Program Analysis for Adaptive Data Analysis

ANONYMOUS AUTHOR(S)

Data analyses are usually designed to identify some property of the population from which the data are drawn, generalizing beyond the specific data sample. For this reason, data analyses are often designed in a way that guarantees that they produce a low generalization error. That is, to guarantee that the result of a data analysis run on a sample data does not differ too much from the result one would achieve by running the analysis over the entire population.

An adaptive data analysis can be seen as a process composed by multiple queries interrogating some data, where the choice of which query to run next may rely on the results of previous queries. The generalization error of each individual query/analysis can be controlled by using an array of well-established statistical techniques. However, when queries are arbitrarily composed, the different errors can propagate through the chain of different queries and bring to high generalization error. To address this issue, data analysts are designing several techniques that not only guarantee bounds on the generalization errors of single queries, but that also guarantee bounds on the generalization error of the composed analyses. The total number of queries and the depth of the chain of queries are of great significance to guarantee the generalization error, when the composed data analyses are adaptive. The choice of which of these techniques to use, often depends on the process of the chain of queries that an adaptive data analysis can generate.

In this work, we propose a program analysis that can help data analysts design adaptive data analyses controlling their generalization error. Given an input program implementing an adaptive data analysis, our program analysis generates an upper bound on the total number of queries that the data analysis will run, and more interestingly also an upper bound on the depth of the chain of queries. These two measures can be used to select the right technique to guarantee a bound on the generalization error of the data analysis. Our program analysis is based on an analysis of the dependency graph between different queries, representing the potential chain an adaptive data analysis may generate. We show how the proposed program analysis can help to analyze the generalization error of several concrete data analyses with different adaptivity structures.

Additional Key Words and Phrases: Adaptive data analysis, program analysis, dependency graph

1 INTRODUCTION

Consider a dataset \(X\) consisting of \(n\) independent samples from some unknown population \(P\). How can we ensure that the conclusions drawn from \(X\) generalize to the population \(P\)? Despite decades of research in statistics and machine learning on methods for ensuring generalization, there is an increased recognition that many scientific findings generalize poorly (e.g. [Gelman and Loken 2014; Ioannidis 2005]). While there are many reasons a conclusion might fail to generalize, one that is receiving increasing attention is adaptivity, which occurs when the choice of method for analyzing the dataset depends on previous interactions with the same dataset [Gelman and Loken 2014].

Adaptivity can arise from many common practices, such as exploratory data analysis, using the same data set for feature selection and regression, and the re-use of datasets across research projects. Unfortunately, adaptivity invalidates traditional methods for ensuring generalization and statistical validity, which assume that the method is selected independently of the data. The misinterpretation of adaptively selected results has even been blamed for a "statistical crisis" in empirical science [Gelman and Loken 2014].

A line of work initiated by Dwork et al. [2015c], Hardt and Ullman [2014] posed the question: Can we design general-purpose methods that ensure generalization in the presence of adaptivity, together with guarantees on their accuracy? The idea that has emerged in these works is to use randomization to help ensure generalization. Specifically, these works have proposed to mediate
Fig. 1. Overview of our Adaptive Data Analysis model. We have a population that we are interested in studying, and a dataset containing individual samples from this population. The adaptive data analysis we are interested in running has access to the dataset through queries of some pre-determined family (e.g., statistical or linear queries) mediated by a mechanism. This mechanism uses randomization to reduce the generalization error of the queries issued to the data.

The access of an adaptive data analysis to the data by means of queries from some pre-determined family (we will consider here statistical or linear queries) that are sent to a mechanism which uses some randomized process to guarantee that the result of the query does not depend too much on the specific sampled dataset. This guarantees that the result of the queries generalizes well. This approach is described in Figure 1. This line of work has identified many new algorithmic techniques for ensuring generalization in adaptive data analysis, leading to algorithms with greater statistical power than all previous approaches. Common methods proposed by these works include, the addition of noise to the result of a query, data splitting, etc. Moreover, these works have also identified problematic strategies for adaptive analysis, showing limitations on the statistical power one can hope to achieve. Subsequent works have then further extended the methods and techniques in this approach and further extended the theoretical underpinning of this approach, e.g. [Bassily et al. 2016; Dwork et al. 2015a,b; Feldman and Steinke 2017; Jung et al. 2020; Rogers et al. 2020; Steinke and Zakynthinou 2020; Ullman et al. 2018].

A key development in this line of work is that the best method for ensuring generalization in an adaptive data analysis depends to a large extent on the number of rounds of adaptivity, the depth of the chain of queries. As an informal example, the program \( x \leftarrow q_1(D); y \leftarrow q_2(D, x); z \leftarrow q_3(D, y) \) has three rounds of adaptivity, since \( q_2 \) depends on \( D \) not only directly because it is one of its input but also via the result of \( q_1 \), which is also run on \( D \), and similarly, \( q_3 \) depends on \( D \) directly but also via the result of \( q_2 \), which in turn depends on the result of \( q_1 \). The works we discussed above showed that, not only does the analysis of the generalization error depend on the number of rounds, but knowing the number of rounds actually allows one to choose methods that lead to the smallest possible generalization error. As an example, when a study includes queries with a large number of rounds of adaptivity, then a low generalization error can be achieved by adding Gaussian noise scaled to the number of rounds to the result of each query. When instead a study includes queries with a low number of rounds of adaptivity, then a low generalization error can be achieved by using more specialized methods, such as the reusable holdout technique from Dwork et al. [2015c].

This scenario motivates us to explore the design of program analysis techniques that can be used to estimate the number of rounds of adaptivity that a program implementing a data analysis can perform. These techniques could be ultimately be integrated into a tool for adaptive data analysis such as the Guess and Check framework by Rogers et al. [2020].

The first problem we face is how to define formally a model for adaptive data analysis which is general enough to support the methods we discussed above and would permit to formulate the notion of adaptivity these methods use. We take the approach of designing a programming framework for submitting queries to some mechanism giving access to the data mediated by one of the techniques we mentioned before, e.g., adding Gaussian noise, randomly selecting a subset of the data, using the reusable holdout technique, etc. In this approach, a program models an analyst asking a sequence of queries to the mechanism. The mechanism runs the queries on the data...
applying one of the methods discussed above and returns the result to the program. The program
then use this result to decide which query to run next. Overall, we are interested in controlling
the generalization of the results of the queries which are returned by the mechanism, by means of
the adaptivity.

To define adaptivity we consider a dependency graph between the different queries that we
synthesize from the possible execution traces of the program representing the data analysis. The
dependency graph is built by inspecting all the possible traces of execution and by identifying
situations where the execution of a query causes the execution of another query. Intuitively, a
query \( Q \) may depend on another query \( P \), if there are two values that \( P \) can return which affect
in different ways the execution of \( Q \). For example, as depicted in [Dwork et al. 2015b], a machine
learning algorithm for constructing a classifier can be modeled by first computing each feature’s
correlations with the label via a sequence of queries, and then constructing the classifier based on
the correlation values. If one feature’s correlation changes, the classifier depending on features
is also affected. This notion of dependency builds on the execution trace as a causal history. In
particular, we are interested in the history or provenance of a query up until this is executed, we
are not then concerned about how the result is used — except for tracking whether the result of the
query may further cause some other query. This is because we focus on the generalization error of
queries and not their post-processing.

The second problem we face is how to estimate the adaptivity of a given program. The adaptive
data analysis model we consider and our definition of adaptivity suggest that for this task we
can use a program analysis that is reminiscent of information flow control. However, this is not
sufficient since, in general, a query \( Q \) is not a monolithic block but rather it may depend, through
the use of variables and values, on other parts of the program. Hence, we also need to consider
some form of data-flow analysis. Our program analysis, named AdaptFun, combines information
flow and data-flow analysis using an adjacency matrix \( M \) representing the dependency between
different variables, and a vector \( V \) representing different queries. These two components allow us
to over-approximate the dependency graph and estimate the adaptivity of the program.

To simplify our analysis, we do not directly apply the program analysis to the source program.
Instead, we first transform the program into static single assignment form, SSA form. In this form
all the variables are assigned once, including variables in loops, and this helps our analysis in
avoiding the complexity of handling variables reassignment. Moreover, we show that by analyzing
programs in SSA form, we get a bound on the number of rounds of adaptivity that is also a bound
for the source program.

To Summarize, our work aims at the design of a static analysis for programs implementing
adaptive analysis that can estimate their rounds of adaptivity. Specifically, our contributions are as
follows:

1. A programming framework for adaptive data analyses where the program represents an
analyst that can query a generalization-preserving mechanism mediating the access to some
data.
2. A formal definition of the notion of adaptivity under the analyst-mechanism model. This
definition is built on a query-based dependency graph built out of all the possible program
execution traces.
3. A program analysis algorithm AdaptFun which provides an upper bound on the adaptivity
via a variable-based dependency graph.
4. A soundness proof of the program analysis showing that the adaptivity estimated by AdaptFun
bounds the true adaptivity of the program.

2 OVERVIEW

Some results in Adaptive Data Analysis. In Adaptive Data Analysis an analyst is interested in studying some distribution $P$ over some domain $X$. Following previous works [Bassily et al. 2016; Dwork et al. 2015c; Hardt and Ullman 2014], we focus on the setting where the analyst is interested in answers to statistical queries (also known as linear queries) over the distribution. A statistical query is usually defined by some function $f : X \rightarrow [-1, 1]$ (often other codomains such as $[0, 1]$ or $[-R, +R]$, for some $R$, are considered). The analyst wants to learn the population mean, which (abusing notation) is defined as

$$f(P) = \mathbb{E}_{X \sim P} [f(X)].$$

However, the distribution $P$ can only be accessed via a set of samples $X_1, \ldots, X_n$ drawn from $P$. We assume that the samples are drawn independently and identically distributed (i.i.d.). These samples are held by a mechanism $M(X_1, \ldots, X_n)$ who receives the query $f$ and computes an answer $a \approx f(P)$.

