, t', w'\rangle$ *and* $Labels(c) \cap Keys(w) = \emptyset$*, then*

$$w = w'$$

Proof of Lemma 8. Proof in File: "`lem_wunchange.tex`" ■

]] [[

Lemma 9 (Trace is Written Only)**.** *Given a program c with starting trace* t_1 *and* t_2*, for arbitrary starting memory m and while map w, if there exist evaluations*

$$\langle m, c, t_1, w\rangle \rightarrow^* \langle m_1', \texttt{skip}, t_1', w_1'\rangle$$

$$\langle m, c, t_2, w\rangle \rightarrow^* \langle m_2', \texttt{skip}, t_2', w_2'\rangle$$

then:

$$m_1' = m_2' \wedge w_1' = w_2'$$

Proof of Lemma 9. Proof in File: "`lem_twriteonly.tex`" ■

]] [[

Lemma 10 (Trace Uniqueness)**.** *Given a program c with a starting memory m, [WQ: a while map w,] for any starting trace* t_1 *and* t_2*, if there exist evaluations*

$$\langle m, c, t_1, w\rangle \rightarrow^* \langle m_1', \texttt{skip}, t_1', w_1'\rangle$$

$$\langle m, c, t_2, w\rangle \rightarrow^* \langle m_2', \texttt{skip}, t_2', w_2'\rangle$$

then:

$$t_1' - t_1 = t_2' - t_2$$

Proof of Lemma 10. Proof in File: "`lem_tunique.tex`" ■

]] [[

Corollary 3.1.1.

$$\texttt{av} \in_{\texttt{av}} t \implies \exists t_1, t_2, \texttt{av}'.\ s.t.,\ (\texttt{av} =_{\texttt{av}} \texttt{av}') \wedge t_1 +\!+ [\texttt{av}'] +\!+ t_2 = t$$

Proof. Proof in File: "`coro_aqintrace.tex`" ■

]]

Lemma 11 (Trace Non-Decreasing)**.** .
[JL: For any program c with a starting memory m, trace t and while map w:

$$\langle m, c, t, w\rangle \rightarrow \langle m, c', t', w'\rangle \implies \exists\ t'',\ s.t.,\ t + + t'' = t'$$

]

Proof. Proof is obvious by induction on the operational semantic rules applied in the transition .
By induction on the operational semantic rules applied in the transition $\langle m, c, t, w\rangle \rightarrow \langle m, c', t', w'\rangle$, we have cases for each rule. By observation on the rules, the trace t remains unchanged in all the rules except the only one **query-v**. So, the rule **query-v** is the only interesting case to be discussed as following.

- **case:**

$$\frac{\texttt{query}(\alpha) = v}{\langle m, [x \leftarrow \texttt{query}(\alpha)]^l, t, w\rangle \rightarrow \langle m, \texttt{skip}, t + + [(\alpha, l, w)], w\rangle}\ \textbf{query-v}$$

  In this case, we have $c' = \texttt{skip}$, $t' = t + + [(\alpha, l, w)]$, $m' = m[v/x]$ and $w' = w$.
  Let $t'' = [(\alpha, l, w)]$, we have $t + + [(\alpha, l, w)] = t'$, i.e., $t + + t'' = t'$. This case is proved.

□

]]

3.5 Trace-based Adaptivity

We define adaptivity through a query-based dependency graph. In our model, an *analyst* asks a sequence of queries to the mechanism, and the analyst receives the answers to these queries from the mechanism. A query is adaptively chosen by the analyst when the choice of this query is affected by answers from previous queries. In this model, the adaptivity we are interested in is the length of the longest sequence of such adaptively chosen queries, among all the queries the data analyst asks to the mechanism. Also, when the analyst asks a query, the only information the analyst will have will be the answers to previous queries and the state of the program. It means that when we want to know if this query is adaptively chosen, we only need to check whether the choice of this query will be affected by changes of answers to previous queries. There are two possible situations that can affect the choice of a query, either the query argument directly uses the results of previous queries (data dependency), or the control flow of the program with respect to a query (whether to ask this query or not) depends on the results of previous queries (control flow dependency).

As a first step, we give a definition of when one query may depend on a previous query, which is supposed to consider both control dependency and data dependency. We first look at two possible candidates:

1. One query may depend on a previous query if and only if a change of the answer to the previous query may also change the result of the query.

2. One query may depend on a previous query if and only if a change of the answer to the previous query may also change the appearance of the query.

The first candidate works well by witnessing the result of one query according to the change of the answer of another query. We can easily find that the two queries have nothing to do with each other in a simple example $c = x \leftarrow \texttt{query}(\chi(1)); y \leftarrow \texttt{query}(\chi(2))$. This candidate definition works well with respect to data dependency. However, if fails to handle control dependency since it just monitors the changes to the answer of a query when the answer of previous queries returned change. The key point is that this query may also not be asked because of an analyst decision which depend on the answers of previous queries. An example of this situation is shown in program c_1 as follows.

$$c_1 = x \leftarrow \texttt{query}(\chi(1)); \texttt{if}\ (x > 2, y \leftarrow \texttt{query}(\chi(2)), \texttt{skip})$$

We choose the second candidate, which performs well by witnessing the appearance of one query $\texttt{query}(\chi(2))$ upon the change of the result of one previous query $\texttt{query}(\chi(1))$ in c_1. It considers the control dependency, and at the same, does not miss the data dependency. In particular, the arguments of a query characterizes it. In this sense, if the data used in the arguments changes due to a different answer to a certain previous query, the appearance of the query may change as well. This situation is also captured by our definition. Let us look at another variant of program c, p_2, in which the queries equipped with functions using previously assigned variables storing answer of its previous query.

$$c_2 = x \leftarrow \texttt{query}(\chi(2)); y \leftarrow \texttt{query}(x + \chi(3))$$

As a reminder, in the `While` language, the query request is composed by two components: a symbol `query` representing a linear query type and the argument e, which represents the function specifying what the query asks. So we do think $\texttt{query}(\chi(1))$ is different from $\texttt{query}(\chi(2))$. Informally, we think $\texttt{query}(x + \chi(3))$ may depend on the query $\texttt{query}(\chi(2))$, because equipped function of the former $x + \chi(3)$ depend on the data assigned with $\texttt{query}(\chi(2))$. We can see the appearance definition catches data dependency in such a way, since $\texttt{query}(x + \chi(2))$ will not be the same query if the value of x is changed.

We give a formal definition of variable may dependency based on the trace-based operational semantics as follows. [JL:

Definition 24 (Annotated Variables / Events May Dependency)**.** .
One event $\texttt{av}_2 = (x_2, v_2, l_2, n_2)$ *may depend on another one* $\texttt{av}_1 = (x_1, v_1, l_1, n_1)$ *in a program c, with a starting memory m and hidden database D, denoted as* $\mathsf{DEP}_{\texttt{av}}(\texttt{av}_1, \texttt{av}_2, c, m, D)$ *is defined below.*

$$\begin{array}{l}
\exists \mathbf{m}_1, \mathbf{m}_3, \tau_1, \tau_3, \mathbf{c}_2, v_1, (\alpha_1 \vee \mathbf{e}_1). \\
\left(\begin{array}{l}
\langle \mathbf{m}, \mathbf{c}, [], [] \rangle \rightarrow^* \langle \mathbf{m}_1, [\mathbf{x}_1 \leftarrow \texttt{query}(\alpha_1)(/\, v_1)]^{l_1}; \mathbf{c}_1, \texttt{qt}_1, \tau_1, w_1 \rangle \rightarrow^{\textit{ssa-query-v (/ assn-v)}} \\
\langle \mathbf{m}_1[v_1/\mathbf{x}], c_2, \texttt{qt}_1', \tau_1 + +[\texttt{av}_1], w_1 \rangle \rightarrow^* \langle \mathbf{m}_3, \texttt{skip}, \texttt{qt}_3, \tau_3, w_3 \rangle \\
\bigwedge \left(\begin{array}{l}
\texttt{av}_2 \in (\tau_3' - (\tau_1 + +[\texttt{av}_1])) \\
\implies \exists v \in \mathcal{QD}, v \neq v_1, \mathbf{m}_3', \texttt{qt}_3', \tau_3', w_3'. \langle \mathbf{m}_1[v/\mathbf{x}], \mathbf{c}_2, \texttt{qt}_1', \tau_1 + +[\texttt{av}_1], w_1 \rangle \\
\quad \rightarrow^* (\langle \mathbf{m}_3', \texttt{skip}, \texttt{qt}_3', \tau_3', w_3' \rangle \\
\quad \wedge \texttt{av}_2 \notin (\tau_3' - (\tau_1 + +[\texttt{av}_1])))
\end{array} \right) \\
\bigwedge \left(\begin{array}{l}
\texttt{av}_2 \notin (\tau_3 - (\tau_1 + +[\texttt{av}_1])) \\
\implies \exists v \in \mathcal{QD}, v \neq v_1, \mathbf{m}_3', \texttt{qt}_3', \tau_3', w_3'. \langle \mathbf{m}_1[v/\mathbf{x}], \mathbf{c}_2, \texttt{qt}_1', \tau_1 + +[\texttt{av}_1], w_1 \rangle \\
\quad \rightarrow^* (\langle \mathbf{m}_3', \texttt{skip}, \texttt{qt}_3', \tau_3', w_3' \rangle \\
\quad \wedge \texttt{av}_2 \in (\tau_3' - (\tau_1 + +[\texttt{av}_1])))
\end{array} \right)
\end{array} \right)
\end{array}$$

Definition 25 (Annotated Variables May Dependency – Version 2)**.** .
One event $\texttt{av}_2$ *may depend on another one* $\texttt{av}_1$ *in a program* $\mathbf{c}$*, with a starting memory* $\mathbf{m}$ *and hidden*

database D, denoted as $\mathsf{DEP}_{\mathtt{av}}(\mathtt{av}_1, \mathtt{av}_2, c, \mathbf{m}, D)$ is defined below.

$$\exists \mathbf{m}, \mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3, \mathbf{m}'_2, \mathbf{m}'_3, \tau_1, \tau_2, \tau'_2, t_1, t_2, t'_2, \mathbf{c}_1, \mathbf{c}_2, v'_1, \mathbf{e}_2.$$
$$\left(\begin{array}{l} \langle \mathbf{m}, \mathbf{c}, [] \rangle \rightarrow^* \langle \mathbf{m}_1, [\mathbf{x}_1 \leftarrow v_1]^{l_1}; \mathbf{c}_1, \tau_1, t_1 \rangle \\ \bigwedge \langle \mathbf{m}_1[v_1/\mathbf{x}_1], c_1, \tau_1 + +[\mathtt{av}_1], t_1[\mathbf{x}_1] + + \rangle \\ \qquad \rightarrow^* \langle \mathbf{m}_2, [\mathbf{x}_2 \leftarrow \mathbf{e}_2]^{l_2}; \mathbf{c}_2, \tau_2, t_2 \rangle \\ \qquad \rightarrow^* \langle \mathbf{m}_3, \mathbf{c}_2, \tau_2 + +[\mathtt{av}_2], t_2[\mathbf{x}_2] + + \rangle \\ \bigwedge \langle \mathbf{m}_1[v'_1/\mathbf{x}_1], \mathbf{c}_1, \tau_1, t_1 \rangle \rightarrow^* \langle \mathbf{m}'_2, \mathbf{c}_2, \tau'_2, t'_2 \rangle \\ \bigwedge \mathtt{av}_2 \notin \tau'_2 \end{array}\right) \quad \begin{array}{l} \mathtt{av}_1 = (\mathbf{x}_1, v_1, l_1, n_1) \\ \mathtt{av}_2 = (\mathbf{x}_2, v_2, l_2, n_2) \end{array}$$

$$\exists \mathbf{m}, \mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3, \mathbf{m}'_2, \mathbf{m}'_3, \tau_1, \tau_2, \tau'_2, t_1, t_2, t'_2, \mathbf{c}_1, \mathbf{c}_2, v'_1, \psi_2.$$
$$\left(\begin{array}{l} \langle \mathbf{m}, \mathbf{c}, [] \rangle \rightarrow^* \langle \mathbf{m}_1, [\mathbf{x}_1 \leftarrow \mathtt{query}(\alpha_1)]^{l_1}; \mathbf{c}_1, \tau_1, t_1 \rangle \\ \bigwedge \langle \mathbf{m}_1[v_1/\mathbf{x}_1], c_1, \tau_1 + +[\mathtt{av}_1], t_1[\mathbf{x}_1] + + \rangle \\ \qquad \rightarrow^* \langle \mathbf{m}_2, [\mathbf{x}_2 \leftarrow \mathtt{query}(\psi_2)]^{l_2}; \mathbf{c}_2, \tau_2, t_2 \rangle \\ \qquad \rightarrow^* \langle \mathbf{m}_3, \mathbf{c}_2, \tau_2 + +[\mathtt{av}_2], t_2[\mathbf{x}_2] + + \rangle \\ \bigwedge \langle \mathbf{m}_1[v'_1/\mathbf{x}_1], \mathbf{c}_1, \tau_1, t_1 \rangle \rightarrow^* \langle \mathbf{m}'_2, \mathbf{c}_2, \tau'_2, t'_2 \rangle \\ \bigwedge \mathtt{av}_2 \notin \tau'_2 \end{array}\right) \quad \begin{array}{l} \mathtt{av}_1 = (\mathbf{x}_1, \alpha_1, l_1, n_1) \\ \mathtt{av}_2 = (\mathbf{x}_2, \alpha_2, l_2, n_2) \end{array}$$

