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, they are designed so 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 choice of which of these techniques to use, often depends on the chain of queries that an adaptive data analysis can generate.

In this work, we consider adaptive data analyses implemented as while-like programs and we design a program analysis which can help with identifying which technique to use to control their generalization error. More specifically, we formalize the intuitive notion of adaptivity as a quantitative property of programs. We do this because the adaptivity level of a data analysis is a key measure to choose the right technique. Based on this definition, we design a program analysis for soundly approximating this quantity. The program analysis generates a representation of the data analysis as a weighted dependency graph, where the weight is an upper bound on the number of times each variable can be reached, and uses a path search strategy to guarantee an upper bound on the adaptivity. We implement our program analysis and show that it can help to analyze the adaptivity 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 the access of an adaptive data analysis to the data by means of queries from some pre-determined...
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.

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 - we will discuss this further in Section 2.

For example, these works showed that when an adaptive data analysis uses a large number of rounds of adaptivity then a low generalization error can be achieved by mechanism of adding to the result of each query Gaussian noise scaled to the number of rounds. When instead an adaptive data analysis uses a small number of rounds of adaptivity then a low generalization error can be achieved by using more specialized methods, such as data splitting mechanism or the reusable holdout technique from Dwork et al. [2015c]. To better understand this idea, we show in Figure 2 two experiments showcasing these situations. More precisely, in Figure 2(a) we show the results of a specific analysis with two rounds of adaptivity. This analysis can be seen as a classifier which first runs 500 non-adaptive queries on the first 500 attributes of the data, looking for correlations between the attributes and a label, and then runs one last query which depends on all these correlations. Without any mechanism the generalization error is pretty large, and the lower generalization error is achieved when the data-splitting method is used. In Figure 2(b), we show the results of a specific analysis with four hundreds rounds of adaptivity. This analysis can be seen as a classifier which

---

1: We will use formally a program implementing this analysis (Figure 3) as a running example in the rest of the paper.
2: We will present this analysis formally in Section 6.
at each step runs an adaptive query based on the result of the previous ones. Again, without any mechanism the generalization error is pretty large, and the lower generalization error is achieved when the Gaussian noise is used.

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 used to help a data analyst in the choice of the mechanism to use, and they 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 can 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.

The second problem we face is how to define the adaptivity of a given program. 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 shown in [Dwork et al. 2015b], and as we did in our example in Figure 2(a), one can design a machine learning algorithm for constructing a classifier which first computes each feature’s correlations with the label via a sequence of queries, and then constructs 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. To formalize this intuition as a quantitative program property, we use a trace semantics recording the execution history of programs on some given input — and we create a dependency graph, where the dependency between different variables (query is also assigned to variable) is explicit and track which variable
is associated with a query request. We then enrich this graph with weights describing the maximal number of times each variable is evaluated in a program evaluation starting with an initial state. The adaptivity is then defined as the length of the walk visiting most query-related variables on this graph.

The third 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 based on some form of dependency analysis. This analysis needs to take into consideration: 1) the fact that, 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, it needs to consider some form of data flow analysis. 2) the fact that, in general, the decision on whether to run a query or not may depend on some other value. Hence, it needs to consider some form of control flow analysis. 3) the fact that, in general, we are not only interested in whether there is a dependency or not, but in the length of the chain of dependencies. Hence, it needs to consider some quantitative information about the program dependencies. To address these considerations and be able to estimate a sound upper bound on the adaptivity of a program, we develop a program analysis algorithm, named \textit{AdaptFun}, which combines data flow and control flow analysis with reachability bound analysis \cite{Gulwani:2010}. This new program analysis gives tighter bounds on the adaptivity of a program than the ones one would achieve by directly using the data and control flow analyses or the ones that one would achieve by directly using reachability bound analysis techniques alone.

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 variable-based dependency graph that is constructed using sets of program execution traces.
3. A static program analysis algorithm \textit{AdaptFun} combining data flow, control flow and reachability bound analysis in order to provide tight bounds on the adaptivity of a program.
4. A soundness proof of the program analysis showing that the adaptivity estimated by \textit{AdaptFun} bounds the true adaptivity of the program.
5. An implementation of \textit{AdaptFun} and an experimental evaluation the bounds this implementation provides on several examples.

2 OVERVIEW

Some results in Adaptive Data Analysis. In Adaptive Data Analysis an analyst is interested in studying some distribution \( P \) over some domain \( \mathcal{X} \). Following previous works \cite{Bassily:2016,Dwork:2015:GII,Hardt: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 \( \text{query} : \mathcal{X} \rightarrow [-1, 1] \) (often other codomains such as \([0, 1]\) or \([-R,+R]\), for some \( R \), are considered). The analyst wants to learn the \textit{population mean}, which (abusing notation) is defined as \( \text{query}(P) = \mathbb{E}_{X \sim P}[\text{query}(X)] \).

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 $\text{query}(P)$. The naïve way to approximate the population mean is to use the empirical mean, which (abusing notation) is defined as $\text{query}(X_1, \ldots, X_n) = \frac{1}{n} \sum_{i=1}^{n} \text{query}(X_i)$. 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 $\text{query}_1, \ldots, \text{query}_k$. 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 $\text{query}_j$ depend on the prefix $\text{query}_1, a_1, \ldots, \text{query}_{j-1}, a_{j-1}$ 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 $\max_{j=1,\ldots,k} |a_j - \text{query}_j(P)| = O\left(\sqrt{\frac{\log k}{n}}\right)$, and for any $r \geq 2$ and any $r$-round adaptive statistical queries, it satisfies $\max_{j=1,\ldots,k} |a_j - \text{query}_j(P)| = O\left(\sqrt{\frac{k}{n}}\right)$

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.

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

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

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. The examples in Figure 2 are experimental results on realistic two rounds and multiple rounds data analysis algorithms, illustrating these theorems.

**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
we multiply this result by some constant, we are also changing the incurred error. Moreover, this
way, by means of data dependency or control flow dependency. In order to find the right definition
of 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
queries and sending this query to the mechanism. More specifically, we assume that, in this example,
the analyst asks in total \( k + 1 \) queries to the mechanism in two phases. In the
first phase, the analyst asks \( k \) queries 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 \( k \) numeric attributes, which we index just by natural numbers. The
queries inside the while loop correspond to the first phase and compute an approximation of the
product of the empirical mean of the first \( k \) 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 \( k \) attributes. Since statistical queries
counts the empirical mean of a function on rows, we use \( \chi \) to abstract a possible row in the
database and queries are of the form \( \text{query}(\psi) \), where \( \psi \) is a special expression (see syntax in
Section 3) representing a function \( X \rightarrow U \) on rows. We use \( U \) to denote the codomain of queries
and it could be \([-1, 1], [0, 1] \) or \([-R, +R] \), for some \( R \) we consider. This function characterizes the
linear query we are interested in running. As an example, \( x \leftarrow \text{query}(\chi[j] \cdot \chi[k]) \) computes an
approximation, according to the used mechanism, of the empirical mean of the product of the \( j^{th} \)
attribute and \( k^{th} \) attribute, identified by \( \chi[j] \cdot \chi[k] \). Notice that we don’t materialize the mechanism
but we assume that it is implicitly run when we execute the query.

Fig. 3. (a) The program \texttt{twoRounds}(k), an example with two rounds of adaptivity (b) The corresponding
execution-based dependency graph (c) The corresponding program-based dependency graph from AdaptFun.

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 (the query \( \text{query}(\chi[j] \cdot \chi[k]) \))
at line 3 only relies on the counter \( j \) and input \( k \), while the last query (at line 6) depends on the
results of 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
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.
To address these challenges, we first build a directed graph representing the possible dependencies between queries of a program and then we define the adaptivity of the program using this graph. We call this graph: execution-based dependency graph. The vertices in the graph represent the variables that are assigned in some command of the program. The edges represent dependency relations between vertices. The dependency relations are computed by looking at the execution traces of a program (see Section 4). Additionally, we add weights to every vertex. The weight of a vertex is a function that given a starting state returns a natural number representing the number of times the vertex is visited when the program is executed starting from this state.

As an example, let us consider the graph in Figure 3(b). This graph is built by considering all the possible execution traces of the program in Figure 3(a). Each vertex in this graph has a superscript representing its weight, and a subscript 1 or 0 telling if the vertex corresponds to a query or not. We will call this subscript a query annotation. For example the vertex \( l_6 : w_1^1 \), has weight \( w_1 \), a constant function which returns 1 for every starting state, since this query at line 6 is at most executed once regardless of the initial trace. The query annotation of this vertex is 1, which indicates that \([l ← \text{query}(\chi[k] \ast a)]\) is a query request. Another vertex, \( x_3 : w_k^k \), appears in the while loop. It has as weight a function \( w_k \) that for every initial state returns the value that \( k \) has in this state, since this is also the number the while loop will be iterated. The node \( j_4 : w_0^0 \) has as a subscript 0 representing a non-query assignment.