The naïve way to approximate the population mean is to use the empirical mean, which (abusing notation) is defined as

$$f(X_1, \ldots, X_n) = \frac{1}{n} \sum_{i=1}^{n} f(X_i).$$

However, the mechanism $M$ can then adopt some methods for improving the generalization error. In this work we consider analysts that ask a sequence of $k$ queries $f_1, \ldots, f_k$. If the queries are all chosen in advance, independently of the answers of each one of them, then we say they are non-adaptive. If the choice of each query $f_j$ depend on the prefix $f_1, a_1, \ldots, f_{j-1}, a_{j-1}$ then they are fully adaptive. An important intermediate notion is $r$-round adaptive, where the sequence can be partitioned into $r$ batches of non-adaptive queries. Note that non-interactive queries are 1-round and fully adaptive queries are $k$ rounds.

We now review what is known about the problem of answering $r$-round adaptive queries.

**Theorem 2.1.** For any distribution $P$, and any $k$ non-adaptive statistical queries, the empirical mean satisfies

$$\max_{j=1, \ldots, k} |a_j - f_j(P)| = O\left(\sqrt{\frac{\log k}{n}}\right)$$

For any $r \geq 2$ and any $r$-round adaptive statistical queries, it satisfies

$$\max_{j=1, \ldots, k} |a_j - f_j(P)| = O\left(\sqrt{\frac{k}{n}}\right)$$

In fact, these bounds are tight (up to constant factors) which means that even allowing one extra round of adaptivity leads to an exponential increase in the generalization error of the empirical mean, from $\log k$ to $k$.

Dwork et al. [2015c] and Bassily et al. [2016] showed that by using an alternative mechanism $M$ which uses randomization in order to limit the dependency of a single query on the specific data instance, one can actually achieve much stronger generalization error as a function of the number of queries, specifically.
we index just by natural numbers. The queries inside the loop correspond to the first phase and we assume that, in this example, the domain $X$ contains at least four numeric attributes, which are of the form $q(e)$ where $e$ is an expression with a special variable $\chi$ representing a possible row. Mainly $e$ represents a function from $X$ to some domain $U$, for example $U$ could be $[-1, 1]$ or $[0, 1]$. This function characterizes the linear query we are interested in running. As an example, $x \leftarrow q(\chi[2])$ computes an approximation, according to the used mechanism, of the empirical mean $\frac{1}{n} \sum_{x \in D} q(\chi[2])$. The queries inside the loop correspond to the second phase and computes an approximation of the empirical mean where each record is weighted by the sum of the empirical mean of the first three attributes. Queries are of the form $q(e)$ where $e$ is an expression with a special variable $\chi$ representing a possible row. Mainly $e$ represents a function from $X$ to some domain $U$, for example $U$ could be $[-1, 1]$ or $[0, 1]$. This function characterizes the linear query we are interested in running. As an example, $x \leftarrow q(\chi[2])$ computes an approximation, according to the used mechanism, of the empirical mean $\frac{1}{n} \sum_{x \in D} q(\chi[2])$.

**Theorem 2.2 ([Bassily et al. 2016; Dwork et al. 2015c]).** For any $k$, there exists a mechanism such that for any distribution $P$, and any $r \geq 2$ any $r$-round adaptive statistical queries, it satisfies

$$\max_{j=1,\ldots,k} |a_j - f_j(P)| = O \left( \frac{\sqrt{k}}{\sqrt{n}} \right)$$

Notice that Theorem 2.2 has different quantification in that the optimal choice of mechanism depends on the number of queries. Thus, we need to know the number of queries a priori to choose the best mechanism.

Dwork et al. [2015c] also gave more refined bounds in terms of the number of rounds of adaptivity.

**Theorem 2.3 ([Dwork et al. 2015c]).** For any $r$ and $k$, there exists a mechanism such that for any distribution $P$, and any $r \geq 2$ any $r$-round adaptive statistical queries, it satisfies

$$\max_{j=1,\ldots,k} |a_j - f_j(P)| = O \left( \frac{r \sqrt{\log k}}{\sqrt{n}} \right)$$

This suggests that if one knows a good a priori upper bound on the number of rounds of adaptivity, one can get a much better guarantee of generalization error, but only by using an appropriate choice of the mechanism.

A formal model for adaptivity. Motivated by the results discussed above, we will present a static analysis aimed at giving good a priori upper bounds on the number of rounds of adaptivity of a program. Before introducing the static analysis, we motivate the definition of adaptivity we will use through a simple example illustrated in Figure 2(a), which implements a simple "two rounds strategy".

In this example the analyst asks queries to the mechanism in two phases. In the first phase, the analyst asks a fixed number $k$ of queries (in the example $k = 3$) and stores the answers that are provided by the mechanism. In the second phase, the analyst constructs a new query based on the results of the previous $k$ queries and sends this query to the mechanism. More specifically, we assume that, in this example, the domain $X$ contains at least four numeric attributes, which we index just by natural numbers. The queries inside the loop correspond to the first phase and compute an approximation of the empirical mean of the first three attributes. The query outside the loop corresponds to the second phase and computes an approximation of the empirical mean where each record is weighted by the sum of the empirical mean of the first three attributes. Queries are of the form $q(e)$ where $e$ is an expression with a special variable $\chi$ representing a possible row. Mainly $e$ represents a function from $X$ to some domain $U$, for example $U$ could be $[-1, 1]$ or $[0, 1]$. This function characterizes the linear query we are interested in running. As an example, $x \leftarrow q(\chi[2])$ computes an approximation, according to the used mechanism, of the empirical mean $\frac{1}{n} \sum_{x \in D} q(\chi[2])$.
of the second attribute, identified by $\chi[2]$. Notice that we don’t materialize the mechanism but we assume that it is implicitly run when we execute the query.

In order to analyze programs like the one we just discussed, it is convenient to work with a version of the program where similar commands can be easily distinguished. For this reason, we use labeled versions of programs, where labels correspond to lines of code. As an example, we give the labeled version of the two rounds example program in Figure 2(b).

This example is intuitively 2-rounds adaptive since we have two clearly distinguished phases, and the queries that we ask in the first phase do not depend on each other, while the last query depends on all the previous queries. However, capturing this concept formally is surprisingly difficult. The difficulty comes from the fact that a query can depend on the result of another query in multiple ways, by means of data dependency or control flow dependency. In order to find the right definition for our goal we take inspiration from the known results on the data analysis model we discussed above. This theory tells us that what we want to measure is the generalization error on the result of a query, and not an arbitrary manipulation of the query. Indeed, arbitrary manipulations can change the generalization error. As an example, suppose that $v$ is the result we get from running a query, if we multiply this result by some constant, we are also changing the incurred error. Moreover, this theory tells us that we can always consider a non-adaptive set of queries as to being adaptive, and more importantly, that we can transform an adaptive query into a non-adaptive one, incurring an exponential blow up of the number of queries. For example, we could ask many queries upfront and depending on the results of some of them, we could return the results of others. For these reasons, we define adaptivity in terms of the possible execution traces of the program on all possible inputs. A trace of execution is a list of query requests of the form $[q_1(v_1)^{(l_1,w_1)}, \ldots, q_n(v_n)^{(l_n,w_n)}]$, where every occurrence of a query is labeled with the line of code $l$ it appears at, and the counter $w$ identifying the possible loop iteration happening when a query is called. For example, in $q_1^{(3,1)}$, the superscript $(3,1)$ indicates that the query is asked at line 3 and that the query is requested in the first iteration of the loop. When the query is not in a loop, we omit the counter.

Using traces we can identify situations when one query can affect the execution of another one. Using this information we can build a directed graph, called query-based dependency graph, where the nodes represent the queries that are executed and the edges between two nodes represent the fact that one query may depend on the other. We show such a graph for our running example in Figure 2(c). We can then define adaptivity as the longest possible path in this graph. Looking again at our example, it is easy to see that the longest path in the graph in Figure 2(c), which we mark with a red dashed arrow, is 2, as we were expecting.

**Static analysis for adaptivity.** The high level architecture of our static analysis framework is presented in Figure 3. The input of the analysis is a labeled program $P$ for which the adaptivity $A$ is defined by means of a trace-based definition, as discussed above. In order to estimate an upper bound on $A$, our program analysis first transforms the program $P$ into static single assignment (SSA) form. The goal of this step is to guarantee that each variable is assigned only once. We show
We introduce the syntax of the Loop language we use to write our data analyses. It is standard that expressions can be either arithmetic expressions or boolean expressions. An arithmetic expression can be a constant $n$ denoting integer, a variable $x$ from some countable set $\text{Var}$, a combination of arithmetic expressions by means of the symbol $\oplus$, denoting basic operations including addition, product, subtraction, etc. A boolean expression can be $\text{true}$ or $\text{false}$, the negation of a boolean expression, or a combination of boolean expressions by means of $\oplus$, denoting basic boolean operations.

### 3 LOOP LANGUAGE

In this section, we formally introduce the language we will focus on for writing data analyses. This is a simple loop language with some primitives for calling queries. After defining the syntax of the language and showing an example, we will define its trace-based operational semantics. This is the main technical ingredient we will use to define the program’s adaptivity. We will conclude this section by discussing the limitation of this language with respect to static analysis for adaptivity.

