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 a 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 errors. 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. [22, 29]). 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 [22]. 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 [22].

A line of work initiated by Dwork et al. [18], Hardt and Ullman [28] 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 family (we will consider here a specific family of queries often called "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 Fig. 1. This line of work has identified

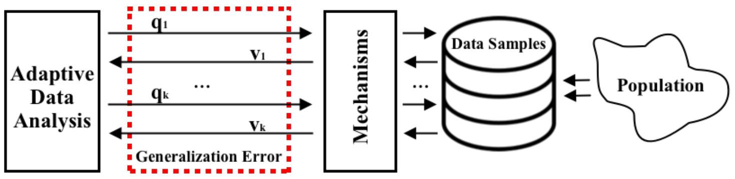


Fig. 1. Overview of our Adaptive Data Analysis model. We have a population that we are interested in studying, and a dataset containing individual samples from this population. The adaptive data analysis we are interested in running has access to the dataset through queries of some pre-determined family (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.

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, using sampling methods, 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. [7, 10, 14, 16, 17, 19, 31, 35, 39, 40].

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 a mechanism 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. [18]. To better understand this idea, we show in Fig. 2 three experiments showcasing these situations. More precisely, in Fig. 2(a) we show the results of a specific analysis[1] with two rounds of adaptivity. This analysis can be seen as a classifier which first runs 400 non-adaptive queries on the first 400 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 of the last query is pretty large, and the lower generalization error is achieved when the data-splitting method is used. Fig. 2(c) shows how this situation also changes with the number of queries. Specifically, it shows the root mean square error of the last *adaptive* query when the number of queries varies. This also highlights the fact that different mechanisms, for the same analysis, produce results with very different generalization errors. In Fig. 2(b), we show the results of a specific analysis[2] with four hundreds rounds of adaptivity. At each step, this analysis runs an adaptive query based on the results of the

[1]We will use formally a program implementing this analysis (Fig. 3) as a running example in the rest of the paper.
[2]We will present this analysis formally in Section 6.

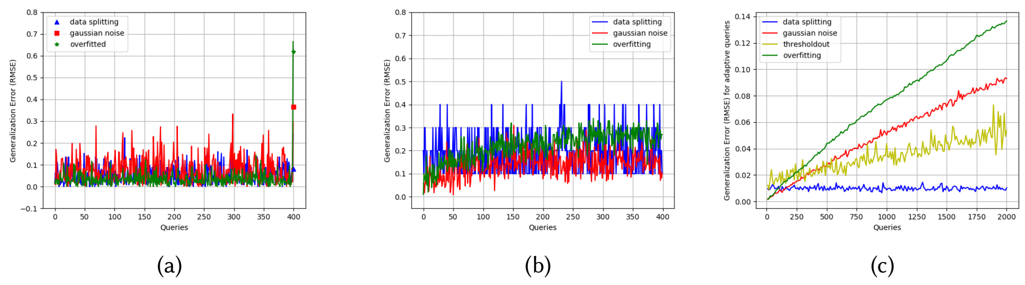


Fig. 2. The generalization errors of two adaptive data analysis examples, under different choices of mechanisms. (a) Data analysis with 2 rounds adaptivity, (b) Data analysis with 400 rounds adaptivity. (c) Same one as (a)

previous ones. Without any mechanism, the generalization error of most of the queries is pretty large, and this error can be lowered by using Gaussian noise.

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 ultimately be integrated into a tool for adaptive data analysis such as the *Guess and Check* framework by Rogers et al. [35].

The first problem we face is *how to formally define* a model for adaptive data analysis which is general enough to support the methods we discussed above and which 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 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 error of the query results 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 [17], and as we did in our example in Fig. 2(a), one can design a machine learning algorithm for constructing a classifier which first computes each feature's correlation 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 queries. 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 (queries are also assigned to variables) is explicit and tracks which variable is associated with a query request. We then enrich this graph with weights describing the 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[3]. In other words, we define adaptivity as a *quantitative form of program dependency*.

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 static program analysis algorithm, named AdaptFun, which combines data flow and control flow analysis with reachability bound analysis [25]. This combination 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. We evaluate AdaptFun on a number of examples showing that it is able to efficiently estimate precise upper bounds on the adaptivity of different programs. All the proofs and extended definitions can be found in the supplementary material.

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:

(1) A programming framework for adaptive data analyses where programs represent analysts that can query generalization-preserving mechanisms 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 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 AdaptFun bounds the true adaptivity of the program.
(5) A prototype implementation of AdaptFun and an experimental evaluation showing the accuracy and efficiency of the adaptivity estimation provided by this implementation on several examples. We also provide an evaluation showing how the generalization error of several real-world data analyses can be effectively reduced by using the information provided by AdaptFun.

2 OVERVIEW

2.1 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 [7, 18, 28], 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 $\mathtt{query} : \mathcal{X} \to [-1, 1]$ (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 $\mathtt{query}(P) = \underset{X \sim P}{\mathbb{E}}[\mathtt{query}(X)]$. We assume that the distribution P can

[3]Formally, graphs will be well-defined only for terminating programs, this will guarantee that the walk is finite

only be accessed via a set of *samples* $X_1, \dots, X_n$ drawn independently and identically distributed (i.i.d.) from P. These samples are held by a mechanism $M(X_1, \dots, X_n)$ who receives the query $\texttt{query}$ and computes an answer $a \approx \texttt{query}(P)$. The naïve way to approximate the population mean is to use the *empirical mean*, which (abusing notation) is defined as $\texttt{query}(X_1, \dots, X_n) = \frac{1}{n} \sum_{i=1}^{n} \texttt{query}(X_i)$. However, the mechanism M can adopt some methods for improving the generalization error $|a - \texttt{query}(P)|$.

In this work we consider analysts that ask a sequence of k queries $\texttt{query}_1, \dots, \texttt{query}_k$. If the queries are all chosen in advance, independently of the answers of each other, then we say they are *non-adaptive*. If the choice of each query $\texttt{query}_j$ depends on the prefix $\texttt{query}_1, a_1, \dots, \texttt{query}_{j-1}, a_{j-1}$ 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-adaptive queries are 1-round and fully adaptive queries are k-round adaptive.

