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 $\mathtt{query}(\psi)$ 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 $\mathtt{query}(x)$, 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.

$$\begin{array}{l} x \leftarrow \mathtt{query}(0); \\ \mathtt{if}\ (x_1 > 0) \\ y \leftarrow \mathtt{query}(x) \end{array}$$

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 $\mathtt{query}(0)$ does not depend on another query $\mathtt{query}(x)$ when the result of $\mathtt{query}(0)$ remains the same regardless of the modification of the result of $\mathtt{query}(x)$. Hence, it becomes hard to distinguish whether the variance of result of $\mathtt{query}(0)$ 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, – $\mathtt{query}(0)$ –. 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	$::=$	$[\texttt{skip}]^l \mid [x \leftarrow e]^l \mid [x \leftarrow \texttt{query}\,(\psi)]^l$
			$\mid \texttt{while}\ [b]^l\ \texttt{do}\ c \mid c;c \mid \texttt{if}\,([b]^l, c, c)$
Memory	m	$::=$	$[] \mid (x^l \to v) :: m$
Annotated Query	$\texttt{aq}$	$::=$	(α, l, w)
Annotated Variable	$\texttt{aq}$	$::=$	$[JL: (x, l, w)]$
Query Trace	$\texttt{qt}$	$::=$	$[] \mid \texttt{aq} :: t$
[[Variable Trace]]	$\texttt{vt}$	$::=$	$[] \mid \texttt{av} :: t$
While Map	w	$::=$	$[] \mid w[l \to n]$

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

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

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: [WQ: While Map is a standard map from label to natural number.]

Definition 1 (While Map (w)). .
While map is defined as a map, mapping from the key (labels l) to iteration number n, as follows:

```
type wm = | empty
          | Cons of (int * int) & wm

let rec minus (w:wm) (l:int) : wm =
    match w with
      | [] ⇒ []
      | (k, v) :: tl ⇒
       if k = l then tl else (k,v) :: minus tl l.

let rec plus (w:wm) (l:int) : wm =
    match w with
      | [] ⇒ [(l,0)]
      | (k, v) :: tl ⇒
       if k = l then (k,v+1):: tl else (k,v) :: plus tl l.
```

A mapping in the while map $[l \to n]$ gives the accurate information on which while command (`while` $[b]^l$ `do` c) the statement is in by finding its corresponding label l, (i.e., the key in the map). The label l specifies the line number of the guard $[b]^l$ of the while command, where the current statement lives in. The mapped result n indicates the iteration number, which the current statement belongs to. For example, the while map $w = [3:1, 4:2]$ indicates that the statement is currently in a nested while loop, the first level while loop starting from line of label 3 (i.e., the line of the outside guard is) and in its first iteration, the statement is now in the nested while loop starting from line of label 4 (i.e., label of the guard for the nested while loop) and in the second iteration. We use $\emptyset$ to represent an empty map, indicating the statement is not in any while loop statement. The two operations ($w - l$ and $w + l$) on w is defined as follows:

$$w - l := \begin{cases} w & l \not\in Keys(w) \\ w' & w = w'[l \to \_] \end{cases} \tag{1}$$

$$w + l := \begin{cases} w[l \to 0] & l \not\in Keys(w) \\ w'[l \to n+1] & w = w'[l \to n] \end{cases} \tag{2}$$

We use $w - l$ to remove the mapping of the key l from the while map w. This is used when exiting the loop at line l. We record in w the first iteration of a while loop marked with label l by assigning l with the iteration 1. The mapped number increases when going into the next iteration.

$Keys(w)$ is used to return the list containing all the keys of the while map w. $MinKeys(w)$ returns the minimal key in $Keys(w)$.

Definition 2 (While Map Order). *The label with while map is partially ordered and defined as follows:* $<_w$ *and* $=_w$.

$$\begin{array}{lcl} w_1 =_w w_2 & \triangleq & Keys(w_1) = Keys(w_2) \land \forall k \in Keys(w_1).w_1(k) = w_2(k) \\ \emptyset =_w \emptyset & & \end{array}$$

$mk(w_i) = MinKey(w_i)$

$$w_1 <_w w_2 \quad \triangleq \quad \begin{array}{ll} & w_1 = \emptyset \\ mk(w_1) < mk(w_2) & w_1, w_2! = \emptyset \\ w_1(mk(w_1)) < w_2(mk(w_2)) & mk(w_1) = mk(w_2) \\ (w_1 - mk(w_1)) <_w (w_2 - mk(w_2)) & o.w. \end{array}$$

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.

[JL: The annotated query $\mathtt{aq} = (\alpha, l, w)$ is a triple contains 3 elements. α is a query value representing the corresponding query request $x \leftarrow \mathtt{query}(\alpha)$ during the execution of the program. l is the label of the query request command in a program. w is the while map indicating if or not this command is in a while loop, and which iteration it is in. Given the label l and while map w are ordered, the annotated queries also preserve this property. Its order and equivalence relation are defined in Definition 3.

Definition 3 (Order of Annotated Queries)**.** .
Given 2 annotated queries $\mathtt{aq}_1 = (\alpha_1, l_1, w_1), \mathtt{aq}_2 = (\alpha_2, l_2, w_2)$ *:*

$$\mathtt{aq}_1 <_{aq} \mathtt{aq}_2 \triangleq \begin{cases} l_1 < l_2 & w_1 = \emptyset \lor w_2 = \emptyset \lor w_1 =_w w_2 \\ w_1 <_w w_2 & \text{Otherwise} \end{cases}$$

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

] A query trace $\mathtt{qt}$ 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 $\langle m, c, t, w\rangle$, of form $\langle m, c, t, w\rangle \to \langle m', \mathtt{skip}, t', w'\rangle$.
[[The annotated variable $\mathtt{av}$**.]]**
[[A variable trace $\mathtt{vt}$**.]]**
[[Operations on variable trace $\mathtt{vt}$**.]]**
[[

Definition 4 (Equivalence of Annotated Variables)**.** *Given 2 annotated queries* $\mathtt{av}_1 = (\alpha_1, l_1, w_1), \mathtt{av}_2 = (\alpha_2, l_2, w_2)$ *:*

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

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

]] The evaluation rules for arithmetic and boolean expressions are standard. They have the form $\langle m, a\rangle \to_a a'$, evaluating an arithmetic expression a in the memory m, and similar for the boolean expressions $\langle m, b\rangle \to_b b'$, defined as follows: [JL:

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

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

] [JL: 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 5.

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 $FV(\psi_2) \in \mathtt{dom}(m)$:*

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

, 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 6:

Definition 6 (Equivalence of Annotated Queries)**.** *Given 2 annotated queries* $\mathtt{aq}_1 = (\alpha_1, l_1, w_1), \mathtt{aq}_2 = (\alpha_2, l_2, w_2)$ *:*

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

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

] [JL: Given an annotated query $\mathtt{aq}$ and a trace t, the appending operation $\mathtt{aq} :: t$ is the standard list appending operation, appends $\mathtt{aq}$ 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:

$$t_1 ++ t_2 \triangleq \begin{cases} t_2 & t_1 = [] \\ \mathtt{aq} :: (t_1' ++ t_2) & t_1 = \mathtt{aq} :: t_1' \end{cases} \tag{3}$$

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

$$t_1 - t_2 \triangleq t_3\ s.t., t_2 ++ t_3 = t_1 \tag{4}$$

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

$$\mathtt{aq} \in_{\mathtt{aq}} t \triangleq \begin{cases} \mathtt{false} & t = [] \\ \mathtt{true} & t = \mathtt{aq}' :: t' \quad \mathtt{aq} =_{\mathtt{aq}} \mathtt{aq}' \\ \mathtt{aq} \in t' & t = \mathtt{aq}' :: t' \quad \mathtt{aq} \neq_{\mathtt{aq}} \mathtt{aq}' \end{cases} \tag{5}$$

] [JL:

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

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

$c_1 \neq_c c_2$ 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:

$$c' \in_c c \triangleq \exists c_1, c_2, c''.\ s.t.,\ c =_c c_1; c''; c_2 \land c' =_c c'' \tag{6}$$

]

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 `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 9. [[

Lemma 1 (Determinism of Variable Trace).

]]
[[

Definition 8 (Well-formed Variable Trace). *A trace t is well formed if and only if it preserves the following two properties:*

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

]]
[[

Theorem 2.1 (Variable Trace Generated from Operational Semantics is Well-formed). . *Given a program c, with arbitrary starting memory m, trace t and while map w if $\langle m, c, t, w \rangle \rightarrow^* \langle m', \texttt{skip}, t', w' \rangle$, then $(t' - t)$ is a well formed trace with respect to program c, m and w, denoted as $m, w, c \vDash t' - t$.*

Proof. □

]]

[JL:

Definition 9 (Well-formed Query Trace). *A trace t is well formed if and only if it preserves the following two properties:*

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

]

[JL:

Theorem 2.2 (Query Trace Generated from Operational Semantics is Well-formed). .
Given a program c, with arbitrary starting memory m, trace t and while map w if $\langle m, c, t, w\rangle \to^* \langle m', \mathtt{skip}, t', w'\rangle$, then $(t' - t)$ is a well formed trace with respect to program c, m and w, denoted as $m, w, c \vDash t' - t$.

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

case: $[x \leftarrow e]^l$
We assume a configuration $\langle m, [x \leftarrow e]^l, t, w\rangle$ s.t. .
From the evaluation rule assn1 and rule assn2, we know there exists an evaluation as follows:

$$\langle m, [x \leftarrow e]^l, t, w\rangle \to \langle m', \mathtt{skip}, t, w\rangle$$

when we assume $\langle m, e\rangle \to^* v$.
The resulting trace is t.
By unfolding the trace subtraction operation, we know $t - t = []$ is an empty list, which is well formed according to Definition 9.

case: $[x \leftarrow \mathtt{query}(\psi)]^l$
By repeatedly applying the operational semantics rule query-e, we know there exist α s.t.,:

$$\langle m, [x \leftarrow \mathtt{query}(\psi)]^l, t, w\rangle \to^* \langle m', [x \leftarrow \mathtt{query}(\alpha)]^l, t, w\rangle$$

According to the evaluation rule query-v, we have the following transition:

$$\langle m', [x \leftarrow \mathtt{query}(\alpha)]^l, t, w\rangle \to \langle m', \mathtt{skip}, t ++ [(\alpha, l, w)], w\rangle$$

The resulting trace is $t ++ [(\alpha, l, w)]$.
By unfolding the trace subtraction operation, we know $(t ++ [(\alpha, l, w)]) - t = [(\alpha, l, w)]$, which is well formed by Definition 9.

case: $\mathtt{if}\ ([b]^l, c_1, c_2)$
Let $\langle m', \mathtt{skip}, t, w\rangle$ $(\star)$ be the configuration s.t. $\langle m,\ \mathtt{if}\ ([b]^l, c_1, c_2), t, w\rangle \to^* \langle m', \mathtt{skip}, t', w'\rangle$. It is sufficient to show that $t' - t$ is well-formed.
According to the evaluation rule if, we know there exists an evaluation by repeatedly applying if

$$\langle m,\ \mathtt{if}\ ([b]^l, c_1, c_2), t, w\rangle \to^* \langle m,\ \mathtt{if}\ ([v]^l, c_1, c_2), t, w\rangle\ (1)$$

, where $\langle m, b\rangle \to_b v$ and v is either $\mathtt{true}$ or $\mathtt{false}$.
Considering 2 subcases where v is either $\mathtt{true}$ or $\mathtt{false}$:

subcase: $v = \texttt{true}$
By applying the evaluation rule if-t, we have:

$$\langle m, \texttt{ if }([\texttt{true}]^l, c_1, c_2), t, w\rangle \to \langle m, c_1, t, w\rangle \ (t1)$$

By induction hypothesis on c_1 with starting memory m, trace t and while map w, let $\langle m_1, \texttt{skip}, t_1, w_1\rangle$ be the configuration s.t.

$$\langle m, c_1, t, w\rangle \to^* \langle m_1, \texttt{skip}, t_1, w_1\rangle \ (t2)$$

we know $(t_1 - t)$ is a well formed trace.
According to the evaluation (a), $(t1)$ and $(t2)$, we know $\langle m_1, \texttt{skip}, t_1, w_1\rangle$ is the assumed configuration $\langle m', \texttt{skip}, t', w'\rangle$ such that

$$\langle m, \texttt{ if }([b]^l, c_1, c_2), t, w\rangle \to^* \langle m_1, \texttt{skip}, t_1, w_1\rangle$$

Since $(t_1 - t)$ is well-formed and $t' - t = t_1 - t$, this subcase is proved.

subcase: $v = \texttt{false}$
By applying the evaluation rule if-f, we have:

$$\langle m, \texttt{ if }([\texttt{false}]^l, c_1, c_2), t, w\rangle \to \langle m, c_2, t, w\rangle \ (f1)$$

By induction hypothesis on c_1 with starting memory m, trace t and while map w, let $\langle m_2, \texttt{skip}, t_2, w_2\rangle$ be the configuration s.t.

$$\langle m, c_2, t, w\rangle \to^* \langle m_2, \texttt{skip}, t_2, w_2\rangle \ (f2)$$

we know $(t_2 - t)$ is a well formed trace.
According to the evaluation (a), $(f1)$ and $(f2)$, we know $\langle m_2, \texttt{skip}, t_2, w_2\rangle$ is the assumed configuration $\langle m', \texttt{skip}, t', w'\rangle$ in $(\star)$ such that

$$\langle m, \texttt{ if }([b]^l, c_1, c_2), t, w\rangle \to^* \langle m_2, \texttt{skip}, t_2, w_2\rangle$$

Since $(t_2 - t)$ is well-formed and $t' = t_2$, this subcase is proved.

case: $c_1; c_2$
According to the evaluation rule seq1, we have the following evaluation by repeatedly applying seq1

$$\langle m, c_1; c_2, t, w\rangle \to^* \langle m_1, [\texttt{skip}]^l; c_2, t_1, w_1\rangle \ (1)$$

where

$$\langle m, c_1, t, w\rangle \to^* \langle m_1, [\texttt{skip}]^l, t_1, w_1]\rangle \ (2)$$

By induction hypothesis on c_1 and (2), we know $(t_1 - t)$ is well-formed. According to the evaluation rule seq2, we have:

$$\langle m_1, [\texttt{skip}]^l; c_2, t_1, w_1\rangle \to \langle m_1, c_2, t_1, w_1\rangle \ (3)$$

By induction hypothesis on c_2 and let $\langle m_2, \texttt{skip}, t_2, w_2]\rangle$ be the configuration s.t.,:

$$\langle m_1, c_2, t_1, w_1\rangle \to^* \langle m_2, \texttt{skip}, t_2, w_2]\rangle \ (4)$$

we know $(t_2 - t_1)$ is well-formed.
According to evaluations $(1), (2), (3)$ and (4), we have following evaluation:

$$\langle m, c_1; c_2, t, w\rangle \to^* \langle m_2, \texttt{skip}, t_2, w_2]\rangle$$

It is sufficient to show $(t_2 - t)$ is well-formed. Unfolding the definition 9, it is sufficient to show that the trace $(t_2 - t)$ holds both the *Uniqueness* and *Ordering* property.
Let $t_1' = t_1 - t$ and $t_2' = t_2 - t_1$, unfolding the subtraction operations, we have:

$$(t_2 - t) = t_1' + + t_2'$$

case: Uniqueness $(\diamond)$
By the distinction properties of labels in program c,

$$\forall \mathtt{aq}_1 \in t_1', \mathtt{aq}_2 \in t_2'.\ \mathtt{aq}_1 \neq_{\mathtt{aq}} \mathtt{aq}_2$$

Since $t = t_1' + + t_2'$ and both t_1' and t_2' are well-formed, we know:

$$\forall \mathtt{aq}_1, \mathtt{aq}_2 \in (t_1' + + t_2').\ \mathtt{aq}_1 \neq_{\mathtt{aq}} \mathtt{aq}_2$$

THe Uniqueness property for $(t_2 - t)$ is proved.

case: Ordering $(\Box)$
By the definition of ordering property, we need to show $\forall \mathtt{aq}_1, \mathtt{aq}_2 \in_{\mathtt{aq}} (t_2 - t)$:

$$(\mathtt{aq}_1 <_{aq} \mathtt{aq}_2) \Longleftrightarrow \exists t^1, t^2, t^3, \mathtt{aq}_1', \mathtt{aq}_2'.\ s.t.,\ (\mathtt{aq}_1 =_{\mathtt{aq}} \mathtt{aq}_1') \wedge (\mathtt{aq}_2 =_{\mathtt{aq}} \mathtt{aq}_2') \wedge t^1 ++ [\mathtt{aq}_1'] ++ t^2 ++ [\mathtt{aq}_2'] ++ t^3 = t$$

Since $(t_2 - t) = t_1' + + t_2'$, there are 4 cases need to be proved:
(1) $\mathtt{aq}_1, \mathtt{aq}_2 \in_{\mathtt{aq}} t_1'$
(2) $\mathtt{aq}_2, \mathtt{aq}_2 \in_{\mathtt{aq}} t_2'$
(3) $\mathtt{aq}_2 \in_{\mathtt{aq}} t_1', \mathtt{aq}_1 \in_{\mathtt{aq}} t_2'$
(4) $\mathtt{aq}_1 \in_{\mathtt{aq}} t_1', \mathtt{aq}_2 \in_{\mathtt{aq}} t_2'$
Given both t_1' and t_2' are well-formed, we know this property is true in case (1) and (2).
By the ordering properties of labels in program $c_1; c_2$, we know all the labels in c_2 are greater than the labels in c_1, i.e.,

$$\forall \mathtt{aq}_1 \in_{\mathtt{aq}} t_1', \mathtt{aq}_2 \in_{\mathtt{aq}} t_2'.\ \mathtt{aq}_1 <_{aq} \mathtt{aq}_2\ (a)$$

Then we have the case (3) proved because the premise is `false` in this case. To prove the ordering property in case (4), it is sufficient to show the following 2 implications:

$$(\mathtt{aq}_1 <_{aq} \mathtt{aq}_2) \Longrightarrow \exists t^1, t^2, t^3, \mathtt{aq}_1', \mathtt{aq}_2'.\ s.t.,\ (\mathtt{aq}_1 =_{\mathtt{aq}} \mathtt{aq}_1') \wedge (\mathtt{aq}_2 =_{\mathtt{aq}} \mathtt{aq}_2') \wedge t^1 ++ [\mathtt{aq}_1'] ++ t^2 ++ [\mathtt{aq}_2'] ++ t^3 = t$$

$$\exists t^1, t^2, t^3, \mathtt{aq}_1', \mathtt{aq}_2'.\ s.t.,\ (\mathtt{aq}_1 =_{\mathtt{aq}} \mathtt{aq}_1') \wedge (\mathtt{aq}_2 =_{\mathtt{aq}} \mathtt{aq}_2') \wedge t^1 ++ [\mathtt{aq}_1'] ++ t^2 ++ [\mathtt{aq}_2'] ++ t^3 = t \Longrightarrow (\mathtt{aq}_1 <_{aq} \mathtt{aq}_2)$$

By Corollary 2.2.1 and the hypothesis (4), we know:

$$\exists t_{11}, t_{12}, \mathtt{aq}_1'.\ (\mathtt{aq}_1 =_{\mathtt{aq}} \mathtt{aq}_1') \wedge t_{11} + + [\mathtt{aq}_1'] + + t_{12} = t_1'$$

$$\exists t_{21}, t_{22}, \mathtt{aq}_2'.\ (\mathtt{aq}_2 =_{\mathtt{aq}} \mathtt{aq}_2') \wedge t_{21} + + [\mathtt{aq}_2'] + + t_{22} = t_2'$$

Then according to the list concatenation operations, we have:

$$\exists t_{11}, t_{12}, \mathtt{aq}_1', t_{21}, t_{22}, \mathtt{aq}_2'.\ (\mathtt{aq}_1 =_{\mathtt{aq}} \mathtt{aq}_1') \wedge (\mathtt{aq}_2 =_{\mathtt{aq}} \mathtt{aq}_2') \wedge t_{11} ++ [\mathtt{aq}_1'] ++ t_{12} ++ t_{21} ++ [\mathtt{aq}_2'] ++ t_{22} = t_1' ++ t_2'$$

Let $t^1 = t_{11}$, $t^2 = t_{12} + + t_{21}$ and $t^3 = t_{22}$, since $(t_2 - t) = t_1' + + t_2'$, we have:

$$\exists t^1, t^2, t^3, \mathtt{aq}_1', \mathtt{aq}_2'.\ s.t.,\ (\mathtt{aq}_1 =_{\mathtt{aq}} \mathtt{aq}_1') \wedge (\mathtt{aq}_2 =_{\mathtt{aq}} \mathtt{aq}_2') \wedge t^1 + + [\mathtt{aq}_1'] + + t^2 + + [\mathtt{aq}_2'] + + t^3 = (t_2 - t)$$