$$\exists \mathbf{m}, \mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3, \mathbf{m}'_2, \mathbf{m}'_3, \tau_1, \tau_2, \tau'_2, t_1, t_2, t'_2, \mathbf{c}_1, \mathbf{c}_2, v'_1, \psi_2.$$
$$\left(\begin{array}{l} \langle \mathbf{m}, \mathbf{c}, [] \rangle \rightarrow^* \langle \mathbf{m}_1, [\mathbf{x}_1 \leftarrow v_1]^{l_1}; \mathbf{c}_1, \tau_1, t_1 \rangle \\ \bigwedge \langle \mathbf{m}_1[v_1/\mathbf{x}_1], c_1, \tau_1 + +[\mathtt{av}_1], t_1[\mathbf{x}_1] + + \rangle \\ \qquad \rightarrow^* \langle \mathbf{m}_2, [\mathbf{x}_2 \leftarrow \mathtt{query}(\psi_2)]^{l_2}; \mathbf{c}_2, \tau_2, t_2 \rangle \\ \qquad \rightarrow^* \langle \mathbf{m}_3, \mathbf{c}_2, \tau_2 + +[\mathtt{av}_2], t_2[\mathbf{x}_2] + + \rangle \\ \bigwedge \langle \mathbf{m}_1[v'_1/\mathbf{x}_1], \mathbf{c}_1, \tau_1, t_1 \rangle \rightarrow^* \langle \mathbf{m}'_2, \mathbf{c}_2, \tau'_2, t'_2 \rangle \\ \bigwedge \mathtt{av}_2 \notin \tau'_2 \end{array}\right) \quad \begin{array}{l} \mathtt{av}_1 = (\mathbf{x}_1, v_1, l_1, n_1) \\ \mathtt{av}_2 = (\mathbf{x}_2, \alpha_2, l_2, n_2) \end{array}$$

$$\exists \mathbf{m}, \mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3, \mathbf{m}'_2, \mathbf{m}'_3, \tau_1, \tau_2, \tau'_2, t_1, t_2, t'_2, \mathbf{c}_1, \mathbf{c}_2, v'_1, \mathbf{e}_2.$$
$$\left(\begin{array}{l} \langle \mathbf{m}, \mathbf{c}, [] \rangle \rightarrow^* \langle \mathbf{m}_1, [\mathbf{x}_1 \leftarrow v_1]^{l_1}; \mathbf{c}_1, \tau_1, t_1 \rangle \\ \bigwedge \langle \mathbf{m}_1[v_1/\mathbf{x}_1], c_1, \tau_1 + +[\mathtt{av}_1], t_1[\mathbf{x}_1] + + \rangle \\ \qquad \rightarrow^* \langle \mathbf{m}_2, [\mathbf{x}_2 \leftarrow \mathbf{e}_2]^{l_2}; \mathbf{c}_2, \tau_2, t_2 \rangle \\ \qquad \rightarrow^* \langle \mathbf{m}_3, \mathbf{c}_2, \tau_2 + +[\mathtt{av}_2], t_2[\mathbf{x}_2] + + \rangle \\ \bigwedge \langle \mathbf{m}_1[v'_1/\mathbf{x}_1], \mathbf{c}_1, \tau_1, t_1 \rangle \rightarrow^* \langle \mathbf{m}'_2, \mathbf{c}_2, \tau'_2, t'_2 \rangle \\ \bigwedge \mathtt{av}_2 \notin \tau'_2 \end{array}\right) \quad \begin{array}{l} \mathtt{av}_1 = (\mathbf{x}_1, \alpha_1, l_1, n_1) \\ \mathtt{av}_2 = (\mathbf{x}_2, v_2, l_2, n_2) \end{array}$$

Definition 26 (Variable May Dependency)**.** .
Given a program $\mathbf{c}$ with its assigned variables $\mathtt{aVar}_{\mathbf{c}}$, one variable $\mathbf{x}_2 \in \mathtt{aVar}_{\mathbf{c}}$ may depend on another variable $\mathbf{x}_1 \in \mathtt{aVar}_{\mathbf{c}}$ in $\mathbf{c}$ denoted as $\mathsf{DEP}_{\mathsf{var}}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{c})$ is defined below.

$$\exists \mathtt{av}_1, \mathtt{av}_2, \mathbf{m}, D.\ \pi_1(\mathtt{av}_1) = \mathbf{x}_1 \wedge \pi_1(\mathtt{av}_2) = \mathbf{x}_2 \wedge \mathsf{DEP}_{\mathtt{av}}(\mathtt{av}_1, \mathtt{av}_2, c, \mathbf{m}, D)$$

Definition 27 (Execution Based Dependency Graph)**.** .
Given a program $\mathbf{c}$, a database D, a starting memory $\mathbf{m}$ with its assigned variables $\mathtt{aVar}_c$ and initial variable counter $\mathtt{vcnt}_c^0$ with its corresponding execution: $\langle \mathbf{m}, \mathbf{c}, [], \mathtt{vcnt}_{\mathbf{c}}^0 \rangle \rightarrow^* \langle \mathbf{m}', \mathtt{skip}, \tau, \mathtt{vcnt} \rangle$, the dependency graph $\mathbf{G}_{\mathbf{trace}}(\mathbf{c}, \mathbf{m}, \mathbf{D}) = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{qF})$ is defined as:

$$\begin{array}{rlll} \textit{Vertices} & \mathtt{V} & := & \{x \in \mathcal{VAR} \mid x = \mathtt{aVar}_{\mathbf{c}}(i); i = 0, \ldots, |\mathtt{aVar}_{\mathbf{c}}|\} \\ \textit{Directed Edges} & \mathtt{E} & := & \{(x, x') \in \mathcal{VAR} \times \mathcal{VAR} \mid \mathsf{DEP}_{\mathsf{var}}(x, x', c); x = \mathtt{aVar}_{\mathbf{c}}(i); x' = \mathtt{aVar}_{\mathbf{c}}(j); i, j = 0, \ldots, |\mathtt{aVar}_{\mathbf{c}}|\} \\ \textit{Weights} & \mathtt{W} & := & \{(x, n) \in \mathcal{VAR} \times \mathbb{N} \mid n = \mathtt{vcnt}(x); x = \mathtt{aVar}_{\mathbf{c}}(i); i = 0, \ldots, |\mathtt{aVar}_{\mathbf{c}}|\} \\ \textit{Query Flags} & \mathtt{qF} & := & \left\{(x, n) \in \mathtt{V} \times \{0, 1\} \;\middle|\; \left\{ \begin{array}{ll} n = 1 & x \in \mathtt{qVar}_{\mathbf{c}} \\ n = 0 & o.w. \end{array} \right\}; x = \mathtt{aVar}_{\mathbf{c}}(i); i = 0, \ldots, |\mathtt{aVar}_{\mathbf{c}}| \right\} \end{array}$$

Definition 28 (Finite Walk (k))**.** .
Given a labeled weighted graph $G = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{qF})$*, a* finite walk k *in* G *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:*

- $e_i = (v_i, v_{i+1})$ *for every* $1 \le i < n$.
- *every vertex* $v \in \mathtt{V}$ *appears in this vertices sequence* $(v_1, \dots, v_n)$ *of* k *at most* $W(v)$ *times.*

$(v_1, \dots, v_n)$ *is the vertex sequence of this walk.*
Length of this finite walk k *is the number of vertices in its vertex sequence, i.e.,* $\mathtt{len}(k) = n$.

Given a labeled weighted graph $G = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{qF})$, we use $\mathcal{WALK}(G)$ to denote a set containing all finite walks k in G; and $k_{v_1 \to v_2} \in \mathcal{WALK}(G)$where $v_1, v_2 \in \mathtt{V}$ denotes the walk from vertex v_1 to v_2 .

Definition 29 (Length of Finite Walk w.r.t. Query ($\mathtt{len}_\mathtt{q}$))**.** .
Given a labeled weighted graph $G = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{qF})$ *and a* finite walk k *in* G *with its vertex sequence* $(v_1, \dots, v_n)$*, the length of* k *w.r.t query is defined as:*

$$\mathtt{len}_\mathtt{q}(k) = \mathtt{len}\big(v \mid v \in (v_1, \dots, v_n) \wedge \mathtt{F}(v) = 1\big)$$

, where $\big(v \mid v \in (v_1, \dots, v_n) \wedge \mathtt{F}(v) = 1\big)$ *is a subsequence of* k*'s vertex sequence.*

Given a program c with a starting memory m and database D, we generate its program-based graph $\mathbf{G_{trace}(c, m, D)} = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{qF})$. Then the adaptivity bound based on program analysis for $\mathbf{c}$ is the number of query vertices on a finite walk in $\mathbf{G_{prog}(c)}$. This finite walk satisfies:

- the number of query vertices on this walk is maximum
- the visiting times of each vertex v on this walk is bound by its weight $\mathtt{W}(v)$.

It is formally defined in 30.

Definition 30 (Adaptivity of A Program)**.** .
Given a program $\mathbf{c}$ *in SSA language, its adaptivity is defined for all possible starting SSA memory* $\mathbf{m}$ *and database* D *as follows:*

$$A(c) = \max\big\{\mathtt{len}_\mathtt{q}(k) \mid \mathbf{m} \in \mathcal{SM}, D \in \mathcal{DB}, k \in \mathcal{WALK}(\mathbf{G_{trace}(c, m, D)}\big\}$$

]

[[The following lemma describes a property of the trace-based dependency graph. For any program c with a database D and a starting memory m, the directed edges in its trace-based dependency graph can only be constructed from nodes representing smaller annotated queries to annotated queries of greater order. There doesn't exist backward edges with direction from greater annotated queries to smaller ones.]]

Lemma 12. *[Edges are Forwarding Only].*
Given a program c*, a database* D*, a starting memory* m *and the corresponding trace-based dependency graph* $G(c, D, m) = (\mathtt{V}, \mathtt{E})$*, for any directed edge* $(\mathtt{av}', \mathtt{av}) \in \mathtt{E}$*, this is not the case that:*

$$\mathtt{av}' \ge_{\mathtt{av}} \mathtt{av}$$

Proof. Proof in File: "`edge_forward.tex`". □

Lemma 13. *[Trace-based Dependency Graph is Directed Acyclic].*
Every trace-based dependency graph is a directed acyclic graph.

Proof. Proof is obvious based on the Lemma 12. □

Lemma 14 (Adaptivity is Bounded)**.** .
Given the program c with a certain database D and starting memory m, the $A(c)$ w.r.t. the D and m is bounded, i.e.,:

$$\langle m, c, [], [] \rangle \rightarrow^* \langle m', \texttt{skip}, t', w' \rangle \implies A_{D,m}(c) \leq |t'|$$

Proof. Proof is obvious based on the Lemma 13. □

3.6 SSA Transformation and Soundness of Transformation

in File "`ssa_transform_sound.tex`"

4 AdaptFun

There are four steps to get the adaptivity of a program $\mathbf{c}$ based on analyzing the program.

1. Collecting the variables that are newly assigned in the program (via assignment expressions). These variables are stored in an assigned variable vector `aVar`. We also track extra information of each assigned variable (whether it is assigned by a query result, or showing up in loop, or showing up in `if` expression or o.w.) and store it in a vector `F` of the same size as `aVar`.

2. Tracking the data flow relations between all these assigned variables. These informations are stored in a matrix `M`, whose size is $|\texttt{aVar}| \times |\texttt{aVar}|$.

3. Estimating the reachability bound of each variable in `aVar`.

4. With all these informations from previous steps, generating a program-based dependency graph $\mathbf{G_{prog}}$ and compute the adaptivity bound.

In the following subsections, we first define the notations and symbols being used in AdaptFunwith a simple example for understanding these definitions. Then we present the algorithmic analysis rules, which is the core of the AdaptFun, with 3 examples illustrating how AdaptFunworks. In the following subsections, we present the adaptivity analysis based on the AdaptFun's analyzing results, and the soundness w.r.t. the trace-based analyzing results in previous sections.