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

Theorem 2.1 ([7]). *(1) For any distribution P, and any k* non-adaptive *statistical queries,*
$$\max_{j=1,\dots,k} |a_j - \texttt{query}_j(P)| = O\left(\sqrt{\frac{\log k}{n}}\right).$$
(2) For any distribution P, and any k r-round adaptive *statistical queries, with $r \geq 2$, the empirical mean (rounded to an appropriate number of bits of precision)*[4] *satisfies:*
$$\max_{j=1,\dots,k} |a_j - \texttt{query}_j(P)| = O\left(\sqrt{\frac{k}{n}}\right)$$

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, from $\log k$ to k.

Dwork et al. [18] and Bassily et al. [7] showed that by using carefully calibrated Gaussian noise 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 ([7, 18]). *For any distribution P, any k, any $r \geq 2$ and any r*-round adaptive *statistical queries, if we answer queries with carefully calibrated Gaussian noise we have:*
$$\max_{j=1,\dots,k} |a_j - \texttt{query}_j(P)| = O\left(\frac{\sqrt[4]{k}}{\sqrt{n}}\right)$$

More interestingly, Dwork et al. [18] also gave a refined bounds that can be achieved with different mechanisms depending on the number of rounds of adaptivity.

Theorem 2.3 ([18]). *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*
$$\max_{j=1,\dots,k} |a_j - \texttt{query}_j(P)| = O\left(\frac{r\sqrt{\log k}}{\sqrt{n}}\right)$$

Notice that Theorem 2.3 has different quantification in that the optimal choice of mechanism depends on the number of queries and number of rounds of adaptivity. This suggests that if one knows a good *a priori upper bound on the number of rounds of adaptivity*, one can choose the appropriate mechanism and get a much better guarantee in terms of the generalization error. As an example, as we can see in Fig. 2, if we know that an algorithm is 2-rounds adaptive, we can choose data splitting as the mechanism, while if we know that an algorithm has many rounds of adaptivity we can choose Gaussian noise. It is worth to stress that by knowing the number of rounds of adaptivity one can also compute a concrete upper bound on the generalization error of a data analysis. This information allows one to have a quantitative, a priori, estimation of the

[4]With infinite precision even two queries may give unbounded error, when the first query's result encodes the whole data.

```
twoRounds(k) ≜
$[a \leftarrow 0]^0; [j \leftarrow k]^1;$
while $[j > 0]^2$ do $\left([x \leftarrow \texttt{query}(\chi[j] \cdot \chi[k])]^3;\ [j \leftarrow j - 1]^4; [a \leftarrow x + a]^5\right);$
$[l \leftarrow \texttt{query}(\chi[k] \cdot a)]^6$
```

(a)

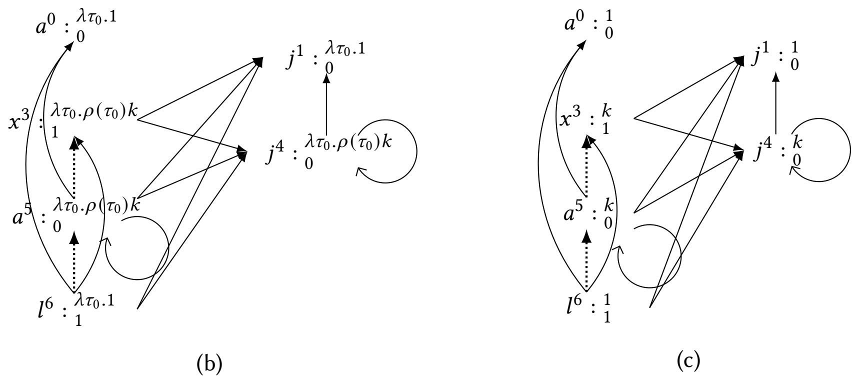

$a^0 : \lambda\tau_0.1$, 0
$j^1 : \lambda\tau_0.1$, 0
$x^3 : \lambda\tau_0.\rho(\tau_0)k$, 1
$j^4 : \lambda\tau_0.\rho(\tau_0)k$, 0
$a^5 : \lambda\tau_0.\rho(\tau_0)k$, 0
$l^6 : \lambda\tau_0.1$, 1
(b)
$a^0 : 1$, 0
$j^1 : 1$, 0
$x^3 : k$, 1
$j^4 : k$, 0
$a^5 : k$, 0
$l^6 : 1$, 1
(c)

Fig. 3. (a) The program twoRounds(k), an example with two rounds of adaptivity (b) The corresponding semantics-based dependency graph (c) The estimated dependency graph from AdaptFun.

effectiveness of a data analysis. This motivates us to design a static program analysis aimed at giving good *a priori* upper bounds on the number of rounds of adaptivity of a program.

2.2 AdaptFun formally through an example.

We illustrate the key technical components of our framework through a simple adaptive data analysis with two rounds of adaptivity. In this analysis, an analyst asks $k+1$ queries to a 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. The mechanism is abstract here and our goal is to use static analysis to provide an upper bound on adaptivity to help choose the mechanism. This data analysis assumes that the data domain $\mathcal{X}$ contains at least k numeric attributes (every query in the first phase focuses on one), which we index just by natural numbers. The implementation of this data analysis in the language of AdaptFun is presented in Fig. 3(a).

The AdaptFun language extends a standard while language[5] with a query request constructor denoted query. Queries have the form query(ψ), where ψ is a special expression (see syntax in Section 3) representing a function : $\mathcal{X} \to U$ 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. Indeed, as we discussed in the previous section, linear queries compute the empirical mean of a function on rows — we use χ to abstract a possible row in the database. As an example, $x \leftarrow$ 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. In Fig. 3(a), the queries inside the while loop correspond to the first phase of the data analysis 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

[5]Programs components are labeled, so that we can uniquely identify every component.

computes an approximation of the empirical mean where each record is weighted by the sum of the empirical mean of the first k attributes.