This case is proved.
For the other directions, according to the hypothesis (4) and (a) proved above, we know:

$$(\mathtt{aq}_1 <_{aq} \mathtt{aq}_2)$$

This case is proved.

case: $\texttt{while}\ [b]^l\ \texttt{do}\ c'$
According to the while-b and ifw-b rule, we have following evaluation:

$$\langle m,\ \texttt{while}\ [b]^l\ \texttt{do}\ c', t, w\rangle \to^* \langle m,\ \texttt{if}\ _w([v]^l, c';\ \texttt{while}\ [b]^l, 2\ \texttt{do}\ c', \texttt{skip}), t, w\rangle\ (w1)$$

where $\langle m, b\rangle \to_b^* v$ and v is either $\texttt{true}$ or $\texttt{false}$.

subcase: $v = \texttt{true}$
By evaluation rule ifw-true, we have:

$$\langle m,\ \texttt{if}\ _w([\texttt{true}]^l, c';\ \texttt{while}\ [b]^l, 2\ \texttt{do}\ c', \texttt{skip}), t, w\rangle \to^* \langle m, c';\ \texttt{while}\ [b]^l, 2\ \texttt{do}\ c', t, [l \mapsto 1]\rangle\ (w2)$$

By rule seq1, we have the following evaluation:

$$\langle m, c';\ \texttt{while}\ [b]^l, 2\ \texttt{do}\ c', t, [l \mapsto 1]\rangle \to^* \langle m', \texttt{skip};\ \texttt{while}\ [b]^l, 2\ \texttt{do}\ c', t', w'\rangle\ (w3)$$

where $\langle m, c', t, [l \mapsto 1]\rangle \to^* \langle m', \texttt{skip}, t', w'\rangle\ (w4)$.
By induction hypothesis on c' and the evaluation $(w4)$, we know $t' - t$ is well-formed.
By the rule seq2, we have:

$$\langle m', \texttt{skip};\ \texttt{while}\ [b]^l, 2\ \texttt{do}\ c', t', w'\rangle \to \langle m',\ \texttt{while}\ [b]^l, 2\ \texttt{do}\ c', t', w'\rangle\ (w5)$$

By the termination of the while loop, we know there exist an evaluation by repeating the evaluation in $(w1), (w2), (w3)$ and $(w5)$:

$$\langle m',\ \texttt{while}\ [b]^l, 2\ \texttt{do}\ c', t', w'\rangle \to^* \langle m^n,\ \texttt{if}\ _w([\texttt{false}]^l, c';\ \texttt{while}\ [b]^l, n+1,\ \texttt{do}\ c', \texttt{skip}), t^n, w^n\rangle$$

By the conclusion from $(w3)$ and $(w4)$, i.e., $(t' - t)$ is well-formed, we know for every resulting trace t^i in the evaluation of each iteration $i = 2, \ldots, n$, where $n \geq 1$, the following invariant holds:
$t^i - t^{i-1}$ is well formed.
By evaluation rule ifw-false, we have:

$$\langle m,\ \texttt{if}\ _w([\texttt{false}]^l, c';\ \texttt{while}\ [b]^l, n+1\ \texttt{do}\ c', \texttt{skip}), t^n, w^n\rangle \to^* \langle m, \texttt{skip}, t^n, w^n / l\rangle$$

It is sufficient to show $(t^n - t)$ is well-formed.
Unfolding the definition 9, it is sufficient to show that the trace $(t^n - t)$ holds both the *Uniqueness* and *Ordering* property.
Let $t_1 = t' - t$ and $t_i = t^i - t^{i-1}$ for $i = 2, \ldots, n$, unfolding the subtraction operations, we have:

$$(t^n - t) = t_1 + + t_2 + + \ldots + + t_n$$

By repeating the proof $(\diamond)$ and $(\Box)$ in the sequence case, we have this case proved.

subcase: $v = \texttt{false}$
By evaluation rule ifw-false, we have:

$$\langle m,\ \texttt{if}\ _w([\texttt{false}]^l, c';\ \texttt{while}\ [b]^l\ \texttt{do}\ c', \texttt{skip}), t, w\rangle \to^* \langle m, \texttt{skip}, t, w / l\rangle$$

Because $t - t = []$, this sub-case is proved. $\Box$

]
[JL:

$[\![\ \boxed{Memory \times Command \times QTrace \times VTrace \times WhileMap \to Memory \times Command \times QTrace \times VTrace \times W}$

$$\frac{\langle m, e\rangle \to \langle m, e'\rangle}{\langle m, [x \leftarrow e]^l, t, w\rangle \to \langle m, [x \leftarrow e']^l, t, w\rangle}\ \textbf{assn1} \qquad \frac{}{\langle m, [x \leftarrow v]^l, t, w\rangle \to \langle m[v/x], [\texttt{skip}]^l, t, w\rangle}\ \textbf{assn2}$$

$$\frac{}{\langle m, \texttt{while}\ [b]^l\ \texttt{do}\ c, t, w\rangle \to \langle m, c; \texttt{if}\,_w(b, c; \texttt{while}\ [b]^l\ \texttt{do}\ c, \texttt{skip}), t, w\rangle}\ \textbf{while-b}$$

$$\frac{m, b \to b'}{\langle m, \texttt{if}\,_w(b, c, \texttt{skip}), t, w\rangle \to \langle m, \texttt{if}\,_w(b', c, \texttt{skip}), t, w\rangle}\ \textbf{ifw-b}$$

$$\frac{}{\langle m, \texttt{if}\,_w(b, c; \texttt{while}\ [b]^l\ \texttt{do}\ c, \texttt{skip}), t, w\rangle \to \langle m, c; \texttt{while}\ [b]^l\ \texttt{do}\ c, t, (w + l)\rangle}\ \textbf{ifw-true}$$

$$\frac{}{\langle m, \texttt{if}\,_w(\texttt{false}, c; \texttt{while}\ [b]^l\ \texttt{do}\ c, \texttt{skip}), t, w\rangle \to \langle m, \texttt{skip}, t, (w - l)\rangle}\ \textbf{ifw-false}$$

$$\frac{\langle m, \psi\rangle \to_q \psi'}{\langle m, [x \leftarrow \texttt{query}(\psi)]^l, t, w\rangle \to \langle m, [x \leftarrow \texttt{query}(\psi')]^l, t, w\rangle}\ \textbf{query-e}$$

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

$$\frac{\langle m, c_1, t, w\rangle \to \langle m', c_1', t', w'\rangle}{\langle m, c_1; c_2, t, w\rangle \to \langle m', c_1'; c_2, t', w'\rangle}\ \textbf{seq1} \qquad \frac{}{\langle m, [\texttt{skip}]^l; c_2, t, w\rangle \to \langle m, c_2, t, w\rangle}\ \textbf{seq2}$$

$$\frac{\langle m, b\rangle \to_b b'}{\langle m, \texttt{if}\ ([b]^l, c_1, c_2), t, w\rangle \to \langle m, \texttt{if}\ ([b']^l, c_1, c_2), t, w\rangle}\ \textbf{if}$$

$$\frac{}{\langle m, \texttt{if}\ ([\texttt{true}]^l, c_1, c_2), t, w\rangle \to \langle m, c_1, t, w\rangle}\ \textbf{if-t} \qquad \frac{}{\langle m, \texttt{if}\ ([\texttt{false}]^l, c_1, c_2), t, w\rangle \to \langle m, c_2, t, w\rangle}\ \textbf{if-f}$$

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

Lemma 2 (While Map Remains Unchanged (Invariant))**.** *Given a program c with a starting memory m, trace t and while map w, s.t., $\langle m, c, t, w\rangle \to^* \langle m', \texttt{skip}, t', w'\rangle$ and $Labels(c) \cap Keys(w) = \emptyset$, then*

$$w = w'$$

Proof of Lemma 2. Proof in File: "`lem_wunchange.tex`" ■

] [JL:

Lemma 3 (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*

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

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

then:

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

Proof of Lemma 3. Proof in File: "`lem_twriteonly.tex`" ■

] [JL:

Lemma 4 (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*

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

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

then:

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

Proof of Lemma 4. Proof in File: "`lem_tunique.tex`" ■

] [JL:

Corollary 2.2.1.

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

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

]

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 $\mathtt{query}(x+\chi(2))$ will not be the same query if the value of x is changed.

We give a formal definition of query may dependency based on the trace-based operational semantics as follows.

Definition 10 (Query May Dependency). .
[JL: One annotated query $\mathsf{aq}_2 = (\alpha_2, l_2, w_2)$ may depend on another query $\mathsf{aq}_1 = (\alpha_1, l_1, w_1)$ in a program c, with a starting memory m and a hidden database D, denoted as $\mathsf{DEP}_{\mathsf{q}}(\mathsf{aq}_1, \mathsf{aq}_2, c, m, D)$ is defined below.

$$\exists m_1, m_3, t_1, t_3, c_2, v_1. \left(\begin{array}{l} \langle m, c, [], [] \rangle \to^* \langle m_1, [x \leftarrow \mathtt{query}(\alpha_1)]^{l_1}; c_2, t_1, w_1 \rangle \to^{\textit{query-v}} \\ \langle m_1[v_1/x], c_2, t_1 ++ [\mathsf{aq}_1], w_1 \rangle \to^* \langle m_3, \mathtt{skip}, t_3, w_3 \rangle \\ \land \left(\begin{array}{l} \mathsf{aq}_2 \in_{\mathsf{aq}} (t_3 - (t_1 ++ [\mathsf{aq}_1])) \\ \implies \exists v \in \mathcal{QD}, v \neq v_1, m_3', t_3', w_3'. \langle m_1[v/x], c_2, t_1 ++ [\mathsf{aq}_1], w_1 \rangle \\ \quad \to^* (\langle m_3', \mathtt{skip}, t_3', w_3' \rangle \\ \quad \land \mathsf{aq}_2 \notin_{\mathsf{aq}} (t_3' - (t_1 ++ [\mathsf{aq}_1]))) \end{array} \right) \\ \land \left(\begin{array}{l} \mathsf{aq}_2 \notin_{\mathsf{aq}} (t_3 - (t_1 ++ [\mathsf{aq}_1])) \\ \implies \exists v \in \mathcal{QD}, v \neq v_1, m_3', t_3', w_3'. \langle m_1[v/x], c_2, t_1 ++ [\mathsf{aq}_1], w_1 \rangle \\ \quad \to^* (\langle m_3', \mathtt{skip}, t_3', w_3' \rangle \\ \quad \land \mathsf{aq}_2 \in_{\mathsf{aq}} (t_3' - (t_1 ++ [\mathsf{aq}_1]))) \end{array} \right) \end{array} \right)$$

]

Definition 11 (Trace-based Dependency Graph). .
[JL: Given a program c, a database D, a starting memory m, the trace-based dependency graph $G(c, D, m) = (\mathtt{V}, \mathtt{E})$ is defined as:

$$\begin{array}{rcl} \textit{Vertices} & \mathtt{V} & := & \left\{ \mathsf{av} \in \mathcal{AV} \mid \exists \mathbf{m}', w, \mathtt{qt}, \mathtt{vt}.\ \langle \mathbf{m}, \mathbf{c}, [], [], [] \rangle \to^* \langle \mathbf{m}', \mathtt{skip}, \mathtt{qt}, \mathtt{vt}, w \rangle \land \mathsf{av} \in \mathtt{vt} \right\} \\ \textit{Directed Edges} & \mathtt{E} & := & \left\{ (\mathsf{av}, \mathsf{av}') \in \mathcal{AV} \times \mathcal{AV} \mid \mathsf{DEP}_{\mathsf{var}}(\mathsf{av}, \mathsf{av}', c, m, D) \right\} \end{array}$$

]

The edges are directed. When an annotated query aq may depend on its previous one aq', we have the corresponded edge $(\mathsf{aq}, \mathsf{aq}')$ from aq to aq'.

Definition 12 (Path (p)). .
Given a directed graph $G = (\mathtt{V}, \mathtt{E})$, a path p in G is defined as a sequence of edges $(e_1, \dots, e_{n-1})$ where $e_1, \dots, e_{n-1} \in \mathtt{E}$, for which there is a sequence of vertices $(v_1, \dots, v_n)$ where $v_1, \dots, v_n \in \mathtt{V}$ such that for every $1 \leq i < n$, $e_i = (v_i, v_{i+1})$ and all edges and vertices are distinct.

$(v_1, \dots, v_n)$ is the vertices sequence of this path.

[JL: Length of the path p is the number of vertices in its vertices sequence $(v_1, \dots, v_n)$, i.e., $\mathtt{len}(p) = n$.]

[JL: We use $\mathcal{PATH}(G)$ to denote the finite set containing all paths p in a directed acyclic graph $G = (\mathtt{V}, \mathtt{E})$; and $p_{\mathsf{aq}_1 \to \mathsf{aq}_2} \in \mathcal{PATH}(G)$ where $\mathsf{aq}_1, \mathsf{aq}_2 \in \mathtt{V}$ denotes the path from vertex aq_1 to aq_2 .]

Definition 13 (Adaptivity of A Program in While Language). .
Given a program c, its adaptivity is defined as the length of the longest path in its trace-based dependency graph $G(c, D, m)$, for all possible starting memory m and database D.

$$A(c) = \max \left\{ \mathtt{len}(p) \mid m \in \mathcal{M}, D \in \mathcal{DB}, p \in \mathcal{PATH}(G(c, D, m)) \right\}$$

We proved some useful properties for our language.

Lemma 5 (Trace Non-Decreasing)**.** .
[JL: For any program c with a starting memory m, trace t and while map w:

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

]

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 $\langle m, c, t, w\rangle \to \langle m, c', t', w'\rangle$, 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:**

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

  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.

□

[JL: 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 6. *[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* $(\texttt{aq}', \texttt{aq}) \in \texttt{E}$, *this is not the case that:*

$$\texttt{aq}' \geq_{\texttt{aq}} \texttt{aq}$$

Proof. This lemma is proved by showing there is a contradiction.

[JL: Assume there exists an edge $(\texttt{aq}', \texttt{aq}) \in \texttt{E}$ and $\texttt{aq}' \geq_{\texttt{aq}} \texttt{aq}$, where $\texttt{aq}' = (\alpha', l', w')$ and $\texttt{aq} = (\alpha, l, w)$. According to the Definition 11, we have:

$$DEP(\texttt{aq}', \texttt{aq}, c, m, D)\ (1)$$

Unfolding the Definition 10 in (1), we know: there exists t_1, t_3, m_1, m_3, c_2 s.t.,

$$\langle m, c, [], []\rangle \to^{*} \langle m_1, [x \leftarrow \texttt{query}(\alpha')]^{l'}; c_2, t_1, w'\rangle \to^{\textbf{query-v}} \langle m_1[v_1/x], c_2, (t_1 ++ [\texttt{aq}'], w_1\rangle \to^{*} \langle m_3, \texttt{skip}, t_3, w_3\rangle\ (a)$$

and

$$\bigwedge \left\{ \begin{array}{l} \left(\begin{array}{l} \texttt{aq} \in_{\texttt{aq}} (t_3 - (t_1 ++ [\texttt{aq}'])) \\ \implies \exists v \in \mathcal{QD}, v \neq v_1, m_3', t_3', w_3'.\ \langle m_1[v/x], c_2, t_1 ++ [\texttt{aq}'], w_1\rangle \\ \quad \to^{*} (\langle m_3', \texttt{skip}, t_3', w_3'\rangle \wedge \texttt{aq} \notin_{\texttt{aq}} (t_3' - (t_1 ++ [\texttt{aq}']))) \end{array}\right) (b) \\ \left(\begin{array}{l} \texttt{aq} \notin_{\texttt{aq}} (t_3 - (t_1 ++ [\texttt{aq}'])) \\ \implies \exists v \in \mathcal{QD}, v \neq v_1, m_3', t_3', w_3'.\langle m_1[v/x], c_2, t_1 ++ [\texttt{aq}'], w_1\rangle \\ \quad \to^{*} (\langle m_3', \texttt{skip}, t_3', w_3'\rangle \wedge \texttt{aq} \in_{\texttt{aq}} (t_3' - (t_1 ++ [\texttt{aq}']))) \end{array}\right) (c) \end{array}\right.$$

According to the Theorem 2.2 and (a), we know both t_1, t_3 are well-formed traces.
Consider 2 cases:

$$\mathtt{aq} \in_{\mathtt{aq}} (t_3 - (t_1 ++ [\mathtt{aq}'])) \ (d) \quad \vee \quad \mathtt{aq} \notin_{aq} (t_3 - (t_1 ++ [\mathtt{aq}'])) \ (e)$$

- **case: (d)**

  $$\mathtt{aq} \in_{\mathtt{aq}} (t_3 - (t_1 ++ [\mathtt{aq}'])) \ (d)$$

  By unfolding the trace subtraction operations, we have;

  $$\exists t_2. \ s.t., \ t_1 ++ [\mathtt{aq}'] ++ t_2 = t_3 \wedge \mathtt{aq} \in_{\mathtt{aq}} t_2 \ (3)$$

  According to the Corollary 2.2.1 and $\mathtt{aq} \in_{\mathtt{aq}} t_2$, we have:

  $$\exists t_{21}, t_{22}, \mathtt{aq}' \ s.t., \ (\mathtt{aq} =_{\mathtt{aq}} \mathtt{aq}') \wedge t_{21} ++ [\mathtt{aq}'] ++ t_{22} = t_2 \ (4)$$

  By rewriting (4) inside (3), we have:

  $$\exists t_{21}, t_{22}, \mathtt{aq}'' \ s.t., \ (\mathtt{aq} =_{\mathtt{aq}} \mathtt{aq}'') \wedge t_1 ++ [\mathtt{aq}'] ++ t_{21} ++ [\mathtt{aq}''] ++ t_{22} = t_3 \ \star$$

  By the *ordering* property in definition 9 and $(\star)$, we know

  $$\mathtt{aq}' <_{aq} \mathtt{aq}$$

  This is contradict to our assumption, where $\mathtt{aq}' \geq_{\mathtt{aq}} \mathtt{aq}$.

- **case: (e),**

  $$\mathtt{aq} \notin_{aq} (t_3 - (t_1 ++ [\mathtt{aq}'])) \ (e)$$

  According to the condition (c), we know: $\exists v \in \mathcal{QD}, v \neq v_1, m_3', t_3', w_3'$.

  $$\langle m_1[v/x], c_2, t_1 ++ [\mathtt{aq}'], w_1 \rangle \to^* (\langle m_3', \mathtt{skip}, t_3', w_3' \rangle \ (5) \wedge \mathtt{aq} \in_q (t_3' - (t_1 ++ [\mathtt{aq}']))) \ (6)$$

  According to the Theorem 2.2, Sub-Lemma-t and (5), we know $t_3' - (t_1 ++ [\mathtt{aq}'])$ is a well-formed trace.
  From (a), we know that the minimal line number of c_2 is greater than l', so we know that :

  $$\forall \mathtt{aq}'' \in t_3' - (t_1 ++ [\mathtt{aq}']), \mathtt{aq}'' >_{aq} \mathtt{aq}'$$

  Unfolding the trace subtraction operations in (6), we have;

  $$\exists t_2'. \ s.t., \ t_1 ++ [\mathtt{aq}'] ++ t_2' = t_3' \wedge \mathtt{aq} \in_{\mathtt{aq}} t_2' \ (7)$$

  According to the sub-lemma and (7), we have:

  $$\exists t_{21}', t_{22}' \ s.t., \ t_{21}' ++ [\mathtt{aq}] ++ t_{22}' = t_2' \ (8)$$

  By rewriting (8) inside (7), we have:

  $$t_1 ++ [\mathtt{aq}'] ++ t_{21}' ++ [\mathtt{aq}] ++ t_{22}' = t_3' \ \diamond$$

  By the *ordering* property in definition 9 and $(\diamond)$, we know

  $$\mathtt{aq}' <_{aq} \mathtt{aq}$$

  This is contradict to the assumption, $\mathtt{aq}' \geq_{\mathtt{aq}} \mathtt{aq}$.