4.1 Notations

Definition 31 (Assigned Variables (`aVar`))**.** *Given a program* $\mathbf{c}$*, its assigned variables* `aVar` *is a vector containing all variables newly assigned in the program preserving the order. It is defined as follows:*

$$\texttt{aVar}_{\mathbf{c}} \triangleq \begin{cases} [\mathbf{x}] & \mathbf{c} = [\mathbf{x} \leftarrow \mathbf{e}]^{(l,w)} \\ [\mathbf{x}] & \mathbf{c} = [\mathbf{x} \leftarrow \texttt{query}\ (\boldsymbol{\psi})]^{(l,w)} \\ \texttt{aVar}_{\mathbf{c_1}} +\!+ \texttt{aVar}_{\mathbf{c_2}} & \mathbf{c} = \mathbf{c_1}; \mathbf{c_2} \\ \texttt{aVar}_{\mathbf{c_1}} +\!+ \texttt{aVar}_{\mathbf{c_2}} +\!+ [\bar{\mathbf{x}}, \bar{\mathbf{y}}, \bar{\mathbf{z}}] & \mathbf{c} = \texttt{if}\ ([\mathbf{b}]^{(l,w)}, [\bar{\mathbf{x}}, \bar{\mathbf{x}_2}, \bar{\mathbf{x}_2}], [\bar{\mathbf{y}}, \bar{\mathbf{y}_2}, \bar{\mathbf{y}_3}], [\bar{\mathbf{z}}, \bar{\mathbf{z}_2}, \bar{\mathbf{z}_3}], \mathbf{c_1}, \mathbf{c_2}) \\ \texttt{aVar}_{\mathbf{c}'} +\!+ [\bar{\mathbf{x}}] & \mathbf{c} = \texttt{while}\ ([\mathbf{b}]^{(l,w)}, [\bar{\mathbf{x}}, \bar{\mathbf{x}_2}, \bar{\mathbf{x}_2}], \mathbf{c}') \end{cases}$$

[JL: We are abusing the notations and operators from list here. The notation [] represents an empty vector and $x :: A$ represents add an element x to the head of the vector A. The concatenation operation between 2 vectors A_1 and A_2, i.e., $A_1 +\!+ A_2$ is mimic the standard list concatenation operations as follows:

$$A_1 +\!+ A_2 \triangleq \begin{cases} A_2 & A_1 = [] \\ x :: (A_1' +\!+ A_2) & A_1 = x :: A_1' \end{cases} \tag{13}$$

We use index within parenthesis to denote the access to the element of corresponding location, $A(i)$ denotes the element at location i in the vector A and $M(i, j)$ denotes the element at location i-th raw, i-th column in the matrix M.]

Consider the program c below in the left hand side as an example, its assigned variables `aVar` (short for $\texttt{aVar}(\mathbf{c})$) is as in the right hand side is shown as follows:

$$\mathbf{c} = \begin{array}{l} [\mathbf{x_1} \leftarrow \texttt{query}(\mathbf{0})]^1; \\ [\mathbf{x_2} \leftarrow \mathbf{x_1} + \mathbf{1}]^2; \\ [\mathbf{x_3} \leftarrow \mathbf{x_2} + \mathbf{2}]^3 \end{array} \qquad \texttt{aVar} = \begin{bmatrix} \mathbf{x_1} \\ \mathbf{x_2} \\ \mathbf{x_3} \end{bmatrix}$$

Lemma 15. *For any program* $\mathbf{c}$*, every variable in* $\mathtt{aVar}(\mathbf{c})$ *is distinct*

Proof. It is due to the SSA nature. We can prove it by induction on $\mathbf{c}$. □

[JL:

Definition 32 (Variable Flags ($\mathtt{F}$)). .
Given a program $\mathbf{c}$ *with its assigned variables* $\mathtt{aVar}$*, the* $\mathtt{F}$ *is a vector of the same length as* $\mathtt{aVar}$*, s.t. for each variable* $\mathbf{x}$ *showing up as the* i*-th element in* $\mathtt{aVar}$ *(i.e.,* $\mathbf{x} = \mathtt{aVar}(i)$*),* $\mathtt{F}(i) \in \{0,1,2\}$ *is defined as follows:*

$$\mathtt{F}(i) := \left\{ \begin{array}{ll} 2 & \mathbf{x} = \mathtt{aVar}(i) \land (\exists \boldsymbol{\psi}.\ s.t.,\ [\mathbf{x} \leftarrow \mathtt{query}(\boldsymbol{\psi})]^{l} \in_c \mathbf{c}) \\ & \mathbf{x} = \mathtt{aVar}(i) \bigwedge \\ 1 & \left(\begin{array}{l} \left(\exists\, \mathbf{c}', \mathbf{e}, \mathbf{b}, l, l'.\ \mathtt{while}\ [\mathbf{b}]^{l}\ \mathtt{do}\ \mathbf{c}' \in_c \mathbf{c} \land [\mathbf{x} \leftarrow \mathbf{e}]^{l'} \in_c \mathbf{c}'\right) \bigvee \\ \left(\exists\, \mathbf{b}, l, l_1, l_2, \mathbf{c_1}, \mathbf{c_2}, \mathbf{e}_1, \mathbf{e}_2.\ \mathtt{if}\ ([\mathbf{b}]^{l}, \mathbf{c_1}, \mathbf{c_2}) \in_c \mathbf{c} \land ([\mathbf{x} \leftarrow \mathbf{e_1}]^{l1} \in_c \mathbf{c_1} \lor [\mathbf{x} \leftarrow \mathbf{e_2}]^{l2} \in_c \mathbf{c_2})\right) \end{array} \right) \\ 0 & o.w. \end{array} \right\}.$$

Operations on $\mathtt{F}$ are defined as follows:

$$\begin{array}{llll} \mathtt{F}_1 \uplus \mathtt{F}_2(i) & := & \left\{ \begin{array}{ll} k & k = \max\{\mathtt{F}_1(i), \mathtt{F}_2(i)\} \land |\mathtt{F}_1| = |\mathtt{F}_2| \\ 0 & o.w. \end{array} \right. & i = 1, \cdots, |\mathtt{F}_1| \\ \mathtt{F} \uplus n(i) & := & \max\{\mathtt{F}(i), n\} & i = 1, \ldots, |\mathtt{F}| \\ [n]^{k}(i) & := & n & i = 1, \ldots, k\ \land\ |[n]^{k}| = k \end{array} \tag{14}$$

[[Given a program c with its assigned variables $\mathtt{aVar}$**, and two variables x, y showing up as** i**-th,** j**-th elements in** $\mathtt{aVar}$ **(i.e.,** $\mathbf{x} = \mathtt{aVar}(i)$ **and** $\mathbf{y} = \mathtt{aVar}(j)$**), we say y flows to x in c if and only if** $j < i$ **and the value of y directly or indirectly influence the evaluation of the value of x as follows:**

- **(Directly Influence) The program c contains either a command** $\mathbf{x} \leftarrow \mathbf{e}$ **or** $\mathbf{x} \leftarrow \mathtt{query}(\boldsymbol{\psi})$**, such that y shows up as a free variable in e or** $\boldsymbol{\psi}$**. We use** $\mathtt{flowsTo}(\mathbf{x}, \mathbf{y}, \mathbf{c})$ **to denote y flows to x in c.**
- **(Indirectly Influence) The program c contains either a while loop command or if condition command, such that y shows up in the guard and x shows up in the left hand of an assignment command in the body.**

This is formally defined in 33. We use $FV(e)$**,** $FV(\mathbf{b})$ **and** $FV(\psi)$ **denote the set of free variables in expression** e**, boolean expression b and query expression** ψ **respectively.**

Definition 33 (Data Flows between Assigned Variables ($\mathtt{flowsTo}$)). .
Given a program $\mathbf{c}$ *with its assigned variables* $\mathtt{aVar}$*, and two variables* $\mathbf{x}$*,* $\mathbf{y}$ *s.t.,* $\mathbf{x} = \mathtt{aVar}(i)$ *and* $\mathbf{y} = \mathtt{aVar}(j)$*,* $\mathbf{y}$ *flows to* $\mathbf{x}$ *in* $\mathbf{c}$*, i.e.,* $\mathtt{flowsTo}(\mathbf{x}, \mathbf{y}, \mathbf{c})$ *is defined as:*

$$\begin{array}{l} \mathtt{flowsTo}(\mathbf{x}, \mathbf{y}, \mathbf{c}) \triangleq (j < i) \land \\ \left(\bigvee \begin{array}{l} (\exists \mathbf{e}, l.\ [\mathbf{x} \leftarrow \mathbf{e}]^{l} \in_c \mathbf{c} \land \mathbf{y} \in FV(\mathbf{e})) \\ (\exists \boldsymbol{\psi}, l.\ [\mathbf{x} \leftarrow \mathtt{query}(\boldsymbol{\psi})]^{l} \in_c \mathbf{c} \land \mathbf{y} \in FV(\boldsymbol{\psi})) \\ \left(\exists\, \mathbf{c}', \mathbf{e}, \mathbf{b}, l, l'.\ \mathtt{while}\ [\mathbf{b}]^{l}\ \mathtt{do}\ \mathbf{c}' \in_c \mathbf{c} \land [\mathbf{x} \leftarrow \mathbf{e}]^{l'} \in_c \mathbf{c}' \land \mathbf{y} \in FV(\mathbf{b})\right) \\ \left(\exists\, \mathbf{b}, l, l_1, l_2, \mathbf{c_1}, \mathbf{c_2}, \mathbf{e}_1, \mathbf{e}_2.\ \mathtt{if}\ ([\mathbf{b}]^{l}, \mathbf{c_1}, \mathbf{c_2}) \in_c \mathbf{c} \land ([\mathbf{x} \leftarrow \mathbf{e_1}]^{l1} \in_c \mathbf{c_1} \lor [\mathbf{x} \leftarrow \mathbf{e_2}]^{l2} \in_c \mathbf{c_2}) \land \mathbf{y} \in FV(\mathbf{b})\right) \end{array} \right). \end{array}$$

]]]

Definition 34 (Data Flow Matrix ($\mathtt{M}$))**.** *Given a program* $\mathbf{c}$ *with its assigned variables* $\mathtt{aVar}$ *of length* N*, its data flow matrix* $\mathtt{M}$ *is a matrix of size* $N \times N$ *s.t.* $\forall \mathbf{x},\mathbf{y} \in \mathtt{aVar}.\, \mathbf{x} = \mathtt{aVar}(i), \mathbf{y} = \mathtt{aVar}(j)$*:*

$$\mathtt{M}(i,j) \triangleq \begin{cases} 1 & \mathtt{flowsTo}(\mathbf{x},\mathbf{y},\mathbf{c}) \\ 0 & o.w. \end{cases} ,\mathbf{x} = \mathtt{aVar}(i); \mathbf{y} = \mathtt{aVar}(j); i,j = 1,\ldots,N.$$

Operations on the data flow matrices are defined as follows:

$$\mathtt{M}_1;\mathtt{M}_2 := \mathtt{M}_2 \cdot \mathtt{M}_1 + \mathtt{M}_1 + \mathtt{M}_2 \tag{15}$$

Consider the same program c as above, its data flow matrix $\mathtt{M}$ and $\mathtt{F}$ for the program c is as follows:

$$\mathbf{c} = \begin{array}{l} [\mathbf{x_1} \leftarrow \mathtt{query}(\mathbf{0})]^1; \\ [\mathbf{x_2} \leftarrow \mathbf{x_1} + \mathbf{1}]^2; \\ [\mathbf{x_3} \leftarrow \mathbf{x_2} + \mathbf{2}]^3 \end{array} \qquad \mathtt{M} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix}, \mathtt{F} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

There are two special matrices used for generating the data flow matrix $\mathtt{M}$ in the analysis algorithm. They are the left matrix $\mathtt{LM}_i$ and right matrix $\mathsf{R}_{(\mathsf{e},\mathsf{i})}$.

Given a program $\mathbf{c}$ with its assigned variables $\mathtt{aVar}$ of length N, the left matrix $\mathtt{LM}_i$ generates a matrix of 1 column, N rows, where the i-th row is 1 and all the other rows are 0.

Definition 35 (Left Matrix ($\mathtt{LM}_i$))**.** .
Given a program $\mathbf{c}$ *with its assigned variables* $\mathtt{aVar}$ *of length* N*, the left matrix* $\mathtt{LM}_i$ *is defined as follows:*

$$\mathtt{LM}_i(j) := \begin{cases} 1 & j = i \\ 0 & o.w. \end{cases} , j = 1,\ldots,N.$$

Given a program $\mathbf{c}$ with its assigned variables $\mathtt{aVar}$ of length N, the right matrix $\mathtt{RM}_{e,i}$ generates a matrix of one row and N columns, where the locations of free variables in e is marked as 1.