#### 3.1 Syntax

We introduce the syntax of the Loop language we use to write our data analyses. After defining the syntax of the language and showing an example, we will define its trace-based operational semantics. This is the main technical ingredient we will use to define the program’s adaptivity. We will conclude this section by discussing the limitation of this language with respect to static analysis for adaptivity.

```plaintext
[a_1 \leftarrow 0]^1;
[i_1 \leftarrow 0]^2;
\text{loop } [3]^3 \text{ do }
\begin{align*}
(i_3, i_1, i_2), (a_3, a_2, a_1) \\
x_1 \leftarrow q_1(\chi[i_3])^4 \\
a_2 \leftarrow a_2 + x_1^3 \\
i_2 \leftarrow i_3 + 1^6 \\
[l_1 \leftarrow q_2(\chi[4] * a_3)]^7
\end{align*}
```

Fig. 4. (a) Example of a ssa program with two rounds of adaptivity (b) The corresponding variable-based dependency graph.
We use $∅$ to represent an empty map, indicating the statement not in any loop. Some operations on loops are supported and $[]$ stands for the empty list. The access to elements in the list can be achieved through $x[a]$ when variable $x$ is referred to a list. The value $v$ now contains the natural number $n$, and the boolean primitives $true$ and $false$, the special row $\chi$ and access to it $\chi[v]$, the empty list $[]$ and non-empty list $[v, \ldots, v]$.

A command $c$ can either be $skip$, an assignment command $x ← e$, the composition of two commands $c; c$, an if statement $if(b, c, c)$, a loop statement $loop a$ do $c$. The main novelty of the syntax is the query request command $x ← q(e)$. As a reminder, the aforementioned tight bound of adaptive data analysis in Section 2 focuses on linear queries, specified by a function from rows to $[0, 1]$ or $[-1, +1]$. To express these functions, we introduce the special variable $\chi$ to represent the rows of the database in the arithmetic expressions. In this sense, a simple linear query which returns the first element of the row is written as $q(\chi[1])$, representing a query $q(\chi) = \chi(1)$. The query can also take variables as input, the aforementioned two round example in Figure 2.(a) uses the loop counter $i$ to construct the query $q(\chi[i])$, and a more complicated one $q(\chi[4] + a)$ at the end.

### 3.2 Trace-based Operational Semantics

We evaluate programs in our Loop language by means of the trace-based operational semantics, to capture the dependency between queries. For easier looking up in the trace, we add a label to Loop language as follow:

Labeled commands $c ::= [x ← e]^l | [x ← q(e)]^l | loop a^l do c | c; c | if(b^l, c, c) | [skip]^l$

The commands are now labeled with $l$, a natural number standing for the line of code where they appear. We put the label $l$ to the conditional predicate $b$ in the if statement, and to the loop counter $a$ in the loop statement. It is used when we introduce the Loop Maps $w$ as shown below.

Loop Maps $w ∈ Label → \mathbb{N}$ Memory $m ::= [] | m[x → v]$

Annotated Query $AQ ::= \{q(v)^{(1, w)}\}$ Trace $t ::= [] | q(v)^{(1, w)} ::= t$

An important component of the labeled Loop language is loop maps $w$, designed for loops. It is a map from label $l$ to the iteration number $n$, as a complementary annotation to the label. Because the statement in the loop from varied iterations shares the same line number, the label $l$ in commands alone is not enough to distinguish statements. The loop maps $w$ with the mapping of $[k → n]$ gives accurate information on which loop the statement is in by its key($l$ at loop counter), and which iteration $n$ the statement belongs to. For example, the loop maps $w = [3 : 1, 4 : 2]$ indicates that the statement is currently in a nested loop, the outer loop starting from label 3 and in its first iteration, the statement is now in the inner loop starting from label 4 and in the second iteration. We use $∅$ to represent an empty map, indicating the statement not in any loop. Some operations on
\[\{m, c, t, w\} \to \{m', c', t', w'\}\]  
\[\{m, \text{loop } [v_N]^{t} \text{ do } c, t, w\} \to \{m, \text{loop } \{\{v_N - 1\}\}^{t} \text{ do } c, t, (w + l)\}\]

\[w_N > 0\]  
\[\text{low-loop}\]

\[\{m, \text{loop } [v_N]^{t} \text{ do } c, t, w\} \to \{m, \text{loop } \{\{v_N - 1\}\}^{t} \text{ do } c, t, (w + l)\}\]

\[w_N = 0\]  
\[\text{low-loop-exit}\]

\[\{m, \text{loop } [v_N]^{t} \text{ do } c, t, w\} \to \{m, \text{skip}^{t}, t, (w \setminus l)\}\]

\[\text{low-seq2}\]

\[\{m, \text{if}([\text{false}]^{t}, c_1, c_2), t, w\} \to \{m, c_2, t, w\}\]

\[\text{low-if-f}\]

\[\{m, e \to e'\}\]

\[\{m, e\} \to \{m, [x \leftarrow q(e)]^{t}, t, w\}\]

\[\text{low-query-e}\]

\[\{m, c_1, t, w\} \to \{m', c_1', t', w'\}\]

\[\text{low-seq1}\]

\[\{m, [x \leftarrow q(v)]^{t}, t, w\} \to \{m, [v_q/x], \text{skip}, t + [q(v)]^{t/l}, w\}\]

\[\text{low-query-v}\]

\[\{m, [b]^{t}, c_1, c_2), t, w\} \to \{m, \text{if}([b]^{t}, c_1, c_2), t, w\}\]

\[\text{low-if-t}\]

\[\{m, b \to b'\}\]

\[\{m, c_1, t, w\} \to \{m, \text{if}([b]^{t}, c_1, c_2), t, w\}\]

Fig. 6. Trace-based operational semantics

We denote \(w \setminus l\) to remove the mapping of the key \(l\) in the loop maps \(w\), used when exiting the loop at line \(l\). The special loop maps \(w_l\) expresses a map identical to \(w\), but without the mapping of label \(l\). We register to \(w\) in the first iteration of a loop marked by label \(l\) and assign \(l\) with the iteration 1. The mapped number increase when going into another iteration of the same loop. We denote \(\text{Keys}(w)\) to return all the keys of the loop maps \(w\).

A memory is standard, a map from variables to values. Queries can be uniquely annotated as \(\mathcal{AQ}\), 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 loop statement, specified by \(w\). A trace \(t\) is supposed to accumulate along with the execution of the program. It is a list of annotated queries \(\mathcal{AQ}\).

A trace can be regarded as a history, of queries asked by the analyst during the execution of the program. We used the trace-based small-step operational semantics in Figure 6, of the form \(\{m, c, t, w\} \to \{m', \text{skip}, t', w'\}\). It states that a configuration \(\{m, c, t, w\}\) evaluates to another configuration with the trace and loop maps updated along with the evaluation of the command \(c\) to the normal form of the command \(\text{skip}\). A configuration contains four elements: a memory \(m\), the command \(c\) to be evaluated, a starting trace \(t\), a starting loop maps \(w\). Most of the time, the loop maps remains empty until the evaluation goes into loops. We also have the evaluation of arithmetic expressions of the form \(\{m, a\} \to_a a'\), evaluating an arithmetic expression \(a\) in the memory \(m\), and similar for the boolean expressions \(\{m, b\} \to_b b'\).

We introduce the selected parts of the operational semantics in Figure 6. The rule \text{low-query-e} evaluates the argument of a query request. When the argument in normal form, this query will be answered. The rule \text{low-query-v} modifies the starting memory \(m\) to \(m[v_q/x]\) using the answer \(q(v)\) of the query \(q(v)\) from the mechanism, with the trace expanded by appending the query \(q(v)\) 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 loop modify the loop maps \(w\). In the rule \text{low-loop}, the loop maps \(w\) is updated by \(w + l\) because the execution goes into another iteration when the condition \(v_N > 0\) is satisfied. When \(v_N\) reaches 0, the loop exits and the loop maps \(w\) eliminates the label \(l\) of this loop statement by \(w \setminus l\) in the rule \text{low-loop-exit}. 

3.3 Query-based Dependency Graph

We give the adaptivity through a query-based dependency graph. In the model for adaptive data analyses we choose, 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 of 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 one query, the only information he has at hand will be the answers to previous queries. 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 the change of answers to previous queries in a direct way. There are two possible cases that the choice of the query will be affected, either when it uses the results of previous queries (data dependency), or the control flow of this query (whether to ask this query or not) depends on these results (control dependency).

As the first step, we give the definition of one query may depend on its 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 one previous query if and only if the change of the answer to the previous query may also change the result of the query.
2. One query may depend on one previous query if and only if the 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 $p = x \leftarrow q(\chi(1)); y \leftarrow q(\chi(2))$. It also works well to consider data dependency. However, if fails to catch control dependency by just monitoring the change of the answer of query when the answer of its target previous query returned by the mechanism changes. The key point is that this query may also not be asked if the analyst was given different answers of queries before and this should also be counted as may dependency, shown in program $p_1$ as follows.

$$p_1 = x \leftarrow q(\chi(1)); \text{if}(x > 2, y \leftarrow q(\chi(2)), \text{skip})$$

We choose the second candidate, which performs well by witnessing the appearance of one query $q(\chi(2))$ upon the change of the result of one previous query $q(\chi(1))$ in $p_1$. It considers the control dependency, and at the same, does not miss the data dependency. In particular, the arguments of a query decide its identification. 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. In the definition, it is considered as may dependency. Let us look at another variant of program $p$, $p_2$, in which the queries equipped with functions using previously assigned variables storing answer of its previous query.

$$p_2 = x \leftarrow q(\chi(2)); y \leftarrow q(x + \chi(3))$$

As a reminder, in the Loop language, the query request is composed by two components: a symbol $q$ representing a linear query type and the argument $e$, which represents the function specifying what the query asks. So we do think $q(\chi(1))$ is different from $q(\chi(2))$. Informally, we think $q(x + \chi(3))$ may depend on the query $q(\chi(2))$, because equipped function of the former $x + \chi(3)$ depend on the data assigned with $q(\chi(2))$. We can see the appearance definition catches data dependency in such a way, since $q(x + \chi(2))$ 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.
The introduction of a language supports SSA and the adaptivity in this language follows. At last, the length of the longest path in this graph. We denote the path from starting with different configurations, we can construct a corresponding dependency graph.