From both cases, we derive $\mathtt{aq}' \geq_{\mathtt{aq}} \mathtt{aq}$, which is contradict to the hypothesis, i.e., $\mathtt{aq}' \geq_{\mathtt{aq}} \mathtt{aq}$. Then, we can conclude that for any directed edge $(\mathtt{aq}', \mathtt{aq}) \in \mathtt{E}$, this is not the case that:

$$\mathtt{aq}' \geq_{\mathtt{aq}} \mathtt{aq}$$

] $\square$

Lemma 7. *[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 6. $\square$

Lemma 8 (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.,:

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

Proof. Proof is obvious based on the Lemma 7. $\square$

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 three simple but relevant examples to show why the loop language suffers. 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 Dependency in SSA Language

[[

3.3 SSA form Language with Different DEPENDENCY Graph

While Map	w	$::=$	$[] \mid w[l \to n]$
Annotated Query	$\texttt{aq}$	$::=$	(α, l, w)
Annotated Variable	$\texttt{av}$	$::=$	$(\mathbf{x}, v, l, w)$
Query Trace	$\texttt{qt}$	$::=$	$[] \vert \texttt{aq} :: \texttt{qt}$
Variable Trace	$\texttt{vt}$	$::=$	$[] \vert \texttt{av} :: \texttt{qt}$

We use $\mathcal{SVAR}$ and $\mathcal{SM}$ to denote the set of SSA Variables and SSA Memories respectively.

Annotated Variable	$\mathtt{av}$	$::=$	$(\mathbf{x}, v, n)$
Variable Trace	$\mathtt{vt}$	$::=$	$[] \mid \mathtt{av} :: \mathtt{vt}$
Variable Visits	t	$::=$	$\mathcal{SVAR} \to \mathbb{N}$

We use $\mathcal{SVAR}$ and $\mathcal{SM}$ to denote the set of SSA Variables and SSA Memories respectively.

[[

Definition 14 (Assigned Variables ($\mathtt{aVar}$)). *Given a program* $\mathbf{c}$*, its assigned variables* $\mathtt{aVar}_{\mathbf{c}}$ *is a vector containing all variables newly assigned in the program preserving the order,* $\forall \mathbf{x} \in \mathtt{aVar}, \mathbf{x} \in \mathcal{SVAR}$*. It is defined as follows:*

$$\mathtt{aVar}_{\mathbf{c}} \triangleq \begin{cases} [\mathbf{x}] & \mathbf{c} = [\mathbf{x} \leftarrow \mathbf{e}]^{(l,w)} \\ [\mathbf{x}] & \mathbf{c} = [\mathbf{x} \leftarrow \mathtt{query}\,(\psi)]^{(l,w)} \\ \mathtt{aVar}_{\mathbf{c}_1} ++ \mathtt{aVar}_{\mathbf{c}_2} & \mathbf{c} = \mathbf{c}_1 ; \mathbf{c}_2 \\ \mathtt{aVar}_{\mathbf{c}_1} ++ \mathtt{aVar}_{\mathbf{c}_2} ++ [\bar{\mathbf{x}}, \bar{\mathbf{y}}, \bar{\mathbf{z}}] & \mathbf{c} = \mathtt{if}\,([\mathbf{b}]^{(l,w)}, [\bar{\mathbf{x}}, \bar{\mathbf{x}}_2, \bar{\mathbf{x}}_2], [\bar{\mathbf{y}}, \bar{\mathbf{y}}_2, \bar{\mathbf{y}}_3], [\bar{\mathbf{z}}, \bar{\mathbf{z}}_2, \bar{\mathbf{z}}_3], \mathbf{c}_1, \mathbf{c}_2) \\ \mathtt{aVar}_{\mathbf{c}'} ++ [\bar{\mathbf{x}}] & \mathbf{c} = \mathtt{while}\,([\mathbf{b}]^{(l,w)}, [\bar{\mathbf{x}}, \bar{\mathbf{x}}_2, \bar{\mathbf{x}}_2], \mathbf{c}') \end{cases}$$

Definition 15 (Query Variables ($\mathtt{qVar}$)). .
Given a program $\mathbf{c}$*, its query variables* $\mathtt{qVar}$ *is a vector containing all variables newly assigned by a query in the programm,* $\mathtt{qVar} \subset \mathcal{SVAR}$*. It is defined as follows:*

$$\mathtt{qVar}_{\mathbf{c}} \triangleq \begin{cases} [] & \mathbf{c} = [\mathbf{x} \leftarrow \mathbf{e}]^{(l,w)} \\ [\mathbf{x}] & \mathbf{c} = [\mathbf{x} \leftarrow \mathtt{query}\,(\psi)]^{(l,w)} \\ \mathtt{qVar}_{\mathbf{c}_1} ++ \mathtt{qVar}_{\mathbf{c}_2} & \mathbf{c} = \mathbf{c}_1 ; \mathbf{c}_2 \\ \mathtt{qVar}_{\mathbf{c}_1} ++ \mathtt{qVar}_{\mathbf{c}_2} ++ [\bar{\mathbf{x}}, \bar{\mathbf{y}}, \bar{\mathbf{z}}] & \mathbf{c} = \mathtt{if}\,([\mathbf{b}]^{(l,w)}, [\bar{\mathbf{x}}, \bar{\mathbf{x}}_2, \bar{\mathbf{x}}_2], [\bar{\mathbf{y}}, \bar{\mathbf{y}}_2, \bar{\mathbf{y}}_3], [\bar{\mathbf{z}}, \bar{\mathbf{z}}_2, \bar{\mathbf{z}}_3], \mathbf{c}_1, \mathbf{c}_2) \\ \mathtt{qVar}_{\mathbf{c}'} ++ [\bar{\mathbf{x}}] & \mathbf{c} = \mathtt{while}\,([\mathbf{b}]^{(l,w)}, [\bar{\mathbf{x}}, \bar{\mathbf{x}}_2, \bar{\mathbf{x}}_2], \mathbf{c}') \end{cases}$$

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:

$$A_1 ++ A_2 \triangleq \begin{cases} A_2 & A_1 = [] \\ x :: (A_1' ++ A_2) & A_1 = x :: A_1' \end{cases} \qquad (7)$$

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.]]

3.4 Trace-based Adaptivity in SSA Language

Definition 16 (remove :?: Query May Dependency in SSA Language). .
One annotated query $\mathtt{aq}_1 = (\alpha_1, l_1, w_1)$ *may depend on another query* $\mathtt{aq}_2 = (\alpha_2, l_2, w_2)$ *in a program* $\mathbf{c}$*, with a starting memory* $\mathbf{m}$ *and hidden database* D*, denoted as* $\mathsf{DEP_q}^{ssa}(\mathtt{aq}_1, \mathtt{aq}_2, \mathbf{c}, \mathbf{m}, D)$ *is defined*

[[

$$\boxed{\langle \mathbf{m}, \mathbf{a} \rangle \to \langle \mathbf{a} \rangle} \qquad \boxed{\langle \mathbf{m}, \mathbf{c}, t \rangle \to \langle \mathbf{m}', \mathbf{c}', t' \rangle}$$

$$Memory \times Command \times QTrace \times VTrace \times WhileMap \Rightarrow Memory \times Command \times QTrace \times VTrace \times Whil$$

$$\frac{\langle \mathbf{m}, \psi \rangle \to_{\mathbf{q}} \psi'}{\langle \mathbf{m}, [\mathbf{x} \leftarrow \mathtt{query}(\psi)]^{\mathbf{l}}, \mathtt{qt}, \mathtt{vt}, \mathbf{w} \rangle \to \langle \mathbf{m}, [\mathbf{x} \leftarrow \mathtt{query}(\psi')]^{\mathbf{l}}, \mathtt{qt}, \mathtt{vt}, \mathbf{w} \rangle} \textbf{ ssa-query-e}$$

$$\frac{\mathtt{query}(\alpha)(\mathbf{D}) = \mathbf{v}}{\langle \mathbf{m}, [\mathbf{x} \leftarrow \mathtt{query}(\alpha)]^{\mathbf{l}}, \mathtt{qt}, \mathtt{vt}, \mathbf{w} \rangle \to \langle (\mathbf{x} \mapsto \mathbf{v}) :: \mathbf{m}, \mathtt{skip}, \mathtt{qt} ++ [(\alpha, \mathbf{l}, \mathbf{w})], \mathtt{vt} ++ [(\mathbf{x}, \mathbf{v}, \mathbf{l}, \mathbf{w})], \mathbf{w} \rangle} \textbf{ ssa-query-v}$$

$$\frac{t'(\mathbf{x}) = t(\mathbf{x}) + 1}{\langle \mathbf{m}, [\mathbf{x} \leftarrow \mathbf{v}]^{\mathbf{l}}, \mathtt{qt}, \mathtt{vt}, \mathbf{w} \rangle \to \langle (\mathbf{x} \mapsto \mathbf{v}) :: \mathbf{m}, [\mathtt{skip}]^{\mathbf{l}}, \mathtt{qt}, \mathtt{vt} ++ [(\mathbf{x}, \mathbf{v}, \mathbf{l}, \mathbf{w})], \mathbf{w} \rangle} \textbf{ ssa-assn}$$

]] [[

$$\boxed{\langle \mathbf{m}, \mathbf{a} \rangle \to \langle \mathbf{a} \rangle} \qquad \boxed{\langle \mathbf{m}, \mathbf{c}, t \rangle \to \langle \mathbf{m}', \mathbf{c}', t' \rangle}$$

$$Memory \times Command \times VTrace \times Visits \Rightarrow Memory \times Command \times VTrace \times Visits$$

$$\frac{\langle \mathbf{m}, \psi \rangle \to_{\mathbf{q}} \psi'}{\langle \mathbf{m}, [\mathbf{x} \leftarrow \mathtt{query}(\psi)]^{\mathbf{l}}, \mathtt{vt}, \mathbf{t} \rangle \to \langle \mathbf{m}, [\mathbf{x} \leftarrow \mathtt{query}(\psi')]^{\mathbf{l}}, \mathtt{vt}, \mathbf{t} \rangle} \textbf{ ssa-query-e}$$

$$\frac{\mathtt{query}(\alpha)(\mathbf{D}) = \mathbf{v} \qquad t'[\mathbf{x}] = t[\mathbf{x}] + 1}{\langle \mathbf{m}, [\mathbf{x} \leftarrow \mathtt{query}(\alpha)]^{\mathbf{l}}, \mathtt{vt}, \mathbf{t} \rangle \to \langle (\mathbf{x} \mapsto \mathbf{v}) :: \mathbf{m}, \mathtt{skip}, \mathtt{vt} ++ [(\mathbf{x}, \alpha, \mathbf{t}'[\mathbf{x}])], \mathbf{t}' \rangle} \textbf{ ssa-query-v}$$

$$\frac{t'(\mathbf{x}) = t(\mathbf{x}) + 1}{\langle \mathbf{m}, [\mathbf{x} \leftarrow \mathbf{v}]^{\mathbf{l}}, \mathbf{t}, \mathbf{w} \rangle \to \langle (\mathbf{x} \mapsto \mathbf{v}) :: \mathbf{m}, [\mathtt{skip}]^{\mathbf{l}}, \mathtt{vt} ++ [(\mathbf{x}, \mathbf{v}, \mathbf{t}'[\mathbf{x}])], \mathbf{t}' \rangle} \textbf{ ssa-assn}$$

]]

Figure 2: Operational Semantics for the SSA Language

below.

$$\exists \mathbf{m}_1, \mathbf{m}_3, t_1, t_3, \mathbf{c}_2, v_1 .
\left(\begin{array}{l}
\langle \mathbf{m}, \mathbf{c}, [], [] \rangle \to^{*} \langle \mathbf{m}_1, [\mathbf{x} \leftarrow \mathtt{query}(\alpha_1)]^{l_1}; c_2, t_1, w_1 \rangle \to^{\textit{ssa-query-v}} \\
\langle \mathbf{m}_1[v_1/\mathbf{x}], c_2, t_1 ++ [\mathtt{aq}_1], w_1 \rangle \to^{*} \langle \mathbf{m}_3, \mathtt{skip}, t_3, w_3 \rangle \\
\bigwedge \left(\begin{array}{l}
\mathtt{aq}_2 \in_{\mathtt{aq}} (t_3 - (t_1 ++ [\mathtt{aq}_1])) \\
\implies \exists v \in \mathcal{QD}, v \neq v_1, \mathbf{m}_3', t_3', w_3' . \langle \mathbf{m}_1[v/\mathbf{x}], \mathbf{c}_2, t_1 ++ [\mathtt{aq}_1], w_1 \rangle \\
\qquad \to^{*} (\langle \mathbf{m}_3', \mathtt{skip}, t_3', w_3' \rangle \land \mathtt{aq}_2 \not\in_{\mathtt{aq}} (t_3' - (t_1 ++ [\mathtt{aq}_1])))
\end{array} \right) \\
\bigwedge \left(\begin{array}{l}
\mathtt{aq}_2 \not\in_{\mathtt{aq}} (t_3 - (t_1 ++ [\mathtt{aq}_1])) \\
\implies \exists v \in \mathcal{QD}, v \neq v_1, \mathbf{m}_3', t_3', w_3' . \langle \mathbf{m}_1[v/\mathbf{x}], \mathbf{c}_2, t_1 ++ [\mathtt{aq}_1], w_1 \rangle \\
\qquad \to^{*} (\langle \mathbf{m}_3', \mathtt{skip}, t_3', w_3' \rangle \land \mathtt{aq}_2 \in_{\mathtt{aq}} (t_3' - (t_1 ++ [\mathtt{aq}_1])))
\end{array} \right)
\end{array} \right)$$

Definition 17 (remove :?: Query Variable May Dependency)**.** .
One annotated ssa variable $\mathtt{av}_2 = (\mathbf{x}_2, l_2, w_2)$ *may depend on another one* $\mathtt{av}_1 = (\mathbf{x}_1, l_1, w_1)$ *in a program c, with a starting memory m and a hidden database D, denoted as* $\mathsf{DEP_{var}}^{ssa}(\mathtt{av}_1, \mathtt{av}_2, c, m, D)$ *is defined below.*

$$\exists \alpha_1, \alpha_2 .\ \mathtt{aq}_1 = (\alpha_1, l_1, w_1) \land \mathtt{aq}_2 = (\alpha_2, l_2, w_2) \land \mathsf{DEP_q}^{ssa}(\mathtt{aq}_1, \mathtt{aq}_2, c, m, D)$$

Definition 18 (Annotated Variables May Dependency in SSA Language)**.** .
One annotated variable $\mathtt{av}_2 = (\mathbf{x}_2, v_2, l_2, w_2)$ *may depend on another one* $\mathtt{av}_1 = (\mathbf{x}_1, v_1, l_1, w_1)$*in a program* $\mathbf{c}$*, with a starting memory* $\mathbf{m}$ *and hidden database D, denoted as* $\mathsf{DEP_{av}}(\mathtt{av}_1, \mathtt{av}_2, \mathbf{c}, \mathbf{m}, D)$ *is defined below.*

$$\exists \mathbf{m}_1, \mathbf{m}_3, \mathtt{vt}_1, \mathtt{vt}_3, \mathbf{c}_2, v_1, (\alpha_1 \lor \mathbf{e}_1) .
\left(\begin{array}{l}
\langle \mathbf{m}, \mathbf{c}, [], [] \rangle \to^{*} \langle \mathbf{m}_1, [\mathbf{x}_1 \leftarrow \mathtt{query}(\alpha_1)(/\mathbf{e}_1)]^{l_1}; \mathbf{c}_1, \mathtt{qt}_1, \mathtt{vt}_1, w_1 \rangle \to^{\textit{ssa-query-v (/ assn-v)}} \\
\langle \mathbf{m}_1[v_1/\mathbf{x}], c_2, \mathtt{qt}_1', \mathtt{vt}_1 ++ [\mathtt{av}_1], w_1 \rangle \to^{*} \langle \mathbf{m}_3, \mathtt{skip}, \mathtt{qt}_3, \mathtt{vt}_3, w_3 \rangle \\
\bigwedge \left(\begin{array}{l}
\mathtt{av}_2 \in (\mathtt{vt}_3' - (\mathtt{vt}_1 ++ [\mathtt{av}_1])) \\
\implies \exists v \in \mathcal{QD}, v \neq v_1, \mathbf{m}_3', \mathtt{qt}_3', \mathtt{vt}_3', w_3' . \langle \mathbf{m}_1[v/\mathbf{x}], \mathbf{c}_2, \mathtt{qt}_1', \mathtt{vt}_1 ++ [\mathtt{av}_1], w_1 \rangle \\
\qquad \to^{*} (\langle \mathbf{m}_3', \mathtt{skip}, \mathtt{qt}_3', \mathtt{vt}_3', w_3' \rangle \\
\qquad \land \mathtt{av}_2 \not\in (\mathtt{vt}_3' - (\mathtt{vt}_1 ++ [\mathtt{av}_1])))
\end{array} \right) \\
\bigwedge \left(\begin{array}{l}
\mathtt{av}_2 \not\in (\mathtt{vt}_3 - (\mathtt{vt}_1 ++ [\mathtt{av}_1])) \\
\implies \exists v \in \mathcal{QD}, v \neq v_1, \mathbf{m}_3', \mathtt{qt}_3', \mathtt{vt}_3', w_3' . \langle \mathbf{m}_1[v/\mathbf{x}], \mathbf{c}_2, \mathtt{qt}_1', \mathtt{vt}_1 ++ [\mathtt{av}_1], w_1 \rangle \\
\qquad \to^{*} (\langle \mathbf{m}_3', \mathtt{skip}, \mathtt{qt}_3', \mathtt{vt}_3', w_3' \rangle \\
\qquad \land \mathtt{av}_2 \in (\mathtt{vt}_3' - (\mathtt{vt}_1 ++ [\mathtt{av}_1])))
\end{array} \right)
\end{array} \right)$$

Definition 19 (Annotated Variables May Dependency in SSA Language – Version 2)**.** .
One annotated variable $\mathtt{av}_2 = (\mathbf{x}_2, v_2, n_2)$ *may depend on another one* $\mathtt{av}_1 = (\mathbf{x}_1, v_1, n_1)$*in a program* $\mathbf{c}$*, with a starting memory* $\mathbf{m}$ *and hidden database D, denoted as* $\mathsf{DEP_{av}}(\mathtt{av}_1, \mathtt{av}_2, \mathbf{c}, \mathbf{m}, D)$ *is defined below.*

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

where $\mathbf{m}(x) = \mathtt{null}$ *if* $x \notin \mathtt{dom}(\mathbf{m})$.

[[

Definition 20 (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* $\mathsf{DEP}_{\mathsf{var}}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{c})$ *is defined below.*

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

Definition 21 (Execution Based Dependency Graph). .
Given a program $\mathbf{c}$*, a database* D*, a starting memory* $\mathbf{m}$ *with its corresponding execution:* $\langle \mathbf{m}, \mathbf{c}, [], [], [] \rangle \to^* \langle \mathbf{m}', \texttt{skip}, \texttt{qt}, \texttt{vt}, [] \rangle$*, the dependency graph* $\mathbf{G_{trace}}(\mathbf{c}, D, \mathbf{m}) = (\texttt{V}, \texttt{E})$ *is defined as:*

$$\begin{array}{rlll}
\textit{Vertices} & \texttt{V} & := & \left\{ x \in \mathcal{SVAR} \mid x \in \texttt{aVar}_{\mathbf{c}} \right\} \\
\textit{Directed Edges} & \texttt{E} & := & \left\{ (x, x') \in \mathcal{SVAR} \times \mathcal{SVAR} \;\middle|\; \mathsf{DEP}_{\mathsf{var}}(x, x', \mathbf{c}) \right\} \\
\textit{Query Vertices} & \texttt{QV} & := & \left\{ x \in \mathcal{SVAR} \;\middle|\; x \in \texttt{qVar}_{\mathbf{c}} \right\} \\
\textit{Weights} & \texttt{W} & := & \left\{ (x, n) \in \mathcal{SVAR} \times \mathbb{N} \;\middle|\; \texttt{len}(\{ \texttt{av} \mid \texttt{av} \in \texttt{vt} \wedge \pi_1(\texttt{av}) = x \}) = n, \forall x \in \texttt{qVar}_{\mathbf{c}} \right\}
\end{array}$$

]]

Definition 22 (Adaptivity of A Program in SSA Language). .
Given a program $\mathbf{c}$ *in SSA language, its adaptivity is defined as the length of the longest path in its trace-based dependency graph in SSA language* $\mathbf{G_{trace}}(\mathbf{c}, D, \mathbf{m}) = (\texttt{V}, \texttt{E})$*, for all possible starting SSA memory* $\mathbf{m}$ *and database* D*.*

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

]]

3.5 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	::=	$n \mid \texttt{true} \mid \texttt{false} \mid [] \mid [v, \dots, v]$
SSA Expression	$\mathbf{e}$	::=	$v \mid \mathbf{a} \mid \mathbf{b} \mid [e, \dots, e]$
Labeled SSA Command	$\mathbf{c}$	::=	$[\mathbf{x} \leftarrow \mathbf{e}]^l \mid [\mathbf{x} \leftarrow \texttt{query}(\boldsymbol{\psi})]^l \mid \texttt{ifvar}(\bar{\mathbf{x}}, \bar{\mathbf{x}}')$
			$\texttt{while}\ [\mathbf{b}]^l, n,\ [\bar{\mathbf{x}}, \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2]\ \texttt{do}\ \mathbf{c}$
			$\mid \mathbf{c}; \mathbf{c} \mid [\,\texttt{if}\ (\mathbf{b}, [\bar{\mathbf{x}}, \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2], \mathbf{c}, \mathbf{c})]^l \mid [\texttt{skip}]^l$
SSA Memory	$\mathbf{m}$	::=	$[] \mid (\mathbf{x} \to v) :: \mathbf{m}$
While Map	w	::=	$[] \mid w[l \to n]$
Annotated Query	$\texttt{aq}$	::=	(α, l, w)
Trace	t	::=	$[] \mid \texttt{aq} :: t$

[JL: We use $\mathcal{SVAR}$, $\mathcal{SAV}$ and $\mathcal{SM}$ to denote the set of SSA Variables, Annotated SSA Variables and SSA Memories respectively.] 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.