Since the edges between two vertices represent the fact that one program variable may depend on the other, we can define the program adaptivity with respect to a initial trace by means of a walk traversing the graph, visiting each vertex no more than its weight with respect to the initial trace, and visiting as many query nodes as possible. So, looking again at our example, we can see that in the walk along the dotted arrows, \( l_6 : w_1^1 \rightarrow a_5 : w_k^k \rightarrow x_3 : w_k^k \), there are 2 vertices with query annotation 1 and that this number is maximal, i.e. we cannot find another walk having more than 2 vertices with query annotation 1, under the assumption that \( k \geq 1 \). So the adaptivity of the program in Figure 3(a) is 2, as expected.

**Static analysis for adaptivity.** Our static analysis provides an upper bound on the adaptivity for this example as follows. AdaptFun constructs a program-based dependency graph, for our example we show this graph in Figure 3(c). The edges of this graph are built by considering both control flow and data flow between assigned variables (the algorithm is presented in Section 5.3.3). The weight of every vertex is estimated by using a reachability-bound estimation algorithm (presented in Section 5.3.2), which can be symbolic and provide a sound upper bound on the weight of the corresponding vertex in the execution-based dependency graph. For instance, the weight \( k \) of the vertex \( x_3 \) in Figure 3(c) is a sound upper bound on the weight \( w_k \) of vertex \( x_3 \) in Figure 3(b), with the same starting trace. The soundness of this step is proved in Theorem 5.1.

AdaptFun search a walk on this graph which overapproximate the adaptivity of the program (this is done by an algorithm AdaptSearch presented in Section 5.4). For instance, in Figure 3(c), AdaptSearch first finds a path \( l_6 : 1 \rightarrow a_5 : 1 \rightarrow x_3 : 1 \), and then approximate a walk with this path. Every vertex on this walk is visited once, and the number of vertices with query annotation 1 traversed in this path is 2, which is the upper bound we expect. It is worth to note here that even...
though the node $x^3$ has weight depending on $k$, it is only visited once, similarly for $l^k$, hence the overall upper bound on the adaptivity is 2, as we expect.

3 Labeled Query While Language

In this section, we formally introduce the language we will focus on for writing data analyses. This is a standard while 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.

3.1 Syntax

### Arithmetic Expression

$$a ::= n | x | a \oplus_a a | \log a | \text{sign } a | \max(a, a) | \min(a, a)$$

### Boolean Expression

$$b ::= \text{true} | \text{false} | \neg b | b \oplus b | a \sim a$$

### Expression

$$e ::= v | a | b | [e, \ldots, e]$$

### Value

$$v ::= n | \text{true} | \text{false} | |] | [v, \ldots, v]$$

### Query Expression

$$\psi ::= a | a \oplus_a \psi | \chi[a]$$

### Query Value

$$\alpha ::= n | \chi[n] | a \oplus_a a | n \oplus_a \chi[n] | \chi[n] \oplus_a n$$

### Label

$$l \in \mathbb{N} \cup \{in, ex\}$$

### Labeled Command

$$c ::= [x \leftarrow e]^I | [x \leftarrow \text{query(}\psi\text{)}]^I | \text{while } [b]^I \text{ do } c$$

We use $VAR, VAL, QVAL, C, DB$ and $QD$ to stand for the set of variables, values, query values, commands, databases and the codomain of queries, respectively.

An expression is either a standard arithmetic expression or a boolean expression, or a list of expressions. Our language supports primitives for queries, where a specific query is specified by a query expression $\psi$. A query expression contains the necessary information for a query request, for example, $\chi[a]$ represents the values at a certain index $a$ in a row $\chi$ of the database. Query expressions combine access to the database with other expressions, for example, $\chi[3] + 5$ represents a query which asks the value from the column 3 of each database raw $\chi$, adds 5 to each of these values, and then computes the average of these values.

A labeled command $c$ is just a command with a label — we assume that labels are unique, so that they can help to identify uniquely every subexpression. We have skip, assignment $x \leftarrow e$, the composition of two commands $c; c$, an if statement if $(b, c, c)$, a while statement while $b$ do $c$. The main novelty of the syntax is the query request command $x \leftarrow q(\psi)$. For instance, if a data analyst wants to ask a simple linear query which returns the first element of the row, they can simply use the command $x \leftarrow q(\chi[1])$ in their data analysis program. We use $\mathcal{L}V$ to represent the universe of all the labeled variables.

3.2 Trace-based Operational Semantics

An event tracks useful information about each step of the evaluation, as a quadruple. Its first element is either an assigned variable (from an assignment command) or a boolean expression (from the guard of if or while command), follows by the label associated to this event, the value evaluated either from the expression assigned to the variable, or the boolean expression in the guard. The last element stores the query information, which a query value whose default is $\bullet$. We declare event projection operator $\pi_i$ which projects the $i$th element from an event.

$$\text{Event } e ::= (x, l, v, \bullet) | (x, l, v, a) \text{ Assignment Event } | (b, l, v, \bullet) \text{ Testing Event}$$

A trace $\tau \in \mathcal{T}$ is a list of events, collecting the events generated along the program execution. $\mathcal{T}$ represents the set of traces. There are some useful operators: the trace concatenation operator $\cdot : \mathcal{T} \rightarrow \mathcal{T} \rightarrow \mathcal{T}$, combines two traces. The belongs to operator $\in: \mathcal{E} \rightarrow \mathcal{T} \rightarrow \{\text{true, false}\}$

and boolean expressions proved in the appendix.

It states that a configuration \((c, \tau)\) to the normal form of the command

\[
\begin{align*}
\theta_{(\tau::(x, l, v, \bullet))} & x \doteq u \\
\theta_{(\tau::(y, l, v, \bullet))} & x \doteq \theta(\tau)x, y \neq x \\
\theta_{(\tau::(b, l, v, \bullet))} & x \doteq \theta(\tau)x \\
\theta_{(\tau::(y, l, v, \alpha))} & x \doteq \theta(\tau)x, y \neq x \\
\theta_{(\tau::(t))} & x \doteq \bot 
\end{align*}
\]

We give a selection of rules of the trace-based operational semantics in Figure 5.

The rule \texttt{assn} evaluates a standard assignment \(x \leftarrow e\), the expression \(e\) is first evaluated by our expression evaluation \((\tau, e) \Downarrow e\), presented below. And the result \(v\) of evaluating \(e\) is used to construct a new event \(e = (x, l, v, \bullet)\) and attach to the previous trace.

\[
\frac{(\tau, a) \rightarrow a \Downarrow e}{(\tau, a) \Downarrow e \Downarrow v} \hspace{1cm} \frac{(\tau, b) \rightarrow b \Downarrow e}{(\tau, b) \Downarrow e \Downarrow v} \hspace{1cm} \frac{(\tau, e_1) \Downarrow e \Downarrow v_1 \cdots (\tau, e_n) \Downarrow e \Downarrow v_n}{(\tau, e_1, \cdots, e_n) \Downarrow e \Downarrow v_1 \cdots \Downarrow v_n} \hspace{1cm} \frac{(\tau, v) \Downarrow e \Downarrow v}{(\tau, v) \Downarrow e \Downarrow v}
\]

The expression evaluation rules also rely on the evaluation of arithmetic expressions \((\tau, a) \rightarrow a \Downarrow v\) and boolean expressions \((\tau, b) \rightarrow b \Downarrow v\). The full rules can be found in the appendix.