Finally, we reach the definition of adaptivity, by means of the query-based dependency graph.

Definition 2 (Query-based Dependency Graph). Given a program $c$, a database $D$, a starting memory $m$, an initial loop maps $w$, the query-based dependency graph $G(c, D, m, w) = (V, E)$ is defined as:

$$V = \{ q(v)^{(l, w)} \in AQ \mid \forall t. \exists m', w', t'. \langle m, c, t, w \rangle \rightarrow^* \langle m', \text{skip}, t', w' \rangle \land q(v)^{(l, w)} \in (t' - t) \}.$$  

$$E = \{ (q(v)^{(l, w)}, q(v')^{(l', w')}) \in AQ \times AQ \mid \text{DEP}(q(v')^{(l', w')}, q(v)^{(l, w)}, c, w, m, D) \}.$$

The edge is directed, when an annotated query $q(v)^{(l, w)}$ may depend on its previous query $q(v')^{(l', w')}$, we have the directed edge $(q(v)^{(l, w)}, q(v')^{(l', w')})$ from $q(v)^{(l, w)}$ to $q(v')^{(l', w')}$.

The query-based dependency graph only considers the newly generated annotated queries during the execution of the program $c$, so we see the nodes coming from the trace $t' - t$. The previous trace before the execution of $c$ is excluded when constructing the graph. To summary, for every execution of a program $c$ staring with different configurations, we can construct a corresponding dependency graph.

Definition 3 (Adaptivity in Loop Language). Given a program $c$, and a memory $m$, a database $D$, a starting loop maps $w$, the adaptivity of the dependency graph $G(c, D, m, w) = (V, E)$ is the length of the longest path in this graph. We denote the path from $q(v)^{(l, w)}$ to $q(v')^{(l', w')}$ as $p(q(v)^{(l, w)}, q(v')^{(l', w')})$. The adaptivity denoted as $A(c, D, m, w)$.

$$A(c, D, m, w) = \max_{q(v)^{(l, w)}, q(v')^{(l', w')}} \{|p(q(v)^{(l, w)}, q(v')^{(l', w')})|\}$$

4 TOWARDS STATIC SINGLE ASSIGNMENT

In this section, we first present the limitations of the Loop languages for static analysis, followed by our solution – towards the static single assignment form [Alpern et al. 1988; Rosen et al. 1988]. The introduction of a language supports SSA and the adaptivity in this language follows. At last, we show a transformation of the two languages and give the soundness of the transformation.

4.1 The Limit of Loop Language for Static Analysis

The labeled Loop language supports the notion of adaptivity semantically, through a query-based dependency graph. However, syntactically, it is not good enough for program analysis. The reason is it allows variables to be reassigned, making the decision on where used variables come from tricky, especially there are controlled branches. We use three examples in Loop language to show
the dilemma, assuming \( q_1, q_2, q_3 \) are three linear queries.

\[
\begin{align*}
  s_1 & = \begin{cases} 
   \{ x \leftarrow q_1 \} & \text{if} \; (x < 0) \\
   \{ x \leftarrow q_2 \} & \text{else} \\\n   \{ \text{skip} \} & \text{else}
  \end{cases} \\
  s_2 & = \begin{cases} 
   \{ x \leftarrow q_1 \} & \text{if} \; (x < 0) \\
   \{ x \leftarrow q_2 \} & \text{else} \\\n   \{ \text{skip} \} & \text{else}
  \end{cases} \\
  s_3 & = \begin{cases} 
   \{ z \leftarrow q_2 \} & \text{if} \; (x < 0) \\
   \{ x \leftarrow q_3 \} & \text{else} \\\n   \{ \text{skip} \} & \text{else}
  \end{cases}
\end{align*}
\]

In these three examples, the variable \( x \) used in the query \( q(x + \chi(3)) \) at line 5 is implicit, when we statically analyze the statement \( y \leftarrow q(x + \chi(3)) \). In program \( s_1 \), it refers to the either \( x \) at line 1, or \( x \) at line 3. When we have a look at the other two programs \( s_2 \) and \( s_3 \), the query \( q(x + \chi(3)) \) may depend on either \( q_2(x) \) at line 3 or \( q_3(x) \) at line 4 in \( s_2 \), while it only depends on \( q_1(x) \) at line 1 in program \( s_3 \). These structural similar three examples, however, have quite dissimilar dependencies between variables. It increases the challenge to track dependency in static analysis. Still look at the analysis on the statement \( y \leftarrow q(x + \chi(3)) \), extra information is needed such as value of \( x \) used in the statement may come from the result of \( q_1 \) or the answer to \( q_2 \) when this statement lies in \( s_1 \). Similarly, when in program \( s_2 \), the extra information that the value of \( x \) used in the same statement relies on answers to queries \( q_2 \) or \( q_3 \) in both branches of the if statement starting from line 2 is necessary for static analysis on dependency. Additionally, in program \( s_2 \), we also need to update the information that \( x \) assigned at line 1 is overwritten by both branches when analyzing the statement at line 5. To simplify the program analysis, we choose to conduct the static analysis on the static single assignment form of our target programs.

\[
\begin{align*}
  s^*_1 & = \{ [x_1 \leftarrow q_1] \} \\
  s^*_2 & = \{ [x_1 \leftarrow q_1] \} \\
  s^*_3 & = \{ [x_1 \leftarrow q_1] \}
\end{align*}
\]

To distinguish between the Loop language and in SSA form, we denote the SSA variable \( x \) in bold. As we can see, the reachability of assigned variables becomes explicit in the SSA form. In the SSA version \( s^*_1 \) of \( s_1 \), still looking at the statement at line 5, which becomes \( y_1 \leftarrow q(x_1 + \chi(3)) \), we can syntactically figure out that the query may depend on the variable \( x_3 \), which may come from \( x_1 \) or \( x_2 \), without extra information like in \( s_1 \). This benefit also applies to the analysis over the same statement at line 5 in \( s^*_2 \) and \( s^*_3 \).

### 4.2 The SSA Loop Language

We present the syntax of the SSA loop language, a language based on the Loop language, representing programs in the static single assignment form.

The expression inherits from the Loop language, except that the SSA arithmetic expressions now contain SSA variable \( x \in \mathcal{S}V \), and the boolean expression as \( b \). The SSA expression mimics its counterpart in Loop language. In the language, variables can also be annotated, denoted as \( \mathcal{L}V \), in a similar way as the annotated queries in the Loop language. For instance, \( x^{(l,w)} \in \mathcal{L}V \). The SSA memory now is map from SSA variables to values.

The SSA labeled command \( c \) inherits from the Loop language, except that the expressions and variables in these commands are now in its SSA version as shown below.

\[
c ::= \begin{cases} 
   \{ \text{skip} \} & \text{if} \; (x \leftarrow q(e)) \text{ and } \text{if} \; (x \leftarrow \text{skip}) \text{ and } \text{loop} \; \{ [a] \}, n, [\bar{x}, x_1, x_2] \text{ do } c | \\
   c; c & \text{if} \; (b^{l,w}) \text{ and } \text{loop} \; \{ [\bar{y}, y_1, y_2] \}, [\bar{z}, z_1, z_2]) \text{, } c, c
\end{cases}
\]
When switching to the SSA loop language, we show that we are still able to achieve what we will discuss more about it when used in the operational semantics for the SSA loop language.

Take the query request as an example, the argument $s$ evaluated to a value $v$ before the request is sent to the mechanism in rule SSA-query-arg. The if command now contains the extra part $\bar{y} \iff \bar{y}$ evaluated to a value $v$. The second part $\bar{x}$ may depend on the corresponding element of the same location, $x_1$ from $x_1$ collected in the then branch or the corresponding element $x_2$ from $x_2$ collected in the else branch. The size of these three lists are required to be the same.

Every tuple $(x, x_1, x_2)$ from $\bar{x}$ can be understood as $x = \phi(x_1, x_2)$ in the normal SSA form. Every tuple $(c, x_1, x_2)$ or $\bar{x}$ can be used for reference. The second part $\bar{y}$ focuses on the then branch. The list of SSA variables $y_1$ stores the assigned SSA variables before the if statement, whose non-SSA version (variables in the Loop language) will be modified only in the then branch. We can look at program $s_1$ as a reference, in which $x$ at line 1 may be modified only in the then branch at line 3. The list $y_2$ tracks the SSA variables assigned only in the then branch. If the variables are assigned in both branches such as in the program $s_2$, they go into $\bar{x}$. Then we think every SSA variable in $\bar{y}$ may come from the corresponding variable $y_1$ in $y_1$ before the if command or $y_2$ in $y_2$ in the then branch. In this sense, we can also regard every tuple $(y, y_1, y_2)$ as $y = \phi(y_1, y_2)$. The rest part $[z, z_1, z_2]$ focus on the else branch and can be understood similarly. The loop command also has similar part $[x, x_1, x_2]$, focusing on the loop body. The new command $if\var{\bar{x}, x'}$ does not have explicit label because it is only used for evaluation internally, we will discuss more about it when used in the operational semantics for the SSA loop language.