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 $\chi[j] \cdot \chi[k]$ 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.

2.2.1 Adaptivity definition. The central property we are after in this work is the *adaptivity of a program*. We define formally this notion in three steps, which we will describe in details in Section 4. First, we define a notion of dependency, or better *may-dependency*, between variables. To do this we take inspiration from previous works on dependency analysis and information flow control and we say that a variable *may depend* on another one if changing the execution of the latter can affect the execution of the former. We can see in Fig. 3(a) that the value of the variable l, which corresponds to the result of the execution of the query in the second phase (in the command with label 6), is affected by the value of the variable x, which corresponds to the result of the execution of the query at line 3 in the first phase, via the variable a. To formally define this notion of dependency, as in information flow control, we use the execution history of programs recorded by a trace semantics (see Definition 3).

Second, we build an annotated weighted directed graph representing the possible dependencies between labeled variables. We call this graph *semantics-based dependency graph* to stress that this graph summarizes the dependencies we could see if we knew the overall behavior of the program. The vertices of the graph are the assigned program variables with the label of their assignments, edges are pairs of labeled variables which satisfy the dependency relations, weights are functions associated with vertices and describe the number of times the assignment corresponding to the vertex is executed when the program is run in a given starting state[6], and the annotations, which we call *query annotations*, are bits associated with vertices and describe if the corresponding assignment comes from a query (1) or not (0). The *semantics-based dependency graph* of the `twoRounds(k)` program we gave in Fig. 3(a) is described in Fig. 3(b) (we use dashed arrows for two edges that will be highlighted in the next step, for the moment these can be considered similar to the other edges—i.e. solid arrows). We have all the variables that are assigned in the program with their labels, and edges representing dependency relations between them. For example, we have two edges (l^6, a^5) and (a^5, x^3) describing the dependency between the variables assigned by queries. The vertices l^6 and x^3 are the only ones with query annotation 1 (the subscript), since they are the only two variables that are in assignments involving queries. Notice that the graph contains cycles—in this example it contains two self-loops. These cycles capture the fact that the variables a^5 and j^4 are updated at every iteration of the loop using their previous values. Cycles are essential to capture mutual dependencies like the ones that are generated in loops. Adaptivity is a quantitative notion, so capturing this form of dependencies is not enough. This is why we also use weights. The weight of a vertex is a function that given an initial state returns a natural number representing the number of times the assignment corresponding to a vertex is visited during the program execution starting in this initial state. For example, the vertex l^6 has weight $\lambda\tau.1$ since for every initial state τ the corresponding assignment will be executed one time, the vertex a^5 on the other hand has weight $\lambda\tau.\rho(\tau)k$ since the corresponding assignment will be executed a number of times that correspond to the value of k in the initial state τ, and ρ is the operator reading value of k from τ.

[6]In our trace semantics the state is recorded in the trace, so an initial state is actually represented by an initial trace. We will use this terminology in later sections.

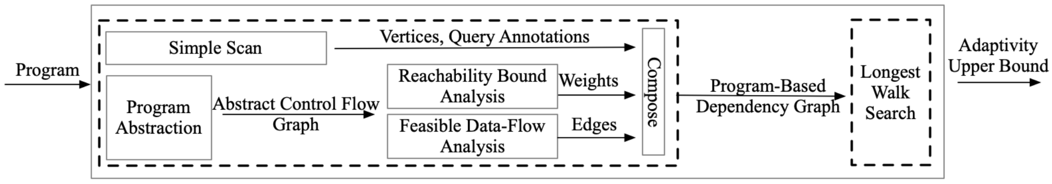

Program
Simple Scan
Vertices, Query Annotations
Program Abstraction
Abstract Control Flow Graph
Reachability Bound Analysis
Weights
Feasible Data-Flow Analysis
Edges
Compose
Program-Based Dependency Graph
Longest Walk Search
Adaptivity Upper Bound

Fig. 4. The overview of AdaptFun

Third, we can finally define adaptivity using the semantics-based dependency graph. We actually define this notion with respect to an initial state τ, since different states can give very different adaptivities. We consider the walk that visits each vertex v of the semantics-based dependency graph no more than the value that the weight w_v applying to the initial state τ, and visits the maximal number of query nodes. The number of query nodes visited is the adaptivity of the program with respect to τ. Looking again at Fig. 3(b), and assuming that $\tau(k) \geq 1$, we can see that the walk along the dashed arrows, $l^6 \to a^5 \to x^3$ has two vertices with query annotation 1, and we cannot find another walk having more than 2 vertices with query annotation 1. So the adaptivity of the program in Fig. 3(a) with respect to τ is 2. If we consider an initial state τ such that $\tau(k) = 0$ we have that the adaptivity with respect to τ is instead 1.

2.2.2 Static analysis. To compute statically a sound and accurate upper bound on the *adaptivity* of a program c, we design a program analysis framework named AdaptFun which we will describe formally in 5. The structure of AdaptFun (Fig. 4) reflects in part the definition of adaptivity we discussed in the previous section. Specifically, AdaptFun is composed by two algorithms (the ones in dashed boxes in the figure), one for building a dependency graph, which we call *estimated dependency graph*, and the other to estimate the adaptivity from this graph. The first algorithm, which we will describe formally in Section 5, generates the *estimated dependency graph* using several program analysis techniques. Specifically, AdaptFun extracts the vertices and the query annotations by looking at the assigned variables of the program, it estimates the edges by using control flow and data flow analysis, and it estimates the weights by using symbolic reachability-bound analysis—weights in this graph are symbolic expressions over input variables. The second algorithm estimates the walk which respects the weights and which visits the maximal number of query nodes. The two algorithms together gives us an upper bound on the program's *adaptivity*.