Distinguished from the standard assignment evaluation, the rule \texttt{query} evaluates a query requesting command \([x \leftarrow \text{query}(\psi)]\) in two steps. The query expression \(\psi\) is first evaluated into a query value \(\alpha\) by following the rules below. Then, by sending this query request \texttt{query}(\(\alpha\)) to a hidden mechanism, this query is evaluated to a result value returned from it, \(v = \text{query}(\alpha)\). Also, the generated event stores both the query value \(\alpha\) here, and the result value of the query request.

\[
\frac{(\tau, a) \rightarrow a \Downarrow n}{(\tau, a) \Downarrow q \Downarrow n} \hspace{1cm} \frac{(\tau, \psi_1) \Downarrow q \Downarrow \alpha_1}{(\tau, \psi_1 \oplus \alpha_2) \Downarrow q \Downarrow \alpha_1 \oplus \alpha_2} \hspace{1cm} \frac{(\tau, a) \rightarrow a \Downarrow n}{(\tau, \chi[a]) \Downarrow q \Downarrow \chi[n]} \hspace{1cm} \frac{(\tau, \alpha) \Downarrow q \Downarrow \alpha}{(\tau, \psi_2) \Downarrow q \Downarrow \alpha_2} \hspace{1cm} \frac{(\tau, \alpha) \Downarrow q \Downarrow \alpha}{(\tau, \psi_2) \Downarrow q \Downarrow \alpha_2}
\]

The rules for if and while both have two versions, when the boolean expressions in the guards are evaluated to true and false, respectively. In these rules, the evaluation of the guard generates a testing event and the trace is updated as well by appending this event.

If we observe the operational semantics rules, we can find that no rule will shrink the trace. It is proved in the appendix.
4 TRACE-BASED DEPENDENCY AND ADAPATIVITY

In this section, we present our definition of adaptivity on basis of an execution-based dependency graph. The construction of this graph requires us to think about the dependency relation between two queries using what we have at hand - the trace generated in Section 3.

4.1 Design choice of Dependency

In the data analysis model our programming framework supports, we define that a query is adaptively chosen when it is affected by answers of previous queries. The next thing is to decide how do we define whether one query is "affected" by previous answers, with the limited information we have? As a reminder, when the analyst asks a query, the only known information will be the answers to previous queries and the current execution trace of the program.

There are two possible situations that a query will be "affected", either when the query expression directly uses the results of previous queries (data dependency), or when 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).

Since the the results of previous queries can be stored or used in variables which aren’t associated to the query request, it is necessary to track the dependency between queries, through all the program’s variables, and then we can distinguish variables which are assigned with query requests. We give a definition of when one variable may-depend on a previous variable with two candidates.

(1) One variable may depend on a previous variable if and only if a change of the value assigned to the previous variable may also change the value assigned to the variable.

(2) One variable may depend on a previous variable if and only if a change of the value assigned to the previous variable may also change the appearance of the assignment command to this variable during execution.

The first definition is defined as the witness of a variation on the value assigned to the same variable through two executions, according to the change of the value assigned to another variable in pre-trace. In particular for query requests, the variation we observe is on the query value instead of on the query requesting results. In the simple program $c_1 = x ← \text{query}(\chi[2]); y ← \text{query}(\chi[3] + x)$. From our perspective, $\text{query}(\chi[1])$ is different from $\text{query}(\chi[2])$. Informally, we think $\text{query}(\chi[3] + x)$ may depend on the query $\text{query}(\chi[2])$, because equipped function of the former $\chi[3] + x$ may depend on the data stored in $x$ assigned with the result of $\text{query}(\chi[2])$, according to this definition.

Nevertheless, the first definition fails to catch control dependency because it just monitors the changes to a query, but misses the appearance of the query when the answers of its previous queries change. For instance, it fails to handle $c_2 = x ← \text{query}(\chi[1]); if(x > 2, y ← \text{query}(\chi[2]), \text{skip})$, but the second definition can. However, it only considers the control dependency and misses the data dependency. This reminds us to define a may-dependency relation between labeled variables by combining the two definitions to capture the two situations.

4.2 Dependency

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

**Definition 1 (Events Different up to Value (Diff)).** Two events $e_1, e_2 ∈ E$ are Different up to Value, denoted as $\text{Diff}(e_1, e_2)$ if and only if:

\[
\pi_1(e_1) = \pi_1(e_2) \land \pi_2(e_1) = \pi_2(e_2) \\
\land \left( (\pi_3(e_1) \neq \pi_3(e_2) \lor (\pi_4(e_1) = \pi_4(e_2) = \bullet) \lor (\pi_4(e_1) \neq \bullet \land \pi_4(e_2) \neq \bullet \land \pi_4(e_1) \neq_p \pi_4(e_2)) \right)
\]
We compare two events by defining \( \text{Diff}(\epsilon_1, \epsilon_2) \). We use \( \psi_1 \equiv_q \psi_2 \) and \( \psi_1 \neq_q \psi_2 \) to note query expression equivalence and inequivalence, distinct from standard equality. A program \( c \)'s labeled variables is a subset of the labeled variables \( \mathcal{LV} \), denoted by \( \mathcal{LV}(c) \subseteq \mathcal{LV} \). We also define the set of query variables for a program \( c \), \( \mathcal{QV}(c) \). A program \( c \)'s query variables is a subset of its labeled variables, \( \mathcal{QV}(c) \subseteq \mathcal{LV}(c) \). We have the operator \( \text{TL} : \mathcal{T} \to \mathcal{LV} \), which gives the set of labels in every event belonging to the trace. Then we introduce a counting operator \( \text{cnt} : \mathcal{T} \to \mathbb{N} \to \mathbb{N} \), which counts the occurrence of a labeled variable in the trace, whose behavior is defined as follows,

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

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

**Definition 2 (Event May-Dependency).**

An event \( \epsilon_2 \) is in the event may-dependency relation with an assignment event \( \epsilon_1 \in \mathcal{E}^{\text{asn}} \) in a program \( c \) with a hidden database \( D \) and a trace \( \tau \in \mathcal{T} \) denoted as \( \text{DEP}_c(\epsilon_1, \epsilon_2, [\epsilon_1]_{\tau} \ast [\epsilon_2], c, D) \), iff

\[
\exists \tau_0, \tau_1, \tau' \in \mathcal{T}, \epsilon_1' \in \mathcal{E}^{\text{asn}}, c_1, c_2 \in C \cdot \text{Diff}(\epsilon_1, \epsilon_1') \land \\
(\exists \epsilon_2' \in \mathcal{E} \cdot \land \langle c, \tau_0 \rangle \rightarrow^* \langle c_1, \tau_1 \ast [\epsilon_1] \rangle \rightarrow^* \langle c_2, \tau_1 \ast [\epsilon_1] \ast \tau_1 \ast [\epsilon_2] \rangle) \\
(\forall \exists \tau_3, \epsilon_3' \in \mathcal{T}, c_3 \in c \in \mathcal{E}^{\text{test}} \cdot \\
\langle c, \tau_0 \rangle \rightarrow^* \langle c_1, \tau_1 \ast [\epsilon_1] \rangle \rightarrow^* \langle c_2, \tau_1 \ast [\epsilon_1] \ast \tau_1 \ast [\epsilon_2] \rangle \\
\land \text{Diff}(\epsilon_2, \epsilon_2') \land \text{cnt}(\tau, \epsilon_2(\epsilon_2)) = \text{cnt}(\tau', \epsilon_2(\epsilon_2')) \land \epsilon_2 \in \tau_3 \land \epsilon_2 \notin \tau_3')
\]

Our event may-dependency relation of two events \( \epsilon_1 \in \mathcal{E}^{\text{asn}} \) and \( \epsilon_2 \in \mathcal{E} \), for a program \( c \) and hidden database \( D \) w.r.t to a trace \( [\epsilon_1]_{\tau} \ast [\epsilon_2] \). The \( \epsilon_1 \in \mathcal{E}^{\text{asn}} \) is an assignment event because only a change on an assignment event will affect the execution trace, according to our operational semantics. In order to observe the changes of \( \epsilon_2 \) under the modification of \( \epsilon_1 \), this trace \( [\epsilon_1]_{\tau} \ast [\epsilon_2] \) starts with \( \epsilon_1 \) and ends with \( \epsilon_2 \). The may-dependency relation considers both the value dependency and value control dependency as discussed in Section 4.1. The relation can be divided into two parts naturally in Definition 2 (line 2 – 4, 5 – 8 respectively, starting from line 1). The idea of the event \( \epsilon_1 \) may depend on \( \epsilon_2 \) can be briefly described: we have one execution of the program as reference (See line 2 and 6, for the two kinds of dependency). When the value assigned to the first variable in \( \epsilon_1 \) is modified, the reference trace \( \tau_1 \ast [\epsilon_1] \) is modified correspondingly to \( \tau_1 \ast [\epsilon_1] \). We use \( \text{Diff}(\epsilon_1, \epsilon_1') \) at line 1 to express this modification, which guarantees that \( \epsilon_1 \) and \( \epsilon_1' \) only differ in their assigned values and are equal on variable name and label. We perform a second run of the program by continuing the execution of the same program from the same execution point, but with the modified trace \( \tau_1 \ast [\epsilon_1] \) (See line 3, 7). The expected may dependency will be caught by observing two different possible changes (See line 4, 8 respectively) when comparing the second execution with the reference one (similar definitions as in [Cousot 2019]).