3.6 Trace-based Operational Semantics for SSA Language

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 $\langle \mathbf{m}, \mathbf{c}, t, w \rangle \to \langle \mathbf{m}', \texttt{skip}, t', w' \rangle$. 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 3. 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

$$\boxed{\langle \mathbf{m}, \mathbf{a} \rangle \to \langle \mathbf{a} \rangle} \qquad\qquad \boxed{\langle \mathbf{m}, \mathbf{c}, t, w \rangle \to \langle \mathbf{m}', \mathbf{c}', t', w' \rangle}$$

$[JL: Memory \times Command \times Trace \times WhileMap \Rightarrow Memory \times Command \times Trace \times WhilMap]$

$$\frac{\langle \mathbf{m}, \psi \rangle \to_{\mathbf{q}} \psi'}{\langle \mathbf{m}, [\mathbf{x} \leftarrow \mathtt{query}(\psi)]^{\mathbf{l}}, \mathbf{t}, \mathbf{w} \rangle \to \langle \mathbf{m}, [\mathbf{x} \leftarrow \mathtt{query}(\psi')]^{\mathbf{l}}, \mathbf{t}, \mathbf{w} \rangle}\ \textbf{ssa-query-e}$$

$$\frac{\mathtt{query}(\alpha)(\mathbf{D}) = \mathbf{v}}{\langle \mathbf{m}, [\mathbf{x} \leftarrow \mathtt{query}(\alpha)]^{\mathbf{l}}, \mathbf{t}, \mathbf{w} \rangle \to \langle (\mathbf{x} \mapsto \mathbf{v}) :: \mathbf{m}, \mathtt{skip}, \mathbf{t} + +[(\alpha, \mathbf{l}, \mathbf{w})], \mathbf{w} \rangle}\ \textbf{ssa-query-v}$$

$$\frac{}{\langle \mathbf{m}, [\mathbf{x} \leftarrow \mathbf{v}]^{\mathbf{l}}, \mathbf{t}, \mathbf{w} \rangle \to \langle (\mathbf{x} \mapsto \mathbf{v}) :: \mathbf{m}, [\mathtt{skip}]^{\mathbf{l}}, \mathbf{t}, \mathbf{w} \rangle}\ \textbf{ssa-assn}$$

$$\frac{\langle \mathbf{m}, \mathbf{b} \rangle \to_{b} \mathbf{b}'}{\langle \mathbf{m}, [\ \mathtt{if}\ (\mathbf{b}, [\bar{\mathbf{x}}, \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2], [\bar{\mathbf{y}}, \bar{\mathbf{y}}_1, \bar{\mathbf{y}}_2], [\bar{\mathbf{z}}, \bar{\mathbf{z}}_1, \bar{\mathbf{z}}_2], \mathbf{c}_1, \mathbf{c}_2)]^{\mathbf{l}}, \mathbf{t}, \mathbf{w} \rangle \to \langle \mathbf{m}, [\ \mathtt{if}\ (\mathbf{b}', [\bar{\mathbf{x}}, \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2], [\bar{\mathbf{y}}, \bar{\mathbf{y}}_1, \bar{\mathbf{y}}_2], [\bar{\mathbf{z}}, \bar{\mathbf{z}}_1, \bar{\mathbf{z}}_2], \mathbf{c}_1, \mathbf{c}_2)]^{\mathbf{l}}, \mathbf{t}, \mathbf{w} \rangle}\ \textbf{ssa-if}$$

$$\frac{}{\langle \mathbf{m}, [\ \mathtt{if}\ (\mathtt{true}, [\bar{\mathbf{x}}, \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2], [\bar{\mathbf{y}}, \bar{\mathbf{y}}_1, \bar{\mathbf{y}}_2], [\bar{\mathbf{z}}, \bar{\mathbf{z}}_1, \bar{\mathbf{z}}_2], \mathbf{c}_1, \mathbf{c}_2)]^{\mathbf{l}}, \mathbf{t}, \mathbf{w} \rangle \to \langle m, c_1;\ \mathtt{ifvar}(\bar{\mathbf{x}}, \bar{\mathbf{x}}_1); \mathtt{ifvar}(\bar{\mathbf{y}}, \bar{\mathbf{y}}_2); \mathtt{ifvar}(\bar{\mathbf{z}}, \bar{\mathbf{z}}_1), t, w \rangle}\ \textbf{ssa-}$$

$$\frac{}{\langle \mathbf{m}, [\ \mathtt{if}\ (\mathtt{false}, [\bar{\mathbf{x}}, \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2], [\bar{\mathbf{y}}, \bar{\mathbf{y}}_1, \bar{\mathbf{y}}_2], [\bar{\mathbf{z}}, \bar{\mathbf{z}}_1, \bar{\mathbf{z}}_2], \mathbf{c}_1, \mathbf{c}_2)]^{\mathbf{l}}, \mathbf{t}, \mathbf{w} \rangle \to \langle m, c_2; \mathtt{ifvar}(\bar{\mathbf{x}}, \bar{\mathbf{x}}_2); \mathtt{ifvar}(\bar{\mathbf{y}}, \bar{\mathbf{y}}_1); \mathtt{ifvar}(\bar{\mathbf{z}}, \bar{\mathbf{z}}_2), t, w \rangle}\ \textbf{ssa}$$

$$\frac{}{\langle \mathbf{m}, \mathtt{ifvar}(\bar{\mathbf{x}}, \bar{\mathbf{x}}'), \mathbf{t}, \mathbf{w} \rangle \to \langle (\bar{\mathbf{x}} \to \mathbf{m}(\bar{\mathbf{x}}')) :: \mathbf{m}, \mathtt{skip}, \mathbf{t}, \mathbf{w} \rangle}\ \textbf{ssa-}\mathtt{ifvar}$$

$$\frac{n = 0 \to i = 1 \qquad n > 0 \to i = 2}{\begin{array}{c}\langle \mathbf{m},\ \mathtt{while}\ [\mathbf{b}]^{\mathbf{l}}, \mathbf{n}, [\bar{\mathbf{x}}', \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2]\ \mathtt{do}\ \mathbf{c}, t, w \rangle \to \\ \langle \mathbf{m},\ \mathtt{if}\ _{w}(\mathbf{b}[\bar{\mathbf{x}}_{\mathbf{i}}/\bar{\mathbf{x}}'], [\bar{\mathbf{x}}', \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2], \mathbf{n}, \mathbf{c}[\bar{\mathbf{x}}_{\mathbf{i}}/\bar{\mathbf{x}}'];\ \mathtt{while}\ [\mathbf{b}]^{\mathbf{l}}, \mathbf{n} + \mathbf{1}, [\bar{\mathbf{x}}', \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2]\ \mathtt{do}\ \mathbf{c}, \mathtt{skip}), t, w \rangle\end{array}}\ \textbf{ssa-while-b}$$

$$\frac{\mathbf{m}, \mathbf{b} \to \mathbf{b}'}{\langle \mathbf{m},\ \mathtt{if}\ _{\mathbf{w}}(\mathbf{b}, [\bar{\mathbf{x}}', \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2], \mathbf{n}, \mathbf{c}_1, \mathbf{c}_2), t, w \rangle \to \langle \mathbf{m},\ \mathtt{if}\ _{\mathbf{w}}(\mathbf{b}', [\bar{\mathbf{x}}', \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2], \mathbf{n}, \mathbf{c}_1, \mathbf{c}_2), t, w \rangle}\ \textbf{ssa-ifw-b}$$

$$\frac{}{\begin{array}{l}\langle \mathbf{m},\ \mathtt{if}\ _{\mathbf{w}}(\mathtt{true}, [\bar{\mathbf{x}}', \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2], \mathbf{n}, \mathbf{c};\ \mathtt{while}\ [\mathbf{b}]^{\mathbf{l}}, \mathbf{n}, [\bar{\mathbf{x}}', \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2]\ \mathtt{do}\ \mathbf{c}, \mathtt{skip}), t, w \rangle \\ \to \langle \mathbf{m},\ \mathtt{if}\ _{\mathbf{w}}(\mathtt{true}, [\bar{\mathbf{x}}', \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2], \mathbf{n}, \mathbf{c};\ \mathtt{while}\ [\mathbf{b}]^{\mathbf{l}}, \mathbf{n}, [\bar{\mathbf{x}}', \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2]\ \mathtt{do}\ \mathbf{c}, t, (w + l) \rangle\end{array}}\ \textbf{ssa-ifw-true}$$

$$\frac{n = 0 \to i = 1 \qquad n > 0 \to i = 2}{\begin{array}{c}\langle \mathbf{m},\ \mathtt{if}\ _{\mathbf{w}}(\mathtt{false}, [\bar{\mathbf{x}}', \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2], \mathbf{n}, \mathbf{c};\ \mathtt{while}\ [\mathbf{b}]^{\mathbf{l}}, \mathbf{n}, [\bar{\mathbf{x}}', \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2]\ \mathtt{do}\ \mathbf{c}, \mathtt{skip})), t, w \rangle \\ \to \langle \mathbf{m}, \mathtt{skip}; \mathtt{ifvar}(\bar{\mathbf{x}}', \bar{\mathbf{x}}_{\mathbf{i}}), t, (w - l) \rangle\end{array}}\ \textbf{ssa-ifw-false}$$

Figure 3: Operational Semantics for the SSA Language

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 $\mathbf{m}(\bar{\mathbf{x_2}})$.

3.7 SSA Transformation

We use a translation environment δ, to map variables x in the `While` language to those variables $\mathbf{x}$ in the SSA language. We use a name environment denoted as Σ as a set of ssa variables, to get a fresh variable by $fresh(\Sigma)$. We define $\delta_1 \bowtie \delta_2$ in a similar way as [3].

$$\delta_1 \bowtie \delta_2 = \{(x, \mathbf{x_1}, \mathbf{x_2}) \in \mathcal{VAR} \times \mathcal{SVAR} \times \mathcal{SVAR} \mid x \mapsto \mathbf{x_1} \in \delta_1, x \mapsto \mathbf{x_2} \in \delta_2, \mathbf{x_1} \neq \mathbf{x_2}\}$$

$$\delta_1 \bowtie \delta_2 / \bar{x} = \{(x, \mathbf{x_1}, \mathbf{x_2}) \in \mathcal{VAR} \times \mathcal{SVAR} \times \mathcal{SVAR} \mid x \notin \bar{x} \land x \mapsto \mathbf{x_1} \in \delta_1, x \mapsto \mathbf{x_2} \in \delta_2, \mathbf{x_1} \neq \mathbf{x_2}\}$$

We call a list of variables $\bar{x}$.

$$[\bar{x}, \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}] = \{(x, x_1, x_2) | \forall 0 \leq i < |\bar{x}|, x = \bar{x}[i] \land x_1 = \bar{x_1}[i] \land x_2 = \bar{x_2}[i] \land |\bar{x}| = |\bar{x_1}| = |\bar{x_2}|\}$$

$$\boxed{\delta; e \hookrightarrow \mathbf{e}} \qquad \frac{}{\delta; x \hookrightarrow \delta(x)}\ \textbf{S-VAR} \qquad \boxed{\Sigma; \delta; c \hookrightarrow \mathbf{c}; \delta'; \Sigma'}$$

$$\frac{\begin{array}{c}\delta; b \hookrightarrow \mathbf{b} \qquad \Sigma; \delta; c_1 \hookrightarrow \mathbf{c_1}; \delta_1; \Sigma_1 \qquad \Sigma_1; \delta; c_2 \hookrightarrow \mathbf{c_2}; \delta_2; \Sigma_2 \\ [\bar{x}, \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}] = \delta_1 \bowtie \delta_2 \qquad [\bar{y}, \bar{\mathbf{y_1}}, \bar{\mathbf{y_2}}] = \delta \bowtie \delta_1 / \bar{x} \qquad [\bar{z}, \bar{\mathbf{z_1}}, \bar{\mathbf{z_2}}] = \delta \bowtie \delta_2 / \bar{x} \\ \delta' = \delta[\bar{x} \mapsto \bar{\mathbf{x}}'][\bar{y} \mapsto \bar{\mathbf{y}}'][\bar{z} \mapsto \bar{\mathbf{z}}'] \qquad \bar{\mathbf{x}}', \bar{\mathbf{y}}', \bar{\mathbf{z}}'\ fresh(\Sigma_2) \qquad \Sigma' = \Sigma_2 \cup \{\bar{\mathbf{x}}', \bar{\mathbf{y}}', \bar{\mathbf{z}}'\}\end{array}}{\Sigma; \delta; [\,\texttt{if}\ (b, c_1, c_2)]^l \hookrightarrow [\,\texttt{if}\ (\mathbf{b}, [\bar{\mathbf{x}}', \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}], [\bar{\mathbf{y}}', \bar{\mathbf{y_1}}, \bar{\mathbf{y_2}}], [\bar{\mathbf{z}}', \bar{\mathbf{z_1}}, \bar{\mathbf{z_2}}], \mathbf{c_1}, \mathbf{c_2})]^l; \delta'; \Sigma'}\ \textbf{S-IF}$$

$$\frac{\delta; e \hookrightarrow \mathbf{e} \qquad \delta' = \delta[x \mapsto \mathbf{x}] \qquad \mathbf{x}\ fresh(\Sigma) \qquad \Sigma' = \Sigma \cup \{\mathbf{x}\}}{\Sigma; \delta; [x \leftarrow e]^l \hookrightarrow [\mathbf{x} \leftarrow \mathbf{e}]^l; \delta'; \Sigma'}\ \textbf{S-ASSN}$$

$$\frac{\delta; \texttt{query} \hookrightarrow \texttt{query} \qquad \delta; \psi \hookrightarrow \boldsymbol{\psi} \qquad \delta' = \delta[x \mapsto \mathbf{x}] \qquad \mathbf{x}\ fresh(\Sigma) \qquad \Sigma' = \Sigma \cup \{\mathbf{x}\}}{\Sigma; \delta; [x \leftarrow \texttt{query}(\psi)]^l \hookrightarrow [\mathbf{x} \leftarrow \texttt{query}(\boldsymbol{\psi})]^l; \delta'; \Sigma'}\ \textbf{S-QUERY}$$

$$\frac{\begin{array}{c}\Sigma; \delta; c \hookrightarrow \mathbf{c_1}; \delta_1; \Sigma_1 \qquad [\bar{x}, \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}] = \delta \bowtie \delta_1 \\ \bar{\mathbf{x}}'\ fresh(\Sigma_1) \qquad \delta' = \delta[\bar{x} \mapsto \bar{\mathbf{x}}'] \qquad \delta'; b \hookrightarrow \mathbf{b} \qquad \mathbf{c}' = \mathbf{c_1}[\bar{\mathbf{x}}'/\bar{\mathbf{x_1}}]\end{array}}{\Sigma; \delta; \texttt{while}\ [b]^l\ \texttt{do}\ c \hookrightarrow \texttt{while}\ [\mathbf{b}]^{\mathbf{l}}, \mathbf{0}, [\bar{\mathbf{x}}', \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}]\ \texttt{do}\ \mathbf{c}; \delta'; \Sigma_1 \cup \{\bar{\mathbf{x}}'\}}\ \textbf{S-WHILE}$$

$$\frac{\Sigma; \delta; c_1 \hookrightarrow \mathbf{c_1}; \delta_1; \Sigma_1 \qquad \Sigma_1; \delta_1; c_2 \hookrightarrow \mathbf{c_2}; \delta'; \Sigma'}{\Sigma; \delta; c_1; c_2 \hookrightarrow \mathbf{c_1}; \mathbf{c_2}\ ; \delta'; \Sigma'}\ \textbf{S-SEQ}$$

$$
\begin{array}{lcl}
|\,\mathtt{while}\ [\mathbf{b}]^{l}, n, [\bar{\mathbf{x}}, \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}]\ \mathtt{do}\ \mathbf{c}| & = & \mathtt{while}\ [|\mathbf{b}|]^{l},\ \mathtt{do}\ |\mathbf{c}| \\
|\mathbf{c_1};\mathbf{c_2}| & = & |\mathbf{c_1}|;|\mathbf{c_2}| \\
|[\,\mathtt{if}\ (\mathbf{b}, [\bar{\mathbf{x}}, \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}], [\bar{\mathbf{y}}, \bar{\mathbf{y_1}}, \bar{\mathbf{y_2}}], [\bar{\mathbf{z}}, \bar{\mathbf{z_1}}, \bar{\mathbf{z_2}}], \mathbf{c_1}, \mathbf{c_2})]^{l}| & = & [\,\mathtt{if}\ (|\mathbf{b}|, |\mathbf{c_1}|, |\mathbf{c_2}|)]^{l} \\
|[\mathbf{x} \leftarrow \mathbf{e}]^{l}| & = & [|\mathbf{x}| \leftarrow |\mathbf{e}|]^{l} \\
|[\mathbf{x} \leftarrow \mathtt{query}(\boldsymbol{\psi})]^{l}| & = & [|\mathbf{x}| \leftarrow |\mathtt{query}(\boldsymbol{\psi})|]^{l} \\
|\mathbf{x}_i| & = & x \\
|n| & = & n \\
|\mathbf{a}_1 \oplus_a \mathbf{a}_2| & = & |\mathbf{a_1}| \oplus_a |\mathbf{a_2}| \\
|\mathbf{b}_1 \oplus_b \mathbf{b}_2| & = & |\mathbf{b}_1| \oplus_b |\mathbf{b}_2|
\end{array}
$$

Figure 4: The Erasure of SSA

Concrete examples.

$$
c_1 \triangleq
\begin{array}{l}
\left[x \leftarrow \mathtt{query}(1)\right]^{1}; \\
\mathtt{if}\ (x == 0)^{2} \\
\mathtt{then}\ \left[y \leftarrow \mathtt{query}(2)\right]^{3} \\
\mathtt{else}\ \left[y \leftarrow 0\right]^{4}; \\
\mathtt{if}\ (x == 1)^{5} \\
\mathtt{then}\ \left[y \leftarrow 0\right]^{6} \\
\mathtt{else}\ \left[y \leftarrow \mathtt{query}(2)\right]^{7}
\end{array}
\quad \hookrightarrow \quad
\begin{array}{l}
\left[\mathbf{x_1} \leftarrow \mathtt{query}(1)\right]^{1}; \\
\mathtt{if}\ (\mathbf{x_1} == \mathbf{0})^{2}, [\mathbf{y_3}, \mathbf{y_1}, \mathbf{y_2}], [], [] \\
\mathtt{then}\ \left[\mathbf{y_1} \leftarrow \mathtt{query}(2)\right]^{3} \\
\mathtt{else}\ \left[\mathbf{y_2} \leftarrow \mathbf{0}\right]^{4}; \\
\mathtt{if}\ (\mathbf{x_1} == \mathbf{1})^{5}, [\mathbf{y_6}, \mathbf{y_4}, \mathbf{y_5}] \\
\mathtt{then}\ \left[\mathbf{y_4} \leftarrow \mathbf{0}\right]^{6} \\
\mathtt{else}\ \left[\mathbf{y_5} \leftarrow \mathtt{query}(2)\right]^{7}
\end{array}
$$

$$
c_2 \triangleq
\begin{array}{l}
\left[x \leftarrow \mathtt{query}(1)\right]^{1}; \\
\left[y \leftarrow \mathtt{query}(2)\right]^{2}; \\
\mathtt{if}\ (x + y == 5)^{3} \\
\mathtt{then}\ \left[z \leftarrow \mathtt{query}(3)\right]^{4} \\
\mathtt{else}\ \left[\mathtt{skip}\right]^{5}; \\
\left[w \leftarrow q_4\right]^{6};
\end{array}
\quad \hookrightarrow \quad
\begin{array}{l}
\left[\mathbf{x_1} \leftarrow \mathtt{query}(1)\right]^{1}; \\
\left[\mathbf{y_1} \leftarrow \mathtt{query}(2)\right]^{2}; \\
\mathtt{if}\ (\mathbf{x_1} + \mathbf{y_1} == \mathbf{5})^{3}, [], [], [] \\
\mathtt{then}\ \left[\mathbf{z_1} \leftarrow \mathtt{query}(3)\right]^{4} \\
\mathtt{else}\ \left[\mathtt{skip}\right]^{5}; \\
\left[\mathbf{w_1} \leftarrow \mathtt{query}(4)\right]^{6};
\end{array}
$$

$$
c_3 \triangleq
\begin{array}{l}
\left[x \leftarrow \mathtt{query}(1)\right]^{1}; \\
[i \leftarrow 0]^{2}; \\
\mathtt{while}\ [i < 100]^{3}\ \mathtt{do} \\
\Big(\left[z \leftarrow \mathtt{query}(3)\right]^{4}; \\
[x \leftarrow z + x]^{5}; \\
[i \leftarrow i + 1]^{6}\Big);
\end{array}
\quad \hookrightarrow \quad
\begin{array}{l}
\left[\mathbf{x_1} \leftarrow \mathtt{query}(1)\right]^{1}; \\
[\mathbf{i_1} \leftarrow 0]^{2}; \\
\mathtt{while}\ [\mathbf{i_1} < 100]^{3}, 0,\ [\mathbf{x_3}, \mathbf{x_1}, \mathbf{x_2}], [\mathbf{i_3}, \mathbf{i_1}, \mathbf{i_2}]\ \mathtt{do} \\
\Big(\left[\mathbf{z_1} \leftarrow \mathtt{query}(3)\right]^{4}; \\
[\mathbf{x_2} \leftarrow \mathbf{z_1} + \mathbf{x_3}]^{5}; \\
[\mathbf{i_2} \leftarrow \mathbf{i_3} + 1]^{6}\Big);
\end{array}
$$

3.8 Trace-based Adaptivity in SSA Language

With the trace-based operational semantics of the SSA language at hand, we are able to provide the trace-based adaptivity definition in the SSA language, by mimicking the corresponding definitions in the `While` language.