We show in Fig. 3(c) the estimated dependency graph that our static analysis algorithm returns for the program `twoRounds(k)` in Fig. 3(a). Vertices and query annotations are the same as the ones in Fig. 3(b) and they are simply inferred by scanning the program. As we said before, the edges are estimated using control flow and data flow analysis. For the `twoRounds(k)` example, every edge in Fig. 3(b) is precisely inferred by our combined analysis, this is why Fig. 3(c) contains exactly the same edges. The weight of every vertex is computed using a reachability-bound estimation algorithm which outputs a symbolic expression over the input variables, in the example only k, representing an upper bound on the number of times each assignment is executed. For example, consider the vertex x^3, its weight is k and this provides an upper bound on the value returned by the weight function $\lambda\tau.\rho(\tau)k$ associated with vertex x^3 in Fig. 3(b) for any initial state.

The algorithm searching for the walk first finds a path $l^6 : {}^1_1 \to a^5 : {}^k_1 \to x^3 : {}^k_1$, and then constructs a walk based on this path. Every vertex on this walk is visited once, and the number of vertices with query annotation 1 in this walk is 2, which is the upper bound we expect. It is worth noting here that x^3 and a^5 can only be visited once because there isn't an edge to go back to them, even though they both have the weight k. In this sense, instead of simply computing the weighted length of this path $(2k + 1)$ as adaptivity, the algorithm AdaptBD computes the upper bound 2. Note that 2 is not always tight, for example when $k = 0$.

3 LABELED QUERY WHILE LANGUAGE

The language of AdaptFun is a standard while language with labels to identify different components and with primitives for queries, and equipped with a trace-based operational semantics which is the main technical tool we will use to define the program's adaptivity.

$$
\begin{array}{llll}
\text{Arithmetic Expression} & a & ::= & n \mid x \mid a \oplus_a a \mid \log a \mid \mathtt{sign}\ a \mid \max(a,a) \mid \min(a,a) \\
\text{Boolean Expression} & b & ::= & \mathtt{true} \mid \mathtt{false} \mid \neg b \mid b \oplus_b b \mid a \sim a \\
\text{Expression} & e & ::= & v \mid a \mid b \mid [e, \dots, e] \\
\text{Value} & v & ::= & n \mid \mathtt{true} \mid \mathtt{false} \mid [] \mid [v, \dots, v] \\
\text{Query Expression} & \psi & ::= & \alpha \mid a \mid \psi \oplus_a \psi \mid \chi[a] \\
\text{Query Value} & \alpha & ::= & n \mid \chi[n] \mid \alpha \oplus_a \alpha \mid n \oplus_a \chi[n] \mid \chi[n] \oplus_a n \\
\text{Label} & l & \in & \mathbb{N} \cup \{\mathtt{in}, \mathtt{ex}\} \\
\text{Labeled Command} & c & ::= & [x \leftarrow e]^l \mid [x \leftarrow \mathtt{query}(\psi)]^l \mid \mathtt{while}\ [b]^l\ \mathtt{do}\ c \\
 & & & \mid c; c \mid \mathtt{if}\ ([b]^l, c, c) \mid [\mathtt{skip}]^l
\end{array}
$$

Expressions include standard arithmetic (with value $n \in \mathbb{N} \cup \{\infty\}$) and boolean expression, (a and b) and extended query expressions ψ. A query expression ψ can be either a simple arithmetic expression a, an expression of the form $\chi[a]$ where χ represents a row of the database and a represents an index used to identify a specific attribute of the row χ, a combination of two query expressions, $\psi \oplus_a \psi$, or a normal form α. For example, the query expression $\chi[3] + 5$ denotes the computation that obtains the value in the 3rd column of χ in one row and then adds 5 to it. Command are the typical ones from while languages with an additional command $x \leftarrow \mathtt{query}(\psi)$ for query requests which can be used to interrogate the database and compute the linear query corresponding to ψ. Each command is annotated with a label l, we will use natural numbers as labels and we will use them to record the location of each command, so that we can uniquely identify them. We also have a set $\mathcal{LV}$ of labeled variables, these are simply variables with a label. We denote by $\mathbb{LV}(c)$ the set of labeled variables which are assigned in an assignment command in the program c. We denote by $\mathtt{QV}(c)$ the set of labeled variables that are assigned to the result of a query in the program c.

3.1 Trace-based Operational Semantics

We use a trace-based operational semantics tracking the history of program execution. The operational semantics is parameterized by a database that can be accessed only through queries. Since this database is fixed, we omit it from the semantics but it is important to keep in mind that this database exists and it is what allows us to evaluate queries. A *trace* τ is a list of *events* generated when executing specific commands. We denote by $\mathcal{T}$ the set of traces and we will use list notation for traces, where $[]$ is the empty trace, the operator :: combines an event and a trace in a new event, and the operator $++$ concatenates two traces.

We have two kinds of events: *assignment events* and *testing events*. Each event consists of a quadruple, and we use $\mathcal{E}^{\mathsf{asn}}$ and $\mathcal{E}^{\mathsf{test}}$ to denote the set of all assignment events and testing events, respectively.

$$
\begin{array}{llll}
\text{Event} & \epsilon & ::= & (x, l, v, \bullet) \mid (x, l, v, \alpha) & \text{Assignment Event} \\
 & & & \mid (b, l, v, \bullet) & \text{Testing Event}
\end{array}
$$

An assignment event tracks the execution of an assignment or a query request and consists of the assigned variable, the label of the command that generates it, the value assigned to the variable, and the normal form α of the query expression that has been requested, if this command is a query request, otherwise a default value $\bullet$. A testing event tracks the execution of if or while command and consists of the guard of the command, the label of the command, the result of evaluating the

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

$$\frac{\langle \tau, e \rangle \Downarrow_e v \qquad \epsilon = (x, l, v, \bullet)}{\langle [x \leftarrow e]^l, \tau \rangle \to \langle [\mathtt{skip}]^l, \tau{::}\epsilon \rangle}\ \textbf{assn} \qquad \frac{\tau, \psi \Downarrow_q \alpha \qquad \mathtt{query}(\alpha) = v \qquad \epsilon = (x, l, v, \alpha)}{\langle [x \leftarrow \mathtt{query}(\psi)]^l, \tau \rangle \to \langle [\mathtt{skip}]^l, \tau{::}\epsilon \rangle}\ \textbf{query}$$