In the first part (line 2 – 4 of Definition 2), we witness the appearance of \( \epsilon_1' \) in the second execution, and a variation between \( \epsilon_2 \) and \( \epsilon_2' \) on their values. We have special requirement \( \text{Diff}(\epsilon_2, \epsilon_2') \), which guarantees that they have the same variable name and label but only differ in their evaluated values. In particularly for queries, if \( \epsilon_2 \) and \( \epsilon_2' \) are generated from query requesting, then \( \text{Diff}(\epsilon_2, \epsilon_2') \) guarantees that they differ in their query values rather than the query requesting results. Additionally, in order to handle multiple occurrences of the same event through iterations of the while loop, where \( \epsilon_2 \) and \( \epsilon_2' \) could be in different while loops, we restrict the same occurrence of \( \epsilon_2 \)'s label in \( \tau \) from the first execution with the occurrence of \( \epsilon_2' \)'s label in \( \tau' \) from the second execution, through \( \text{cnt}(\tau, \epsilon_2(\epsilon_2)) = \text{cnt}(\tau', \epsilon_2(\epsilon_2')) \) at line 4.
In the second part (line 5–8 of Definition 2), we witness the disappearance of $e_2$ through observing the change of a testing event $e_b$. To witness the disappearance, the command that generates $e_2$ must not be executed in the second execution. The only way to control whether a command will be executed, is through the change of a guard’s evaluation result in an if or while command, which generates a testing event $e_b$ in the first place. So we observe when $e_b$ changes into $\neg e_b$ in the second execution firstly, whether it follows with the disappearance of $e_2$ in the second trace. We restrict the occurrence of $e_b$’s label in the two traces being the same through $\text{cnt}(\tau',\pi_2(e_b)) = \text{cnt}(\tau,\pi_2(e_b))$ to handle the while loop. Again, for queries, we observe the disappearance based on the query value equivalence.

Considering all events generated during a program’s executions under an initial trace, as long as there is one pair of events satisfying the event may-dependency relation in Definition 2, we say the two related variables satisfy the variable may-dependency relation, in Definition 3.

**Definition 3 (Variable May-Dependency).** A variable $x_1^i \in LV(c)$ is in the variable may-dependency relation with another variable $x_2^j \in LV(c)$ in a program $c$, denoted as $\text{DEP}_{\text{var}}(x_1^i, x_2^j, c)$, if and only if:

$$\exists e_1, e_2 \in E^{\text{asn}}, \tau \in T, D \in DB. \pi_1(e_1)\varpi_2(e_1) = x_1^i \land \pi_1(e_2)\varpi_2(e_2) = x_2^j \land \text{DEP}_{\text{o}}(e_1, e_2, \tau, c, D)$$

### 4.3 Execution Based Dependency Graph

Based on the variable may-dependency relation, we define the execution-based dependency graph.

**Definition 4 (Execution Based Dependency Graph).** Given a program $c$, its execution-based dependency graph $G_{\text{trace}}(c) = (V_{\text{trace}}(c), E_{\text{trace}}(c), W_{\text{trace}}(c), Q_{\text{trace}}(c))$ is defined as follows:

- **Vertices** $V_{\text{trace}}(c) := \{x^i \in LV | x^i \in LV(c)\}$
- **Directed Edges** $E_{\text{trace}}(c) := \{ (x^i, y^j) \in LV(c) \land \text{DEP}_{\text{var}}(x^i, y^j, c) \}$
- **Weights** $W_{\text{trace}}(c) := \{ (x^i, w) | w : T \rightarrow N \land x^i \in LV(c) \land \forall r \in T_0(c), \tau' \in T, (c, \tau) \rightarrow^* (\text{skip}, \tau, \tau') \implies w(\tau) = \text{cnt}(\tau', I) \}$
- **Query Annotation** $Q_{\text{trace}}(c) := \{ (x^i, n) \mid x^i \in LV(c) \land n = 1 \iff x^i \in QV(c) \land n = 0 \iff x^i \notin QV(c) \}$

There are four components of the execution-based dependency graph. The vertices $V_{\text{trace}}(c)$ is the set of program $c$’s labeled variables $LV(c)$, which are statically collected. The query annotation is a set of pairs $Q_{\text{trace}}(c) \in \mathcal{P}(LV \times \{0, 1\})$ mapping each $x^i \in V_{\text{trace}}(c)$ to 0 or 1, indicating whether this labeled variable is in program $c$’s query variable set $QV(c)$. The weights is a set of pairs, $(x^i, w) \in LV \times (T \rightarrow N)$, with a labeled variable as first component and its weight $w$ the second component. Weight $w$ for $x^i$ is a function $w : T \rightarrow N$ mapping from a starting trace to a natural number. When program executes under this starting trace $\tau$, $(c, \tau) \rightarrow^* (\text{skip}, \tau, \tau')$, it generates an execution trace $\tau'$. This natural number is the evaluation times of the labeled command corresponding to the vertex, computed by the counter operator $w(\tau) = \text{cnt}(\tau', I)$. We can see in the execution-based dependency graph of twoRounds in Figure 3(b), the weight of vertices in the while loop is $\theta(\tau)k$, which depends on the value of the user input $k$ specified in the starting trace $\tau$. The directed edges $E_{\text{trace}}(c)$ is also a set of pairs with two labeled variables $(x^i, y^j) \in LV \times LV$, from $x^i$ pointing to $y^j$ in the graph. The edges are constructed directly from our variable may-dependency relation. For any two vertices $x^i$ and $y^j$ in $V_{\text{trace}}(c)$, if they satisfy the variable may-dependency relation $\text{DEP}_{\text{var}}(x^i, y^j, c)$, there is a direct edge between the two vertices in our execution-based dependency graph for program $c$.

In most data analysis programs $c$ we are interested, there are usually some user input variables, such as $k$ in twoRounds. We denote $T_0(c)$ as the set of initial traces in which all the input variables in $c$ are initialized, it is also reflected in $W_{\text{trace}}(c)$. 

4.4 Trace-based Adaptivity

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

**Definition 5 (Finite Walk ($k$)).** Given the execution-based dependency graph $G_{\text{trace}}(c) = (V_{\text{trace}}(c), E_{\text{trace}}(c), W_{\text{trace}}(c), Q_{\text{trace}}(c))$ of a program $c$, a finite walk $k$ in $G_{\text{trace}}(c)$ is a function $k : \mathcal{T} \rightarrow \text{sequence of edges}$. For an initial trace $\tau_0 \in \mathcal{T}_0(c)$, $k(\tau_0)$ is a sequence of edges $(e_1 \ldots e_{n-1})$ for which there is a sequence of vertices $(v_1, \ldots, v_n)$ such that:

- $e_i = (v_i, v_{i+1}) \in E_{\text{trace}}(c)$ for every $1 \leq i < n$.
- every $v_i \in V_{\text{trace}}(c)$ and $(v_i, w_i) \in W_{\text{trace}}(c)$, $v_i$ appears in $(v_1, \ldots, v_n)$ at most $w(\tau_0)$ times.

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

We use $\mathcal{W}K(G_{\text{trace}}(c))$ to denote the set containing all finite walks $k$ in $G_{\text{trace}}(c)$; and $k_{v_1 \rightarrow v_2} \in \mathcal{W}K(G_{\text{trace}}(c))$ with $v_1, v_2 \in V_{\text{trace}}(c)$ denotes the walk from vertex $v_1$ to $v_2$.

We are interested in queries, so we need to recover the variables corresponding to queries from the walk. We define the query length of a walk, instead of counting all the vertices in $k$’s vertices sequence, we just count the number of vertices which correspond to query variables in this sequence.

**Definition 6 (Query Length of the Finite Walk (len$^q$)).** Given the execution-based dependency graph $G_{\text{trace}}(c) = (V_{\text{trace}}(c), E_{\text{trace}}(c), W_{\text{trace}}(c), Q_{\text{trace}}(c))$ of a program $c$, and a finite walk $k \in \mathcal{W}K(G_{\text{trace}}(c))$. The query length of $k$ is a function $\text{len}^q(k) : \mathcal{T} \rightarrow \mathbb{N}$, such that with an initial trace $\tau_0 \in \mathcal{T}_0(c)$, $\text{len}^q(k)(\tau_0)$ is the number of vertices which correspond to query variables in the vertices sequence of the walk $k(\tau_0) (v_1, \ldots, v_n)$ as follows,

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

The definition of adaptivity is then presented in Def 7 below.