4.3 Trace-based Operational Semantics of SSA Loop Language

When switching to the SSA loop language, we show that we are still able to achieve what we can get in Section 3. The operational semantics of the SSA loop language mimics its counterpart, of the form $T = \langle m, c, t, w \rangle \rightarrow \langle m', c, t', w' \rangle$. The SSA memory $m$ is a map from SSA variable $x$ to values. It still uses a trace to track the query requests during the execution, starting from an SSA configuration with SSA memory $m$ and program in its SSA form $c$, which allows a similar construction of the query-based dependency graph in the SSA language as in the Loop language.

We showed Early evaluation rules in Figure 7. 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 $e$ which may contain SSA variables will be evaluated to a value $v$ first before the request is sent to the mechanism 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 query, makes no difference in the two languages.

Since we add the extra part \([x, x_1, x_2], [y, y_1, y_2], [z, z_1, z_2] \) in the if statement compared to its counterpart in the Loop language introduced before, the rules relevant to the if conditional (SSA-if-t and SSA-if-f) use the extra command if\( var(x, x') \) to update the SSA memory \( m \) with the the mapping from all the new generated variable \( x \) in the list \( x \) to the appropriate value \( m(x') \).

The SSA variable \( x' \) is the corresponding variable with respect to \( x \) in \( x' \). There is a one-on-one correspondence between the two SSA variable lists \( x \) and \( x' \), based on the position in the list, which requires the two lists of the same length. The rule SSA-ifvar reflects the usage of if\( var(x, x') \). It is easier to understand the usage of if\( var(x, x') \) 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. \( x_1 \) and \( x_2 \) in the rule), it generates a new SSA variable \( x \), assigning it with \( \phi(x_1, x_2) \), and replaces the variable to be used with this newly assigned \( x \). We know that in the future program after this if statement, only the variables appeared in \( x \) will be available, instead of \( x_1, x_2 \) from two branches. For the evaluation of the program after this if statement, we need to tell the memory the exact value of the newly generated variable \( x \), which is the value stored in \( x_1 \) when the conditional predicate \( b \) is true, or the value in \( x_2 \) when \( b \) is false. To this end, the internal command if\( var(x, x') \) plays its role. For the if the rule, we need to instantiate those variables from \( x \) whose values come from two branches, \( y \) whose values from then branch or assignment before the if command, and \( z \) whose values from else branch or before the if command. Correspondingly, we need to have three extra ifvar commands.

The evaluation of loop depends on the loop counter \( a \) in the rule SSA-loop, which will be evaluated to a value \( v_N \). When \( v_N \) is greater than 0, the loop is still executing, and all the variables \( x \) in \( \bar{x} \) of the loop body \( c \) are replaced as the corresponding variables in \( x_2 \) in the first iteration \( n = 0 \), or \( x_2 \) in other iterations \( n > 0 \). The loop turns to an exit described in the rule SSA-loop-exist when \( v_N > 0 \), and the memory \( m \) updates the mapping of variables in \( \bar{x} \) with \( x_1 \) if the iteration counter goes to zero \( n = 0 \), which means the loop body is not executed once. When the loop enters the exit after executing the body a few times \( n \), the variables in \( \bar{x} \) is instantiated with the value from the body \( m(x_2) \).

The trace-based operational semantics of the SSA loop language allows us to provide our query-based dependency graph in the SSA version.

**Definition 4 (Query May Dependency in SSA).** One query \( q(v_2) \) may depend on another query \( q(v_1) \) in a program \( c \), with a starting loop maps \( w \), denoted as \( \text{DEP}_{\text{SSA}}(q(v_1))(l,w), q(v_2))(l,w), c, w, m, D) \).

We omit the formal definition of may dependency in SSA, which mimics its counterpart of the Loop language. The query-based dependency graph as well the adaptivity in SSA loop language can be similarly defined.

**Definition 5 (Dependency Graph in SSA).**

Given a program \( c \), a database \( D \), a starting loop maps \( w \), the dependency graph \( G_s(c, D, m, w) = (V, E) \) is defined as:

\[ V = \{q(v)^{(l,w)} \in A Q \mid \forall t. \exists m', w', t'. (m, c, t, w) \rightarrow^{*} (m', \text{skip}, t', w') \wedge q(v)^{(l,w)} \in (t' - t)\} \]

\[ E = \{(q(v)^{(l,w)}, q(v')^{(l',w')}) \in A Q \times A Q \mid \text{DEP}_{\text{SSA}}(q(v)^{(l,w)}, q(v')^{(l',w')}, c, w, m, D)\} \]

**Definition 6 (Adaptivity in SSA).** Given a program \( c \), and a meory \( m \), a database \( D \), a starting loop maps \( w \), the adaptivity of the dependency graph \( G_s(c, D, m, w) = (V, E) \) is the length of the longest path in this graph. We denote the path from \( q(v)^{(l,w)} \) to \( q(v')^{(l',w')} \) as \( p_s(q(v)^{(l,w)}, q(v')^{(l',w')}) \).

The adaptivity denoted as \( A_s(c, D, m, w) \).

\[ A_s(c, D, m, w) = \max_{q(v)^{(l,w)}, q(v')^{(l',w')} \in V} \{||p_s(q(v)^{(l,w)}, q(v')^{(l',w')})||\} \]
4.4 Transformation

We build a bridge between the two languages through a transformation in spirit of the work [Vekris et al. 2016]. The command transformation of the form $\Sigma; \delta; c \leftrightarrow c; \delta'; \Sigma'$ translates the labelled command $c$ in the Loop language to its counterpart in SSA loop language. The SSA name environment $\Sigma$, a set of ssa variables already used before the transformation process, is used to generate a fresh SSA variable via a function $\text{fresh}(\Sigma)$. Additionally, translating variables read in the program in the Loop language to its unique SSA variable requires a translation environment $\delta$, a map from variable $x \in \mathcal{V}$ to its SSA form $x \in \mathcal{S}$V. Also, the translation environment $\delta$ and the SSA name environment $\Sigma$ will be updated to $\delta'$ and $\Sigma'$ respectively, along the transformation of the target command. The transformation of the expression is much simpler, of the form $\delta; e \leftrightarrow e'$, which transforms the variables in $e$ to ssa variables stored in the translation environment $\delta$, shown in the rule $s$-var.

We present selected transformation rules in Figure 8. The rules $s$-assn and $s$-query both use $\text{fresh}(\Sigma)$ to generate a new fresh SSA variable $x$ to guarantee the unique assignment of SSA variables. The translation environment is updated with the mapping from variable $x$ in Loop language to the new generated SSA variable $x$ for reference to $x$ used in the future. The SSA name environment is also modified by recording $x$. The transformation of the sequence is standard, with both environments $\delta$ and $\Sigma$ updated during the transformation procedure.

We look at the rule $s$-if by first introducing the binary operation $\bowtie$ on two translation environments $\delta_1$ and $\delta_2$.

$$
\begin{align*}
\delta_1 \bowtie \delta_2 &= \{(x, x_1, x_2) \in \mathcal{V}\mathcal{AR} \times \mathcal{S}\mathcal{V} \times \mathcal{S}\mathcal{V} \mid x \mapsto x_1 \in \delta_1, x \mapsto x_2 \in \delta_2, x_1 \neq x_2\} \\
\delta_1 \bowtie \delta_2 / \bar{x} &= \{(x, x_1, x_2) \in \mathcal{V}\mathcal{AR} \times \mathcal{S}\mathcal{V} \times \mathcal{S}\mathcal{V} \mid x \notin \bar{x} \land x \mapsto x_1 \in \delta_1, x \mapsto x_2 \in \delta_2, x_1 \neq x_2\}
\end{align*}
$$

This operation $\delta_1 \bowtie \delta_2$ combines two translation environments by only keeping the mappings of the same key in both environments. It returns a set of tuples with three elements $(x, x_1, x_2)$ to show that a variable $x$ in the Loop language may be translated to either $x_1$ or $x_2$, depending on the control flow. We use $\bar{x}$ to represent a list of variables $x$, in this sense, the results of $\delta_1 \bowtie \delta_2$ is denoted as $[\bar{x}, \bar{x}_1, \bar{x}_2]$ as follows.

$$
[\bar{x}, \bar{x}_1, \bar{x}_2] = \{(x, x_1, x_2) \mid \forall 0 \leq i < |\bar{x}|, x = \bar{x}[i] \land x_1 = \bar{x}_1[i] \land x_2 = \bar{x}_2[i] \land |\bar{x}| = |\bar{x}_1| = |\bar{x}_2|\}
$$

In the rule $s$-if, a variable $x$ in Loop language may be translated to two possible SSA variables in three cases: (1) the variable $x$ is assigned in both two branches, whose mapping of $x$ is stored in $\delta_1$(then branch) and $\delta_2$(else branch), (2) the variable $x$ is assigned before the if statement (in $\delta$) and only assigned in the then branch $\delta_1$ (3) the variable $x$ is assigned before the if statement (in $\delta$) and only assigned in the else branch $\delta_2$. This corresponds to the aforementioned discussion of the if statement in SSA loop language. We leave these mappings explicitly in the if command of the SSA loop language syntactically. We also use the variant of $\delta = \delta_1 \bowtie \delta_2 / \bar{x}$ to guarantee that the variables stored in $[\bar{y}, \bar{y}_1, \bar{y}_2]$ only appear in the then branch, not in the else branch. Similarly for $[\bar{z}, \bar{z}_1, \bar{z}_2]$. After the transformation, the variable in $x, \bar{y}, \bar{z}$ is replaced with the fresh SSA variables stored in $x, \bar{y}, \bar{z}$ and the translation environment is updated accordingly.