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

$$\frac{\langle c_1, \tau \rangle \to \langle c_1', \tau' \rangle}{\langle c_1; c_2, \tau \rangle \to \langle c_1'; c_2, \tau' \rangle}\ \textbf{seq1} \qquad \frac{\langle c_2, \tau \rangle \to \langle c_2', \tau' \rangle}{\langle [\mathtt{skip}]^l; c_2, \tau \rangle \to \langle c_2', \tau' \rangle}\ \textbf{seq2} \qquad \frac{\tau, b \Downarrow_b \mathtt{true} \qquad \epsilon = (b, l, \mathtt{true}, \bullet)}{\langle\ \mathtt{if}\ ([b]^l, c_1, c_2), \tau \rangle \to \langle c_1, \tau{::}\epsilon \rangle}\ \textbf{if-t if-f}$$

Fig. 5. Trace-based Operational Semantics for Language.

guard, while the last element is $\bullet$. We use the operator $\rho(\tau)x$ to fetch the latest value assigned to x in the trace τ. We use the operator $\mathtt{cnt}$ to count the occurrence of a labeled variable in the trace. We denote by $\mathbb{TL}(\tau) \subseteq \mathcal{L}$ the set of the labels occurring in τ. Finally, we use $\mathcal{T}_0(c) \subseteq \mathcal{T}$ to denote the set of *initial traces*, the ones which assign a value to the input variables.

The trace-based operational semantics is described in terms of a small step evaluation relation $\langle c, \tau \rangle \to \langle c', \tau \rangle'$ describing how a configuration program-trace evaluates to another configuration program-state. The rules for the operational semantics are described in Fig. 5. The rules for assignment and query generate assignment events, while the rules for while and if generate testing events. The rules for the standard while language constructs correspond to the usual rules extended to deal with traces. We have relations $\langle \tau, e \rangle \Downarrow_e v$ and $\langle \tau, b \rangle \Downarrow_b v$ to evaluate expressions and boolean expressions, respectively. Their definitions are in the supplementary material. The only rule that is non-standard is the **query** rule. When evaluating a query, the query expression ψ is first simplified to its normal form α using an evaluation relation $\langle \tau, \psi \rangle \Downarrow_q \alpha$. Then normal form α characterizes the linear query that is run against the database. The query result v is the expected value of the function $\lambda\chi.\alpha$ applied to each row of the dataset. We summarize this process with the notation $\mathtt{query}(\alpha) = v$ which we use in the rule **query**. Once the answer of the query is computed, the rules record all the needed information in the trace. As usual, we will use $\to^*$ for the reflexive and transitive closure of $\to$.

The query expression evaluation relation $\langle \tau, \psi \rangle \Downarrow_q \alpha$ is defined by the following rules which reduce a query expression to its normal form.

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

4 DEFINITION OF ADAPTIVITY

In this section, we formally present the definition of adaptivity for a given program. As we discussed in Section 2.2.1, we first define a dependency relation between program variables, we then define a semantics-based dependency graph, and finally look at the walk visiting the maximal number of query nodes in this graph.

4.1 May-dependency between variables

We are interested in defining a notion of dependency between program variables since assigned variables are a good proxy to study dependencies between queries—we can recover query requests from variables associated with queries. We consider dependencies that can be generated by either data or control flow. For example, in the program

$$c_1 = [x \leftarrow \mathtt{query}(\chi[2])]^1; [y \leftarrow \mathtt{query}(\chi[3] + x)]^2$$

the query $\texttt{query}(\chi[3]+x)$ depends on the query $\texttt{query}(\chi[2]))$ through a *value dependency* via x^1. Conversely, in the program

$$c_2 = [x \leftarrow \texttt{query}(\chi[1])]^1;\ \texttt{if}\ ([x > 2]^2, [y \leftarrow \texttt{query}(\chi[2])]^3, [\texttt{skip}]^4)$$

the query $\texttt{query}(\chi[2])$ depends on the query $\texttt{query}(\chi[1])$ via the *control dependency* of the guard of the if command involving the labeled variable x^1.

To define dependency between program variables we will consider two events that are generated from the same command, hence they have the same variable name or boolean expression and label, but have either different value or different query expression. This is captured by the following definition.

Definition 1. *Two events $\epsilon_1, \epsilon_2 \in \mathcal{E}$ are in the relation $\texttt{Diff}(\epsilon_1, \epsilon_2)$, if and only if:*

$$\pi_1(\epsilon_1) = \pi_1(\epsilon_2) \land \pi_2(\epsilon_1) = \pi_2(\epsilon_2) \tag{1a}$$

$$\land \big((\pi_3(\epsilon_1) \neq \pi_3(\epsilon_2) \land \pi_4(\epsilon_1) = \pi_4(\epsilon_2) = \bullet) \lor (\pi_4(\epsilon_1) \neq \bullet \land \pi_4(\epsilon_2) \neq \bullet \land \pi_4(\epsilon_1) \neq_q \pi_4(\epsilon_2))\big) \tag{1b}$$

where $\psi_1 =_q \psi_2$ denotes the semantics equivalence between query values[7]*, and π_i projects the i-th element from the quadruple of an event.*

It is worth stressing that in the definition above, if two events are both generated from query requests, we are not comparing the query answers (the element in the third position of the tuple), but the query expressions (the element in the fourth position of the tuple). For example in the running program in Fig. 3(a), given different inputs $k = 1$ and $k = 2$ both events $\epsilon_1 = (x, 3, 0, \chi[0] \cdot \chi[1])$ and $\epsilon_2 = (x, 3, 0, \chi[0] \cdot \chi[2])$ can be generated from query request $[x \leftarrow \texttt{query}(\chi[j] \cdot \chi[k])]^3$. Even though $\pi_3(\epsilon_1) = \pi_3(\epsilon_2)$, these two events are still different by our definition. The Equation 1(b) captures this by first checking $\pi_4(\epsilon_1) \neq \bullet \land \pi_4(\epsilon_1) \neq \bullet$ to guarantee both events are from query requests. Then we check again the forth element $\pi_4(\epsilon_1) \neq_q \pi_4(\epsilon_2)$ to guarantee that the two events come from different query requests.