Definition 36 (Right Matrix ($\mathtt{RM}_e$))**.** .
Given a program $\mathbf{c}$ *with its assigned variables* $\mathtt{aVar}$ *of length* N*, the right matrix* $\mathtt{RM}_e$ *is defined as follows:*

$$\mathtt{RM}_e(j) := \begin{cases} 1 & \mathbf{x} \in FV(e) \\ 0 & o.w. \end{cases} ,\mathbf{x} = \mathtt{aVar}(j)\ ,\ j = 1,\ldots,N.$$

Using the same example program $\mathbf{c}$ as above with assigned variables $\mathtt{aVar} = [\mathbf{x_1},\mathbf{x_2},\mathbf{x_3}]$, the left and right matrices w.r.t. its 2-nd command $[\mathbf{x_2} \leftarrow \mathbf{x_1} + \mathbf{1}]^2$ are as follows:

$$\mathtt{LM}_1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \qquad \mathtt{RM}_{\mathbf{x}_1+1} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$

4.2 Algorithmic Analysis Rules

Variable Collection Algorithm, VetxCol The **VetxCol** algorithm shows how the assigned variables $\mathtt{aVar}$ are collected (via the command $\mathbf{x} \leftarrow \mathbf{e}$ or $\mathbf{x} \leftarrow \mathtt{query}(\boldsymbol{\psi})$) from the program $\mathbf{c}$ in the first step, along with constructing the flag for each variable, i.e., $\mathtt{F}$. The algorithmic rules for **VetxCol** algorithm is defined in Figure 3. It has the form: [JL: $\mathtt{VetxCol}(\mathtt{aVar};\mathtt{F};\mathbf{c}) \rightarrow (\mathtt{aVar}';\mathtt{F}')$]. The input of **VetxCol** is a program $\mathbf{c}$, the assigned variables $\mathtt{aVar}$ collected before the program $\mathbf{c}$ as well as the flags $\mathtt{F}$ for every corresponding variable . The output of the algorithm is the updated assigned variables $\mathtt{aVar}'$ and flags

[JL:

$$\frac{}{\mathsf{VetxCol}(\mathtt{aVar};\mathtt{F};[\mathbf{x}\leftarrow\mathbf{e}]^{\mathbf{l}}) \rightarrow (\mathtt{aVar}++[\mathbf{x}];\mathtt{F}++[0])}\ \textbf{VetxCol-asgn}$$

$$\frac{}{\mathsf{VetxCol}(\mathtt{aVar};\mathtt{F};[\mathbf{x}\leftarrow \mathtt{query}(\boldsymbol{\psi})]^{l}) \rightarrow (\mathtt{aVar}++[\mathbf{x}];\mathtt{F}++[2])}\ \textbf{VetxCol-query}$$

$$\frac{\begin{array}{c}\mathsf{VetxCol}(\mathtt{aVar};[];\mathbf{c_1}) \rightarrow (\mathtt{aVar}_1;\mathtt{F}_1) \qquad \mathsf{VetxCol}(\mathtt{aVar}_1;[];\mathbf{c_2}) \rightarrow (\mathtt{aVar}_2;\mathtt{F}_2) \qquad \mathtt{aVar}' = [\bar{\mathbf{x}}]++[\bar{\mathbf{y}}]++[\bar{\mathbf{z}}] \\ k = \mathtt{len}(\mathtt{aVar}') \qquad \mathtt{aVar}_3 = \mathtt{aVar}_2++\mathtt{aVar}' \qquad \mathtt{F}_3 = \mathtt{F}++((\mathtt{F}_1++\mathtt{F}_2)\uplus 1)++([1]^k)\end{array}}{\mathsf{VetxCol}(\mathtt{aVar};\mathtt{F};[\ \mathtt{if}\ (\mathbf{b},[\bar{\mathbf{x}},\bar{\mathbf{x_1}},\bar{\mathbf{x_2}}],[\bar{\mathbf{y}},\bar{\mathbf{y_1}},\bar{\mathbf{y_2}}],[\bar{\mathbf{z}},\bar{\mathbf{z_1}},\bar{\mathbf{z_2}}],\mathbf{c_1},\mathbf{c_2})]^{l}) \rightarrow (\mathtt{aVar}_3;\mathtt{F}_3)}\ \textbf{VetxCol-if}$$

$$\frac{\mathsf{VetxCol}(\mathtt{aVar};\mathtt{F}\mathbf{c_1}) \rightarrow (\mathtt{aVar}_1;\mathtt{F}_1) \qquad \mathsf{VetxCol}(\mathtt{aVar}_1;\mathtt{F}_1;\mathbf{c_2}) \rightarrow (\mathtt{aVar}_2;\mathtt{F}_2)}{\mathsf{VetxCol}(\mathtt{aVar};\mathtt{F};(\mathbf{c_1};\mathbf{c_2})) \rightarrow (\mathtt{aVar}_2;\mathtt{F}_2)}\ \textbf{VetxCol-seq}$$

$$\frac{\mathsf{VetxCol}(\mathtt{aVar};[];\mathbf{c}) \rightarrow (\mathtt{aVar}';\mathtt{F}') \qquad \mathtt{aVar}'' = \mathtt{aVar}'++[\bar{\mathbf{x}}] \qquad \mathtt{F}'' = \mathtt{F}++(\mathtt{F}'\uplus 1)++([1]^{\mathtt{len}([\bar{\mathbf{x}}])})}{\mathsf{VetxCol}(\mathtt{aVar};\mathtt{F};\ \mathtt{while}\ [\mathbf{b}]^{l},\mathbf{n},[\bar{\mathbf{x}},\bar{\mathbf{x_1}},\bar{\mathbf{x_2}}]\ \mathtt{do}\ \mathbf{c}) \rightarrow (\mathtt{aVar}'';\mathtt{F}'')}\ \textbf{VetxCol-while}$$

]

Figure 3: The Algorithmic Rules of **VetxCol**

$\mathtt{F}'$ thorough the program $\mathbf{c}$ The assignment commands are the source of variables **VetxCol** collecting, in the case **VetxCol-asgn** and **VetxCol-query**, the output assigned variables are extended by $\mathbf{x}$.
When it comes to the `if` ... `then` ... `else` command in the rule **VetxCol-if**, variables assigned in the then branch $\mathbf{c_1}$, as well as the variables assigned in the else branch $\mathbf{c_2}$, and the new generated variables $\bar{\mathbf{x}},\bar{\mathbf{y}},\bar{\mathbf{z}}$ in $[\bar{\mathbf{x}},\bar{\mathbf{x_1}},\bar{\mathbf{x_2}}],[\bar{\mathbf{y}},\bar{\mathbf{y_1}},\bar{\mathbf{y_2}}],[\bar{\mathbf{z}},\bar{\mathbf{z_1}},\bar{\mathbf{z_2}}]$.
The sequence command $\mathbf{c_1};\mathbf{c_2}$ is standard by accumulating the predicted variables in the two commands $\mathbf{c_1}$ and $\mathbf{c_2}$ preserving their order.
The while command `while` $\mathbf{b},[\bar{\mathbf{x}}]\ldots$ `do` $\mathbf{c}$ considers the newly generated variables by SSA transformation $\bar{\mathbf{x}}$ as well and the newly assigned variables in its body $\mathbf{c}$.

Below we present the definition for a valid index, to have a clear understanding on the variable collecting algorithm: [JL:

Definition 37 (Valid Index (Remove?))**.** *Given an assigned variable list* $\mathtt{aVar}$, $\mathtt{aVar};\vDash(\mathbf{c},i_1,i_2)$ *iff* $\mathtt{aVar}' = \mathtt{aVar}[0,\ldots,i_1-1],\mathtt{aVar}';\mathbf{c}\rightarrow\mathtt{aVar}''\wedge\mathtt{aVar}'' = \mathtt{aVar}[0,\ldots,i_2-1]$.

]

Data Flow Matrix Generating Algorithm In this data flow matrix generating algorithm, we analyze the data flow information among all assigned variables $\mathtt{aVar}$ collected via the the **VetxCol** algorithm of length N. We track the data flow relations between all these assigned variables. These informations are stored in a matrix $\mathtt{M}$, whose size is $N\times N$. The algorithm to fill in the matrix is of the form: [JL: $\mathsf{FlowGen}(\Gamma;\mathbf{c};\mathtt{aVar})\rightarrow(\mathtt{M})$] $\mathsf{FlowGen}(\Gamma;\mathbf{c};i_1,i_2)\rightarrow(\mathtt{M};\mathtt{F})$. Γ is a vector records the variables the current program $\mathbf{c}$ depends on, the index i_1 is a pointer which refers to the position of the first new-generated variable in $\mathbf{c}$ in the assigned variables $\mathtt{aVar}$, and i_2 points to the first new variable that is not in $\mathbf{c}$ (if exists). [JL:

Definition 38 (Valid Gamma (Remove?)). $\Gamma \vDash i_1$ *iff* $\forall i \geq i_1, \Gamma(i_1) = 0$.

] $\boxed{\Gamma \vdash^{i_1,i_2}_{\mathtt{M},\mathtt{F}} c}$

$$\frac{\mathtt{M} = \mathtt{LM}_i * (\mathtt{RM}_{\mathbf{e},i} + \Gamma)}{\mathsf{FlowGen}(\Gamma; [\mathbf{x} \leftarrow \mathbf{e}]^l; i) \rightarrow (\mathtt{M}; \mathtt{F}_0; i+1)}\ \textbf{FlowGen-asgn}$$

$$\frac{\mathtt{M} = \mathtt{LM}_i * (\mathtt{RM}_{\mathbf{e},i} + \Gamma) \qquad \mathtt{F} = \mathtt{LM}_i \qquad \mathtt{F}(i) = 1}{\mathsf{FlowGen}(\Gamma; [\mathbf{x} \leftarrow \mathtt{query}(\mathbf{e})]^l; i) \rightarrow (\mathtt{M}; \mathtt{F}; i+1)}\ \textbf{FlowGen-query}$$

$$\frac{\begin{array}{c}\mathsf{FlowGen}(\Gamma + \mathtt{RM}_{\mathbf{b},i_1}; \mathbf{c_1}; i_1) \rightarrow (\mathtt{M}_1; \mathtt{F}_1; i_2) \qquad \mathsf{FlowGen}(\Gamma + \mathtt{RM}_{\mathbf{b},i_1}; \mathbf{c_2}; i_2) \rightarrow (\mathtt{M}_2; \mathtt{F}_2; i_3) \\ \mathsf{FlowGen}(\Gamma; [\bar{\mathbf{x}}, \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}]; i_3) \rightarrow (M_x; \mathtt{F}_\emptyset; i_3 + |\bar{\mathbf{x}}|) \\ \mathsf{FlowGen}(\Gamma; [\bar{\mathbf{y}}, \bar{\mathbf{y_1}}, \bar{\mathbf{y_2}}]; i_3 + |\bar{\mathbf{x}}|) \rightarrow (\mathtt{M}_y; \mathtt{F}_\emptyset; i_3 + |\bar{\mathbf{x}}| + |\bar{\mathbf{y}}|) \\ \mathsf{FlowGen}(\Gamma; [\bar{\mathbf{z}}, \bar{\mathbf{z_1}}, \bar{\mathbf{z_2}}]; i_3 + |\bar{\mathbf{x}}| + |\bar{\mathbf{y}}|) \rightarrow (\mathtt{M}_y; \mathtt{F}_\emptyset; i_3 + |\bar{\mathbf{x}}| + |\bar{\mathbf{y}}| + |\bar{\mathbf{z}}|) \qquad \mathtt{M} = (\mathtt{M}_1 + \mathtt{M}_2) + \mathtt{M}_x + \mathtt{M}_y + \mathtt{M}_z\end{array}}{\mathsf{FlowGen}(\Gamma;\ \mathtt{if}\ ([\mathbf{b}]^l, [\bar{\mathbf{x}}, \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}], [\bar{\mathbf{y}}, \bar{\mathbf{y_1}}, \bar{\mathbf{y_2}}], [\bar{\mathbf{z}}, \bar{\mathbf{z_1}}, \bar{\mathbf{z_2}}], \mathbf{c_1}, \mathbf{c_2}); i_1) \rightarrow (\mathtt{M}; \mathtt{F}_1 \uplus \mathtt{F}_2 \uplus 2; i_3 + |\bar{x}| + |\bar{y}| + |\bar{z}|)}\ \textbf{FlowGen-if}$$

$$\frac{\mathsf{FlowGen}(\Gamma; \mathbf{c_1}; i_1) \rightarrow (\mathtt{M}_1; \mathtt{F}_1; i_2) \qquad \mathsf{FlowGen}(\Gamma; \mathbf{c_2}; i_2) \rightarrow (\mathtt{M}_2; \mathtt{F}_2; i_3)}{\mathsf{FlowGen}(\Gamma; (\mathbf{c_1}; \mathbf{c_2}); i_1) \rightarrow ((\mathtt{M}_1; \mathtt{M}_2); \mathtt{F}_1 \uplus V_2; i_3)}\ \textbf{FlowGen-seq}$$

$$\frac{\begin{array}{c}B = |\bar{\mathbf{x}}| \qquad A = |\mathbf{c}| \qquad \mathsf{FlowGen}(\Gamma; [\bar{\mathbf{x}}, \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}]; i + (B + A)) \rightarrow (\mathtt{M}_1; V_1; i + B + (B + A)) \\ \mathsf{FlowGen}(\Gamma; \mathbf{c}; i + B + (B + A)) \rightarrow (\mathtt{M}_2; \mathtt{F}_2; i + B + A + (B + A)) \\ \mathsf{FlowGen}(\Gamma; [\bar{\mathbf{x}}, \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}]; i + (B + A)) \rightarrow (\mathtt{M}; \mathtt{F}; i + (B + A) + B) \\ \mathtt{M}' = \mathtt{M} + (\mathtt{M}_1 + \mathtt{M}_2) \qquad \mathtt{F}' = \mathtt{F} \uplus ((\mathtt{F}_1 \uplus \mathtt{F}_2) \uplus 2)\end{array}}{\mathsf{FlowGen}(\Gamma;\ \mathtt{while}\ [b]^l\ \mathbf{n}\ [\bar{\mathbf{x}}, \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}]\ \mathtt{do}\ c; i) \rightarrow (\mathtt{M}'; \mathtt{F}'; i + (B + A) + B)}\ \textbf{FlowGen-while}$$