Definition 23 (Query May Dependency in SSA Language)**.** .
[JL: One annotated query $\mathtt{aq}_1 = (\alpha_1, l_1, w_1)$ *may depend on another query* $\mathtt{aq}_2 = (\alpha_2, l_2, w_2)$ *in a*

program $\mathbf{c}$*, with a starting memory* $\mathbf{m}$ *and hidden database* D*, denoted as* $\mathsf{DEP_q}^{ssa}(\mathsf{aq_1}, \mathsf{aq_2}, \mathbf{c}, \mathbf{m}, D)$ *is defined below.*

$$\begin{array}{l}
\exists \mathbf{m}_1, \mathbf{m}_3, t_1, t_3, \mathbf{c}_2, v_1. \\
\left(\begin{array}{l}
\langle \mathbf{m}, \mathbf{c}, [], []\rangle \to^* \langle \mathbf{m}_1, [\mathbf{x} \leftarrow \mathtt{query}(\alpha_1)]^{l_1}; c_2, t_1, w_1\rangle \to^{\textit{ssa-query-v}} \\
\langle \mathbf{m}_1[v_1/\mathbf{x}], c_2, t_1 ++ [\mathsf{aq_1}], w_1\rangle \to^* \langle \mathbf{m}_3, \mathtt{skip}, t_3, w_3\rangle \\
\bigwedge \left(\begin{array}{l}
\mathsf{aq_2} \in_{\mathsf{aq}} (t_3 - (t_1 ++ [\mathsf{aq_1}])) \\
\implies \exists v \in \mathcal{QD}, v \neq v_1, \mathbf{m}_3', t_3', w_3'. \langle \mathbf{m}_1[v/\mathbf{x}], \mathbf{c}_2, t_1 ++ [\mathsf{aq_1}], w_1\rangle \\
\quad \to^* (\langle \mathbf{m}_3', \mathtt{skip}, t_3', w_3'\rangle \wedge \mathsf{aq_2} \not\in_{\mathsf{aq}} (t_3' - (t_1 ++ [\mathsf{aq_1}])))
\end{array}\right) \\
\bigwedge \left(\begin{array}{l}
\mathsf{aq_2} \not\in_{\mathsf{aq}} (t_3 - (t_1 ++ [\mathsf{aq_1}])) \\
\implies \exists v \in \mathcal{QD}, v \neq v_1, \mathbf{m}_3', t_3', w_3'. \langle \mathbf{m}_1[v/\mathbf{x}], \mathbf{c}_2, t_1 ++ [\mathsf{aq_1}], w_1\rangle \\
\quad \to^* (\langle \mathbf{m}_3', \mathtt{skip}, t_3', w_3'\rangle \wedge \mathsf{aq_2} \in_{\mathsf{aq}} (t_3' - (t_1 ++ [\mathsf{aq_1}])))
\end{array}\right)
\end{array}\right)
\end{array}$$

]

Definition 24 (SSA Variable May Dependency)**.** .
[JL: One annotated ssa variable $\mathsf{av_2} = (\mathbf{x}_2, l_2, w_2)$ *may depend on another one* $\mathsf{av_1} = (\mathbf{x}_1, l_1, w_1)$ *in a program c, with a starting memory m and a hidden database D, denoted as* $\mathsf{DEP_{var}}^{ssa}(\mathsf{av_1}, \mathsf{av_2}, c, m, D)$ *is defined below.*

$$\exists \alpha_1, \alpha_2.\ \mathsf{aq_1} = (\alpha_1, l_1, w_1) \wedge \mathsf{aq_2} = (\alpha_2, l_2, w_2) \wedge \mathsf{DEP_q}^{ssa}(\mathsf{aq_1}, \mathsf{aq_2}, c, m, D)$$

]

[[

Lemma 9 (Inversion of Query May Dependency in SSA Language)**.** *Given a program* $\mathbf{c}$*, with a starting memory* $\mathbf{m}$ *and hidden database* D*, if one annotated query* $\mathsf{aq_1} = (\alpha_1, l_1, w_1)$ *may depend on another query* $\mathsf{aq_2} = (\alpha_2, l_2, w_2)$*, i.e.,* $\mathsf{DEP_q}^{ssa}(\mathsf{aq_1}, \mathsf{aq_2}, \mathbf{c}, \mathbf{m}, D)$*, Then there must be a sequence of variables* $x_1, \dots, x_n$ *s.t.,*

$$x_i \in FV(e_{i+1}) \wedge x_{i+1} \leftarrow e_{i+1} \in_{\mathbf{c}} \mathbf{c},\ i = 1, \dots, n-1$$

]]

Definition 25 (Trace-Based Dependency Graph in SSA Language)**.** .
[JL: Given a program $\mathbf{c}$*, a database* D*, a starting memory* $\mathbf{m}$*, the dependency graph* $\mathbf{G_{trace}}(\mathbf{c}, D, \mathbf{m}) = (\mathtt{V}, \mathtt{E})$ *is defined as:*

$$\begin{array}{rrcl}
\textit{Vertices} & \mathtt{V} & := & \left\{\mathtt{av} \in \mathcal{SAV} \middle| \exists \mathbf{m}', w, \mathtt{qt}, \mathtt{vt}.\ \langle \mathbf{m}, \mathbf{c}, [], [], []\rangle \to^* \langle \mathbf{m}', \mathtt{skip}, \mathtt{qt}, \mathtt{vt}, w\rangle \wedge \mathtt{av} \in \mathtt{vt}\right\} \\
\textit{Directed Edges} & \mathtt{E} & := & \left\{(\mathtt{av}, \mathtt{av}') \in \mathcal{SAV} \times \mathcal{AQ} \ \middle|\ \mathsf{DEP_{var}}^{ssa}(\mathtt{av}, \mathtt{av}', \mathbf{c}, \mathbf{m}, D)\right\}
\end{array}$$

]

Definition 26 (Adaptivity of A Program in SSA Language)**.** .
Given a program $\mathbf{c}$ *in SSA language, its adaptivity is defined as the length of the longest path in its trace-based dependency graph in SSA language* $\mathbf{G_{trace}}(\mathbf{c}, D, \mathbf{m}) = (\mathtt{V}, \mathtt{E})$*, for all possible starting SSA memory* $\mathbf{m}$ *and database* D*.*

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

Theorem 3.1 (Graph Equivalence)**.** .
Given $\Sigma; \delta; c \hookrightarrow \mathbf{c}; \delta'; \Sigma'$*,* $\forall m. m \vDash \delta. \forall \mathbf{m}, \mathbf{m} \vDash_{ssa} \delta \wedge m = \delta^{-1}(\mathbf{m})$*, then* $G(c, D, m) = \mathbf{G_{trace}}(\mathbf{c}, D, \mathbf{m})$*.*

Proof. The nodes in the two graphs are the same according to soundness of the transformation theorem. We will show that the edges are the same. Unfold the definition of DEP_q, when we find some v in the codomain of one query that may change the appearance of another query, we can always show that $v = \mathbf{v}$ works for $\mathsf{DEP}_\mathsf{q}{}^{ssa}$ as well. In the other direction, for any $\mathbf{v}$ that may change the appearance of another query and lead to a different trace t_3' in the ssa execution, there is the same $v = \mathbf{v}$ so that the trace t_3' is generated in the while language execution, according to the transformation theorem. $\square$

Corollary 3.1.1 (Trace-Based Adaptivity Equivalence)**.** .
Given $\Sigma;\delta;c \hookrightarrow \mathbf{c};\delta';\Sigma'$, $\forall m.m \vDash \delta.\forall \mathbf{m},\mathbf{m} \vDash_{ssa} \delta \wedge m = \delta^{-1}(\mathbf{m})$, *starting with a trace t and while map w, then* $A(c) = A(\mathbf{c})$.

3.9 The Soundness of the Transformation

In this subsection, we show our transformation from the `While` language to its SSA form is sound with respect to the adaptivity. To be specific, a transformed program $\mathbf{c}$ starting with appropriate configuration, generates the same trace as the program before the transformation c, in its corresponding configuration.

Definition 27 ([[Written Variables]])**.** .
We defined the assigned variables in the while language program c as $\mathsf{wr}(c)$,*the assigned variables in the ssa-form program* $\mathbf{c}$ *as* $\mathsf{wr_s}(\mathbf{c})$ *defined as follows.*

$$\begin{array}{lcl}
\mathsf{wr}(x \leftarrow e) & = & \{x\} \\
\mathsf{wr}(x \leftarrow \mathtt{query}(\psi)) & = & \{x\} \\
\mathsf{wr}(c_1; c_2) & = & \mathsf{wr}(c_1) \cup \mathsf{wr}(c_2) \\
\mathsf{wr}(\mathtt{while}\ b\ \mathtt{do}\ c) & = & \mathsf{wr}(c) \\
\mathsf{wr}(\mathtt{if}\ (b, c_1, c_2)) & = & \mathsf{wr}(c_1) \cup \mathsf{wr}(c_2)
\end{array}$$

$$\begin{array}{lcl}
\mathsf{wr_s}(\mathbf{x} \leftarrow \mathbf{e}) & = & \{\mathbf{x}\} \\
\mathsf{wr_s}(\mathbf{x} \leftarrow \mathtt{query}(\boldsymbol{\psi})) & = & \{\mathbf{x}\} \\
\mathsf{wr_s}(\mathbf{c_1};\mathbf{c_2}) & = & \mathsf{wr_s}(\mathbf{c_1}) \cup \mathsf{wr_s}(\mathbf{c_2}) \\
\mathsf{wr_s}(\mathtt{while}\ \mathbf{b}, \mathbf{n}, [\bar{\mathbf{x}}, \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2]\ \mathtt{do}\ \mathbf{c}) & = & \{\bar{\mathbf{x}}\} \cup \mathsf{wr_s}(\mathbf{c}) \\
\mathsf{wr_s}(\mathtt{if}\ (\mathbf{b}, [\bar{\mathbf{x}}, \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2], [\bar{\mathbf{y}}, \bar{\mathbf{y}}_1, \bar{\mathbf{y}}_2], [\bar{\mathbf{z}}, \bar{\mathbf{z}}_1, \bar{\mathbf{z}}_2], \mathbf{c}_1, \mathbf{c}_2)) & = & \{\bar{\mathbf{x}}, \bar{\mathbf{y}}, \bar{\mathbf{z}}\} \cup \mathsf{wr_s}(\mathbf{c_1}) \cup \mathsf{wr_s}(\mathbf{c_2})
\end{array}$$

Definition 28 ([[Read Variables]])**.** .
The variables read in the while language programs c as $\mathsf{rd}(c)$, *variables used in ssa-form program* $\mathbf{c}$ *:*

$$\begin{array}{lcl}
\mathsf{rd}(x \leftarrow e) & = & \mathsf{rd}(e) \\
\mathsf{rd}(x \leftarrow \mathtt{query}(\psi)) & = & \{\} \\
\mathsf{rd}(c_1; c_2) & = & \mathsf{rd}(c_1) \cup \mathsf{rd}(c_2) \\
\mathsf{rd}(\mathtt{loop}\ a\ \mathtt{do}\ c) & = & \mathsf{rd}(a) \cup \mathsf{rd}(c) \\
\mathsf{rd}(\mathtt{if}\ (b, c_1, c_2)) & = & \mathsf{rd}(b) \cup \mathsf{rd}(c_1) \cup \mathsf{rd}(c_2)
\end{array}$$

$$\begin{array}{lcl}
\mathsf{rd_s}(\mathbf{x} \leftarrow \mathbf{e}) & = & \mathsf{rd_s}(\mathbf{e}) \\
\mathsf{rd_s}(\mathbf{x} \leftarrow \mathtt{query}(\boldsymbol{\psi})) & = & \{\} \\
\mathsf{rd_s}(\mathbf{c_1};\mathbf{c_2}) & = & \mathsf{rd_s}(\mathbf{c_1}) \cup \mathsf{rd_s}(\mathbf{c_2}) \\
\mathsf{rd_s}(\mathtt{while}\ \mathbf{b}, \mathbf{n}, [\bar{\mathbf{x}}, \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2]\ \mathtt{do}\ \mathbf{c}) & = & \mathsf{rd_s}(\mathbf{b}) \cup \mathsf{rd_s}(\mathbf{c}) \cup \{\bar{\mathbf{x}}_1\} \cup \{\bar{\mathbf{x}}_2\} \\
\mathsf{rd_s}(\mathtt{if}\ (\mathbf{b}, [\bar{\mathbf{x}}, \bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2], [\bar{\mathbf{y}}, \bar{\mathbf{y}}_1, \bar{\mathbf{y}}_2], [\bar{\mathbf{z}}, \bar{\mathbf{z}}_1, \bar{\mathbf{z}}_2], \mathbf{c}_1, \mathbf{c}_2)) & = & \mathsf{rd_s}(\mathbf{b}) \cup \mathsf{rd_s}(\mathbf{c_1}) \cup \mathsf{rd_s}(\mathbf{c_2}) \cup \{\bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2, \bar{\mathbf{y}}_1, \bar{\mathbf{y}}_2, \bar{\mathbf{z}}_1, \bar{\mathbf{z}}_2\}
\end{array}$$

Definition 29 ([[Necessary Variables]])**.** .
We call the variables needed to be assigned before executing the program c as necessary variables $\mathsf{FV}(c)$. *Its ssa form is :* $\mathsf{nv_s}(\mathbf{c})$.

$$
\begin{array}{lcl}
nv(x \leftarrow e) & = & \mathsf{rd}(e) \\
nv(x \leftarrow \mathtt{query}(\psi)) & = & \{\} \\
nv(\mathtt{while}\ b\ \mathtt{do}\ c) & = & \mathsf{rd}(b) \cup nv(c) \\
nv(\mathtt{if}\ (b, c_1, c_2)) & = & \mathsf{rd}(b) \cup nv(c_1) \cup nv(c_2) \\
nv(c_1; c_2) & = & nv(c_1) \cup (nv(c_2) - \mathsf{wr}(c_1))
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{nv_s}(\mathbf{x} \leftarrow \mathbf{e}) & = & \mathsf{rd_s}(\mathbf{e}) \\
\mathsf{nv_s}(\mathbf{x} \leftarrow \mathtt{query}(\boldsymbol{\psi})) & = & \{\} \\
\mathsf{nv_s}(\mathtt{while}\ \mathbf{b}, \mathbf{n}, [\bar{\mathbf{x}}, \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}]\ \mathtt{do}\ \mathbf{c}) & = & \mathsf{rd_s}(\mathbf{b}) \cup \mathsf{nv_s}(\mathbf{c})[\bar{\mathbf{x_1}}/\bar{\mathbf{x}}] \\
\mathsf{nv_s}(\mathtt{if}\ (\mathbf{b}, [\bar{\mathbf{x}}, \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}], [\bar{\mathbf{y}}, \bar{\mathbf{y_1}}, \bar{\mathbf{y_2}}], [\bar{\mathbf{z}}, \bar{\mathbf{z_1}}, \bar{\mathbf{z_2}}], \mathbf{c_1}, \mathbf{c_2})) & = & \mathsf{rd_s}(\mathbf{b}) \cup \mathsf{nv_s}(\mathbf{c_1}) \cup \mathsf{nv_s}(\mathbf{c_2}) \\
\mathsf{nv_s}(\mathbf{c_1}; \mathbf{c_2}) & = & \mathsf{nv_s}(\mathbf{c_1}) \cup (\mathsf{nv_s}(\mathbf{c_2}) - \mathsf{wr_s}(\mathbf{c_1}))
\end{array}
$$

The Lemma 10 and 11 proved the preserving properties for variables and values during the transformation.

Lemma 10 (Variable Preserving)**.** *If* $\Sigma; \delta; c \hookrightarrow \mathbf{c}; \delta'; \Sigma'$, $\mathsf{nv_s}(\mathbf{c}) = \delta(nv(c))$.

Proof. By induction on the transformation.

- **case: Case** $$\dfrac{\begin{array}{c} \delta; b \hookrightarrow \mathbf{b} \qquad \delta; c_1 \hookrightarrow \mathbf{c_1}; \delta_1 \qquad \delta; c_2 \hookrightarrow \mathbf{c_2}; \delta_2 \\ [\bar{x}, \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}] = \delta_1 \bowtie \delta_2 \qquad [\bar{y}, \bar{\mathbf{y_1}}, \bar{\mathbf{y_2}}] = \delta \bowtie \delta_1 / \bar{x} \qquad [\bar{z}, \bar{\mathbf{z_1}}, \bar{\mathbf{z_2}}] = \delta \bowtie \delta_2 / \bar{x} \\ \delta' = \delta[\bar{x} \mapsto \bar{\mathbf{x}}'] \qquad \bar{\mathbf{x}}', \bar{\mathbf{y}}', \bar{\mathbf{z}}'\ fresh \end{array}}{\delta; [\,\mathtt{if}\ (b, c_1, c_2)]^l \hookrightarrow [\,\mathtt{if}\ (\mathbf{b}, [\bar{\mathbf{x}}', \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}], [\bar{\mathbf{y}}', \bar{\mathbf{y_1}}, \bar{\mathbf{y_2}}], [\bar{\mathbf{z}}', \bar{\mathbf{z_1}}, \bar{\mathbf{z_2}}], \mathbf{c_1}, \mathbf{c_2})]^l; \delta'}\ \textbf{S-IF}$$
  From the definition of $\mathsf{nv_s}([\,\mathtt{if}\ (\mathbf{b}, [\bar{\mathbf{x}}', \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}], \mathbf{c_1}, \mathbf{c_2})]^l) = \mathsf{rd_s}(\mathbf{b}) \cup \mathsf{nv_s}(\mathbf{c_1}) \cup \mathsf{nv_s}(\mathbf{c_2})$. We want to show:
  $$\mathsf{rd_s}(\mathbf{b})) \cup \mathsf{nv_s}(\mathbf{c_1}) \cup \mathsf{nv_s}(\mathbf{c_2}) = \delta(\mathsf{rd}(b)) \cup \delta(\mathsf{FV}(c_1)) \cup \delta(\mathsf{FV}(c_2))$$
  By induction hypothosis on the second and third premise, we know that : $\mathsf{nv_s}(\mathbf{c_1}) = \delta(\mathsf{FV}(c_1))$ and $\mathsf{nv_s}(\mathbf{c_2}) = \delta(\mathsf{FV}(c_2))$. We still need to show that:
  $$\mathsf{rd_s}(\mathbf{b}) = \delta(\mathsf{rd}(b))$$
  From the first premise, we know that $\mathsf{rd}(b) \subseteq \mathtt{dom}(\delta)$. This is goal is proved by the rule **S-VAR** on all the variables in b.