**Definition 7 (Adaptivity of a Program).** Given a program $c$, its adaptivity $A(c)$ is function $A(c) : \mathcal{T} \rightarrow \mathbb{N}$ such that for an initial trace $\tau_0 \in \mathcal{T}_0(c)$,

$$A(c)(\tau_0) = \max \{\text{len}^q(k)(\tau_0) \mid k \in \mathcal{W}K(G_{\text{trace}}(c))\}.$$ 

5 THE STATIC ANALYSIS ALGORITHM ON PROGRAMS

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

5.1 A guide to the static program analysis framework

Our program analysis framework AdaptFun can be divided as two steps: 1) to construct a weighted dependency graph based on $c$. 2) to find a path in this graph, which is used to estimate an upper bound on the adaptivity of $c$. 

5.1.1 Graph Estimation. According to adaptivity defined over program’s execution-based dependency graph (in Definition 4), we first build a similar graph to over-approximate that graph. The construction considers the vertices, edges, and the weight of every vertex, as well as some annotations which mark queries. The overall picture of this step is organized as follows.

1. Vertices are the assigned variables with unique labels, extracted directly from the program, see Section 5.2.
2. Query annotations are also decided directly from the program, when there is a query request, the associated variable will be marked as 1, otherwise, 0. See Section 5.2.
3. Every vertex has a weight, which tells the maximal times this vertex can be visited in a possible execution. This weight is estimated by a reachability bound analysis on each vertex. See Section 5.3.2.
4. Edges between vertices consider both control flow and data flow, See Section 5.3.3.
5. Finally, with all the ingredients ready, we construct the final approximated program-based dependency graph in Section 5.4.

Overall, this program-based graph has a similar topology as the execution-based dependency graph. It has the same vertices and query annotation, but approximated edges and weights. We call this generated approximation graph, program-based dependency graph.

5.1.2 Adaptivity Computation. Likewise the adaptivity is defined as a finite walk in the execution based dependency graph, our static estimation on this adaptivity also relies on finding a walk in the program-based dependency graph. We discuss some challenges in finding the ‘appropriate’ walk in the graph, and how our algorithm responds to these challenges as in Section 5.4.

5.2 Vertices Estimation and Query Annotation Estimation
The vertices in the program-based dependency graph are identical to the execution-based dependency graph, which are assigned variables in the program annotated with unique labels,

\[ V_{\text{prog}}(c) \triangleq V_{\text{trace}}(c) \cdot \]

In the same way, the query annotation in program-based dependency graph is identical to the execution-based dependency graph. We define \( Q_{\text{prog}}(c) \triangleq Q_{\text{trace}}(c) \).

5.3 Weight and Edge Estimation
The weight of every vertex in the execution-based graph relies on program’s execution traces. In order to over-approximate the weight statically but still tightly, we present a symbolic reachability bound analysis for estimation of the weight of each vertex(label) in Section 5.3.2, in spirit of some reachability bound techniques.

Since the edges of the execution-based graph of a program relies on the dependency relation, which handles both control flow and data flow, as an over-approximation of this graph, the edges of our program-based dependency graph also cover these two kind of flows. We develop a feasible data flow relation to catch them, in Section 5.3.3.

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

5.3.1 Abstract Execution Control Flow graph. We define an abstract control flow graph for program \( c \), whose vertices are the unique labels from \( c \)’s labeled commands with a special label \( \text{ex} \) for the exit of the program. Each edge \((l_1, dc, l_2)\) of our abstract control flow graph is annotated with a difference constraint, which is used to describe the execution of \( l_1 \). The edge itself talks about the transition between \( l_1 \) to \( l_2 \). Still for the same twoRounds(k) example as overview, its generated
abstract control flow graph is shown in Figure 7(b). For example, the edge \((0, a \leq 0, 1)\) on the top, tells us the command \([a \leftarrow 0]^0\) is executed with next continuation location 1, where the command
\([j \leftarrow k]^1\) will be executed next. The constraint \(a \leq 0\) is a difference constraint, generated by
abstracting from the assignment command \(a \leftarrow 0\), representing that value of \(a\) is less than or equals
to 0 after location 0 before executing command at line 1. The difference constraint is an inequalityelation. The left-hand side of the inequality talks about variables before the execution and the
right-hand side describes those after the execution. The difference constraint \(a < a + x\) on the edge
\((5, a < a + x, 2)\) describes the execution of the command \([a \leftarrow a + x]^3\). The cycle \(2 \rightarrow 4 \rightarrow 5 \rightarrow 2\)
in Figure 7(b). Please also look at the edge from 3 to 4, which talks about the query. The \(x < Q_m\)
describes the execution of a query request (the command at line 3), the query results stored in \(x\)
is bounded by \(Q_m\). \(Q_m\) is the maximal value for query requesting result from the database DB. \(\top\)
means there is no assignment executed, for example, we have the difference constraint \(\top\) on the
dge 2 to 6, means at line 2, there is no assignment (it is a testing guard \(j > 0\).)

5.3.2 Weight Estimation. In order to estimate weight for every vertex in the program-based de-
dependency graph, we perform the symbolic reachability bound analysis on the abstract control
flow graph and add weight as shown in Figure 7(c). We use absW(c) for the computed weights, a
set of pairs \((l, w)\) where \(w\) is the weight for label \(l\) from the abstract control flow graph of \(c\). \(w\)
is an arithmetic expression over \(\mathbb{N}\) and input variables, denoted by \(\mathcal{A}_{in}\). This analysis is inspired
from the program complexity analysis method in [Sinn et al. 2017a]. The detail of our symbolic
reachability bound analysis which uses the difference constraint of the abstract control flow graph
can be found in the appendix. Then we compute the weight for each vertex in \(V_{prog}(c)\), as a set of
pairs \((x^l, w) \in \mathcal{L} \times \mathcal{A}_{in}\) mapping each \(x^l \in V_{prog}(c)\) to an arithmetic expression over \(\mathbb{N}\) and input
variables. Because the vertices in the two graph share the same unique label, the line number of
the same command, we define \(w_{prog}(c)\) as follows,

\[
W_{prog}(c) = \{(x^l, w) \mid x^l \in V_{prog}(c) \land (l, w) \in \text{absW}(c)\}.
\]

We prove that this arithmetic expression for \(x^l \in V_{prog}(c)\) is a sound upper bound of the maximum
visiting times of \(x^l\) over all execution traces of \(c\), with the full proof in the appendix.

**Theorem 5.1 (Soundness of the Reachability Bounds Estimation).** Given a program \(c\) with
its estimated weight \(w_{prog}(c)\) we have:

\[
V(x^l, w) \in W_{prog}, \tau_0 \in T_0(c), \tau \in T, \nu \in \mathbb{N}, \langle c, \tau_0 \rangle \rightarrow^* \langle \text{skip}, \tau_{0 + r} \rangle \land \langle \tau_0, w \rangle \Downarrow_v \nu \implies \text{cnt}(\tau, I) \leq \nu
\]

**Example.** As in Figure 3(c), the weight for \(a^3\) is \(k\), which is a sound estimated weight. For any
initial \(\tau_0 \in T_0(c)\), we know \(\langle \tau_0, k \rangle \Downarrow_v \theta(\tau_0)k\) and the weight \(w_k\) for vertex \(a^3\) from Figure 3(b)
\(w_k(\tau_0) = \theta(\tau_0)k\). In the same way, the weights for all the other vertices are sound.
5.3.3 Edge Estimation. We show how to estimate the directed edges for the program-based dependency graph. We develop a variant of data flow analysis, called Feasible Data-Flow Generation, which produces a sound approximation of the edges in the execution based dependency graph.

Feasible Data-Flow Generation. We generate edges by using both control and data flow in the following flowsTo relation, which uses the results of reaching definition analysis, as RD(l, c) for every label l in a program c. FV computes the set of free variables in an expression.