[JL: Updated Flow Generation Algorithm] [JL: $\boxed{\Gamma \vdash_{\mathtt{M},\mathtt{aVar}} \mathbf{c}}$

$$\frac{\mathbf{x} = \mathtt{aVar}(i) \qquad \mathtt{M} = \mathtt{LM}_i * (\mathtt{RM}_{\mathbf{e}} + \Gamma)}{\mathsf{FlowGen}(\Gamma; [\mathbf{x} \leftarrow \mathbf{e}]^l; \mathtt{aVar}) \rightarrow (\mathtt{M})}\ \textbf{FlowGen-asgn}$$

$$\frac{\mathbf{x} = \mathtt{aVar}(i) \qquad \mathtt{M} = \mathtt{LM}_i * (\mathtt{RM}_{\mathbf{e}} + \Gamma)}{\mathsf{FlowGen}(\Gamma; [\mathbf{x} \leftarrow \mathtt{query}(\boldsymbol{\psi})]^l; \mathtt{aVar}) \rightarrow (\mathtt{M})}\ \textbf{FlowGen-query}$$

$$\frac{\begin{array}{c}\mathsf{FlowGen}(\Gamma + \mathtt{RM}_{\mathbf{b}}; \mathbf{c_1}; \mathtt{aVar}) \rightarrow (\mathtt{M}_1) \qquad \mathsf{FlowGen}(\Gamma + \mathtt{RM}_{\mathbf{b}}; \mathbf{c_2}; \mathtt{aVar}) \rightarrow (\mathtt{M}_2) \\ \mathsf{FlowGen}(\Gamma; [\bar{\mathbf{x}}, \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}]; \mathtt{aVar}) \rightarrow (\mathtt{M}_x) \qquad \mathsf{FlowGen}(\Gamma; [\bar{\mathbf{y}}, \bar{\mathbf{y_1}}, \bar{\mathbf{y_2}}]; \mathtt{aVar}) \rightarrow (\mathtt{M}_y) \\ \mathsf{FlowGen}(\Gamma; [\bar{\mathbf{z}}, \bar{\mathbf{z_1}}, \bar{\mathbf{z_2}}]; \mathtt{aVar}) \rightarrow (\mathtt{M}_z) \qquad \mathtt{M} = (\mathtt{M}_1 + \mathtt{M}_2) + \mathtt{M}_x + \mathtt{M}_y + \mathtt{M}_z\end{array}}{\mathsf{FlowGen}(\Gamma;\ \mathtt{if}\ ([\mathbf{b}]^l, [\bar{\mathbf{x}}, \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}], [\bar{\mathbf{y}}, \bar{\mathbf{y_1}}, \bar{\mathbf{y_2}}], [\bar{\mathbf{z}}, \bar{\mathbf{z_1}}, \bar{\mathbf{z_2}}], \mathbf{c_1}, \mathbf{c_2})) \rightarrow (\mathtt{M})}\ \textbf{FlowGen-if}$$

$$\frac{\mathsf{FlowGen}(\Gamma; \mathbf{c_1}; \mathtt{aVar}) \rightarrow (\mathtt{M}_1) \qquad \mathsf{FlowGen}(\Gamma; \mathbf{c_2}; \mathtt{aVar}) \rightarrow (\mathtt{M}_2)}{\mathsf{FlowGen}(\Gamma; (\mathbf{c_1}; \mathbf{c_2}); \mathtt{aVar}) \rightarrow ((\mathtt{M}_1; \mathtt{M}_2))}\ \textbf{FlowGen-seq}$$

$$\frac{\mathsf{FlowGen}(\Gamma + \mathtt{RM}_{\mathbf{b}}; \mathbf{c}; \mathtt{aVar}) \rightarrow (\mathtt{M}_1) \qquad \mathsf{FlowGen}(\Gamma; [\bar{\mathbf{x}}, \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}]; \mathtt{aVar}) \rightarrow (\mathtt{M}_2)}{\mathsf{FlowGen}(\Gamma;\ \mathtt{while}\ [\mathbf{b}]^l, \mathbf{n}, [\bar{\mathbf{x}}, \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}]\ \mathtt{do}\ \mathbf{c}; \mathtt{aVar}) \rightarrow (\mathtt{M}_1 + \mathtt{M}_2)}\ \textbf{FlowGen-while}$$

] Below we define the valid data flow matrix, to have a clear understanding on the data flow generating algorithm:

Definition 39 (Valid Matrix)**.** *For a assigned variables* $\texttt{aVar}$, $\texttt{aVar} \vDash (\texttt{M}, \texttt{F})$ *iff the cardinality of* $\texttt{aVar}$ *equals to the one of* $\texttt{F}$, $|\texttt{aVar}| = |\texttt{F}|$ *and the matrix* $\texttt{M}$ *is of size* $|\texttt{F}| \times |\texttt{F}|$.

[JL:

Definition 40 (Valid Matrix)**.** *Given a program* $\mathbf{c}$ *with its assigned variables* $\texttt{aVar}$, $\texttt{aVar} \vDash \texttt{M}$ *iff the cardinality of* $\texttt{M}$ *equals to the product of* $\texttt{aVar}$*'s cardinality, i.e.,* $|\texttt{M}| = |\texttt{aVar}| \times |\texttt{aVar}|$.

]

Reachability Bounds Given a program c with its assigned variables $\texttt{aVar}$, we use the $\mathsf{RechBound}(\mathbf{x}, \mathbf{c})$ algorithm, from paper [2], to estimate the reachability bound for each variable $\mathbf{x} \in \texttt{aVar}$. The input of RechBound is a program $\mathbf{c}$ in SSA language and a variable $\mathbf{x}$ from $\mathbf{c}$. The output of $\mathsf{RechBound}(\mathbf{x}, \mathbf{c})$ is an integer representing the reachability bound of $\mathbf{x}$ in $\mathbf{c}$.

The following example programs $\mathbf{c}2$ and $\mathbf{c}3$ with while loop illustrate how the algorithm works. The collected assigned variables, $\texttt{aVar}_{\mathbf{c}2}$ and $\texttt{aVar}_{\mathbf{c}3}$, data flow matrix $\texttt{M}_{\mathbf{c}2}$ and $\texttt{M}_{\mathbf{c}3}$ and variable flags $\texttt{F}_{\mathbf{c}2}$ and $\texttt{F}_{\mathbf{c}3}$ for program $\mathbf{c}2$ and $\mathbf{c}3$ are presented in the right hand side.

$$\mathbf{c}2 \triangleq \begin{array}{l} [\mathbf{x_1} \leftarrow \texttt{query}(1)]^1; \\ [\mathbf{i_1} \leftarrow 0]^2; \\ \texttt{while } [\mathbf{i_1} < 2]^3 \\ \quad [\mathbf{x_3}, \mathbf{x_1}, \mathbf{x_2}], [\mathbf{i_3}, \mathbf{i_1}, \mathbf{i_2}] \texttt{ do} \\ \quad \big([\mathbf{y_1} \leftarrow \texttt{query}(2)]^4; \\ [\mathbf{x_2} \leftarrow \mathbf{y_1} + \mathbf{x_3}]^5; \\ [\mathbf{i_2} \leftarrow \mathbf{1} + \mathbf{i_3}]^6\big); \\ [\mathbf{z_1} \leftarrow \mathbf{x_3} + 2]^7 \end{array}, \quad \texttt{aVar}_{\mathbf{c}2} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_3 \\ \mathbf{y}_1 \\ \mathbf{x}_2 \\ \mathbf{z}_1 \\ \mathbf{i}_1 \\ \mathbf{i}_2 \\ \mathbf{i}_3 \end{bmatrix}, \quad \texttt{M}_{\mathbf{c}2} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}, \quad \texttt{F}_{\mathbf{c}2} = \begin{bmatrix} 1 \\ 2 \\ 1 \\ 2 \\ 0 \\ 0 \\ 2 \\ 1 \end{bmatrix}$$

$$\mathbf{c}3 \triangleq \begin{array}{l} [\mathbf{x}_1 \leftarrow \texttt{query}(1)]^1; \\ [\mathbf{i_1} \leftarrow 1]^2; \\ \texttt{while } [i < 0]^3, \\ \quad [\mathbf{x_3}, \mathbf{x_1}, \mathbf{x_2}], [\mathbf{i_3}, \mathbf{i_1}, \mathbf{i_2}] \texttt{ do} \\ \quad \big([\mathbf{y_1} \leftarrow \texttt{query}(2)]^3; \\ [\mathbf{x_2} \leftarrow \mathbf{y_1} + \mathbf{x_3}]^5\big); \\ [\mathbf{z_1} \leftarrow \mathbf{x_3} + 2]^6 \end{array}, \quad \texttt{aVar}_{\mathbf{c}3} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{i}_1 \\ \mathbf{x}_3 \\ \mathbf{i}_3 \\ \mathbf{z}_1 \end{bmatrix}, \quad \texttt{M}_{\mathbf{c}3} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \quad \texttt{F}_{\mathbf{c}3} = \begin{bmatrix} 1 \\ 0 \\ 2 \\ 2 \\ 0 \end{bmatrix}$$

We can now look at the if statement.

$$\mathbf{c}4 \triangleq \begin{array}{l} [\mathbf{x}_1 \leftarrow \texttt{query}(1)]^1; \\ [\mathbf{y}_1 \leftarrow \texttt{query}(2)]^2; \\ \texttt{if } (\mathbf{x_1} + \mathbf{y_1} == \mathbf{5})^3, \\ [\mathbf{x_4}, \mathbf{x_2}, \mathbf{x_3}], [], [\mathbf{y_3}, \mathbf{y_1}, \mathbf{y_2}] \\ \texttt{then } [\mathbf{x}_2 \leftarrow \texttt{query}(3)]^4 \\ \texttt{else } [\mathbf{x}_3 \leftarrow \texttt{query}(4)]^5; \\ \mathbf{y}_2 \leftarrow 2) \\ [\mathbf{z_1} \leftarrow \mathbf{x_4} + \mathbf{y_3}]^6 \end{array}, \quad \texttt{aVar}_{\mathbf{c}4} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{y}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \\ \mathbf{y}_2 \\ \mathbf{x}_4 \\ \mathbf{y}_3 \\ \mathbf{z}_1 \end{bmatrix}, \quad \texttt{M}_{\mathbf{c}4} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}, \quad \texttt{F}_{\mathbf{c}4} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

4.3 Adaptivity Based on Program Analysis in AdaptFun

Definition 41 (Program-Based Dependency Graph)**.** .
Given a program $\mathbf{c}$ *with its assigned variables* $\texttt{aVar}$ *of length* N*, s.t.,* $\Gamma \vdash_{\texttt{M}_c,\texttt{F}_c} \mathbf{c}$*, its program-based graph* $G(\mathbf{c}) = (\texttt{V},\texttt{E},\texttt{W},\texttt{F})$ *is. defined as:*

$$
\begin{array}{rrcl}
\textit{Vertices} & \texttt{V} & := & \{\mathbf{x} \mid \mathbf{x} = \texttt{aVar}(i);\ i = 1,\ldots,N\} \\
\textit{Directed Edges} & \texttt{E} & := & \{(\mathbf{x_1},\mathbf{x_2}) \mid (\mathbf{x_1} = \texttt{aVar}(i) \wedge \mathbf{x_2} = \texttt{aVar}(j) \wedge \texttt{M}_c(i,j) \geq 1);\ i,j = 1,\ldots,N\} \\
\textit{Weights} & \texttt{W} & := & \bigcup \begin{array}{l} \{(v,w) \in \texttt{V} \times (\mathbb{N} \cup e) \mid v \in \texttt{V} \wedge \texttt{F}(v) > 0 \wedge w = \mathsf{RechBound}(v,c)\} \\ \{(v,1) \in \texttt{V} \times \{1\} \mid v \in \texttt{V} \wedge \texttt{F}(v) = 0\} \end{array} \\
\textit{Query Flags} & \texttt{qF} & := & \{(\mathbf{x},n) \in \texttt{V} \times \{0,1\} \mid \left\{ \begin{array}{ll} n = 1 & \texttt{F}_c(i) = 2 \\ n = 0 & o.w. \end{array} \right\}; \mathbf{x} = \texttt{aVar}(i); i = 1,\ldots,N\}
\end{array}
$$

Definition 42 (Finite Walk (k))**.** .
Given a labeled weighted graph $G = (\texttt{V},\texttt{E},\texttt{W},\texttt{qF})$*, a* finite walk k *in* G *is a sequence of edges* $(e_1 \ldots e_{n-1})$ *for which there is a sequence of vertices* $(v_1,\ldots,v_n)$ *such that:*

- $e_i = (v_i, v_{i+1})$ *for every* $1 \leq i < n$.
- *every vertex* $v \in \texttt{V}$ *appears in this vertices sequence* $(v_1,\ldots,v_n)$ *of* k *at most* $W(v)$ *times.*

$(v_1,\ldots,v_n)$ *is the vertex sequence of this walk.*
[JL: Length of this finite walk k *is the number of vertices in its vertex sequence, i.e.,* $\texttt{len}(k) = n$*.]*

[JL: Given a labeled weighted graph $G = (\texttt{V},\texttt{E},\texttt{W},\texttt{qF})$, we use $\mathcal{WALK}(G)$ to denote a set containing all finite walks k in G; and $k_{v_1 \to v_2} \in \mathcal{WALK}(G)$where $v_1, v_2 \in \texttt{V}$ denotes the walk from vertex v_1 to v_2 .]