We can now define when an event *may depend* on another one[8].

Definition 2 (Event May-Dependency). *An event $\epsilon_2 \in \mathcal{E}^{\mathsf{asn}}$* may-depend *on an event $\epsilon_1 \in \mathcal{E}^{\mathsf{asn}}$ in a program c denoted $\mathsf{DEP_e}(\epsilon_1, \epsilon_2, c)$, if and only if*

$$\exists \tau, \tau_0, \tau_1, \tau' \in \mathcal{T}, \epsilon_1' \in \mathcal{E}^{\mathsf{asn}}, c_1, c_2 \in C\ .\ \texttt{Diff}(\epsilon_1, \epsilon_1') \land \tag{2a}$$

$$(\exists \epsilon_2' \in \mathcal{E}\ .\left(\begin{array}{ll} & \langle c, \tau_0\rangle \rightarrow^* \langle c_1, \tau_1 {+\!+} [\epsilon_1]\rangle \rightarrow^* \langle c_2, \tau_1 {+\!+} [\epsilon_1] {+\!+} \tau {+\!+} [\epsilon_2]\rangle \\ \land & \langle c_1, \tau_1 {+\!+} [\epsilon_1']\rangle \rightarrow^* \langle c_2, \tau_1 {+\!+} [\epsilon_1'] {+\!+} \tau' {+\!+} [\epsilon_2']\rangle \\ \land & \texttt{Diff}(\epsilon_2, \epsilon_2') \land \texttt{cnt}(\tau, \pi_2(\epsilon_2)) = \texttt{cnt}(\tau', \pi_2(\epsilon_2')) \end{array}\right) \tag{2b}$$

$$\lor \left(\begin{array}{l} \exists \tau_3, \tau_3' \in \mathcal{T}, \epsilon_b \in \mathcal{E}^{\mathsf{test}}\ . \\ \quad \langle c, \tau_0\rangle \rightarrow^* \langle c_1, \tau_1 {+\!+} [\epsilon_1]\rangle \rightarrow^* \langle c_2, \tau_1 {+\!+} [\epsilon_1] {+\!+} \tau {+\!+} [\epsilon_b] {+\!+} \tau_3\rangle \\ \quad \land \langle c_1, \tau_1 {+\!+} [\epsilon_1']\rangle \rightarrow^* \langle c_2, \tau_1 {+\!+} [\epsilon_1'] {+\!+} \tau' {+\!+} [(\neg \epsilon_b)] {+\!+} \tau_3'\rangle \\ \quad \land \mathbb{TL}(\tau_3) \cap \mathbb{TL}(\tau_3') = \emptyset \land \texttt{cnt}(\tau', \pi_2(\epsilon_b)) = \texttt{cnt}(\tau, \pi_2(\epsilon_b)) \land \epsilon_2 \in \tau_3 \land \epsilon_2 \notin \tau_3' \end{array}\right)), \tag{2c}$$

There are several components in this definition. The part with label (2a) requires that ϵ_1 and ϵ_1 differ in their values ($\texttt{Diff}(\epsilon_1, \epsilon_1')$). The next two parts (2b) and (2c) capture the value dependency and control dependency, respectively. As in the literature on non-interference, and following [12], we formulate these dependencies as relational properties, in terms of two different traces of execution. We force these two traces to differ by using the event ϵ_1 in one and ϵ_1' in the other. For the value dependency we check whether the change also creates a change in the value of ϵ_2 or

[7]The formal definition is in the supplementary material

[8]We consider here dependencies between assignment events. This simplifies the definition and is enough for the stating the following definitions. The full definition is in the supplementary material.

not. We additionally check that the two events we consider appear the same number of times in the two traces - this to make sure that if the events are generated by assignments in a loop, we consider the same iteration. For the control dependency we check whether the change in ϵ_1 affects the appearance in the computation of ϵ_2 or not. For this we require the presence of a test event whose value is affected by the change in ϵ_1 in order to guarantee that the computation goes through a control flow guard. Similarly to the previous condition, we additionally check that the two test events we consider appear the same number of times in the two traces.

Looking again to the running example in Fig. 3(a), given the initial trace $[(k, \mathtt{in}, 1, \bullet)]$, the program execution generates an event $\epsilon_1 = (x, 3, v_1, \chi[0] \cdot \chi[1])$ corresponding to the query request $[x \leftarrow \mathtt{query}(\chi[j] \cdot \chi[k])]^3$, and another event $\epsilon_2 = (l, 6, v_2, \chi[1] \cdot v_1)$ corresponding to the query request $[l \leftarrow \mathtt{query}(\chi[k] \cdot a)]^6$. In order to check if we have a may-dependency, we replace ϵ_1 with another event $\epsilon_1' = (x, 3, v_1', \chi[0] \cdot \chi[1])$ where $v_1 \neq v_1'$. We then continue to execute the program. The 6^{th} command generates $\epsilon_2 = (l, 3, v_2, \chi[1] \cdot v_1')$. Since $\pi_4(\epsilon_2) \neq_1 \pi_4(\epsilon_2')$, we have $\mathtt{Diff}(\epsilon_2, \epsilon_2')$ according to Definition 1 and then the dependency relation $\mathsf{DEP}_\mathsf{e}(\epsilon_1, \epsilon_2, \mathtt{twoRounds(k)})$.

We can now extend the dependency relation to variables by considering all the assignment events generated during the program's execution.