- **case: Case** $$\dfrac{\begin{array}{c} \Sigma; \delta; c \hookrightarrow \mathbf{c_1}; \delta_1; \Sigma_1 \qquad [\bar{x}, \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}] = \delta \bowtie \delta_1 \\ \bar{\mathbf{x}}'\ fresh(\Sigma_1) \qquad \delta' = \delta[\bar{x} \mapsto \bar{\mathbf{x}}'] \qquad \delta'; b \hookrightarrow \mathbf{b} \qquad \mathbf{c}' = \mathbf{c_1}[\bar{\mathbf{x}}'/\bar{\mathbf{x_1}}] \end{array}}{\Sigma; \delta; \mathtt{while}\ [b]^l\ \mathtt{do}\ c \hookrightarrow \mathtt{while}\ [\mathbf{b}]^{\mathbf{l}}, \mathbf{0}, [\bar{\mathbf{x}}', \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}]\ \mathtt{do}\ \mathbf{c}; \delta'; \Sigma_1 \cup \{\bar{\mathbf{x}}'\}}\ \textbf{S-WHILE}$$
  Unfolding the definition, we need to show:
  $$\mathsf{rd_s}(\mathbf{b}) \cup \mathsf{nv_s}(\mathbf{c}')[\bar{\mathbf{x_1}}/\bar{\mathbf{x}}] = \delta(\mathsf{rd}(b)) \cup \delta(\mathsf{FV}(c))$$
  We can similarly show that $\mathsf{rd_s}(\mathbf{b}) = \delta(\mathsf{rd}(b))$ as in the if case. We still need to show that:
  $$\mathsf{nv_s}(\mathbf{c_1}[\bar{\mathbf{x}}'/\bar{\mathbf{x_1}}])[\bar{\mathbf{x_1}}/\bar{\mathbf{x}}'] = \delta(\mathsf{FV}(c))$$
  It is proved by induction hypothesis on $\Sigma; \delta; c \hookrightarrow \mathbf{c_1}; \delta_1; \Sigma_1$.

- **case: Case** $\dfrac{\Sigma;\delta;c_1 \hookrightarrow \mathbf{c_1};\delta_1;\Sigma_1 \qquad \Sigma_1;\delta_1;c_2 \hookrightarrow \mathbf{c_2};\delta';\Sigma'}{\Sigma;\delta;c_1;c_2 \hookrightarrow \mathbf{c_1};\mathbf{c_2}\ ;\delta';\Sigma'}$ **S-SEQ**

  To show:
  $$\mathsf{nv_s}(\mathbf{c_1}) \cup (\mathsf{nv_s}(\mathbf{c_2}) - \mathsf{wr_s}(\mathbf{c_1})) = \delta(\mathsf{FV}(c_1)) \cup \delta(\mathsf{FV}(c_2) - \mathsf{wr}(c_1))$$

  By induction hypothesis on the first premise, we know that : $\mathsf{nv_s}(\mathbf{c_1}) = \delta(\mathsf{FV}(c_1))$, still to show:
  $$(\mathsf{nv_s}(\mathbf{c_2}) - \mathsf{wr_s}(\mathbf{c_1})) = \delta(\mathsf{FV}(c_2) - \mathsf{wr}(c_1))$$

  We know that $\delta_1 = \delta[\mathsf{wr}(c_1) \mapsto \mathsf{wr_s}(\mathbf{c_1})]$, so by induction hypothesis, we know: $\mathsf{nv_s}(\mathbf{c_2}) = \delta[\mathsf{wr}(c_1) \mapsto \mathsf{wr_s}(\mathbf{c_1})](\mathsf{FV}(c_2)) = \delta(\mathsf{FV}(c_2)) \cup \mathsf{wr_s}(\mathbf{c_1}) - \delta(\mathsf{wr}(c_1))$.

  This case is proved.

□

We first define a good memory in the `While` language m or in the ssa language $\mathbf{m}$ with respect to a translation environment δ, denoted as $m \vDash \delta$ and $\mathbf{m} \vDash \delta$ respectively.

Definition 30 (Well Defined Memory). .

1. $m \vDash \delta \triangleq \forall x \in \mathtt{dom}(\delta), \exists v, (x, v) \in m$.

2. $\mathbf{m} \vDash_{ssa} \delta \triangleq \forall \mathbf{x} \in \mathtt{codom}(\delta), \exists v, (\mathbf{x}, v) \in \mathbf{m}$.

The part declared in the translation environment δ in a ssa memory $\mathbf{m}$ can be reverted to corresponding part of the memory m with an inverse of δ as follows.

Definition 31 (Inverse of Transformed memory). $m = \delta^{-1}(\mathbf{m}) \triangleq \forall x \in \mathtt{dom}(\delta), (\delta(x), m(x)) \in \mathbf{m}$.

Lemma 11 (Value Preserving). . *Given* $\delta; e \hookrightarrow \mathbf{e}$, $\forall m. m \vDash \delta. \forall \mathbf{m}, \mathbf{m} \vDash_{ssa} \delta \wedge m = \delta^{-1}(\mathbf{m})$, *then* $\langle m, e\rangle \to v$ *and* $\langle \mathbf{m}, \mathbf{e}\rangle \to v$.

Theorem 3.2 (Soundness of transformation). *Given* $\Sigma;\delta;c \hookrightarrow \mathbf{c};\delta';\Sigma'$, $\forall m. m \vDash \delta. \forall \mathbf{m}, \mathbf{m} \vDash_{ssa} \delta \wedge m = \delta^{-1}(\mathbf{m})$, *if there exist an execution of c in the while language, starting with a trace t and loop maps w,* $\langle m, c, t, w\rangle \to^* \langle m', \mathtt{skip}, t', w'\rangle$, *then there also exists a corresponding execution of* $\mathbf{c}$ *in the ssa language so that* $\langle \mathbf{m}, \mathbf{c}, t, w\rangle \to^* \langle \mathbf{m}', \mathtt{skip}, t', w'\rangle$ *and* $m' = \delta'^{-1}(\mathbf{m}')$.

Proof. We assume that q is the same when transformed to $\mathbf{q}$, as the primitive in both languages. And the value remains the same during the transformation. It is proved by induction on the transformation rules.

- **case: Case** $\dfrac{\Sigma;\delta;c_1 \hookrightarrow \mathbf{c_1};\delta_1;\Sigma_1 \qquad \Sigma_1;\delta_1;c_2 \hookrightarrow \mathbf{c_2};\delta';\Sigma'}{\Sigma;\delta;c_1;c_2 \hookrightarrow \mathbf{c_1};\mathbf{c_2}\ ;\delta';\Sigma'}$ **S-SEQ**

  We choose an arbitrary memory m so that $m \vDash \delta$, we choose a trace t and a loop maps w.
  $$\frac{\langle m, c_1, t, w\rangle \to^* \langle m_1, \mathtt{skip}, t_1, w_1\rangle \qquad \langle m_1, c_2, t_1, w_1\rangle \to^* \langle m', \mathtt{skip}, t', w'\rangle}{\langle m, c_1; c_2, t, w\rangle \to^* \langle m', \mathtt{skip}, t', w'\rangle}$$

  We choose the transformed memory $\mathbf{m}$ so that $m = \delta^{-1}(\mathbf{m})$.

  To show: $\langle \mathbf{m}, \mathbf{c_1}; \mathbf{c_2}, t, w\rangle \to^* \langle \mathbf{m}', \mathtt{skip}, t'.w'\rangle$ and $m' = \delta'^{-1}(\mathbf{m}')$.

By induction hypothesis on the first premise, we have:

$$\langle \mathbf{m}, \mathbf{c_1}, t, w\rangle \to^* \langle \mathbf{m_1}, \texttt{skip}, t_1, w_1\rangle \land m_1 = \delta_1^{-1}(\mathbf{m_1})$$

By induction hypothesis on the second premise, using the conclusion $m_1 = \delta_1^{-1}(\mathbf{m_1})$. We have:

$$\langle \mathbf{m_1}, \mathbf{c_2}, t_1, w_1\rangle \to^* \langle \mathbf{m'}, \texttt{skip}, t', w'\rangle \land m' = \delta'^{-1}(\mathbf{m'})$$

So we know that

$$\frac{\langle \mathbf{m}, \mathbf{c_1}, t, w\rangle \to^* \langle \mathbf{m_1}, \texttt{skip}, t_1, w_1\rangle \qquad \langle \mathbf{m_1}, \mathbf{c_2}, t_1, w_1\rangle \to^* \langle \mathbf{m'}, \texttt{skip}, t', w'\rangle}{\langle \mathbf{m}, \mathbf{c_1}; \mathbf{c_2}, t, w\rangle \to^* \langle \mathbf{m'}, \texttt{skip}, t', w'\rangle}$$

- **case: Case** $\dfrac{\delta; e \hookrightarrow \mathbf{e} \qquad \delta' = \delta[x \mapsto \mathbf{x}] \qquad \mathbf{x}\ fresh(\Sigma) \qquad \Sigma' = \Sigma \cup \{x\}}{\Sigma; \delta; [x \leftarrow e]^l \hookrightarrow [\mathbf{x} \leftarrow \mathbf{e}]^l; \delta'; \Sigma'}$ **S-ASSN**

  We choose an arbitrary memory m so that $m \vDash \delta$, we choose a trace t and a loop maps w, we know that the resulting trace is still t from its evaluation rule **assn** when we suppose $m, e \to v$.

  $$\frac{}{\langle m, [x \leftarrow v]^l, t, w\rangle \to \langle m[v/x], [\texttt{skip}]^l, t, w\rangle}\ \textbf{assn}$$

  We choose the transformed memory **m** so that $m = \delta^{-1}(\mathbf{m})$.

  To show: $\langle \mathbf{m}, [\mathbf{x} \leftarrow \mathbf{e}]^l, t, w\rangle \to^* \langle \mathbf{m'}, \texttt{skip}, t, w\rangle$ and $m' = \delta'^{-1}(\mathbf{m'})$.

  From the rule **ssa-assn**, we assume $\mathbf{m}, \mathbf{e} \to \mathbf{v}$, we know that

  $$\frac{}{\langle \mathbf{m}, [\mathbf{x} \leftarrow \mathbf{v}]^{\mathbf{l}}, t, w\rangle \to \langle \mathbf{m}[\mathbf{x} \mapsto \mathbf{v}], [\texttt{skip}]^{\mathbf{l}}, t, w\rangle}\ \textbf{ssa-assn}$$

  We also know that $\delta' = \delta[x \mapsto \mathbf{x}]$ and $m = \delta^{-1}(\mathbf{m})$, $m' = m[v/x]$. We can show that $m[v/x] = \delta[x \mapsto \mathbf{x}]^{-1}(\mathbf{m}[\mathbf{x} \mapsto v])$.

- **case: Case** $\dfrac{\delta; q \hookrightarrow \mathbf{q} \qquad \delta; e \hookrightarrow \mathbf{e} \qquad \delta' = \delta[x \mapsto \mathbf{x}] \qquad \mathbf{x}\ fresh(\Sigma) \qquad \Sigma' = \Sigma \cup \{x\}}{\Sigma; \delta; [x \leftarrow \texttt{query}(\psi)]^l \hookrightarrow [\mathbf{x} \leftarrow \texttt{query}(\boldsymbol{\psi})]^l; \delta'}$ **S-QUERY**

  We choose an arbitrary memory m so that $m \vDash \delta$, we choose a trace t and a loop maps w, we know that when we suppose $\langle m, e\rangle \to v$.

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

  We choose the transformed memory **m** so that $m = \delta^{-1}(\mathbf{m})$.

  To show: $\langle \mathbf{m}, [\mathbf{x} \leftarrow \texttt{query}(\boldsymbol{\psi})]^l, t, w\rangle \to^* \langle \mathbf{m'}, \texttt{skip}, t, w\rangle$ and $m' = \delta'^{-1}(\mathbf{m'})$.

  From the rule **ssa-query**, we know that

  $$\frac{\texttt{query}(\mathbf{v})(\mathbf{D}) = \boldsymbol{\alpha}}{\langle \mathbf{m}, [\mathbf{x} \leftarrow \texttt{query}(\boldsymbol{\psi})]^{\mathbf{l}}, t, w\rangle \to \langle \mathbf{m}[\mathbf{x} \mapsto \mathbf{v}], \texttt{skip}, t +\!\!+ [(q^{(l,w)}, v)], w\rangle}\ \textbf{ssa-query}$$

  We also know that $\delta' = \delta[x \mapsto \mathbf{x}]$ and $m = \delta^{-1}(\mathbf{m})$, $m' = m[v/x]$. We can show that $m[v/x] = \delta[x \mapsto \mathbf{x}]^{-1}(\mathbf{m}[\mathbf{x} \mapsto v])$.

- **case: Case** $$\frac{\begin{array}{c}\delta; b \hookrightarrow \mathbf{b} \qquad \Sigma;\delta;c_1 \hookrightarrow \mathbf{c_1};\delta_1;\Sigma_1 \qquad \Sigma_1;\delta;c_2 \hookrightarrow \mathbf{c_2};\delta_2;\Sigma_2 \\ [\bar{x},\bar{\mathbf{x_1}},\bar{\mathbf{x_2}}] = \delta_1 \bowtie \delta_2 \qquad [\bar{y},\bar{\mathbf{y_1}},\bar{\mathbf{y_2}}] = \delta \bowtie \delta_1 / \bar{x} \qquad [\bar{z},\bar{\mathbf{z_1}},\bar{\mathbf{z_2}}] = \delta \bowtie \delta_2 / \bar{x} \\ \delta' = \delta[\bar{x} \mapsto \bar{\mathbf{x}}'][\bar{y} \mapsto \bar{\mathbf{y}}'][\bar{z} \mapsto \bar{\mathbf{z}}'] \qquad \bar{\mathbf{x}}',\bar{\mathbf{y}}',\bar{\mathbf{z}}'\ fresh(\Sigma_2) \qquad \Sigma' = \Sigma_2 \cup \{\bar{\mathbf{x}}',\bar{\mathbf{y}}',\bar{\mathbf{z}}'\}\end{array}}{\Sigma;\delta;[\,\mathtt{if}\ (b,c_1,c_2)]^l \hookrightarrow [\,\mathtt{if}\ (\mathbf{b},[\bar{\mathbf{x}}',\bar{\mathbf{x_1}},\bar{\mathbf{x_2}}],[\bar{\mathbf{y}}',\bar{\mathbf{y_1}},\bar{\mathbf{y_2}}],[\bar{\mathbf{z}}',\bar{\mathbf{z_1}},\bar{\mathbf{z_2}}],\mathbf{c_1},\mathbf{c_2})]^l;\delta';\Sigma'} \textbf{ S-IF}$$
We choose an arbitrary memory m so that $m \vDash \delta$, we choose a trace t and a loop maps w. There are two possible evaluation rules depending on the the condition b, we choose the case when $b = \mathtt{true}$, we know there is an execution in ssa language so that $\mathbf{b} = \mathtt{true}$, we use the rule **if-t**.

$$\frac{}{\langle m, [\,\mathtt{if}\ (\mathtt{true}, c_1, c_2)]^l, t, w\rangle \to \langle m, c_1, t, w\rangle \to^* \langle m', \mathtt{skip}, t', w'\rangle}$$

We choose the transformed memory $\mathbf{m}$ so that $m = \delta^{-1}(\mathbf{m})$.

To show: $\langle \mathbf{m}, [\,\mathtt{if}\ (\mathtt{true}, [\bar{\mathbf{x}}',\bar{\mathbf{x_1}},\bar{\mathbf{x_2}}],[\bar{\mathbf{y}}',\bar{\mathbf{y_1}},\bar{\mathbf{y_2}}],[\bar{\mathbf{z}}',\bar{\mathbf{z_1}},\bar{\mathbf{z_2}}], c_1, c_2)]^l, t, w\rangle \to^* \langle \mathbf{m}', \mathtt{skip}, t', w'\rangle$ and $m' = \delta'^{-1}(\mathbf{m}')$.

We use the corresponding rule **SSA-IF-T**.

$$\frac{}{\begin{array}{c}\langle \mathbf{m}, [\,\mathtt{if}\ (\mathtt{true}, [\bar{\mathbf{x}}',\bar{\mathbf{x_1}},\bar{\mathbf{x_2}}],[\bar{\mathbf{y}}',\bar{\mathbf{y_1}},\bar{\mathbf{y_2}}],[\bar{\mathbf{z}}',\bar{\mathbf{z_1}},\bar{\mathbf{z_2}}], \mathbf{c_1}, \mathbf{c_2})]^{\mathbf{l}}, t, w\rangle \to \\ \langle \mathbf{m}, \mathbf{c_1}; \mathtt{ifvar}(\bar{\mathbf{x}}',\bar{\mathbf{x_1}}); \mathtt{ifvar}(\bar{\mathbf{y}}',\bar{\mathbf{y_2}}); \mathtt{ifvar}(\bar{\mathbf{z}}',\bar{\mathbf{z_1}}), t, w\rangle\end{array}} \textbf{ ssa-if-t}$$

By induction hypothesis on $\Sigma;\delta;c_1 \hookrightarrow \mathbf{c_1};\delta_1;\Sigma_1$, and we know that $\langle m, c_1, t, w\rangle \to^* \langle m', \mathtt{skip}, t', w'\rangle$, from our assumption that $m = \delta^{-1}(\mathbf{m})$, we know that

$$\langle \mathbf{m}, \mathbf{c_1}, t, w\rangle \to^* \langle \mathbf{m_1}, \mathtt{skip}, t', w'\rangle \land m' = \delta_1^{-1}(\mathbf{m_1})$$

and we then have:

$$\frac{\langle \mathbf{m}, \mathbf{c_1}, t, w\rangle \to^* \langle \mathbf{m_1}, \mathtt{skip}, t', w'\rangle}{\langle \mathbf{m}, \mathbf{c_1}; \mathtt{ifvar}(\bar{\mathbf{x}}',\bar{\mathbf{x_1}}); \mathtt{ifvar}(\bar{\mathbf{y}}',\bar{\mathbf{y_1}}); \mathtt{ifvar}(\bar{\mathbf{z}}',\bar{\mathbf{z_1}}), t, w\rangle \to^* \langle \mathbf{m_1}[\bar{\mathbf{x}}' \mapsto \mathbf{m_1}(\bar{\mathbf{x_1}}), \bar{\mathbf{y}}' \mapsto \mathbf{m_1}(\bar{\mathbf{y_2}}), \bar{\mathbf{z}}' \mapsto \mathbf{m_1}(\bar{\mathbf{z_1}})], \mathtt{skip}, t',}$$

Now, we want to show that $m' = \delta[\bar{x} \mapsto \bar{\mathbf{x}}', \bar{y} \mapsto \bar{\mathbf{y}}', \bar{z} \mapsto \bar{\mathbf{z}}']^{-1}(\mathbf{m_1}[\bar{\mathbf{x}}' \mapsto \mathbf{m_1}(\bar{\mathbf{x_1}}), \bar{\mathbf{y}}' \mapsto \mathbf{m_1}(\bar{\mathbf{y_2}}), \bar{\mathbf{z}}' \mapsto \mathbf{m_1}(\bar{\mathbf{z_1}})])$.

Unfold the definition, we want to show that

$\forall x \in (\mathtt{dom}(\delta) \cup \bar{x} \cup \bar{y} \cup \bar{z}), (\delta[\bar{x} \mapsto \bar{\mathbf{x}}', \bar{y} \mapsto \bar{\mathbf{y}}', \bar{z} \mapsto \bar{\mathbf{z}}'](x), m'(x)) \in \mathbf{m_1}[\bar{\mathbf{x}}' \mapsto \mathbf{m_1}(\bar{\mathbf{x_1}}), \bar{\mathbf{y}}' \mapsto \mathbf{m_1}(\bar{\mathbf{y_2}}), \bar{\mathbf{z}}' \mapsto \mathbf{m_1}(\bar{\mathbf{z_1}})]$.