Definition 43 (Length of Finite Walk w.r.t. Query ($\texttt{len}_\texttt{q}$))**.** .
Given a labeled weighted graph $G = (\texttt{V},\texttt{E},\texttt{W},\texttt{qF})$ *and a* finite walk k *in* G *with its vertex sequence* $(v_1,\ldots,v_n)$*, the length of* k *w.r.t query is defined as:*

$$\texttt{len}_\texttt{q}(k) = \texttt{len}\big(v \mid v \in (v_1,\ldots,v_n) \wedge \texttt{qF}(v) = 2\big)$$

, where $\big(v \mid v \in (v_1,\ldots,v_n) \wedge \texttt{qF}(v) = 2\big)$ *is a subsequence of* k*'s vertex sequence.*

Given a program $\mathbf{c}$, we generate its program-based graph $\mathbf{G_{prog}}(\mathbf{c}) = (\texttt{V},\texttt{E},\texttt{W},\texttt{qF})$. Then the adaptivity bound based on program analysis for $\mathbf{c}$ is the number of query vertices on a finite walk in $\mathbf{G_{prog}}(\mathbf{c})$. This finite walk satisfies:

- the number of query vertices on this walk is maximum
- the visiting times of each vertex v on this walk is bound by its reachability bound $\texttt{W}(v)$.

It is formally defined in 44.

Definition 44 (Program-Based Adaptivity)**.** .
Given a program $\mathbf{c}$ *and its program-based graph* $\mathbf{G_{prog}}(\mathbf{c}) = (\texttt{V},\texttt{E},\texttt{W},\texttt{qF})$*, the program-based adaptivity for* c *is defined as*

$$A_{\mathbf{prog}}(\mathbf{c}) := \max\{\texttt{len}_\texttt{q}(k) \ \mid \ k \in \mathcal{WALK}(\mathbf{G_{prog}}(\mathbf{c}))\}.$$

4.4 [[Soundness of the AdaptFun]]

[JL:

Theorem 4.1 (Soundness of the AdaptFun)**.** . *Given a program* $\mathbf{c}$*, we have:*

$$A_{\mathbf{prog}}(\mathbf{c}) \geq A(\mathbf{c}).$$

] [JL:

Proof. Given a program $\mathbf{c}$, we construct its program-based graph $\mathbf{G_{prog}}(\mathbf{c}) = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{qF})$ by Definition 41
According to the Definition 44, we have:

$$A_{\mathbf{prog}}(\mathbf{c}) := \max\{\mathtt{len_q}(k) \mid k \in \mathcal{WALK}(\mathbf{G_{prog}}(\mathbf{c}))\}.$$

According to the Definition **??**, we have the trace-based adaptivity as follows:

$$A(\mathbf{c}) = \max\{\mathtt{len}(p) \mid \mathbf{m} \in \mathcal{SM}, D \in \mathcal{DB}, p \in \mathcal{PATH}(\mathbf{G_{trace}}(\mathbf{c}, \mathrm{D}, \mathbf{m})\}$$

Then, we need to show:

$$\max\{\mathtt{len}(p) \mid \mathbf{m} \in \mathcal{SM}, D \in \mathcal{DB}, p \in \mathcal{PATH}(\mathbf{G_{trace}}(\mathbf{c}, \mathrm{D}, \mathbf{m})\} \leq \max\{\mathtt{len_q}(\mathbf{k}) \mid \mathbf{k} \in \mathcal{WALK}(\mathbf{G_{prog}}(\mathbf{c}))\}$$

It is sufficient to show that:

$$\forall p, \mathbf{m}, D,\ s.t.,\ p \in \mathcal{PATH}(\mathbf{G_{trace}}(\mathbf{c}, \mathrm{D}, \mathbf{m}), \exists \mathbf{k} \in \mathcal{WALK}(\mathbf{G_{prog}}(\mathbf{c})) \wedge \mathtt{len}(\mathbf{p}) \leq \mathtt{len_q}(\mathbf{k})$$

Taking an arbitrary starting memory m and an arbitrary underlying database D, we construct a trace-based graph $\mathbf{G_{trace}}(\mathbf{c}, \mathrm{D}, \mathbf{m}) = (\mathtt{V}, \mathtt{E})$ by the definition **??**.
Let $\mathbf{G_{mid}}(\mathbf{c}, \mathbf{m}, \mathrm{D}) = \{\mathbf{V_{mid}}, \mathbf{E_{mid}}, \mathbf{qF_{mid}}\}$ be the intermediate graph by Definition 45.
By Lemma 21, we know:

$$\forall p, \mathbf{m}, D,\ s.t.,\ p \in \mathcal{PATH}(\mathbf{G_{trace}}(\mathbf{c}, \mathrm{D}, \mathbf{m}), \exists \mathbf{p'} \in \mathcal{PATH}(\mathbf{G_{mid}}(\mathbf{c}, \mathbf{m}, \mathrm{D})) \wedge \mathtt{len}(\mathbf{p}) = \mathtt{len_q}(\mathbf{p'})$$

Then it is sufficient to show that:

$$\forall p, \mathbf{m}, D,\ s.t.,\ p \in \mathcal{PATH}(\mathbf{G_{mid}}(\mathbf{c}, \mathrm{D}, \mathbf{m}), \exists k \in \mathcal{WALK}(\mathbf{G_{prog}}(\mathbf{c})) \wedge \mathtt{len_q}(p) \leq \mathtt{len_q}(k)$$

We prove a stronger statement instead:

$$\forall p, \mathbf{m}, D,\ s.t.,\ p \in \mathcal{PATH}(\mathbf{G_{mid}}(\mathbf{c}, \mathrm{D}, \mathbf{m}), \exists k \in \mathcal{WALK}(\mathbf{G_{prog}}(\mathbf{c})) \wedge \mathtt{len_q}(p) = \mathtt{len_q}(k)$$

By Lemma 22, let g be the surjective function $g : \mathbf{V_{prog}} \rightarrow \mathbf{V_{mid}}$ s.t.:

$$\forall \mathtt{av} \in \mathbf{V_{mid}}.\ \mathbf{qF_{prog}}(f(\mathtt{av})) = \mathbf{qF_{mid}}(\mathtt{av}) \wedge |\mathtt{image}(f(\mathtt{av}))| \leq W(f(\mathtt{av})).$$

Let $\mathbf{m}$ and D be an arbitrary memory and database D, taking an arbitrary path $p_{\mathtt{av}_1 \rightarrow \mathtt{av}_n} \in \mathcal{PATH}(\mathbf{G_{mid}}(\mathbf{c}, \mathrm{D}, \mathbf{m})$ with:

Edge sequence: $(e, \ldots, e_{n-1})$

Vertices sequence: $(\mathtt{av}_1,\ldots,\mathtt{av}_n)$.
By Lemma 24, let $h : \mathcal{PATH}(\mathbf{G_{mid}}(\mathbf{c},\mathrm{D},\mathbf{m})) \to \mathcal{WALK}(\mathbf{G_{prog}}(\mathbf{c}))$ be the surjective function satisfies:

$$\forall p_{\mathtt{av}_1\to\mathtt{av}_n} \in \mathcal{PATH}(\mathbf{G_{mid}}(\mathbf{c},\mathrm{D},\mathbf{m})) \text{ with } \begin{cases} \text{edge sequence:} & (e,\ldots,e_{n-1}) \\ \text{vertices sequence:} & (\mathtt{av}_1,\ldots,\mathtt{av}_n) \end{cases}$$

$$\exists k_{f(\mathtt{av}_1)\to f(\mathtt{av}_n)} \in \mathcal{WALK}(\mathbf{G_{prog}}(\mathbf{c})) \text{ with } \begin{cases} \text{edge sequence:} & (g(e),\ldots,g(e_{n-1}) \\ \text{vertices sequence:} & (f(\mathtt{av}_1),\ldots,f(\mathtt{av}_n)) \end{cases}$$

We have the walk: $k_{f(\mathtt{av}_1)\to f(\mathtt{av}_n)} \in \mathcal{WALK}(\mathbf{G_{prog}}(\mathbf{c}))$ with:

Edges sequence: $(g(e),\ldots,g(e_{n-1})$

Vertices sequence: $(f(\mathtt{av}_1),\ldots,f(\mathtt{av}_n))$.
It is sufficient to show

$$\mathtt{len}_{\mathtt{q}}(p_{\mathtt{av}_1\to\mathtt{av}_n}) = \mathtt{len}_{\mathtt{q}}(k_{f(\mathtt{av}_1)\to f(\mathtt{av}_n)})$$

Unfold the definition of $\mathtt{len}_{\mathtt{q}}$, it is suffice to show:

$$\mathtt{len}\big(\mathtt{av} \mid \mathtt{av} \in (\mathtt{av}_1,\ldots,\mathtt{av}_n) \wedge \mathbf{qF_{mid}}(\mathtt{av}) = 2\big) = \mathtt{len}\big(f(\mathtt{av}) \mid f(\mathtt{av}) \in (f(\mathtt{av}_1),\ldots,f(\mathtt{av}_n)) \wedge \mathbf{qF_{prog}}(f(\mathtt{av})\big) = 2) \ (a)$$

By Lemma 22, we know:

$$\forall \mathtt{av} \in \mathbf{V_{mid}}.\ \mathbf{qF_{mid}}(\mathtt{av}) = \mathbf{qF_{prog}}(f(\mathtt{av})) \ (b)$$

By rewriting (b) in (a), we have this case proved.
[[

Definition 45 (Intermediate Graph $\mathbf{G_{mid}}$)**.** .
$\mathcal{AV}$: Annotated Variables based on program execution
Given a program $\mathbf{c}$ *with its assigned variables* $\mathtt{aVar}$ *of length N, a database D, a starting memory* $\mathbf{m}$*, s.t.,* $\Gamma \vdash_{\mathtt{M}_c,\mathtt{F}_c} \mathbf{c}$*, the intermediate graph* $\mathbf{G_{mid}}(\mathbf{c},\mathbf{m},D) = (\mathtt{V},\mathtt{E},\mathtt{F})$ *is defined as:*

$$\begin{array}{rrcl} \textit{Vertices} & \mathtt{V} & := & \{\mathtt{av} \in \mathcal{AV} \mid \exists \mathbf{m}', w', \mathtt{qt}, \tau.\ s.t.,\ \langle \mathbf{m},\mathbf{c},[],[],[]\rangle \to^* \langle \mathbf{m}',\mathtt{skip},\mathtt{qt},\tau,w'\rangle \wedge \mathtt{av} \in \tau\} \\ \textit{Directed Edges} & \mathtt{E} & := & \{(\mathtt{av},\mathtt{av}') \in \mathcal{AV} \times \mathcal{AV} \mid \mathtt{flowsTo}(\mathtt{av},\mathtt{av}',\mathbf{c},\mathbf{m},D)\} \\ \textit{Flags} & \mathtt{F} & := & \{(\mathtt{av},n) \in \mathtt{V} \times \{0,1,2\} \mid (\pi_1(\mathtt{av}) = \mathtt{aVar}(i) \wedge n = \mathtt{F}_c(i));\ i = 1,\ldots,N\} \end{array}$$

]]
[[

Lemma 16 ($\mathsf{DEP}_{\mathsf{var}}$ is Transitive)**.** .
Given a program $\mathbf{c}$*, with a starting memory* $\mathbf{m}$ *and a hidden database D, s.t.,* $\langle \mathbf{m},\mathbf{c},[],[],[]\rangle \to^* \langle \mathbf{m}',\mathtt{skip},\mathtt{qt},\tau,w\rangle$*. Then,* $\forall \mathtt{av}_1,\mathtt{av}_2,\mathtt{av}_3 \in \tau$*:*

$$\Big(\mathsf{DEP}_{\mathsf{var}}(\mathtt{av}_1,\mathtt{av}_2,\mathbf{c},\mathbf{m},D) \wedge \mathsf{DEP}_{\mathsf{var}}(\mathtt{av}_2,\mathtt{av}_3,\mathbf{c},\mathbf{m},D)\Big) \implies \mathsf{DEP}_{\mathsf{var}}(\mathtt{av}_1,\mathtt{av}_3,\mathbf{c},\mathbf{m},D)$$

of Lemma 16. Proof by unfolding and rewriting the Definition 26. ■

]]
[[

Lemma 17 ($\mathtt{flowsTo}$ is Transitive ??)**.** .
Given a program $\mathbf{c}$ *with its assigned variables* $\mathtt{aVar}$ *of length N. Then* $\forall x_1,x_2,x_3 \in \mathtt{aVar}$

$$\Big(\mathtt{flowsTo}(x_1,x_2) \wedge \mathtt{flowsTo}(x_2,x_3)\Big) \implies \mathtt{flowsTo}(x_1,x_3)$$

of Lemma 17. Proof by unfolding the Definition 33. ■

]]
[[

Lemma 18 ($\mathsf{DEP_q}$ Implies $\mathsf{DEP_{var}}$)**.** .
Given a program $\mathbf{c}$*, with a starting memory* $\mathbf{m}$ *and a hidden database* D*, s.t.,* $\langle \mathbf{m}, \mathbf{c}, [], [], [] \rangle \rightarrow^* \langle \mathbf{m}', \texttt{skip}, \texttt{qt}, \tau, w \rangle$*. Then,* $\forall \texttt{av}_1, \texttt{av}_2 \in \texttt{qt}$

$$\mathsf{DEP_q}(\texttt{av}_1, \texttt{av}_2, \mathbf{c}, \mathbf{m}, D) \implies \mathsf{DEP_{var}}(\pi_2(\texttt{av}_1), \pi_2(\texttt{av}_2), \mathbf{c}, \mathbf{m}, D)$$

of Lemma 18. Proof by unfolding the Definition 26 and Definition **??**. ■

]]
[[

Lemma 19 ($\mathsf{DEP_{var}}$ Implies $\texttt{flowsTo}$)**.** .
Given a program $\mathbf{c}$*, with a starting memory* $\mathbf{m}$ *and a hidden database* D*, s.t.,* $\langle \mathbf{m}, \mathbf{c}, [], [], [] \rangle \rightarrow^* \langle \mathbf{m}', \texttt{skip}, \texttt{qt}, \tau, w \rangle$*. Then,* $\forall \texttt{av}_1, \texttt{av}_2 \in \tau$

$$\mathsf{DEP_{var}}(\texttt{av}_1, \texttt{av}_2, \mathbf{c}, \mathbf{m}, D) \implies \texttt{flowsTo}(\pi_1(\texttt{av}_1), \pi_1(\texttt{av}_2))$$

of Lemma 18. Proof by showing contradiction based on the Definition 26 and Definition 33. Let $\texttt{av}_1, \texttt{av}_2 \in \tau$ be 2 arbitrary annotated variables in the variable trace τ, s.t., $\mathsf{DEP_{var}}(\texttt{av}_1, \texttt{av}_2, \mathbf{c}, \mathbf{m}, D)$. Unfolding the $\mathsf{DEP_{var}}$ definition, we have: ■