The loop transformation rule $s$-loop deserves a deep discussion. Besides the normal transformation of the loop counter $a$ to $a$, an additional iteration counter is added to its SSA form to support the evaluation, as we have seen in the rule SSA-loop in Figure 7, and is set to 0. For the variables assigned in the loop body $c$, we leave $[\bar{x}'', \bar{x}_1, \bar{x}_2]$ in the transformed SSA loop command, which tracks variables in the loop body whose value may come from two sources: assignment before the loop($\delta$) or assignment in the loop body $\delta_1$. We have two transformations on the body $c$ using the same SSA name environment $\Sigma$ but different translation environments $\delta$ and $\delta_1$. The premise $\Sigma, \Sigma; \delta; c \leftrightarrow c_1; \delta_1; \Sigma_1$ corresponds to the transformation of the loop body in the first iteration with the
variables assigned before the loop execution. The second premise \( \Sigma; \delta; c \leftrightarrow c_2; \delta_1; \Sigma_1 \) corresponds to the transformation in the later iteration with the assigned variables updated by previous execution of the body. Thanks to the extra part \([\bar{x}', \bar{x}_1, \bar{x}_2]\) in the SSA loop command, we know that those variables used in the first iteration are stored in \(\bar{x}_1\) and those updated by the loop body are stored in \(\bar{x}_2\). To finish the SSA transformation, we get the fresh SSA variables \(\bar{x}'\) to replace the appearance of \(\bar{x}_1\) in \(c_1\) or \(\bar{x}_2\) in \(c_2\). We use \(c_1[\bar{x}'/\bar{x}_1]\) and \(c_2[\bar{x}'/\bar{x}_2]\) to represent the replacement, and only the read variables (except for the assigned variables) are replaced. Finally, we get the loop body \(c\) in its SSA form.

### 4.5 The Soundness of Transformation

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

We first define a well defined memory in the Loop language \(m\) or in the SSA loop language \(\bar{m}\) with respect to a transformation environment \(\delta\), denoted as \(m \succeq \delta\) and \(\bar{m} \succeq \delta\) respectively.

**Definition 7 (Well Defined Memory).**

1. \(m \succeq \delta \triangleq \forall x \in \text{dom}(\delta), \exists v, (x, v) \in m\).
2. \(\bar{m} \succeq_{\text{ssa}} \delta \triangleq \forall x \in \text{codom}(\delta), \exists v, (x, v) \in \bar{m}\).

Part of the SSA memory \(\bar{m}\) can also be reverted to a corresponding part of the memory \(m\) with an inverse of \(\delta\).

**Definition 8 (Inverse of Trans Env).** \(m = \delta^{-1}(\bar{m}) \triangleq \forall x \in \text{dom}(\delta), (\delta(x), m(x)) \in \bar{m}\).

We also show that the expression \(e\) in the Loop language and its translated SSA version \(e'\) through some translation environment \(\delta\) evaluates to the same value in Lemma 1.

**Lemma 1 (Value Remains During Transformation).** Given \(\delta; e \leftrightarrow e, \forall m. m \succeq \delta, \bar{m} \succeq_{\text{ssa}} \delta \wedge m = \delta^{-1}(\bar{m}),\) then \((m, e) \rightarrow v\) and \((\bar{m}, e) \rightarrow v\).

Finally, we show the soundness of the transformation. When a program \(c\) is transformed to its SSA form \(c'\) through a transformation environment \(\delta\), when executing these two programs with the corresponding configuration (memories are well-defined w.r.t the transformation environment \(\delta\)), the newly generated traces in the two languages will be the same and the resulting memory \(m'\) and \(\bar{m'}\) will also be related.
We first show the algorithm $VE$. We do not show details of $PS$. We add variables to the global variable list $\mathcal{G}$, consistent with previous estimation, and an input $c$. Given $\delta; c \leftarrow c; \delta'; \Sigma'$, $\forall m. m \in \delta. \forall m. m \in \text{ssa}$, $\delta \land m = \delta^{-1}(m)$, if there exist an execution of $c$ in the Loop language, starting with a trace $t$ and loop maps $w$, $(m, c, t, w) \rightarrow^* (m', \text{skip}, t', w')$, then there also exists a corresponding execution of $c$ in the ssa language so that $(m, c, t, w) \rightarrow^* (m', \text{skip}, t', w')$ and $m' = \delta^{-1}(m')$.

Our dependency graph is constructed based on the trace, we give a lemma that says that the adaptivity remains the same during the transformation.

Lemma 2. Given $\Sigma; \delta; c \leftarrow c; \delta'; \Sigma'$, $\forall m. m \in \delta. \forall m. m \in \text{ssa}$ $\delta \land m = \delta^{-1}(m)$, starting with a trace $t$ and loop maps $w$, then $A(c, D, m, w) = A_s(c, D, m, w)$.

5 THE ANALYSIS ALGORITHM ON SSA PROGRAMS

In this section, we clarify the algorithm $AdaptFun$ that analyzes the adaptivity of a target program in SSA loop language, which consists of three auxiliary algorithms: a variable estimation algorithm $VE$, a matrix-vector based graph generating algorithm $GG$ to generate the weighted variable-based dependency graph, and a path-searching algorithm $PS$ to find the most weighted path in the graph.

We do not show details of $PS$ as we use a standard graph algorithm. We start with the ideas of $AdaptFun$, and illustrate the first two auxiliary algorithms, variable estimation algorithm $VE$ and dependency graph generating algorithm $GG$. In the end, we show the soundness of our algorithm with respect to adaptivity.

5.1 Ideas behind the Algorithm

In consideration of the definition of adaptivity, the longest path in its query-based dependency graph, our analysis targeting a tight upper bound on the adaptivity, is supposed to take care of paths (possible adaptivity candidates) in all the possible dependency graphs (per configuration). To this end, $AdaptFun$ aims to statically construct a weighted directed dependency graph, in which the nodes are annotated SSA variables and directed edges show when one annotated variable may depend on another. The weight of the node shows if the variable is assigned with a query request. Intuitively, every query request in the query-based dependency graph is assigned to variables that appear in the SSA variable-based dependency graph. [DG: I think the last sentence can be deleted.]

The algorithm $AdaptFun$, summarized in Figure 9, estimates the adaptivity from the weighted variable-based dependency graph. Given the input SSA program $c$ to be analyzed, the first step of $AdaptFun$ is the variable estimation via the auxiliary algorithm $VE$, which specifies the nodes of the variable-based dependency graph. The result of $VE$ is stored in a global variable list $G$, fed to the next step. The second step is the graph generation, via a matrix-vector-based graph generating algorithm $GG$. The matrix $M$ records the may-dependency between annotated variables in the global list $G$. It has size $|G| \times |G|$. The vector $V$ has the same size as $G$ and gives a weight to each variable in $G$. This weight is 1 when the variable is assigned with a query request and 0 otherwise. To be precise, the $i$th row, $j$th column of the matrix $M$, written $M[i][j]$, is 1 when there may be a dependency from variable $G[i]$ to $G[j]$. Dually, $M[i][j] = 0$ means no dependency. In a similar way, $V[i] = 1$ means the variable $G[i]$ is assigned with a query request.

The SSA variable-based weighted dependency graph is constructed by the graph generating algorithm $GG$. The final step is to find out the estimated adaptivity – the most weighted path in the graph. We use another auxiliary algorithm $PS$ to find the path with the most weights in the graph.

5.2 Variable Estimation Algorithm

We first show the algorithm $VE$, which adds variables to the global variable list $G$. $VE$ has the form $VE(G; w; c) \rightarrow (G'; w')$, as shown in Figure 10. The input of $VE$ is a list of annotated variables $G$ collected before the program $c$, a loop map $w$ consistent with previous estimation, and an input...
SSA program $c$.

The output of the algorithm is the updated global list $G'$, along with the updated loop maps $w$, for later estimation.

The algorithm adds variables to $G$ at assignments. In the case of expression assignment $\text{ag-asgn}$ and query request $\text{ag-query}$, the output global list is expanded by $x^{(l,w)}$. When it comes to if statements (rule $\text{ag-if}$), variables assigned in both branches, as well as generated variables $\bar{x}, \bar{y}, \bar{z}$ in $[\bar{x}, \bar{x}_1, \bar{x}_2], [\bar{y}, \bar{y}_1, \bar{y}_2], [\bar{z}, \bar{z}_1, \bar{z}_2]$ are all added to $G$. Sequencing $c_1; c_2$ is handled as expected.

For loops, we assume that a loop counter is a natural number (rule $\text{ag-loop}$). The algorithm accounts for variables assigned in every iteration, including newly created variables in $\bar{x}$, and adds them to $G$ with annotations representing the iteration number.