Definition 8 (Feasible Data-Flow). Given a program c and two labeled variables x, y in this program, flowsTo(x, y, c) is

\[ \text{flowsTo}(x^l, y^l, [x \leftarrow e]^l) \triangleq (x^l, y^l) \in \{(y^l, x^l) | y^l \in \text{FV}(e) \land y^l \in \text{RD}(l, [x \leftarrow e]^l)\} \]
\[ \text{flowsTo}(x^l, y^l, [x \leftarrow \text{query}(\psi)]^l) \triangleq (x^l, y^l) \in \{(y^l, x^l) | y^l \in \text{FV}(\psi) \land y^l \in \text{RD}(l, [x \leftarrow \text{query}(\psi)]^l)\} \]
\[ \text{flowsTo}(x^l, y^l, [\text{skip}]^l) = 0 \]
\[ \text{flowsTo}(x^l, y^l, \text{if}([b]^l, c_1, c_2)) \triangleq \text{flowsTo}(x^l, y^l, c_1) \lor \text{flowsTo}(x^l, y^l, c_2) \]
\[ \lor (x^l, y^l) \in \{(x^l, y^l) | x^l \in \text{FV}(b) \land x^l \in \text{RD}(l, \text{if}([b]^l, c_1, c_2)) \land y^l \in \text{RD}(l, c_1)\} \]
\[ \lor (x^l, y^l) \in \{(x^l, y^l) | x^l \in \text{FV}(b) \land x^l \in \text{RD}(l, \text{if}([b]^l, c_1, c_2)) \land y^l \in \text{RD}(l, c_2)\} \]
\[ \text{flowsTo}(x^l, y^l, \text{while} [b]^l \text{ do } c_w) \triangleq \text{flowsTo}(x^l, y^l, c_w) \lor (x^l, y^l) \in \{(x^l, y^l) | x^l \in \text{FV}(b) \land x^l \in \text{RD}(l, \text{while} [b]^l \text{ do } c_w) \land y^l \in \text{RD}(l, c_w)\} \]
\[ \text{flowsTo}(x^l, y^l, c_1; c_2) \triangleq \text{flowsTo}(x^l, y^l, c_1) \lor \text{flowsTo}(x^l, y^l, c_2) \]

We prove that this Feasible Data-Flow relation is a sound approximation of the variable may-Dependency relation over labeled variables for every program, in the appendix.

Edge Estimation. Then we define the estimated directed edges between vertices in \( \text{V}_{\text{prog}}(c) \), as a set of pairs \( \text{E}_{\text{prog}}(c) \subseteq \mathcal{P}(\mathcal{L}^V \times \mathcal{L}^V) \) indicating a directed edge from the first vertex to the second one in each pair as follows,

\[ \text{E}_{\text{prog}}(c) = \{(y^l, x^l) | y^l, x^l \in \text{V}_{\text{prog}}(c) \land \exists n, z_1^n, \ldots, z_n^n \in \mathcal{L}^V(c). n \geq 0 \land \text{flowsTo}(x^l, z_1^n, c) \land \cdots \land \text{flowsTo}(z_n^n, y^l, c)\} \]

We prove that this estimated directed edge set \( \text{E}_{\text{prog}}(c) \) is a sound approximation of the edge set in \( c \)'s execution-based dependency graph in the appendix.

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

5.4 Adativity Upper Bound Computation

We notate our dependency graph as follows:

\[ \text{G}_{\text{prog}}(c) = (\text{V}_{\text{prog}}(c), \text{E}_{\text{prog}}(c), \text{W}_{\text{prog}}(c), \text{Q}_{\text{prog}}(c)) \]

with \( \text{V}_{\text{prog}}(c), \text{E}_{\text{prog}}(c), \text{W}_{\text{prog}}(c) \) and \( \text{Q}_{\text{prog}}(c) \) as computed in each steps above. Its formal definition can be found in the appendix. We compute the adativity upper bound for a program \( c \) by the maximum query length over all finite walks in the program-based data dependency graph \( \text{G}_{\text{prog}}(c) \), defined formally in Definition 9. We use \( \text{W}^K(\text{G}_{\text{prog}}(c)) \) to represent the walks over the program-based dependency graph for \( c \). Different from the walks on \( \text{G}_{\text{trace}}(c) \), \( k \in \text{W}^K(\text{G}_{\text{prog}}(c)) \) doesn’t rely on the initial trace. The occurrence time of every \( v_i \) in \( k \)'s vertices sequence is bound by an arithmetic expression \( w_i \) where \((v_i, w_i) \in \text{W}_{\text{prog}}(c)\) is \( v_i \)'s estimated weight. Then we have \( \text{len}^q(k) \in \mathcal{A}_{\text{in}} \) as well. The full definition for \( \text{W}^K(\text{G}_{\text{prog}}(c)) \) and \( \text{len}^q \) over \( \text{W}^K(\text{G}_{\text{prog}}(c)) \) is in the appendix.
we use this path to approximate a finite walk, and weight of each vertex as its visiting times, we

\[ x \]

weighted path is

\[ x \]

we are interested in because the weight can be a symbolic expression. Look at the simple example

\[ x \]

is sound with respect to the program-based adaptivity.

\[ \text{AdaptSearch} \]

vertices have query annotation

\[ 1 \]

visit, we will never terminate because we only know

\[ k \]

bound on the program with actual adaptivity

\[ \infty \]

approximation to a large extent. This over-approximation could result in an

\[ \text{AdaptSearch} \]

is a tight bound for this program's adaptivity. This approach is the fundamental of

\[ x \]

dotted arrow in Figure 8, where

\[ x \]

dependency graph, and decrease the weight of

\[ k \]

in Figure 8, where

\[ x \]

Non-Termination Challenge: One naive walk finding method is to simply traverse on this

\[ 1 \]

However, this simple traversing strategy leads to a non-termination dilemma for most programs

\[ 1 \]

we are interested in because the weight can be a symbolic expression. Look at the simple example

\[ k \]

domain \( \mathbb{N} \). If we traverse on the program-based dependency graph, and decrease the weight of \( x^3 \) (the weight \( k \) is symbolic) by one after every visit, we will never terminate because we only know \( k \in \mathbb{N} \).

To solve this non-termination challenge, we switch to another walk finding approach: we first

\[ x \]

find a longest path in the program-based dependency graph and then approximate the walk with

\[ x \]

dependency graph, and decrease the weight of

\[ k \]

in Figure 8, where

\[ x \]

Approximation Challenge: We adopt a deep first strategy to search for the longest weighted

\[ 1 \]

weight, and then use the path to approximate the adaptivity. We find that this gives us over-

\[ 1 \]

to a large extend. This over-approximation could result in an \( \infty \) adaptivity upper

\[ 1 \]

bound on the program with actual adaptivity \( 2 \). Look at twoRounds in overview, we find the longest

\[ 1 \]

weighted path is \( x^3 : k \rightarrow a^5 : k \rightarrow l^6 : 1 \) in Figure 3(c) with weighted query length \( 1 + k \). If

\[ 1 \]

we use this path to approximate a finite walk, and weight of each vertex as its visiting times, we

\[ x \]
Algorithm 1 Adaptivity Computation Algorithm on SCC Graph \( \text{AdaptSearch}_{\text{SCC}}(G) \):

\begin{verbatim}
Require: \( G = (V, E, W, Q) \) \{An Strong Connected program based dependency Graph\}
1: init \( r_{\text{SCC}} \): arithmetic expression, initialized 0, the Adaptivity of this SCC
2: \( \text{visited} : (0,1) \) List, \#\{length \( |V| \), initialize with 0 for every vertex\}
3: \( r : a \), arithmetic expression List, \#\{length \( |V| \), initialize with \( Q(v) \) for every vertex\}
4: flowcapacity: \( a \) List, \#\{length \( |V| \), initialize with \( \infty \) for every vertex\}
5: querynum: INT List, \#\{length \( |V| \), initialize with \( Q(v) \) for every vertex\}.
6: if \( |V| = 1 \) and \( |E| = 0 \) then: return \( Q(v) \)
7: def \( \text{dfs}(G, c, \text{visited}) \):
8:     for every vertex \( v \) connected by a directed edge from \( c \):
9:         if \( \text{visited}[v] = false \):
10:             flowcapacity\[v\] = \( \min(W(v), \text{flowcapacity}[c]) \);
11:             querynum\[v\] = \( \text{querynum}[c] + Q(v) \);
12:             \( r[v] = \max(r[v], \text{flowcapacity}[v] \times \text{querynum}[v]) \);
13:             \( \text{visited}[v] = 1 \); \( \text{dfs}(G, v, \text{visited}) \);
14:     else: \#\{There is a cycle finished, update the adaptivity\}
15:         \( r[v] = \max(r[v], r[c] + \min(W(v), \text{flowcapacity}[c]) \times (\text{querynum}[c] + Q(v))) \);
16:     return \( r[c] \)
17:     for every vertex \( v \) in \( V \); initialize the visited, \( r \), flowcapacity, querynum;
18:     \( r_{\text{SCC}} = \max(r_{\text{SCC}}, \text{dfs}(G, v, \text{visited})) \);
19: return \( r_{\text{SCC}} \)
\end{verbatim}