1. For variable x in $\bar{x}$, we can find a corresponding ssa variable $\mathbf{x} \in \bar{\mathbf{x}}'$, so that $(\mathbf{x}, m'(x)) \in \mathbf{m_1}[\bar{\mathbf{x}}' \mapsto \mathbf{m_1}(\bar{\mathbf{x_1}})]$. It is because we know $[x \mapsto \mathbf{x_1}]$ for certain $\mathbf{x_1} \in \bar{\mathbf{x_1}}$ in δ_1, then by unfolding $m' = \delta_1^{-1}(\mathbf{m_1})$ and $\bar{\mathbf{x_1}} \in \mathtt{codom}(\delta_1)$, we know $(\mathbf{x_1}, m'(x)) \in \mathbf{m_1}$ so that $m'(x) = \mathbf{m_1}(\mathbf{x_1})$.
2. For variable $y \in \bar{y}$, we know that $y \in \mathtt{dom}(\delta_1)$, then $[y \mapsto \mathbf{y_2}]$ for certain $\mathbf{y_2} \in \bar{\mathbf{y_2}}$ in δ_1. So we know that $(\delta_1(y), m'(y)) \in \mathbf{m_1}$, and then $m'(y) = \mathbf{m_1}(\mathbf{y_2})$. We can show $(\mathbf{y}, m'(y)) \in \mathbf{m_1}[\bar{\mathbf{y}}' \mapsto \mathbf{m_1}(\bar{\mathbf{y_2}})]$.
3. For variable $z \in \bar{z}$, we know that $z \notin \mathtt{dom}(\delta_1)$ by the definition (otherwise z will appear in $\bar{x}$), then $[z \mapsto \mathbf{z_1}]$ for certain $\mathbf{z_1} \in \bar{\mathbf{z_1}}$ in δ. We know $(\delta(z), m(z)) \in \mathbf{m}$ from our assumption, so we have $m(z) = \mathbf{m}(\mathbf{z_1})$. Because z is not modified in c_1, so that $m(z) = m'(z)$. Also $\mathbf{m}$ will not shrink during execution and $\mathbf{z_1}$ will not be written in $\mathbf{c_1}$, so $(\mathbf{z_1}, m'(z)) \in \mathbf{m_1}$. Then we can show that $(\mathbf{z}, m'(z)) \in \mathbf{m_1}[\bar{\mathbf{z}}' \mapsto \mathbf{m_1}(\bar{\mathbf{z_1}})]$.

4. For variable $k \in \mathtt{dom}(\delta) - \bar{x} - \bar{y} - \bar{z}$. From our assumption $m = \delta^{-1}(\mathbf{m})$, we can show $(\delta(k), m(k)) \in \mathbf{m}$. We know that k is not written in either branch from our definition, so $(\delta(k), m'(k)) \in \mathbf{m_1}$.

- **case: Case** $$\frac{\Sigma;\delta;c \hookrightarrow \mathbf{c_1};\delta_1;\Sigma_1 \qquad [\bar{x},\bar{\mathbf{x_1}},\bar{\mathbf{x_2}}] = \delta \bowtie \delta_1 \qquad \bar{\mathbf{x}}'\ fresh(\Sigma_1) \qquad \delta' = \delta[\bar{x} \mapsto \bar{\mathbf{x}}'] \qquad \delta';b \hookrightarrow \mathbf{b} \qquad \mathbf{c}' = \mathbf{c_1}[\bar{\mathbf{x}}'/\bar{\mathbf{x_1}}]}{\Sigma;\delta;\ \mathtt{while}\ [b]^l\ \mathtt{do}\ c \hookrightarrow \mathtt{while}\ [\mathbf{b}]^{\mathbf{l}},\mathbf{0},[\bar{\mathbf{x}}',\bar{\mathbf{x_1}},\bar{\mathbf{x_2}}]\ \mathtt{do}\ \mathbf{c};\delta';\Sigma_1 \cup \{\bar{\mathbf{x}}'\}}\ \textbf{S-WHILE}$$

  We choose an arbitrary memory m so that $m \vDash \delta$, we choose a trace t and a loop maps w. Suppose $\langle m, a\rangle \to v_N$. There are two cases, when $v_N = 0$, the loop body is not executed so we can easily show that the trace is not modified. When the while loop is still running ($v_N > 0$), we have the following evaluation in the while language:

$$\frac{}{\langle m,\ \mathtt{while}\ [b]^l\ \mathtt{do}\,[c]^{l+1}, t, w\rangle \to \langle m, c;\ \mathtt{if}\ _w(b, c;\ \mathtt{while}\ [b]^l\ \mathtt{do}\,[c]^{l+1}, \mathtt{skip}), t, w\rangle}\ \textbf{while-b}$$

  which follows by the following evaluation:

$$\frac{m, b \to b'}{\langle m,\ \mathtt{if}\ _w(b, c, \mathtt{skip}), t, w\rangle \to \langle m,\ \mathtt{if}\ _w(b', c, \mathtt{skip}), t, w\rangle}\ \textbf{ifw-b}$$

  In the corresponding ssa-form language, we have the corresponding evaluation in the same way by assuming $m = \delta^{-1}(\mathbf{m})$.

$$\frac{n = 0 \to i = 1 \qquad n > 0 \to i = 2}{\begin{array}{c}\langle \mathbf{m},\ \mathtt{while}\ [\mathbf{b}]^{\mathbf{l}},\mathbf{n},[\bar{\mathbf{x}}',\bar{\mathbf{x_1}},\bar{\mathbf{x_2}}]\ \mathtt{do}\ \mathbf{c}, t, w\rangle \to \\ \langle \mathbf{m},\ \mathtt{if}\ _w(\mathbf{b}[\bar{\mathbf{x_i}}/\bar{\mathbf{x}}'],[\bar{\mathbf{x}}',\bar{\mathbf{x_1}},\bar{\mathbf{x_2}}],\mathbf{n},\mathbf{c}[\bar{\mathbf{x_i}}/\bar{\mathbf{x}}'];\ \mathtt{while}\ [\mathbf{b}]^{\mathbf{l}},\mathbf{n+1},[\bar{\mathbf{x}}',\bar{\mathbf{x_1}},\bar{\mathbf{x_2}}]\ \mathtt{do}\ \mathbf{c}, \mathtt{skip}), t, w\rangle\end{array}}\ \textbf{ssa-while-b}$$

  This evaluation is followed by the following evaluation:

$$\frac{\mathbf{m},\mathbf{b} \to \mathbf{b}'}{\langle \mathbf{m},\ \mathtt{if}_{\mathbf{w}}(\mathbf{b},[\bar{\mathbf{x}}',\bar{\mathbf{x_1}},\bar{\mathbf{x_2}}],\mathbf{n},\mathbf{c_1},\mathbf{c_2}), t, w\rangle \to \langle \mathbf{m},\ \mathtt{if}_{\mathbf{w}}(\mathbf{b}',[\bar{\mathbf{x}}',\bar{\mathbf{x_1}},\bar{\mathbf{x_2}}],\mathbf{n},\mathbf{c_1},\mathbf{c_2}), t, w\rangle}\ \textbf{ssa-ifw-b}$$

  Depending on if the counter n is equal to 0 or not, there are two possible execution paths (the variables $\bar{\mathbf{x}}$ is replaced by the $\bar{\mathbf{x_1}}$ or $\bar{\mathbf{x_2}}$). We start from the first iteration (when $n = 0$) when $v_N > 0$. By induction hypothsis on the premise $\Sigma;\delta;c \hookrightarrow \mathbf{c_1};\delta_1;\Sigma_1$, we know that

$$\langle \mathbf{m},\mathbf{c}'[\bar{\mathbf{x_1}}/\bar{\mathbf{x}}'], t, (w+l)\rangle \to^* \langle \mathbf{m}',\mathtt{skip}, t_i', w'\rangle \wedge m' = \delta_1^{-1}(\mathbf{m}')$$

  Hence we can conclude that:

$$\frac{\langle \mathbf{m},\mathbf{c}'[\bar{\mathbf{x_1}}/\bar{\mathbf{x}}'], t, (w+l)\rangle \to^* \langle \mathbf{m}',\mathtt{skip}, t_1', w'\rangle}{\begin{array}{c}\langle \mathbf{m},\mathbf{c}'[\bar{\mathbf{x_1}}/\bar{\mathbf{x}}'];[\mathtt{loop}\ (\mathbf{v_N}-\mathbf{1}),\mathbf{n+1},[\bar{\mathbf{x}}',\bar{\mathbf{x_1}},\bar{\mathbf{x_2}}]\ \mathtt{do}\ \mathbf{c}']^{\mathbf{1}}, t, (w+l)\rangle \to^* \\ \langle \mathbf{m}',[\mathtt{loop}\ (\mathbf{v_N}-\mathbf{1}),\mathbf{n+1},[\bar{\mathbf{x}}',\bar{\mathbf{x_1}},\bar{\mathbf{x_2}}]\ \mathtt{do}\ \mathbf{c}']^{\mathbf{1}}, t_1', w'\rangle\end{array}}$$

  Then there are two cases,

  1. when guard in the `if` $_w$ expression evaluates to `false`, the while loop terminates and exits. The execution in the while language is defined in the evaluation rule **ifw-false** as follows.

$$\frac{}{\begin{array}{c}\langle \mathbf{m},\ \mathtt{if}_{\mathbf{w}}(\mathtt{false},[\bar{\mathbf{x}}',\bar{\mathbf{x_1}},\bar{\mathbf{x_2}}],\mathbf{n},\mathbf{c};\ \mathtt{while}\ [\mathbf{b}]^{\mathbf{l}}\ \mathtt{do}\ \mathbf{c},\mathtt{skip})), t, w\rangle \\ \to \langle \mathbf{m},\mathtt{skip};\mathtt{ifvar}(\bar{\mathbf{x}}',\bar{\mathbf{x_i}}), t, (w-l)\rangle\end{array}}\ \textbf{ifw-false}$$

The corresponding ssa-form evaluation as follows:

$$\frac{n = 0 \to i = 1 \qquad n > 0 \to i = 2}{\begin{array}{c}\langle \mathbf{m}, \texttt{if}_{\mathbf{w}}(\texttt{false}, [\bar{\mathbf{x}}', \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}], \mathbf{n}, \mathbf{c}; \texttt{while}\ [\mathbf{b}]^{\mathbf{l}}, \mathbf{n}, [\bar{\mathbf{x}}', \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}]\ \texttt{do}\ \mathbf{c}, \texttt{skip})), t, w\rangle \\ \to \langle \mathbf{m}, \texttt{skip}; \texttt{ifvar}(\bar{\mathbf{x}'}, \bar{\mathbf{x_i}}), t, (w - l)\rangle\end{array}} \textbf{ ssa-ifw-false}$$

We can see that both traces are not changed during the exit of the while. We need to show that $m' = \delta^{-1}(\mathbf{m}'[\bar{\mathbf{x}} \mapsto \mathbf{m}'(\bar{\mathbf{x_2}})])$. We know that $[\bar{x} \mapsto \bar{x_2}]$ in δ_1 from the definition, so we can show that for any variable $\mathbf{x_2} \in \bar{x_2}$, $(\mathbf{x_2}, m'(x)) \in \mathbf{m}'$. For variables $x \in \mathtt{dom}(\delta) - \bar{x}$, the variable is not modified during the execution of c so that we know $m(x) = m'(x)$, and then we can show that $(\delta(x), m'(x)) \in \mathbf{m}'$ because $\delta(x)$ is not written in $\mathbf{c}'[\bar{\mathbf{x_1}}/\bar{\mathbf{x}}']$.

2. when guard in the $\texttt{if}_{w}$ expression evaluates to $\texttt{true}$, the while terminates and exits. The execution in the while language is defined in the evaluation rule **ifw-true**. We want to show that : assuming in the $i - th$ ($i < \mathbf{n}$) iteration, starting with t_i and w_i and $m_i = \delta_1^{-1}(\mathbf{m_i})$, this command is evaluated according to the while language operation semantics as $\langle m, \texttt{if}_{w}(\texttt{true}, c; \texttt{while}\ [b]^l\ \texttt{do}\ c, , \texttt{skip}), t, w\rangle \to^* \langle m, ct, (w + l)\rangle$. Then the corresponding ssa form evaluation as follows :

$$\begin{array}{l}\langle \mathbf{m}, \texttt{if}_{\mathbf{w}}(\texttt{true}, [\bar{\mathbf{x}}', \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}], \mathbf{n}, \mathbf{c}; \texttt{while}\ [\mathbf{b}]^{\mathbf{l}}, \mathbf{n}, [\bar{\mathbf{x}}', \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}]\ \texttt{do}\ \mathbf{c}, \texttt{skip}), t, w\rangle \\ \to \langle \mathbf{m}, \texttt{if}_{\mathbf{w}}(\texttt{true}, [\bar{\mathbf{x}}', \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}], \mathbf{n}, \mathbf{c}; \texttt{while}\ [\mathbf{b}]^{\mathbf{l}}, \mathbf{n}, [\bar{\mathbf{x}}', \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}]\ \texttt{do}\ \mathbf{c}, t, (w + l)\rangle\end{array}$$

and $m_i = \delta^{-1}(\mathbf{m_i})$. We then have the evaluation in the while language:

$$\frac{}{\langle m, \texttt{if}_{w}(b, c; \texttt{while}\ [b]^l\ \texttt{do}\ c, \texttt{skip}), t, w\rangle \to \langle m, c; \texttt{while}\ [b]^l\ \texttt{do}\ c, t, (w + l)\rangle} \textbf{ ifw-true}$$

We then have the following evaluation:

$$\frac{}{\begin{array}{l}\langle \mathbf{m}, \texttt{if}_{\mathbf{w}}(\texttt{true}, [\bar{\mathbf{x}}', \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}], \mathbf{n}, \mathbf{c}; \texttt{while}\ [\mathbf{b}]^{\mathbf{l}}, \mathbf{n}, [\bar{\mathbf{x}}', \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}]\ \texttt{do}\ \mathbf{c}, \texttt{skip}), t, w\rangle \\ \to \langle \mathbf{m}, \texttt{if}_{\mathbf{w}}(\texttt{true}, [\bar{\mathbf{x}}', \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}], \mathbf{n}, \mathbf{c}; \texttt{while}\ [\mathbf{b}]^{\mathbf{l}}, \mathbf{n}, [\bar{\mathbf{x}}', \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}]\ \texttt{do}\ \mathbf{c}, t, (w + l)\rangle\end{array}} \textbf{ ssa-ifw-true}$$

By induction hypothsis on the premise $\Sigma; \delta_1; c \hookrightarrow \mathbf{c_2}; \delta_1; \Sigma_1$, we know that

$$\langle \mathbf{m_i}, \mathbf{c}'[\bar{\mathbf{x_2}}/\bar{\mathbf{x}}'], t_i, (w_i + l)\rangle \to^* \langle \mathbf{m_{i+1}}, \texttt{skip}, t_{i+1}, w_{i+1}\rangle \wedge m_{i+1} = \delta_1^{-1}(\mathbf{m_{i+1}})$$

Hence we can conclude that:

$$\frac{\langle \mathbf{m_i}, \mathbf{c}'[\bar{\mathbf{x_2}}/\bar{\mathbf{x}}'], t_i, (w_i + l)\rangle \to^* \langle \mathbf{m_{i+1}}, \texttt{skip}, t_{i+1}, w_{i+1}\rangle}{\begin{array}{c}\langle \mathbf{m_i}, \mathbf{c}'[\bar{\mathbf{x_2}}/\bar{\mathbf{x}}']; [\texttt{loop}\ (\mathbf{v_N} - \mathbf{i} - \mathbf{1}), \mathbf{n} + \mathbf{1}, [\bar{\mathbf{x}}', \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}]\ \texttt{do}\ \mathbf{c}']^{\mathbf{l}}, t_i, (w_i + l)\rangle \to^* \\ \langle \mathbf{m_{i+1}}, [\texttt{loop}\ (\mathbf{v_N} - \mathbf{i} - \mathbf{1}), \mathbf{n} + \mathbf{1}, [\bar{\mathbf{x}}', \bar{\mathbf{x_1}}, \bar{\mathbf{x_2}}]\ \texttt{do}\ \mathbf{c}']^{\mathbf{l}}, t_{i+1}, w_{i+1}\rangle\end{array}}$$

So we can show that before the exit of the loop after ($v_N = n$) iterations, we have $t_n = t_n$ and $m_n = \delta_1^{-1}(\mathbf{m_n})$.

This proof is similar when it comes to the exit as in case 1.

□

4 AdaptFun

There are four steps to get the adaptivity of a program **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 32 (Assigned Variables (`aVar`))**.** *Given a program* **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:

$$A_1 ++ A_2 \triangleq \begin{cases} A_2 & A_1 = [] \\ x :: (A_1' ++ A_2) & A_1 = x :: A_1' \end{cases} \tag{8}$$

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 `aVar`(**c**)) is as in the right hand side is shown as follows:

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

Lemma 12. *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 33 (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:*

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

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

$$\begin{array}{lcll} \mathtt{F}_1 \uplus \mathtt{F}_2(i) & := & \left\{ \begin{array}{ll} k & k = \max\{\mathtt{F}_1(i), \mathtt{F}_2(i)\} \land |\mathtt{F}_1| = |\mathtt{F}_2| \\ 0 & o.w. \end{array} \right. & i = 1, \cdots, |\mathtt{F}_1| \\ \mathtt{F} \uplus n(i) & := & \max\{\mathtt{F}(i), n\} & i = 1, \ldots, |\mathtt{F}| \\ [n]^{k}(i) & := & n & i = 1, \ldots, k \ \land\ |[n]^{k}| = k \end{array} \quad (9)$$

[[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 34. 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 34 (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:*

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

]]]

Definition 35 (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.* $\forall \mathbf{x}, \mathbf{y} \in \mathtt{aVar}.\ \mathbf{x} = \mathtt{aVar}(i), \mathbf{y} = \mathtt{aVar}(j)$*:*

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

Operations on the data flow matrices are defined as follows:

$$\mathtt{M}_1; \mathtt{M}_2 := \mathtt{M}_2 \cdot \mathtt{M}_1 + \mathtt{M}_1 + \mathtt{M}_2 \tag{10}$$

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 $\mathsf{R}_{(e,i)}$.

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 36 (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:*

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

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 37 (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:*

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

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:

$$\mathtt{LM}_1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \qquad \mathtt{RM}_{\mathbf{x_1}+\mathbf{1}} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$

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 5. It has the form: [JL: $\mathsf{VetxCol}(\mathtt{aVar}; \mathtt{F}; \mathbf{c}) \to (\mathtt{aVar}'; \mathtt{F}')$]. 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:

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

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

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

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

$$\frac{\mathsf{VetxCol}(\mathtt{aVar};[];\mathbf{c}) \to (\mathtt{aVar}';\mathtt{F}') \qquad \mathtt{aVar}'' = \mathtt{aVar}'++[\bar{\mathbf{x}}] \qquad \mathtt{F}'' = \mathtt{F}++(\mathtt{F}' \uplus 1)++([1]^{\mathtt{len}([\bar{\mathbf{x}}])})}{\mathsf{VetxCol}(\mathtt{aVar};\mathtt{F};\ \mathtt{while}\ [\mathbf{b}]^{l},\mathbf{n},[\bar{\mathbf{x}},\bar{\mathbf{x}}_1,\bar{\mathbf{x}}_2]\ \mathtt{do}\ \mathbf{c}) \to (\mathtt{aVar}'';\mathtt{F}'')}\ \textbf{VetxCol-while}$$

]

Figure 5: The Algorithmic Rules of **VetxCol**

$\mathtt{F}'$ thorough the program **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 **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` **c** considers the newly generated variables by SSA transformation $\bar{\mathbf{x}}$ as well and the newly assigned variables in its body **c**.

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

Definition 38 (Valid Index (Remove?))**.** *Given an assigned variable list* $\mathtt{aVar}$, $\mathtt{aVar};\vDash(\mathbf{c},i_1,i_2)$ *iff* $\mathtt{aVar}' = \mathtt{aVar}[0,\ldots,i_1-1],\mathtt{aVar}';\mathbf{c} \to \mathtt{aVar}'' \wedge \mathtt{aVar}'' = \mathtt{aVar}[0,\ldots,i_2-1]$.

]

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

Definition 39 (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 40 (Valid Matrix)**.** *For a assigned variables* $\mathtt{aVar}$, $\mathtt{aVar} \vDash (\mathtt{M}, \mathtt{F})$ *iff the cardinality of* $\mathtt{aVar}$ *equals to the one of* $\mathtt{F}$, $|\mathtt{aVar}| = |\mathtt{F}|$ *and the matrix* $\mathtt{M}$ *is of size* $|\mathtt{F}| \times |\mathtt{F}|$.

[JL:

Definition 41 (Valid Matrix)**.** *Given a program* $\mathbf{c}$ *with its assigned variables* $\mathtt{aVar}$, $\mathtt{aVar} \vDash \mathtt{M}$ *iff the cardinality of* $\mathtt{M}$ *equals to the product of* $\mathtt{aVar}$*'s cardinality, i.e.,* $|\mathtt{M}| = |\mathtt{aVar}| \times |\mathtt{aVar}|$.