Definition 3 (Variable May-Dependency). *A variable* $x_2^{l_2} \in \mathsf{LV}(c)$ may-depend *on the variable* $x_1^{l_1} \in \mathsf{LV}(c)$ *in a program* c, $\mathsf{DEP}_\mathsf{var}(x_1^{l_1}, x_2^{l_2}, c)$, *if and only if*

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

Notice that in the definition above we can also have that the two variables are the same, this allows us to capture self-dependencies.

Going back to the running example in Fig. 3(a), we have already discussed how $\epsilon_1 = (x, 3, v_1, \chi[0] \cdot \chi[1])$ and $\epsilon_2 = (l, 6, v_2, \chi[1] \cdot v_1)$ are in the event may-dependency relation. Moreover, we have that x^3 and l^6 are in the variable may-dependency relation by Definition 3.

4.2 Semantics-based Dependency Graph

We can now define the *semantics-based dependency graph* of a program c. We want this graph to combine quantitative reachability information with dependency information.

Definition 4 (Semantics-based Dependency Graph). *Given a program c, its* semantics-based dependency graph $\mathsf{G}_{\mathtt{trace}}(c) = (\mathsf{V}_{\mathtt{trace}}(c), \mathsf{E}_{\mathtt{trace}}(c), \mathsf{W}_{\mathtt{trace}}(c), \mathsf{Q}_{\mathtt{trace}}(c))$ *is defined as follows,*

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

A semantics-based dependency graph $\mathsf{G}_{\mathtt{trace}}(c) = (\mathsf{V}_{\mathtt{trace}}(c), \mathsf{E}_{\mathtt{trace}}(c), \mathsf{W}_{\mathtt{trace}}(c), \mathsf{Q}_{\mathtt{trace}}(c))$ *is* well-formed *if and only if* $\{x^l \mid (x^l, w) \in \mathsf{W}_{\mathtt{trace}}(c)\} = \mathsf{V}_{\mathtt{trace}}(c)$.

As we discussed before, vertices and query annotations are just read out from the program c. We have an edge in $\mathsf{E}_{\mathtt{trace}}(c)$ if we have a may-dependency between two labeled variables in c.

For the dependency graph of our running example in Fig. 3(b), the subscript in vertices x^3 and l^6 is 1 because they both come from query requests. There is a directed edge from x^3 and l^6 because we identify the variable may-dependency relation, $\mathsf{DEP}(x^3, l^6, \mathtt{twoRounds(k)})$ according to Definition 3.

A weight function $w \in \mathsf{W}_{\mathtt{trace}}(c)$ is a function that for every starting trace $\tau_0 \in \mathcal{T}_0(c)$ gives the number of times the assignment of the corresponding vertex x^l is visited. Notice that weight functions are total and with range $\mathbb{N}$. This means that if a program c has some non-terminating behavior, the set $\mathsf{W}_{\mathsf{trace}}(c)$ will be empty. To rule out this situation, we consider as well-formed only graphs which have a weight for every vertex. In the rest of the paper we will implicitly consider only well-formed semantics-based dependency graphs.

Going back to the example in Fig. 3(b), the vertices a^0, j^1 and l^6 have weight function $\lambda\tau . 1$ because commands 0, 1 and 6 are executed only once, given any arbitrary initial trace. The vertices corresponding to commands inside the loop body have weight function $\lambda\tau . \rho(\tau)k$ because they will be executed the same number of times as the loop iteration times.

4.3 Adaptivity of a Program

We can now define the adaptivity of a program formally. This notion is formulated in terms of an initial trace, specifying the value of the input variables, and of the walk on the graph $\mathsf{G}_{\mathsf{trace}}(c)$ which has the largest number of query requests.

Definition 5 (Walk). *Given a well-formed program c with its semantics-based dependency graph $\mathsf{G}_{\mathsf{trace}}(c) = (\mathsf{V}_{\mathsf{trace}}, \mathsf{E}_{\mathsf{trace}}, \mathsf{W}_{\mathsf{trace}}, \mathsf{Q}_{\mathsf{trace}})$, a* walk *$k$ is a function that maps an initial trace τ_0 to a sequence of vertices $(v_1, \dots, v_n)$ for which there is a sequence of edges $(e_1 \dots e_{n-1})$ satisfying*

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

We denote by $\mathcal{WK}(\mathsf{G}_{\mathsf{trace}}(c))$ the set of all the walks k in $\mathsf{G}_{\mathsf{trace}}(c)$.

In Fig. 3(b), $\lambda\tau_0 . (l^6 \to x^3)$ is a walk with two vertices and where each vertex is visited only once. With the assumption that $k \geq 1$, $\lambda\tau_0 . (l^6 \to \underbrace{a^4 \to \dots}_{\rho(\tau)k} \to x^3)$ is a walk where the vertex a^4 is visited $\rho(\tau)k$ times. However, $\lambda\tau_0 . (l^6 \to a^4 \to x^3 \to x^3)$ is not a walk because there is no edge from x^3 to x^3.

Because for the adaptivity we are interested in the dependency between queries, we calculate a special "length" of a walk, the *query length*, by counting only the vertices corresponding to queries.

Definition 6 (Query Length). *Given the semantics-based dependency graph $\mathsf{G}_{\mathsf{trace}}(c)$ of a well-formed program c, and a* walk *$k \in \mathcal{WK}(\mathsf{G}_{\mathsf{trace}}(c))$, the* query length *of k is a function $\mathtt{len}^{\mathtt{q}}(k) : \mathcal{T}_0(c) \to \mathbb{N}$ that given an initial trace $\tau_0 \in \mathcal{T}_0(c)$ gives the number of vertices that correspond to query variables in the vertex sequence of $k(\tau_0)$. It is defined as follows.*

$$\mathtt{len}^{\mathtt{q}}(k) = \lambda\tau_0 . \left|\left(v \mid v \in (v_1, \dots, v_n) \land (v, 1) \in \mathsf{Q}_{\mathsf{trace}}(c) \land k(\tau_0) = (v_1, \dots, v_n)\right)\right|,$$

where the notation $|(\dots)|$ gives the number of vertices in a sequence.