Adaptivity Computation Algorithm on SCC Graph (\( \text{AdaptSearch}_{\text{SCC}}(G) \)). This algorithm in Alg. 1 takes a subgraph of the program-based dependency graph (SCC), and the output is the adaptivity of the input. For an SCC containing only one vertex, it returns the query annotation of this vertex as adaptivity. For SCC containing at least one edge, there are three steps: 1. find out all the paths in the input 2. calculate the adaptivity of every path using our designed adaptivity counting method. 3. return the maximal adaptivity among all the paths. Because our input graph is SCC, when we start traversing from a vertex, we will finally go back to this vertex. The paths we find in step 1 are all those with the same starting and ending vertex. The most interesting part is step 2. For the SCC containing at least one edge, we compute the adaptivity for each path on the fly of searching.
nestedWhileMultiVarRecAcross(k) ≜

\[
\begin{cases}
i \leftarrow k^0; [x \leftarrow \text{query}(\chi[0])]; [y \leftarrow \text{query}(\chi[1])]; & x^1 = 1 \\
\text{while} \ [i > 0] \text{ do} & \chi^1 = 0 \\
\quad \left(\left[i \leftarrow i - 1; [j \leftarrow k]^5; [y \leftarrow \text{query}(\chi(\ln(x) + y))]^6; \right) & j^0 = 1 \\
\text{while} \ [j > 0] \text{ do} & j^0 = 0 \\
\quad \left(\left[j \leftarrow j - 1; [x \leftarrow \text{query}(\chi(\ln(y) + \chi[x])]^9)\right) & y^9 = 0 
\end{cases}
\]

Fig. 9. (a) The nested while loop example, (b) The program-based dependency graph generated from AdaptFun.

for the paths in the recursion algorithm dfs. It is designed based on a deep first search strategy from line: 6-16. In order to guarantee the visiting times of each vertex by its weight and compute the adaptivity accurately, we use a special parameter flowcapacity to track the minimum weight along the path during the searching procedure, and a parameter querynum to track the total number of vertices with query annotation 1 along the path.

flowcapacity is a list of arithmetic expression \(A_m\) recording the minimum weight when the path reaches that vertex, which is initialized by \(\infty\).

querynum is a list of integer initialized by query annotation \(Q(o)\) for every vertex.

The updating operations during the traverse (line: 8) and at the end of the traverse (line: 11) are interesting, specifically flowcapacity\([v]\) \(\times\) querynum\([v]\) computes the query length for this path. It guarantees the visiting times of each vertex on the path reaching a vertex \(v\) is no more than the maximum visiting times it can be on a qualified walk, through flowcapacity\([v]\), and in the same time compute the query length instead of weighted length accurately through querynum\([v]\). In this way, we resolve the **Approximation Challenge** without losing the soundness, formally in the appendix.

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

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

6.1 Examples

Example 6.1 (Multiple Rounds Algorithm). We look at an advanced adaptive data analysis algorithm - multiple rounds algorithm, as in Figure 10(a). It takes the user input $k$ which decides the number of iterations. It starts from an initialized empty tracking list $I$, goes $k$ rounds and at every round, tracking list $I$ is updated by a query result of $\text{query}(I)$. After $r$ rounds, the algorithm returns the columns of the hidden database $D$ not specified in the tracking list $I$. We use functions $\text{updnscore}(p, a)$, $\text{updcscore}(p, a)$, $\text{update}(I, ns, cs)$ to simplify the complex update computations of $\text{Nscore}$, $\text{Cscore}$ and the tracking list $I$, which will not affect our analysis.

The interesting part here is the query asked in each iteration is not independent any more. The query in one iteration $j$ now depends on the tracking list $I$ from its previous iteration $j−1$, which is updated by the query result in the same iteration $j−1$. The connection between queries from different iterations, which means these queries are adaptively chosen according to our Theorem 2.3.

The program-based dependency graph is presented in Figure 10(b). Its execution-based dependency graph has the same graph, except different weight so we do not show it again. We can simply differ iterations, which means these queries are adaptively chosen according to our Theorem 2.3.

Next, we show AdaptFun providing the tight upper bound for this example. If first finds a path $a^6 : k \rightarrow I^9 : k \rightarrow ns^7 : k$ with three weighted vertices, and then AdaptSearch approximate this path to a walk, in which $a^6, I^9, ns^7$ is visited $k$ times. So the estimated adaptivity is $k$. We know for any initial trace $\tau_0$ where $\langle \tau_0, k \rangle \not\owns v$ and $w_k(\tau_0) = v$. So $k$ from AdaptFun is a tight bound.

Example 6.2 (Linear Regression Algorithm with Gradient Decent Optimization). The linear regression algorithm with gradient decent optimization works well in our AdaptFun as well. Analysis Result: $A_{\text{prog}}(\text{linearRegressionGD}(k, rate)) = k$

This linear regression algorithm aims to model a linear relationship between a dependent variable $y$ and an independent variable $x$, $y = ax + c$, specifically approximating the model parameter $a$ and $c$. In order to have a good approximation on the model parameter $a$ and $c$, it sends query to a training
linearRegressionGD(k, rate) ≜
[a ← 0; [c ← 0]; [j ← k];
while [j > 0] do
    (da ← query(−2 * (x[1] − (x[0] * a + c)) + (x[0]));
    dc ← query(−2 * (x[1] − (x[0] * a + c)));
    [a ← a − rate * da];
    [c ← c − rate * dc];
    [j ← j − 1];)

Fig. 11. (a) The linear regression algorithm (b) The program-based dependency graph from AdaptFun.

Example 6.3 (Over-approximation Algorithm). The AdaptFun comes across an over-approximation on the estimation due to its path-insensitive nature. It occurs when the control flow can be decided in a particular way in front of conditional branches, while the static analysis fails to witness.

We show the over-approximation, in Figure 12(a), we call it a multiple rounds odd iteration algorithm. In this algorithm, at line 5 of every iteration, a query query(y(x)) based on previous query results stored in x is asked by the analyst like in the multiple rounds strategy. The difference is that only the query answers from the even iterations (i = 0, 2, · · · ) are used in the query in line 7, query(y(x)). Because the execution trace only updates x using the query answers in even iterations, so the answers from odd iterations do not affect the queries in even iterations. From the execution-based dependency graph in Figure 12(b), we can see that the weight for the vertex y5 is wk/2. A function which takes any initial trace τ0, return the value of k/2 evaluated in τ0. However, AdaptFun fails to realize that odd iteration will always execute the then branch and even iteration means else branch, so it considers both branches for every iteration. In this sense, the weight estimated for y5 and p6 are both k as in Figure 12(c). As a result, AdaptFun estimates the longest walk from Figure 12(c), y5 → x5 → y5 → · · · → x5 with each vertex visited k times, as the dotted arrows. And the adaptivity computed is 1 + 2 + k, instead of 1 + k.

Example 6.4 (Over-Defined Adaptivity Example). The program’s adaptivity in our formal model, in Definition 7 also comes across an over-approximation on the program’s intuitive adaptivity rounds.
We implemented AdaptFun with the vertex query will be used in next query request variable may-dependency of an abstract control flow graph generation, weight estimation (as presented in Section 5.3.2), edge estimation (as presented in Section 5.3.3) in Ocaml, and the adaptivity computation algorithm shown in Section 5.4 in Python. The OCaml program takes the labeled command as input and outputs the program-based dependency graph, feeds into the python program and the python program provides the adaptivity upper bound and the query number as the final output.

6.2 Implementation Results

It is resulted from difference between its weight calculation and the variable may-dependency definition. It occurs when the weight is computed over the traces different from the traces used in witness the variable may-dependency relation.

As the program in Figure 13(a), which is a variant of the multiple rounds strategy, named multipleRoundSingle(k) with input k. In this algorithm, at line 7 of every iteration, a query query(χ[y] + p) based on previous query results stored in p and y is asked by the analyst like in the multiple rounds strategy. The difference is that only the query answers from the one single iterations (j = k − 2) are used in this query query(χ[y] + p). Because the execution trace updates p using the constant 0 for all the iterations where (j ≠ k − 2) at line 10 after the query request at line 7. In this way, all the query answers stored in p will not be accessed in next query request at line 7 in the iterations where (j ≠ k − 2). Only query answer at one single iteration where (j = k − 2) will be used in next query request query(χ[y] + p) at line 7. So the adaptivity for this example is 2. However, our adaptivity model fails to realize that there is only dependency relation between p^7 and p^7 in one single iteration, not the others. As shown in the execution-based dependency graph in Figure 13(b), there is an edge from p^7 to itself representing the existence of Variable May-Dependency from p^7 on itself, and the visiting times of labeled variable p^7 is w_k(t_0) with a initial trace t_0. As a result, the walk with the longest query length is p^7 → · · · → p^7 → y^4 → z^1 with the vertex p^7 visited w_k(t_0), as the dotted arrows. The adaptivity based on this walk is 2 + w(t_0), instead of 2. Though the AdaptFun is able to give us 2 + k, as an accurate bound w.r.t this definition.