]]
[[

Lemma 20 (Injective Mapping of vertices from $\mathbf{G_{trace}}$ to $\mathbf{G_{mid}}$)**.** .
$\mathbf{G_{trace}}(\mathbf{c}) = \{\mathbf{V_{trace}}, \mathbf{E_{trace}}\}$
$\mathbf{G_{mid}}(\mathbf{c}, \mathbf{m}, D) = \{\mathbf{V_{mid}}, \mathbf{E_{mid}}, \mathbf{qF_{mid}}\}$

$$\exists\ \texttt{injective}\ f : \mathcal{AQ} \rightarrow \mathcal{AV}.\ \forall \texttt{av} \in \mathbf{V_{trace}}.\ f(\texttt{av}) \in \mathbf{V_{mid}} \wedge \mathbf{qF_{mid}}(f(\texttt{av})) = 2$$

Proof. Proving by Definition 45 and Definition 44. ■

]]
[[

Lemma 21 (One-on-One Mapping from $\texttt{E}$ of $\mathbf{G_{trace}}$ to $\mathcal{PATH}(\mathbf{G_{mid}})$)**.** .
$\mathbf{G_{trace}}(\mathbf{c}) = \{\mathbf{V_{trace}}, \mathbf{E_{trace}}\}$
$\mathbf{G_{mid}}(\mathbf{c}, \mathbf{m}, D) = \{\mathbf{V_{mid}}, \mathbf{E_{mid}}, \mathbf{qF_{mid}}\}$
An injective function $f : \mathbf{V_{trace}} \rightarrow \mathbf{V_{mid}}$ *s.t.,* $\forall \texttt{av} \in \mathbf{V_{trace}}.\ \mathbf{qF_{mid}}(f(\texttt{av})) = 2$

$$\forall e = (\texttt{av}_1, \texttt{av}_2) \in \mathbf{E_{trace}}.\ \exists p_{f(\texttt{av}_1) \rightarrow f(\texttt{av}_2)} \in \mathcal{PATH}(\mathbf{G_{mid}}(\mathbf{c}, D, \mathbf{m}))$$

Proof. Proving by Lemma 20 and Definition 45 and acyclic property of $\mathbf{G_{trace}}$ and $\mathbf{G_{mid}}$. ■

]]
[[

Lemma 22 (Surjective Mapping of Vertices from $\mathbf{G_{mid}}$ to $\mathbf{G_{prog}}$)**.** .
$\mathbf{G_{mid}}(\mathbf{c},\mathbf{m},D) = \{\mathbf{V_{mid}},\mathbf{E_{mid}},\mathbf{qF_{mid}}\}$
$\mathbf{G_{prog}}(\mathbf{c}) = \{\mathbf{V_{prog}},\mathbf{E_{prog}},\mathbf{qF_{prog}},\mathbf{W_{prog}}\}$
$\exists\ \mathtt{surjective}\ f : \mathcal{AV} \to \mathcal{SVAR}$.

$$\forall \mathtt{av} \in \mathbf{V_{mid}}.\ f(\mathtt{av}) \in \mathbf{V_{prog}} \wedge \mathbf{qF_{prog}}(f(\mathtt{av})) = \mathbf{qF_{mid}}(\mathtt{av}) \wedge |\mathtt{image}(f(\mathtt{av}))| \leq W(f(\mathtt{av}))$$

Proof. Proving by Definition 45. ■

]]
[[

Lemma 23 (Surjective Mapping from $\mathtt{E}$ of $\mathbf{G_{mid}}$) to $\mathtt{E}$ of $\mathbf{G_{prog}}$)**.** .
$\mathbf{G_{mid}}(\mathbf{c},\mathbf{m},D) = \{\mathbf{V_{mid}},\mathbf{E_{mid}},\mathbf{qF_{mid}}\}$
$\mathbf{G_{prog}}(\mathbf{c}) = \{\mathbf{V_{prog}},\mathbf{E_{prog}},\mathbf{qF_{prog}},\mathbf{W_{prog}}\}$
A surjective function $f : \mathbf{V_{prog}} \to \mathbf{V_{mid}}$ *s.t.,* $\forall \mathtt{av} \in \mathbf{V_{mid}}.\ \mathbf{qF_{prog}}(f(\mathtt{av})) = \mathbf{qF_{mid}}(\mathtt{av}) \wedge |\mathtt{image}(f(\mathtt{av}))| \leq W(f(\mathtt{av}))$

$$\exists\ \mathtt{surjective}\ g : \mathbf{E_{mid}} \to \mathbf{E_{prog}}.\ \forall e_{mid} = (\mathtt{av}_1,\mathtt{av}_2) \in \mathbf{E_{mid}}.\exists e_{prog} = (f(\mathtt{av}_1),f(\mathtt{av}_2)) \in \mathbf{E_{prog}}$$

Proof. Proving by Lemma 22. ■

]]
[[

Lemma 24 (Surjective Mapping from $\mathcal{PATH}(\mathbf{G_{mid}})$ to $\mathcal{WALK}(\mathbf{G_{prog}})$)**.** .
$\mathbf{G_{mid}}(\mathbf{c},\mathbf{m},D) = \{\mathbf{V_{mid}},\mathbf{E_{mid}},\mathbf{qF_{mid}}\}$
$\mathbf{G_{prog}}(\mathbf{c}) = \{\mathbf{V_{prog}},\mathbf{E_{prog}},\mathbf{qF_{prog}},\mathbf{W_{prog}}\}$
A surjective function $f : \mathbf{V_{prog}} \to \mathbf{V_{mid}}$ *s.t.,* $\forall \mathtt{av} \in \mathbf{V_{mid}}.\ \mathbf{qF_{prog}}(f(\mathtt{av})) = \mathbf{qF_{mid}}(\mathtt{av}) \wedge |\mathtt{image}(f(\mathtt{av}))| \leq W(f(\mathtt{av}))$
A surjective function $g : \mathbf{E_{mid}} \to \mathbf{E_{prog}}$ *s.t.,* $\forall e_{mid} = (\mathtt{av}_1,\mathtt{av}_2) \in \mathbf{E_{mid}}.\exists e_{prog} = (f(\mathtt{av}_1) \to f(\mathtt{av}_2)) \in \mathbf{E_{prog}}$
$\exists\ \mathtt{surjective}\ h : \mathcal{PATH}(\mathbf{G_{mid}}(\mathbf{c},\mathbf{m},D)) \to \mathcal{WALK}(\mathbf{G_{prog}}(\mathbf{c}))$ *s.t.:*

$$\forall p_{\mathtt{av}_1 \to \mathtt{av}_2} \in \mathcal{PATH}(\mathbf{G_{mid}}(\mathbf{c},\mathbf{m},D))\ \textit{with} \begin{cases} \textit{edge sequence:} & (e,\ldots,e_{n-1}) \\ \textit{vertices sequence:} & (\mathtt{av}_1,\ldots,\mathtt{av}_n) \end{cases}$$

$$\exists k_{f(\mathtt{av}_1) \to f(\mathtt{av}_2)} \in \mathcal{WALK}(\mathbf{G_{prog}}(\mathbf{c}))\ \textit{with} \begin{cases} \textit{edge sequence:} & (g(e),\ldots,g(e_{n-1}) \\ \textit{vertices sequence:} & (f(\mathtt{av}_1),\ldots,f(\mathtt{av}_n)) \end{cases}$$

Proof. Proving by induction on the length of $l = p_{\mathtt{av}_1 \to \mathtt{av}_2} \in \mathcal{PATH}(\mathbf{G_{mid}}(\mathbf{c},\mathbf{m},\mathrm{D}))$, and Lemma 23 and Lemma 22.

case: $l = 1$**:**

case: $l = l' + 1,\ l' \geq 1$**:**

■

]] □

]

5 [[Examples]]

Example 5.1 (TwoRound Algorithm)**.**

$$
TR(k) \triangleq \begin{array}{l} [i \leftarrow 1]^1; \\ [a_1 \leftarrow []]^2; \\ \texttt{while } [i < k]^3, 0, \texttt{ do} \\ \Big([x \leftarrow \texttt{query}()]^4; \\ [a \leftarrow x :: a]^5 [i_2 \leftarrow i_3 + 1]^6\Big); \\ \big[l \leftarrow q_{k+1}(a)\big]^7 \end{array} \quad \Rightarrow \quad TR^{ssa} \triangleq \begin{array}{l} [i \leftarrow 1]^1; \\ [a_1 \leftarrow []]^2; \\ \texttt{while } [i < k]^3, 0, [a_3, a_1, a_2][i_3, i_1, i_2] \texttt{ do} \\ \Big([x_1 \leftarrow q]^4; \\ [a_2 \leftarrow x_1 :: a_3]^5 [i_2 \leftarrow i_3 + 1]^6\Big); \\ \big[l \leftarrow q_{k+1}(a_3)\big]^7 \end{array}
$$

Adapt(TR) = 2

Using AdaptFun*, we first generate a assigned variables G from an empty list* [] *and empty while map* [].

$$[]; []; TR^{ssa} \rightarrow G; w \wedge w = []$$

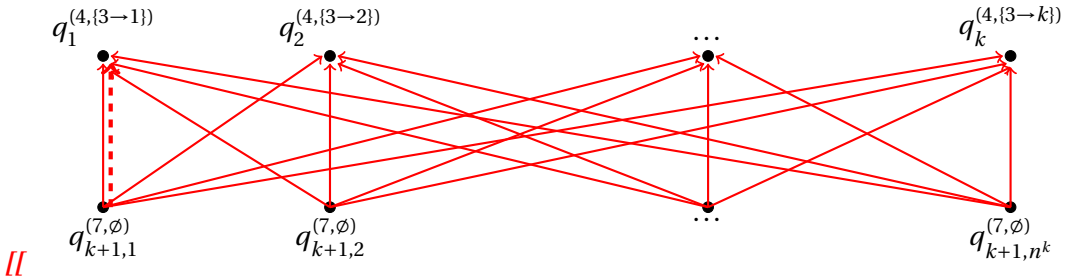
.

$$G_{k=2} = \left[{a_1}^2, {a_3}^{(2,[2:1])}, x_1^4, a_2^5, i_3^3, i_2^6, i_1^2, l_1^7, {l_1}^{(5,[])}\right]$$

We denote a_1^1 *short for* ${a_1}^{(1,[])}$ *and* ${a_3}^{(2,1)}$ *short for* ${a_3}^{(2,[2:1])}$*, where the label* $(2,1)$ *represents at line number* 2 *and in the* 1 *st iteration.*

$$
M = \begin{bmatrix} & a_1^2 & a_3^3 & x_1^4 & a_2^5 & i_3^3 & i_2^6 & i_1^2 & l_1^7 \\ a_1^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_3^3 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ x_1^4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_2^5 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ i_3^3 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ i_2^6 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ i_1^2 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ l_1^7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, V = \begin{bmatrix} a_1^2 & 2 \\ a_3^3 & 2 \\ x_1^4 & 1 \\ a_2^5 & 2 \\ i_3^3 & 2 \\ i_2^6 & 2 \\ i_1^2 & 2 \\ l_1^7 & 1 \end{bmatrix}
$$

$$
M = \begin{bmatrix} & a_1^1 & a_3^{(2,1)} & x_1^{(3,1)} & a_2^{(4,1)} & a_3^{(2,2)} & x_1^{(3,2)} & a_2^{(4,2)} & a_3^2 & l_1^5 \\ a_1^1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_3^{(2,1)} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ x_1^{(3,1)} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_2^{(4,1)} & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_3^{(2,2)} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ x_1^{(3,2)} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_2^{(4,2)} & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ a_3^2 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ l_1^5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}, V = \begin{bmatrix} a_1^1 & 0 \\ a_3^{(2,1)} & 0 \\ x_1^{(3,1)} & 1 \\ a_2^{(4,1)} & 0 \\ a_3^{(2,2)} & 0 \\ x_1^{(3,2)} & 1 \\ a_2^{(4,2)} & 0 \\ a_3^2 & 0 \\ l_1^5 & 1 \end{bmatrix}
$$