### 5.3 Matrix and Vector based Algorithm

Next, we describe the algorithm $GG$, which takes the list $G$ generated by the previous algorithm. $GG$ has the form: $GG(\Gamma; c; i) \rightarrow (M; V; i')$. The input is a tuple consisting of three elements: (1) a 1-row-N-column matrix $\Gamma$ storing the (annotated) variables on which the execution of the current instruction is control dependent; this is needed for if statements. (2) the SSA program $c$ to be analyzed. (3) an index $i$ specifying the location of the first assigned variable of the program $c$ in the global list $G$. The output of $GG$ also consists of three elements: (1) A matrix $M$ representing the may-dependencies from $c$. (2) A vector $V$ representing the (annotated) variables assigned with queries in $c$. (3) the index $i'$ that refers to the next position of the last assigned variable in $c$, if it exists. The existence of the index $i'$ helps to locate the first assigned variable when we need to analyze the continuation of $c$.

We first define some functions which use the indices in $G$. The function $\text{L}(i)$ generates a 1-column-N-rows matrix, where only the $i$-th row is 1 and all the other rows are 0. This function is used to locate the right row when we calculate the matrix during assignment and query commands.

The function $\text{R}(e, i)$ generates a 1-row-N-column matrix. For every variable used in $e$, it finds the corresponding index $i$ in $G$ so that $G[i]$ maps to the variable and marks the $i$th column as 1.

If the variable is not found, we do not mark it. When we say $G[i]$ maps to a target variable, we take off the annotation of $G[i]$ and check if the left variable with no annotation is the same as the
\[ M = L(i) + (R(e, i) + \Gamma) \quad \text{ad-assign} \]

\[ M = L(i) + (R(e, i) + \Gamma) \quad \text{ad-query} \]

\[ \Gamma \vdash \text{if } q(e) \vdash i \rightarrow (M; V ; i + 1) \]

\[ \Gamma \vdash \text{if } R(b, i_1); c_1; i_1 \rightarrow (M_1; V_1 ; i_2) \quad \Gamma \vdash \text{if } R(b, i_1); c_2; i_2 \rightarrow (M_2; V_2 ; i_3) \]

\[ \Gamma \vdash [\bar{x}, x_1, x_2]; i \rightarrow (M_x; V_x ; i + |\bar{x}|) \quad \Gamma \vdash [\bar{y}, y_1, y_2]; i + |\bar{x}| \rightarrow (M_y; V_y ; i + |\bar{x}| + |\bar{y}|) \]

\[ \Gamma \vdash [\bar{x}, z_1, z_2]; i + |\bar{x}| + |\bar{y}| \rightarrow (M_y; V_y ; i + |\bar{x}| + |\bar{y}|) \quad M = (M_1 + M_2) + M_3 + M_4 + M_z \]

\[ \Gamma \vdash [\bar{x}, x_1, x_2], [\bar{y}, y_1, y_2], [\bar{z}, z_1, z_2], c_1, c_2; i_1 \rightarrow (M; V_1 \cup V_2 ; i_3 + |\bar{x}| + |\bar{y}| + |\bar{z}|) \]

\[ \Gamma \vdash ([c_1; i_1] \rightarrow (M_1; V_1 ; i_2)) \quad \Gamma \vdash ([c_2; i_2] \rightarrow (M_2; V_2 ; i_3)) \quad \text{ad-seq} \]

\[ B = |\bar{x}| \quad A = |c| \quad \forall 0 \leq j < N. \Gamma \vdash [\bar{x}, x_1, x_2]; i + j * (B + A) \rightarrow (M_{ij}; V_{ij} ; i + B + j * (B + A)) \]

\[ \Gamma \vdash c; i + B + j * (B + A) \rightarrow (M_{ij}; V_{ij} ; i + B + A + j * (B + A)) \]

\[ \Gamma \vdash [\bar{x}, x_1, x_2]; i + N * (B + A) \rightarrow (M; V_i ; i + N + (B + A) + B) \]

\[ a = N \quad M' = M + \sum_{0 \leq j < N} (M_{ij} + M_{ij}) \quad V' = V \cup \sum_{0 \leq j < N} (V_{ij} \cup V_{ij}) \quad \text{ad-loop} \]

\[ \Gamma \vdash \text{loop } [a]^j, 0, [\bar{x}, x_1, x_2] \text{ do } c, i \rightarrow (M'; V' ; i + N * (B + A) + B) \]

Fig. 11. The key rules of the graph generating algorithm

target variable. The extra argument \( i \) is used to handle loops. For instance, a variable \( y \) may appear many times in \( G \), but with different annotations (iteration numbers). In this case, \( i \) helps find the most recently assigned variable version of \( y \) in \( G \). It is used when analyzing assignment and query request commands. Thanks to our SSA language, where every variable is assigned once, our choice of the most recent assigned variable is reasonable because the variable used in the loop refers to the most recent assignment of itself.

We define \( M_1; M_2 := M_2 \cdot M_1 + M_1 + M_2 \), where \( M_1 + M_2 \) is the standard sum of two matrices. We also define the operator \( \cup \) to combine two vectors.

\[ V_1 \cup V_2 := \begin{cases} 1 & (V_1[i] = 1 \lor V_2[i] = 1) \land 1 \leq i \leq N \land |V_1| = |V_2| \\ 0 & \text{otherwise} \end{cases} \]

For the sake of brevity, we also use some annotations when the algorithm \( GG \) handles the extra part \([\bar{x}, x_1, x_2]\) in the if and loop statements. First, we give unique names for variables in lists \( \bar{x}, x_1, x_2 \) respectively, as follows:

\[ \forall 0 \leq z < |\bar{x}|. \bar{x}(z) = x_z, x_1(z) = x_{1z}, x_2(z) = x_{2z} \]

We treat every tuple \((x_z, x_{1z}, x_{2z})\) in \([\bar{x}, x_1, x_2]\) as the simple may dependency case: \( x_z \) may depend on both \( x_{1z} \) and \( x_{2z} \), just like \( x_z \leftarrow x_{1z} < x_{2z} \), defined as follows.

\[ \Gamma \vdash [\bar{x}, x_1, x_2]; i \rightarrow (M; V_0 ; i + |\bar{x}|) \quad \forall 0 \leq z < |\bar{x}|. \Gamma \vdash [\bar{x}; x_z \leftarrow x_{1z} + x_{2z}; i + z] \rightarrow (M_{xz}; V_0 ; i + z + 1) \]

where \( M = \sum_{0 \leq z < |\bar{x}|} M_{xz} \).

One key idea of algorithm \( GG \) is to track the indices \( i, i' \) in the input and output to synchronize with the previous algorithm \( VE \): The index in \( GG \) increases in the same way as the global list expands after the analysis of a program \( e \) by \( VE \), which helps \( GG \) record the dependency relation from the program \( e \) in the right place of the matrix. For example, in the cases \text{ad-assign} \ and \text{ad-query}, the index increases by 1, which corresponds to the addition of one variable in the algorithm \( VE \).

The if and loop commands have the extra part \([\bar{x}, x_1, x_2]\) and the output index increases by also considering this part as we do in collecting \( G \) in \( VE \).

We show one simple example \( sa \) to illustrate the construction of the matrix.

\[ sa \triangleq [x_1 \leftarrow 2]^1; [x_2 \leftarrow x_1 + 2]^2; [x_3 \leftarrow x_1 + x_2]^3 \]
We would like to show that the query-based dependency graph generated from the trace of the
execution of the target SSA program is a subgraph of the variable-based dependency graph generated
by AdaptFun, and the total number of queries asked in the program implementing an adaptive data
analysis is also bounded by our algorithm.

We first define when a query-based dependency graph whose nodes are queries is a subgraph
of a variable-based dependency graph whose nodes are variables. Intuitively, this happens when
there is a mapping of nodes from the first graph to the second.
Fig. 12. (a) The two round odd algorithm in labeled Loop language. (b) The SSA program for the same example.

**Definition 10 (Subgraph).** Given a query-based dependency graph $G_s = (V_s, E_s)$, a variable-based dependency graph $G_{ssa} = (V_s, E_s)$, $G_s \subseteq G_{ssa}$ iff: \( \exists f \) so that

1. for every $v \in V_s$, $f(v) \in V_s$.
2. $\forall e = (v_i, v_j) \in E_s$, there exists a path from $f(v_i)$ to $f(v_j)$ in $G_{ssa}$.

We now show the soundness of $\text{AdaptFun}$. We use some new definitions. $G \models M, V$ means that $G$ matches $M, V$ in size. For example, if the cardinality of $G$ is $N$, a matched matrix $M$ is of size $N \times N$ and a good vector $V$ has the same size as $G$. The assumption $G; \omega \models (c, i_1, i_2)$ checks that the variables assigned in $c$ estimated by $VE$ match the variables in $G$ from index $i_1$ to $i_2$.

**Theorem 5.1 (Soundness of $AdaptFun$).** Given $GG(\Gamma; c; i_1) \rightarrow (M; V; i_2)$, for any global list $G$, loop maps $w$ such that $G; w \models (c, i_1, i_2) \wedge G \models (M, V)$, let $K$ be the number of queries made during the execution of the piece of program $c$ and $|V|$ be the number of non-zeros in $V$. Then,

$$K \leq |V| \wedge \forall D, m, G_s(c, D, m, w) \subseteq G_{ssa}(M, V, G, i_1, i_2)$$

**Corollary 5.1.1.** Given $GG(\Gamma; c; i_1) \rightarrow (M; V; i_2)$, for any global list $G$, loop maps $w$ such that $G; w \models (c, i_1, i_2) \wedge G \models (M, V)$,

$$K \leq |V| \wedge A_2(c, D, m, w) \leq \text{Adapt}(M, V, G, i_1, i_2)$$

We have already shown in Lemma 2 that for every transformation of a program $c$ to $c'$, the adaptivity of $c$ and $c'$ is equal. It follows that the bound estimated by $AdaptFun$ on $c$ is an upper bound on the adaptivity of $c$.