Fig. 12. (a) The multiple rounds odd example (b) The execution-based dependency graph (c) The program-based dependency graph from AdaptFun.

Fig. 13. (a) The multi rounds single example (b) The execution-based dependency graph.
We evaluated this implementation on 17 example programs with the evaluation results shown in Table 1. In this table, the first column is the name of each program. For each program, the second column is its intuitive adaptivity rounds, the third column is the $A(c)$ we defined through our formal semantic model in Section 4. In the third column, we use $k$ represent the weight function $w_k$ (in program’s execution-based dependency graph) which return value of variable $k$ from an initial trace $r_0$, same for natural numbers. The last column is the output of the AdaptFun implementation, which consists of two expressions. The first one is the upper bound for adaptivity and the second one is the upper bound for the total number of query requests in the program.

The first 3 programs we evaluated are twoRoundsComplete($k$), multipleRoundsComplete($k$), and the linearRegressionGD($k$, rate) which we discussed in overview and Section 6. For these examples, $A(c)$ give the accurate adaptivity definition, simultaneously the AdaptFun outputs the tight bounds for both of the adaptivity and query requesting number as expected. But for the forth program multipleRoundsOdd($k$), AdaptFun outputs an over-approximated upper bound $1 + 2 * k$ for the $A(c)$, which is consistent with our expectation as discussed in Example 6.3. The fifth program is the evaluation results for the example in Example 6.4, where AdaptFun outputs the tight bound for $A(c)$ but $A(c)$ is a loose definition of the program’s actual adaptivity rounds. The programs in the table from seq() to nestedWhileMultiPathMultiVarRecAcross($k$) are designed for testing the programs under different possible situations. These programs contain control dependency, data value dependency, the nested while, dependency through multiple variables, dependency across nested loops, and so on. Overall for these examples, our system gives both the accurate adaptivity definition and adaptivity upper bound simultaneously through the dynamic analysis and static analysis. The full programs can be found in the appendix, and the implementation in Github.

<table>
<thead>
<tr>
<th>Program $c$</th>
<th>adaptivity rounds</th>
<th>$A(c)$</th>
<th>AdaptFun</th>
</tr>
</thead>
<tbody>
<tr>
<td>twoRoundsComplete($k$)</td>
<td>2</td>
<td>2</td>
<td>2, $k$</td>
</tr>
<tr>
<td>multipleRoundsComplete($k$)</td>
<td>$k$</td>
<td>$k$</td>
<td>$k$, $k$</td>
</tr>
<tr>
<td>linearRegressionGD($k$, rate)</td>
<td>$k$</td>
<td>$k$</td>
<td>$k$, $2 + k$</td>
</tr>
<tr>
<td>multipleRoundsOdd($k$)</td>
<td>$1 + k$</td>
<td>$1 + k$</td>
<td>$1 + 2 * k$, $1 + 2 * k$</td>
</tr>
<tr>
<td>multipleRoundsSingle($k$)</td>
<td>$2$</td>
<td>$2 + k$</td>
<td>$2 + k$, $2 + k$</td>
</tr>
<tr>
<td>seq()</td>
<td>4</td>
<td>4</td>
<td>4, 4</td>
</tr>
<tr>
<td>seqMultiVar()</td>
<td>4</td>
<td>4</td>
<td>4, 4</td>
</tr>
<tr>
<td>ifValueDependency()</td>
<td>3</td>
<td>3</td>
<td>3, 3</td>
</tr>
<tr>
<td>ifControlDependency()</td>
<td>3</td>
<td>3</td>
<td>3, 3</td>
</tr>
<tr>
<td>whileRec($k$)</td>
<td>$1 + k$</td>
<td>$1 + k$</td>
<td>$1 + k$</td>
</tr>
<tr>
<td>whileMultiVar($k$)</td>
<td>$1 + 2 * k$</td>
<td>$1 + 2 * k$</td>
<td>$1 + 2 * k$, $2 + 3 * k$</td>
</tr>
<tr>
<td>whileValueControlDependency($k$)</td>
<td>$1 + 2 * k$</td>
<td>$1 + 2 * k$</td>
<td>$1 + 2 * k$, $2 + 2 * k$</td>
</tr>
<tr>
<td>whileMultiPathValueControlDependency($k$)</td>
<td>$2 + k$</td>
<td>$2 + k$</td>
<td>$2 + k$, $1 + 2 * k$</td>
</tr>
<tr>
<td>nestWhileValueDependency($k$)</td>
<td>$2 + k^2$</td>
<td>$2 + k^2$</td>
<td>$2 + k^2$, $1 + k + k^2$</td>
</tr>
<tr>
<td>nestedWhileRecAcross($k$)</td>
<td>$1 + 2 * k$</td>
<td>$1 + 2 * k$</td>
<td>$1 + 2 * k$, $1 + k + k^2$</td>
</tr>
<tr>
<td>nestedWhileMultiVarRecAcross($k$)</td>
<td>$1 + k + k^2$</td>
<td>$1 + k + k^2$</td>
<td>$1 + k + k^2$, $2 + k + k^2$</td>
</tr>
<tr>
<td>nestedWhileMultiPathMultiVarRecAcross($k$)</td>
<td>$1 + k + k^2$</td>
<td>$1 + k + k^2$</td>
<td>$1 + k + k^2$, $2 + k + k^2$</td>
</tr>
</tbody>
</table>

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.

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. 2017a;...
Qu et al. 2019; Radicek et al. 2018]. The idea of defining adaptivity using data flow is inspired by the work of graded Hoare logic [Gaboardi et al. 2021], which reasons about data flows as a resource. One of the most important ingredients of our work is the estimation of the program-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 program-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 program-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 to find a finite walk in the graph.

Moreover, the state-of-art data-flow analysis techniques do not consider the quantitative information on how many times each variable is dependent on the other. Our weight estimation is inspired by works in program complexity analysis and worst case execution time analysis areas, focusing on analyzing the cost of the entire program. The techniques are based on type system [Çiçek et al. 2017b; Rajani et al. 2021], Hoare logic [Carbonneaux et al. 2015], abstract interpretation [Gustafsson et al. 2005; Humenberger et al. 2018], invariant generation through cost equations or ranking functions [Albert et al. 2008; Alias et al. 2010; Brockschmidt et al. 2016; Flores-Montoya and Hähnle 2014] or a combination of program abstraction and invariant inferring [Gulwani et al. 2009; Gulwani and Zuleger 2010; Sinn et al. 2017b]. In general, these techniques give the approximated upper bound of the program’s total running time or resource cost. However, they failed to consider the case where the cost—the adaptivity—could decrease when there isn’t a dependency relation between variables.

**Dynamic program analysis.** Our framework constructs a execution-based dependency graph based on the execution traces 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 execution-based dependency graph only needs data dependency and control dependency between variable assignment results. DDGs have been used in many other domains. Nagar and Jagannathan [2018] use DDGs to find serializability violations. Hammer et al. [2006a] use similar program dependency graphs [Ferrante et al. 1987] for dynamic program slicing. [Hammer et al. 2006b] propose ways of constructing different kinds of program slices, by choose different program dependency. 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.
Our execution-based data dependency relation definition over variables is inspired by the method in [Cousot 2019], where the dependency relation is also identified by looking into the differences on two execution traces. However, Cousot excludes timing channels [Sabelfeld and Myers 2003] and empty observation, which are also not considered as a form of dependency in traditional dependency analysis [Denning and Denning 1977]. Our definition includes timing channels and empty observation by observing both the disappearance and value variation.

**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, as well as the total number of queries asked. 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. Also, our implementation shows the potential application of our work into real world.

In the future, we plan to address the over-approximation 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 to explore the possibility of making AdaptFun path-sensitive. We also see the over-approximation when we estimate weight in Section 5.3.2 in some complicated examples with nested while loops, the corresponding improvement is also in our plan.
REFERENCES