$q_1^{(4,\{3\to 1\})}$
$q_2^{(4,\{3\to 2\})}$
$\cdots$
$q_k^{(4,\{3\to k\})}$
$q_{k+1,1}^{(7,\emptyset)}$
$q_{k+1,2}^{(7,\emptyset)}$
$\cdots$
$q_{k+1,n^k}^{(7,\emptyset)}$

[[*]]*

Example 5.2 (Multi-Round Algorithm)**.**

$$MR \triangleq \begin{array}{l} [i \leftarrow 1]^1; \\ [I \leftarrow []]^2; \\ \texttt{while } [i < k]^3 \texttt{ do} \\ \Big(\big[p \leftarrow c\big]^4; \\ \big[a \leftarrow \texttt{query}(p, I)\big]^5; \\ \big[I \leftarrow \texttt{update}\ (\ I\ , (a, p))\big]^6; \\ [i \leftarrow i + 1]^7 \\ \Big) \end{array} \quad \Rightarrow \quad MR^{ssa} \triangleq \begin{array}{l} [i \leftarrow 1]^1; \\ [I \leftarrow []]^2; \\ \texttt{while } [i < k]^3 0, [I_3, I_1, I_2] \\ \texttt{ do} \\ \Big(\big[p_1 \leftarrow c\big]^4; \\ \big[a \leftarrow \texttt{query}(p_1, I_2)\big]^5; \\ \big[I_2 \leftarrow \texttt{update}\ (\ I_3\ , (a_1, p_1))\big]^6; \\ [i \leftarrow i + 1]^7 \\ \Big) \end{array}$$

Adapt(MR) = k.
Using AdaptFun*, we first generate a assigned variables G from an empty list* [] *and empty whlemap* $\emptyset$*.*

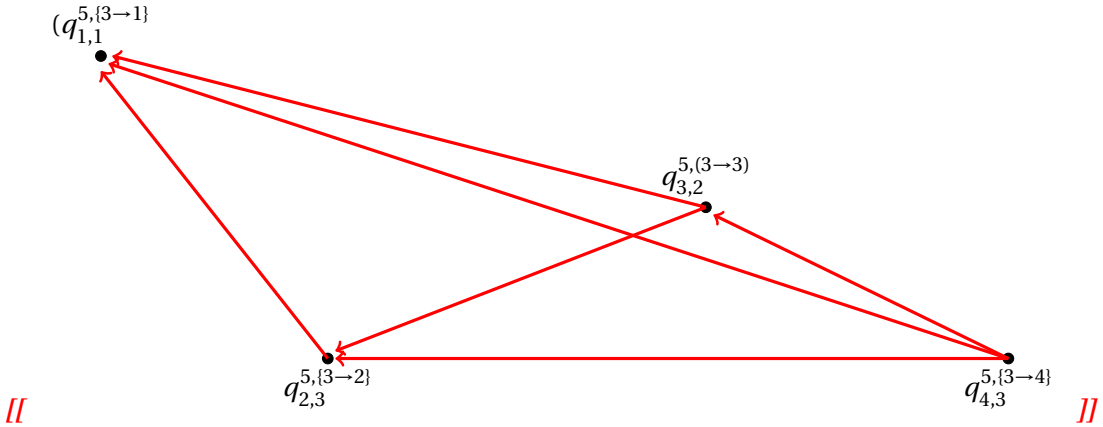
$$[]; \emptyset; MR^{ssa} \rightarrow G; w \wedge w = \emptyset$$

.

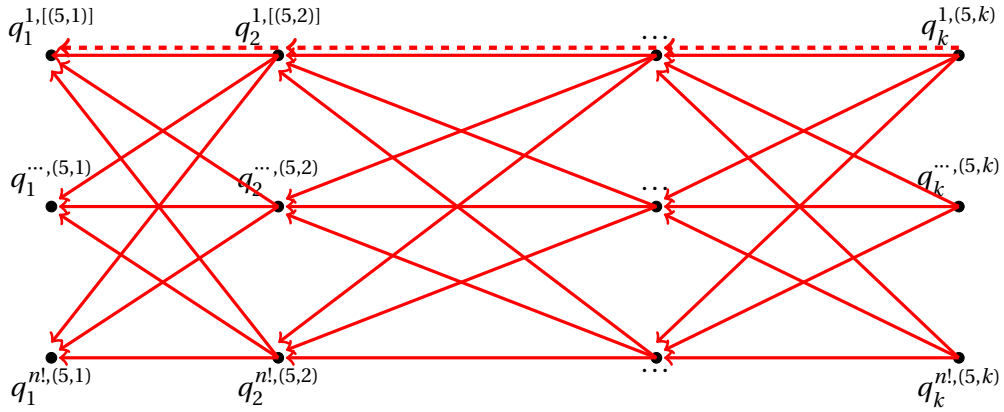
$$G_{k=2} = \left[i_1^1, I_1^2, i_3^3, I_3^3, p_1^4, a_1^5, I_2^6, i_2^7\right]$$

We denote I_1^1 *short for* ${I_1}^{(1,\emptyset)}$ *and* ${I_3}^{(2,1)}$ *short for* ${I_3}^{(2,[2;1])}$*, where the label* $(2,1)$ *represents at line number* 2 *and in the* 1 *st iteration.*

$$M = \begin{bmatrix} i_1^1 & I_1^2 & i_3^3 & I_3^3 & p_1^4 & a_1^5 & I_2^6 & i_2^7 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, V = \begin{bmatrix} i_1^1 & 0 \\ I_1^2 & 0 \\ i_3^3 & 2 \\ I_3^3 & 2 \\ p_1^4 & 2 \\ a_1^5 & 1 \\ I_2^6 & 2 \\ i_2^7 & 2 \end{bmatrix}$$


$(q_{1,1}^{5,\{3\to1\}}$
$q_{3,2}^{5,(3\to3)}$
$q_{2,3}^{5,\{3\to2\}}$
$q_{4,3}^{5,\{3\to4\}}$
[[
]]

$\forall k.\forall D$, *we have* $A(TR^L) = (k-1)$ *given all possible execution traces.* [[



]]

6 Non Determinism

Non-Determinism of queries. When evaluating a query $\texttt{query}(\alpha)$ on a given database D, in addition to obtain a result v from the database $v = \texttt{query}(\alpha)(D)$, we assume there is an underlying mechanism that will perform extra manipulations on v. The mechanism is considered as primitive operations in our language, behaving as black box to programmers. There are different kinds of mechanisms, such as adding noise sampled from certain probabilistic distribution to the result [1]. Because of the randomness of the underlying mechanism, the evaluation of a query $\texttt{query}(\alpha)$ is non-deterministic. That's the reason, in the Definition **??**, given a fixed database D, there will be a query domain $\mathcal{QD}$ where $\texttt{query}(\alpha)(D)$ can be evaluated to different values $v \in \mathcal{QD}$.
On the other hand, in the operational semantics rule **query-v**:

$$\frac{\texttt{query}(\alpha) = v}{\langle m, [x \leftarrow \texttt{query}(\alpha)]^l, t, w\rangle \rightarrow \langle m[v/x], \texttt{skip}, (t{+}{+}[(\alpha, l, w)]), w\rangle}\ \textbf{query-v}$$

, we evaluate the query given database D based on an assumption that the underlying mechanism is fixed. This fixed mechanism only adds constant 0 to the original result v returned from the database, i.e., $v = \texttt{query}(\alpha)(D)$.
The Lemma 25 and 26 formalize this property.

Lemma 25 (Semi-Determinism)**.** .
for any program c with a starting memory m, trace t and while label w, if program c contains neither $[x \leftarrow \texttt{query}(\psi)]^l$ *nor* $[x \leftarrow \texttt{query}(\alpha)]^l$ *for any* ψ *and* α*, then*

$$\bigwedge \left\{ \begin{array}{l} \langle m, c, t, w\rangle \rightarrow^* \langle m_1, \texttt{skip}, t_1, w_1\rangle \\ \langle m, c, t, w\rangle \rightarrow^* \langle m_2, \texttt{skip}, t_2, w_2\rangle \end{array} \right\} \Longrightarrow (m_1 = m_2 \wedge t_1 = t_2 \wedge w_1 = w_2)$$

Proof. Proof is obvious by induction on the operational semantics rules. □

Lemma 26 (Query Semi-Determinism)**.** .
Given a program $c; x \leftarrow \texttt{query}(\psi); c'$ *with a starting memory m, trace t and while label w, s.t. c contains neither* $[x \leftarrow \texttt{query}(\psi)]^l$ *nor* $[x \leftarrow \texttt{query}(\alpha)]^l$ *for any* ψ *and* α*, then:*

$$\bigwedge \left\{ \begin{array}{l} \langle m, c; x \leftarrow \texttt{query}(\psi); c', t, w\rangle \rightarrow^* \langle m_1, x \leftarrow \texttt{query}(\alpha_1); c', t_1, w_1\rangle \\ \langle m, c; x \leftarrow \texttt{query}(\psi); c', t, w\rangle \rightarrow^* \langle m_2, x \leftarrow \texttt{query}(\alpha_2); c', t_2, w_2\rangle \end{array} \right\} \Longrightarrow (\alpha_1 = \alpha_2 \wedge m_1 = m_2 \wedge t_1 = t_2 \wedge w_1 = w_2)$$

Proof. Proof is obvious by induction on the operational semantics rules. □

7 Analysis of Generalization Error

Example 7.1 (Two Round Algorithm)**.**

$$TR^H(k) \triangleq \begin{array}{l} [a_1 \leftarrow []]^1; \\ \texttt{loop}\ [k]^2\ (a_2 \leftarrow f(1, a_1, a_3)) \\ \ \texttt{do} \\ \Big([x_1 \leftarrow \texttt{query}()]^3; \\ [a_3 \leftarrow x_1 :: a_2]^4\Big); \\ \big[l \leftarrow q_{k+1}(a_3)\big]^5 \end{array}$$

Example 7.2 (Multi-Round Algorithm)**.**

$$MR^H \triangleq \begin{array}{l} [I_2 \leftarrow []]^1; \\ \texttt{loop}\ [k]^2\ (I_2 \leftarrow f(2, I_1, I_3)) \\ \ \texttt{do} \\ \Big([p_1 \leftarrow c]^3; \\ \big[a_1 \leftarrow \delta(\texttt{query}(p, I_2))\big]^4; \\ \big[I_3 \leftarrow \texttt{update}\ (\ I_2\ , (a_1, p))\big]^5\Big) \end{array}$$

By applying different mechanisms $\delta()$ over the queries $\texttt{query}(\cdot)$, we have different error bounds.

Gaussian Mechanism: $N(0, \sigma)$ [1]:

Adaptivity $r = 2$: $\sigma = O\left(\frac{\sqrt{r\log(k)}}{\sqrt{n}}\right)$ (also known as expected error);

Adaptivity unknown: $\sigma = O\left(\frac{\sqrt[4]{k}}{\sqrt{n}}\right)$;

Mean Squared Error Bound: $\frac{1}{2n}\min_{\lambda\in[0,1]}\left(\frac{2\rho kn - \ln(1-\lambda)}{\lambda}\right) + 2\mathbb{E}_{Z_i\sim N(0,\frac{1}{2n^2\rho})}\left[\max_{i\in[k]}(Z_i^2)\right]$

Confidence Bounds: minimize τ where $\tau \geq \sqrt{\frac{2}{n\beta}\min_{\lambda\in[0,1]}\left(\frac{2\rho kn - \ln(1-\lambda)}{\lambda}\right)}$ and $\tau \geq \frac{2}{n}\sqrt{\frac{\ln(4n/\beta}{\rho'}}$ with confidence level $1-\beta$.

$(\epsilon, \delta) - DP$ **mechanism**:

Confidence Bounds: $\tau \geq \sqrt{\frac{48}{n}\ln(4/\beta)}$ with $\epsilon \leq \frac{\tau}{4}$ and $\delta = \exp\left(\frac{-4\ln(8/\beta)}{\tau}\right)$

Sample Splitting:

Expected Error: $O\left(\frac{\sqrt{k\log(k)}}{\sqrt{n}}\right)$

Thresholdout: B, σ, T, h

Confidence bounds: $\tau = \max\left\{\sqrt{\frac{2\zeta}{h\beta}}, 2\sigma\ln(\frac{\beta}{2}), \sqrt{\frac{1}{\beta}}\cdot\left(\sqrt{T^2 + 56\sigma^2} + \sqrt{\frac{\zeta}{4h}}\right)\right\}$, for $\zeta = \min_{\lambda\in[0,1)}\left(\frac{2B\ (\sigma^2 h) - \ln(1-\lambda)}{\lambda}\right)$

References

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

[2] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 292–304, New York, NY, USA, 2010. Association for Computing Machinery.