6 **MORE EXAMPLES**

**Example 6.1 (Two Round Odd Elements).** We present a variant of the previous two round example in Figure 12. In this odd example, only the data at odd index of the database is used.

This algorithm only touches the odd part of the database, by adding an extra if statement to checking the index $j$ in Figure 12(a). The extra complexity is added to handle the newly generated variables in the loop and if statement in the SSA version in Figure 12(b). The query-based dependency graph does not change a lot compared to the previous two rounds example in Figure 2(c), but the node does change according to the trace. We assume $a = n$ in the final memory is the result of the sum of previous query results in the loop. We give the trace $t = [q(\chi[1])^5, q(\chi[1])^4, q(\chi[3])^5, q(\chi[1])^3, q(n + \chi[3])^5, q(\chi[3])^5, q(n + \chi[3])^5, q(\chi[3])^5, q(n + \chi[3])^5, q(n + \chi[3])^5]$ and use $q_1$ for $q(\chi[1])$, $q_3$ representing for $q(\chi[3])$, $q_4$ for $q(n + \chi[3])$. The query-based dependency graph based on this trace is shown in Figure 13(a). We show the red path, which is a sequence of adaptively chosen queries of length 2. So among the total 4 queries, we have 2-round adaptive queries. According to the Theorem 2.2 and 2.3, we will have a tighter upper bound on the generalization error if we know the adaptivity 2, obtained from the red path in Figure 13(a).
≥

and

we explore further to look at an advanced adaptive data analysis algorithm - multiple round version.

Fig. 13. (a) The query-based dependency graph for odd example (b) The SSA variable-based weighted dependency graph for the same example, the node in red dashed circle is weighted.

The multiple rounds algorithm starts from an initialized empty tracking list $I$, two scores called $Nscore$ $ns = 0$ and $Cscore$ $cs = 0$, initialized to 0. It goes $k$ rounds and at every round, the two scores $ns$ and $cs$ are updated by the result $a$ of a query $q(f(I))$. The function $f(I)$ specifies a complex linear query using the updated tracking list $I$. The tracking list $I$ is updated by the two scores via a function $update(I, ns, cs)$ at every round. This update function mainly compares $ns$ and $cs$, when $ns \geq cs$, certain elements are added to the tracking list $I$. An implementation of the algorithm is presented in Figure 14(a), in which the round number $k$ are set to 3, and we use $update_cscore(a)$ and $update_nscore(a)$ to simplify the complex update on $Cscore$ and $Nscore$ respectively, for the sake of simplicity.

One complexity of the multiple rounds algorithm in comparison with the two rounds one, is that the query asked in each iteration is not independent(non-adaptive) anymore. For example, the query $q^j$ at iteration $j$ now may depend on the tracking list $I$, which comes from the previous iteration $j-1$. Additionally, this list $I$ at iteration $j-1$ is updated by the query result $q^{j-1}$ at the same iteration. Intuitively, we can see the connection between queries from different iterations, which means these queries are adaptively chosen according to our Theorem 2.3.

We first show its query-based dependency graph in Figure 15(a), the execution trace $t_{mr}$ is generated as follows: $t_{mr} = \{ q(f(I_0))^{5[4:1]}, q(f(I_5))^{5[4:2]}, q(f(I_5))^{5[6:3]} \}$. For a better presentation

We discuss closely related work in both areas.

7 RELATED WORK

In terms of techniques, our work relies on ideas from both static analysis and dynamic analysis. We discuss closely related work in both areas.
\[ b \leftarrow 0 \cdot 1; \]
\[ i \leftarrow 1 \cdot 1; \]
\[ \text{loop} \cdot [5] \cdot 3 \cdot \text{do} \]
\[ x \leftarrow q(b + x[i - 1]) \cdot 4; \]
\[ \text{if} \cdot (i \% 2 == 1) \cdot 5 \cdot \]
\[ [b \leftarrow b + x] \cdot 6; \]
\[ [i \leftarrow i + 1] \cdot 8 \]

**Static program analysis.** Our algorithm in Section 5 is influenced by many areas of static program analysis such as effect systems, control-flow analysis, and data-flow analysis [Ryder and Paull 1988]. The idea of statically estimating a sound upper bound for the adaptivity from the semantics is indirectly inspired from prior work on cost analysis via effect systems [Çiçek et al. 2017; Qu et al. 2019; Radicek et al. 2018]. The idea of using an adjacency matrix to reason about data flows as a resource has been also studied as an instance of graded Hoare logic [Gaboardi et al. 2021]. One of the most important ingredients of our work is the estimation of the variable-based dependency graph. There are many ways to construct a dependency graph statically. Some of the most related work focuses on the testing of graphical user interfaces (GUIs), using an event graph. For example, Memon [2007] proposes an event-flow model using an algorithm to construct an event-flow graph, representing all the possible event interactions. This event-flow graph has a vertex for every GUI event such as click-to-paste and an edge between pairs of events that can be performed immediately one after the other. Our variable-based dependency graph uses the edge to track the may-dependence of one variable with respect to another variable. The main difference is in the way the graph is constructed. AdaptFun relies on the structure of the target program, while the event-flow model only considers the event type. Another work [Arlt et al. 2012] constructs a weighted event-dependency graph, capturing data dependencies between events by analyzing bytecode. Every weighted edge indicates a dependency between two events, meaning one event possibly reads data written by the other event, with the weight showing the intensity of the dependency (the quantity of data involved). Our approach of generating the variable-based dependency graph shares the idea of tracking data dependency via static analysis on the source code. However, because of the different domains, we care about assigned variables, and we use the weight in a different way.

**Dynamic program analysis.** Our framework constructs a dependency graph based on the execution trace of a program. We define semantic dependence on this graph by considering (intraprocedural) data and control dependency [Bilardi and Pingali 1996; Cytron et al. 1991; Pollock and Soffa 1989]. One related work [Austin and Sohi 1992] presents a methodology to construct a dynamic dependency graph (DDG) based on the dynamic execution of a program in an imperative language, where edges represent dependency between instructions. Data dependency, control dependency, storage dependency, and resource dependency between instructions are all considered. Our query-based dependency graph only needs data dependency and control dependency between query requests. Critical path length analysis on DDGs is useful for understanding the scope for parallelization, while we use the length of the longest path to define adaptivity. DDGs have been used in many other domains. Nagar and Jagannathan [2018] use DDGs to find serializability violations. Hammer et al. [2006] use similar program dependency graphs [Ferrante et al. 1987] for dynamic
program slicing. They actually use a combination of static and dynamic dependency graphs but in a manner that is different from how we use the two. Their slicing uses both static and dynamic dependency graphs, while we use the dynamic dependency graph as the basis of a definition, which is then soundly approximated by an analysis based on the static dependency graph.

**Generalization in Adaptive Data Analysis.** Starting from the works by Dwork et al. [2015c] and Hardt and Ullman [2014], several works have designed methods that ensure generalization for adaptive data analyses. Some examples are: [Bassily et al. 2016; Dwork et al. 2015a,b; Feldman and Steinke 2017; Jung et al. 2020; Rogers et al. 2020; Steinke and Zakynthinou 2020; Ullman et al. 2018]. Several of these works drew inspiration from the idea of using methods designed to ensure differential privacy, a notion of formal data privacy, in order to guarantee generalization for adaptive data analyses. By limiting the influence that an individual can have on the result of a data analysis, even in adaptive settings, differential privacy can also be used to limit the influence that a specific data sample can have on the statistical validity of a data analysis. This connection is actually in two directions, as discussed for example by Yeom et al. [2018].

Considering this connection between generalization and privacy, it is not surprising that some of the works on programming language techniques for privacy-preserving data analysis are related to our work. Adaptive Fuzz [Winograd-Cort et al. 2017] is a programming framework for differential privacy that is designed around the concept of adaptivity. This framework is based on a typed functional language that distinguish between several forms of adaptive and non-adaptive composition theorem with the goal of achieving better upper bounds on the privacy cost. Adaptive Fuzz uses a type system and some partial evaluation to guarantee that the programs respect differential privacy. However, it does not include any technique to bound the number of rounds of adaptivity. Lobo-Vesga et al. [2021] propose a language for differential privacy where one can reason about the accuracy of programs in terms of confidence intervals on the error that the use of differential privacy can generate. These are akin to bounds on the generalization error. This language is based on a static analysis which however cannot handle adaptivity. The way we formalize the access to the data mediated by a mechanism is a reminiscence of how the interaction with an oracle is modeled in the verification of security properties. As an example, the recent works by Barbosa et al. [2021] and Aguirre et al. [2021] use different techniques to track the number of accesses to an oracle. However, reasoning about the number of accesses is easier than estimating the adaptivity of these calls, as we do instead here.

**8 CONCLUSION AND FUTURE WORKS**

We presented AdaptFun, a program analysis useful to provide an upper bound on the adaptivity of a data analysis under a specific data analysis model. This estimation can help data analysts to control the generalization errors of their analyses by choosing different algorithmic techniques based on the adaptivity. Besides, a key contribution of our works is the formalization of the notion of adaptivity for adaptive data analysis.

In future work, we plan to address some of the limitations of AdaptFun. Our algorithm may over-estimate the adaptivity of a program, as shown in Section 6, due to its path-insensitive nature. We plan in future work to explore the possibility of making AdaptFun path-sensitive. While we believe that in many concrete situations in data analysis requiring a concrete bound for loops is not a strong limitation, we also plan to explore how to add support for dynamic or unbounded loops. To extend our work in this direction we plan to use classical abstraction techniques, at the cost of a more imprecise estimation.