]

Reachability Bounds Given a program c with its assigned variables $\mathtt{aVar}$, we use the $\mathsf{RechBound}(\mathbf{x}, \mathbf{c})$ algorithm, from paper [2], to estimate the reachability bound for each variable $\mathbf{x} \in \mathtt{aVar}$. The input of $\mathsf{RechBound}$ 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, $\mathtt{aVar}_{\mathbf{c}2}$ and $\mathtt{aVar}_{\mathbf{c}3}$, data flow matrix $\mathtt{M}_{\mathbf{c}2}$ and $\mathtt{M}_{\mathbf{c}3}$ and variable flags $\mathtt{F}_{\mathbf{c}2}$ and $\mathtt{F}_{\mathbf{c}3}$ for program $\mathbf{c}2$ and $\mathbf{c}3$ are presented in the right hand side.

$$
\mathbf{c}2 \triangleq
\begin{array}{l}
[\mathbf{x}_1 \leftarrow \mathtt{query}(1)]^1; \\
[\mathbf{i}_1 \leftarrow 0]^2; \\
\mathtt{while}\ [\mathbf{i}_1 < 2]^3 \\
\ [\mathbf{x}_3,\mathbf{x}_1,\mathbf{x}_2],[\mathbf{i}_3,\mathbf{i}_1,\mathbf{i}_2]\ \mathtt{do} \\
\Big([\mathbf{y}_1 \leftarrow \mathtt{query}(2)]^4; \\
[\mathbf{x}_2 \leftarrow \mathbf{y}_1 + \mathbf{x}_3]^5; \\
[\mathbf{i}_2 \leftarrow \mathbf{1} + \mathbf{i}_3]^6\Big); \\
[\mathbf{z}_1 \leftarrow \mathbf{x}_3 + 2]^7
\end{array}
, \quad
\mathtt{aVar}_{\mathbf{c}2} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_3 \\ \mathbf{y}_1 \\ \mathbf{x}_2 \\ \mathbf{z}_1 \\ \mathbf{i}_1 \\ \mathbf{i}_2 \\ \mathbf{i}_3 \end{bmatrix}, \quad
\mathtt{M}_{\mathbf{c}2} = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 1 & 1 & 1
\end{bmatrix}, \quad
\mathtt{F}_{\mathbf{c}2} = \begin{bmatrix} 1 \\ 2 \\ 1 \\ 2 \\ 0 \\ 0 \\ 2 \\ 1 \end{bmatrix}
$$

$$
\mathbf{c}3 \triangleq
\begin{array}{l}
[\mathbf{x}_1 \leftarrow \mathtt{query}(1)]^1; \\
[\mathbf{i}_1 \leftarrow 1]^2; \\
\mathtt{while}\ [i < 0]^3, \\
\ [\mathbf{x}_3,\mathbf{x}_1,\mathbf{x}_2],[\mathbf{i}_3,\mathbf{i}_1,\mathbf{i}_2]\ \mathtt{do} \\
\Big([\mathbf{y}_1 \leftarrow \mathtt{query}(2)]^3; \\
[\mathbf{x}_2 \leftarrow \mathbf{y}_1 + \mathbf{x}_3]^5\Big); \\
[\mathbf{z}_1 \leftarrow \mathbf{x}_3 + 2]^6
\end{array}
, \quad
\mathtt{aVar}_{\mathbf{c}3} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{i}_1 \\ \mathbf{x}_3 \\ \mathbf{i}_3 \\ \mathbf{z}_1 \end{bmatrix}, \quad
\mathtt{M}_{\mathbf{c}3} = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0
\end{bmatrix}, \quad
\mathtt{F}_{\mathbf{c}3} = \begin{bmatrix} 1 \\ 0 \\ 2 \\ 2 \\ 0 \end{bmatrix}
$$

We can now look at the if statement.

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

4.3 Adaptivity Based on Program Analysis in AdaptFun

Definition 42 (Program-Based Dependency Graph)**.** .
Given a program **c** *with its assigned variables* $\mathtt{aVar}$ *of length* N*, s.t.,* $\Gamma \vdash_{\mathtt{M}_c, \mathtt{F}_c} \mathbf{c}$*, its program-based graph* $G(\mathbf{c}) = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{F})$ *is. defined as:*

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

Definition 43 (Finite Walk (k))**.** .
Given a labeled weighted graph $G = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{F})$*, 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* $1 \leq i < n$*.*
- *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.*
[JL: Length of this finite walk k *is the number of vertices in its vertex sequence, i.e.,* $\mathtt{len}(k) = n$*.]*

[JL: Given a labeled weighted graph $G = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{F})$, 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 44 (Length of Finite Walk w.r.t. Query ($\mathtt{len_q}$))**.** .
Given a labeled weighted graph $G = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{F})$ *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:*

$$
\mathtt{len_q}(k) = \mathtt{len}\big(v \mid v \in (v_1, \dots, v_n) \land \mathtt{F}(v) = 2\big)
$$

, where $\big(v \mid v \in (v_1, \dots, v_n) \land \mathtt{F}(v) = 2\big)$ *is a subsequence of k's vertex sequence.*

Given a program **c**, we generate its program-based graph $\mathbf{G_{prog}}(\mathbf{c}) = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{F})$. Then the adaptivity bound based on program analysis for **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 $\mathtt{W}(v)$.

It is formally defined in 45.

Definition 45 (Program-Based Adaptivity)**.** .
Given a program **c** *and its program-based graph* $\mathbf{G_{prog}}(\mathbf{c}) = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{F})$*, the program-based adaptivity for c is defined as*

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

4.4 [[Soundness of the AdaptFun]]

[JL:

Theorem 4.1 (Soundness of the AdaptFun). . *Given a program* **c**, *we have:*

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

] [JL:

Proof. Given a program **c**, we construct its program-based graph $\mathbf{G_{prog}}(\mathbf{c}) = (\mathtt{V}, \mathtt{E}, \mathtt{W}, \mathtt{F})$ by Definition 42 According to the Definition 45, we have:

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

According to the Definition 13, we have the trace-based adaptivity as follows:

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

Then, we need to show:

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

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 11.
Let $\mathbf{G_{mid}}(\mathbf{c}, \mathbf{m}, \mathrm{D}) = \{\mathbf{V_{mid}}, \mathbf{E_{mid}}, \mathbf{F_{mid}}\}$ be the intermediate graph by Definition 46.
By Lemma 18, 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 19, let g be the surjective function $g : \mathbf{V_{prog}} \to \mathbf{V_{mid}}$ s.t.:

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

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

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

Vertices sequence: $(\mathtt{av}_1, \dots, \mathtt{av}_n)$.
By Lemma 21, 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:

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

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

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: $(g(e), \dots, g(e_{n-1})$

Vertices sequence: $(f(\mathtt{av}_1), \dots, f(\mathtt{av}_n))$.
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:

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

By Lemma 19, we know:

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

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

Definition 46 (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:*

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

]]
[[

Lemma 13 ($\mathsf{DEP}_{\mathsf{var}}$ is Transitive)**.** .
Given a program $\mathbf{c}$*, with a starting memory* $\mathbf{m}$ *and a hidden database D, s.t.,* $\langle \mathbf{m}, \mathbf{c}, [], [], [] \rangle \to^* \langle \mathbf{m}', \mathtt{skip}, \mathtt{qt}, \mathtt{vt}, w \rangle$*. Then,* $\forall \mathtt{av}_1, \mathtt{av}_2, \mathtt{av}_3 \in \mathtt{vt}$*:*

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

of Lemma 13. Proof by unfolding and rewriting the Definition 20. ■

]]
[[

Lemma 14 ($\mathtt{flowsTo}$ is Transitive ??)**.** .
Given a program $\mathbf{c}$ *with its assigned variables* $\mathtt{aVar}$ *of length N. Then* $\forall x_1, x_2, x_3 \in \mathtt{aVar}$

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

of Lemma 14. Proof by unfolding the Definition 34. ■

]]

[[

Lemma 15 (DEP$_\mathsf{q}$ Implies DEP$_\mathsf{var}$)**.** .
Given a program $\mathbf{c}$*, with a starting memory* $\mathbf{m}$ *and a hidden database* D*, s.t.,* $\langle \mathbf{m}, \mathbf{c}, [], [], [] \rangle \to^* \langle \mathbf{m}', \mathtt{skip}, \mathtt{qt}, \mathtt{vt}, w \rangle$*. Then,* $\forall \mathtt{aq}_1, \mathtt{aq}_2 \in \mathtt{qt}$

$$\mathsf{DEP}_\mathsf{q}(\mathtt{aq}_1, \mathtt{aq}_2, \mathbf{c}, \mathbf{m}, D) \implies \mathsf{DEP}_\mathsf{var}(\pi_2(\mathtt{aq}_1), \pi_2(\mathtt{aq}_2), \mathbf{c}, \mathbf{m}, D)$$

of Lemma 15. Proof by unfolding the Definition 20 and Definition 10. ■

]]

[[

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

$$\mathsf{DEP}_\mathsf{var}(\mathtt{av}_1, \mathtt{av}_2, \mathbf{c}, \mathbf{m}, D) \implies \mathtt{flowsTo}(\pi_1(\mathtt{av}_1), \pi_1(\mathtt{av}_2))$$

of Lemma 15. Proof by showing contradiction based on the Definition 20 and Definition 34. Let $\mathtt{av}_1, \mathtt{av}_2 \in \mathtt{vt}$ be 2 arbitrary annotated variables in the variable trace $\mathtt{vt}$, s.t., $\mathsf{DEP}_\mathsf{var}(\mathtt{av}_1, \mathtt{av}_2, \mathbf{c}, \mathbf{m}, D)$. Unfolding the DEP$_\mathsf{var}$ definition, we have: ■

]]

[[

Lemma 17 (Injective Mapping of vertices from $\mathbf{G_{trace}}$ to $\mathbf{G_{mid}}$)**.** .
$\mathbf{G_{trace}}(\mathbf{c}) = \{\mathbf{V_{trace}}, \mathbf{E_{trace}}\}$
$\mathbf{G_{mid}}(\mathbf{c}, \mathbf{m}, D) = \{\mathbf{V_{mid}}, \mathbf{E_{mid}}, \mathbf{F_{mid}}\}$

$$\exists\, \mathtt{injective}\ f : \mathcal{AQ} \to \mathcal{AV}.\ \forall \mathtt{aq} \in \mathbf{V_{trace}}.\ f(\mathtt{aq}) \in \mathbf{V_{mid}} \land \mathbf{F_{mid}}(f(\mathtt{aq})) = 2$$

Proof. Proving by Definition 46 and Definition 45. ■

]]

[[

Lemma 18 (One-on-One Mapping from $\mathtt{E}$ of $\mathbf{G_{trace}}$ to $\mathcal{PATH}(\mathbf{G_{mid}})$)**.** .
$\mathbf{G_{trace}}(\mathbf{c}) = \{\mathbf{V_{trace}}, \mathbf{E_{trace}}\}$
$\mathbf{G_{mid}}(\mathbf{c}, \mathbf{m}, D) = \{\mathbf{V_{mid}}, \mathbf{E_{mid}}, \mathbf{F_{mid}}\}$
An injective function $f : \mathbf{V_{trace}} \to \mathbf{V_{mid}}$ *s.t.,* $\forall \mathtt{aq} \in \mathbf{V_{trace}}.\ \mathbf{F_{mid}}(f(\mathtt{aq})) = 2$

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

Proof. Proving by Lemma 17 and Definition 46 and acyclic property of $\mathbf{G_{trace}}$ and $\mathbf{G_{mid}}$. ■

]]

[[

Lemma 19 (Surjective Mapping of Vertices from $\mathbf{G_{mid}}$ to $\mathbf{G_{prog}}$)**.** .
$\mathbf{G_{mid}}(\mathbf{c},\mathbf{m},D) = \{\mathbf{V_{mid}},\mathbf{E_{mid}},\mathbf{F_{mid}}\}$
$\mathbf{G_{prog}}(\mathbf{c}) = \{\mathbf{V_{prog}},\mathbf{E_{prog}},\mathbf{F_{prog}},\mathbf{W_{prog}}\}$
$\exists$ `surjective` $f:\mathcal{AV}\rightarrow\mathcal{SVAR}$.

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

Proof. Proving by Definition 46. ■

]]
[[

Lemma 20 (Surjective Mapping from E of $\mathbf{G_{mid}}$) to E of $\mathbf{G_{prog}}$)**.** .
$\mathbf{G_{mid}}(\mathbf{c},\mathbf{m},D) = \{\mathbf{V_{mid}},\mathbf{E_{mid}},\mathbf{F_{mid}}\}$
$\mathbf{G_{prog}}(\mathbf{c}) = \{\mathbf{V_{prog}},\mathbf{E_{prog}},\mathbf{F_{prog}},\mathbf{W_{prog}}\}$
A surjective function $f:\mathbf{V_{prog}}\rightarrow\mathbf{V_{mid}}$ *s.t.,* $\forall \mathtt{av}\in\mathbf{V_{mid}}.\ \mathbf{F_{prog}}(f(\mathtt{av})) = \mathbf{F_{mid}}(\mathtt{av}) \wedge |\mathtt{image}(f(\mathtt{av}))| \leq W(f(\mathtt{av}))$

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

Proof. Proving by Lemma 19. ■

]]
[[

Lemma 21 (Surjective Mapping from $\mathcal{PATH}(\mathbf{G_{mid}})$ to $\mathcal{WALK}(\mathbf{G_{prog}})$)**.** .
$\mathbf{G_{mid}}(\mathbf{c},\mathbf{m},D) = \{\mathbf{V_{mid}},\mathbf{E_{mid}},\mathbf{F_{mid}}\}$
$\mathbf{G_{prog}}(\mathbf{c}) = \{\mathbf{V_{prog}},\mathbf{E_{prog}},\mathbf{F_{prog}},\mathbf{W_{prog}}\}$
A surjective function $f:\mathbf{V_{prog}}\rightarrow\mathbf{V_{mid}}$ *s.t.,* $\forall \mathtt{av}\in\mathbf{V_{mid}}.\ \mathbf{F_{prog}}(f(\mathtt{av})) = \mathbf{F_{mid}}(\mathtt{av}) \wedge |\mathtt{image}(f(\mathtt{av}))| \leq W(f(\mathtt{av}))$
A surjective function $g:\mathbf{E_{mid}}\rightarrow\mathbf{E_{prog}}$ *s.t.,* $\forall e_{mid} = (\mathtt{av}_1,\mathtt{av}_2)\in\mathbf{E_{mid}}.\exists e_{prog} = (f(\mathtt{av}_1)\rightarrow f(\mathtt{av}_2))\in\mathbf{E_{prog}}$
$\exists$ `surjective` $h:\mathcal{PATH}(\mathbf{G_{mid}}(\mathbf{c},\mathbf{m},D))\rightarrow\mathcal{WALK}(\mathbf{G_{prog}}(\mathbf{c}))$ *s.t.:*

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

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

Proof. Proving by induction on the length of $l = p_{\mathtt{av}_1\rightarrow\mathtt{av}_2}\in\mathcal{PATH}(\mathbf{G_{mid}}(\mathbf{c},\mathbf{m},\mathrm{D}))$, and Lemma 20 and Lemma 19.

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

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

■

]] □

]

5 [[Examples]]

Example 5.1 (TwoRound Algorithm)**.**

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

Adapt(TR) = 2

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

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

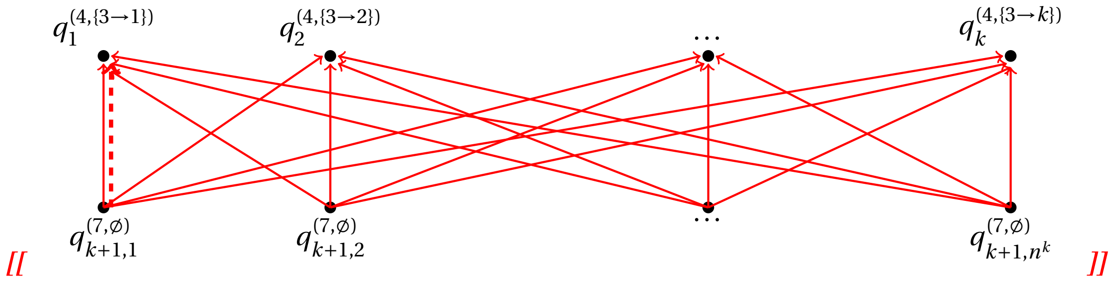
.

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

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)**.**

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

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

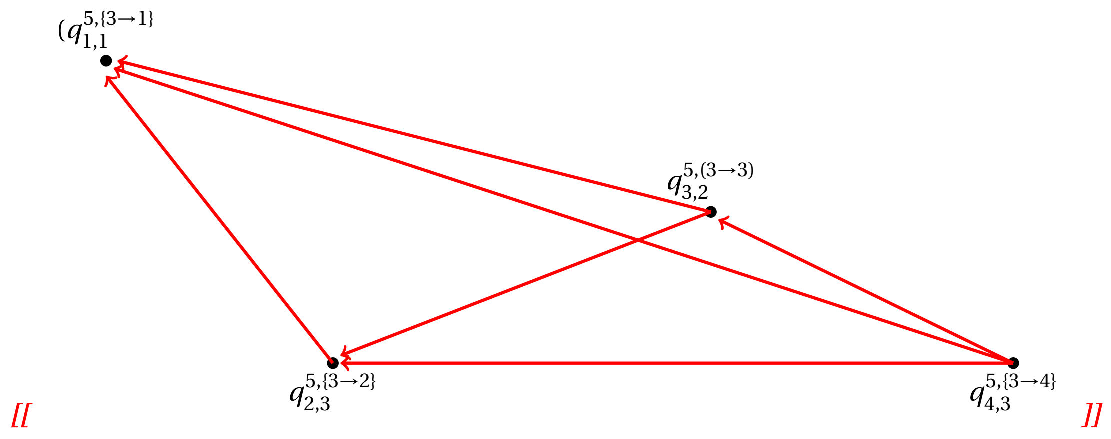
$$[]; \emptyset; MR^{ssa} \to G; w \wedge w = \emptyset$$

.

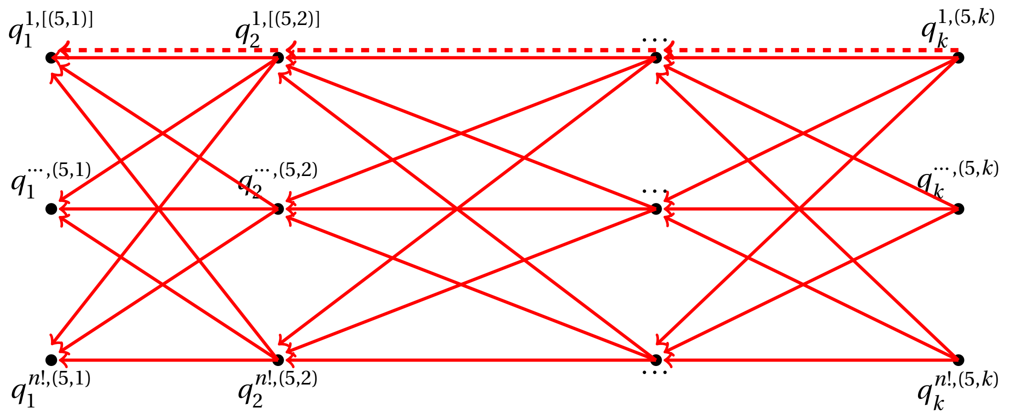
$$G_{k=2} = \big[i_1^1, I_1^2, i_3^3, I_3^3, p_1^4, a_1^5, I_2^6, i_2^7\big]$$

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 10, 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**:

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

, 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 22 and 23 formalize this property.

Lemma 22 (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

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

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

Lemma 23 (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 $[\leftarrow \texttt{query}(\psi)]^l$ nor $[x \leftarrow \texttt{query}(\alpha)]^l$ for any ψ and α, then:

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

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: $\frac{1}{2n} \min\limits_{\lambda \in [0,1]} \left(\frac{2\rho k n - \ln(1-\lambda)}{\lambda}\right) + 2\mathbb{E}_{Z_i \sim N(0, \frac{1}{2n^2\rho})}\left[\max\limits_{i \in [k]}(Z_i^2)\right]$
Confidence Bounds: minimize τ where $\tau \geq \sqrt{\frac{2}{n\beta} \min\limits_{\lambda \in [0,1]} \left(\frac{2\rho k n - \ln(1-\lambda)}{\lambda}\right)}$ 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 $\zeta = \min\limits_{\lambda \in [0,1)} \left(\frac{2B\ (\sigma^2 h) - \ln(1-\lambda)}{\lambda}\right)$

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.

[3] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement types for typescript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 310–325, 2016.