Under the assumption that $k \geq 1$, both walks $\lambda\tau_0 . (l^6 \to x^3)$ and $\lambda\tau_0 . (l^6 \to \underbrace{a^4 \to \dots}_{\rho(\tau)k} \to x^3)$ in Fig. 3(b) have query length $\lambda\tau_0 . 2$ because only vertices x^3 and l^6 come from query requests.

We can now define the adaptivity of a well-formed program as follows.

Definition 7 (Adaptivity of a Program). *Given a well-formed program c, its adaptivity $A(c)$ is a function $A(c) : \mathcal{T}_0(c) \to \mathbb{N}$ defined as follows.*

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

Again, we define the adaptivity by considering only well-formed semantics-based dependency graphs.

In Fig. 3(b), the two dotted edges represent a finite walk with maximal query length, since it contains the only two query requests. Each query request can be visited at most once, and so the adaptivity of this program is $\lambda\tau_0 . 2$.

5 THE ADAPTIVITY ANALYSIS ALGORITHM - ADAPTFUN

In this section, we present our program analysis AdaptFun for computing an upper bound on the *adaptivity* of a given program c. The high level idea behind AdaptFun is to first build an *estimated dependency graph* $\mathsf{G}_{\mathsf{est}}$(c) of a program c (Section 5.1) which overapproximates the semantics-based dependency graph in two dimensions: it overapproximates the dependencies between assigned variables (Section 5.1.2), and, it overapproximates the weights (Section 5.1.3). Then, AdaptFun uses a custom algorithm to estimate the walk of maximal query length on this graph, providing in this way an upper bound on the adaptivity of the program.

Given a program c, the set of vertices $\mathsf{V}_{\mathsf{est}}(c)$ and query annotations $\mathsf{Q}_{\mathsf{est}}(c)$ of the *estimated dependency graph* can be computed by simply scanning the program c. These sets can be computed precisely and correspond to the same sets in the semantics-based dependency graph. This means that $\mathsf{G}_{\mathsf{est}}(c)$ has the same underlying vertex structure as the semantics-based graph $\mathsf{G}_{\mathsf{trace}}(c)$. The differences will be in the sets of edges and weights.

5.1 Weight and Edge Estimation

The set of edges $\mathsf{E}_{\mathsf{est}}(c)$ and the set of weights $\mathsf{W}_{\mathsf{est}}(c)$ of the estimated dependency graph are estimated through an analysis combining control flow, data flow, and loop bound analysis. These analyses are naturally described over an *Abstract Transition Graph* of the input program, which we describe next.

5.1.1 Abstract Transition Graph. We say that we have a *transition* from a program point l to a program point l' if and only if the command with label l' can execute right after the execution of the command with label l. The *Abstract Transition Graph* $\mathsf{absG}(c)$ of a program c is a graph with the set of labels of program points in c (including a label ex for the exit point) as the set of vertices $\mathsf{absV}(c)$, and with the set of transitions in c as the set of edges $\mathsf{absE}(c)$. Each edge of the graph is annotated with either the symbol $\top$, a boolean expression or a *difference constraint* [37].

A difference constraint is an inequality of the form $x' \leq y + v$ or $x' \leq v$ where x, y are variables and $v \in \mathcal{SC}$ is a symbolic constant: either a natural number, the symbol ∞, an input variable or a symbol Q_m representing a query request. We denote by $\mathcal{DC}$ the set of difference constraints.

A difference constraint on an edge, $l \xrightarrow{x' \leq y+v} l'$ or $l \xrightarrow{x' \leq v} l'$, denotes that after executing the command at location l the value of the variable x is at most the value of the expression $y + v$ resp. v before the execution of the command l'. A boolean value b on an edge, $l \xrightarrow{b} l'$, denotes that after evaluating the guard of an if or a while command with label l, b holds and the next command to be executed is the one with label l'. A $\top$ symbol on an edge, $l \xrightarrow{\top} l'$ denotes that the command with label l is a `skip`, and the commands that do not interfere with any loop counter variable.

We compute difference constraints and the other annotation via a simple program abstraction method adopted from [37], described in details in the supplementary material.

Example. We show in Fig. 6(b) the abstract control flow graph, $\mathsf{absG}(\mathsf{twoRounds(k)})$ of the $\mathsf{twoRounds(k)}$ program we gave in Fig. 3(a) and which we also report in Fig. 6(a).

5.1.2 Edge Estimation. The set of edges $\mathsf{E}_{\mathsf{est}}(c)$ is estimated through a combined data and control flow analysis with three components.

Reaching definition analysis: The first component is a reaching definition analysis computing for each label l in the graph $\mathsf{absG}(c)$ the set of labeled variables that may reach l as follows.

(1). For each label l, the analysis generates two initial sets of labeled variables, *in* and *out*, containing all the labeled variables x^l that are newly generated but not yet reassigned before and after executing the command l.

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

(a)

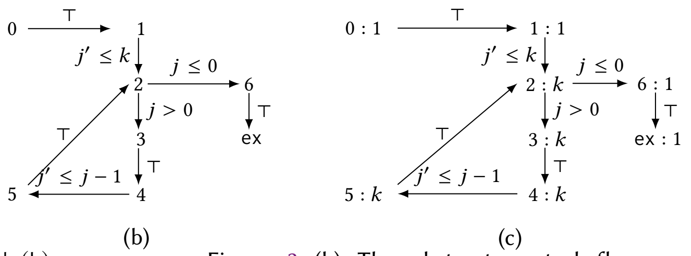


Fig. 6. (a) The same towRounds(k) program as Figure 3 (b) The abstract control flow graph, absG(twoRounds(k)) (c) absG(twoRounds(k)) with the reachability bound.

(2). The analysis iterates over $\texttt{absG}(c)$, and updates $in(l)$ and $out(l)$ until they are stable. The final $in(l)$ is the set of reaching definitions $\texttt{RD}(l, c)$ for l.

Feasible data-flow analysis: The second component is a **feasible data-flow analysis** computing for every pair $x^i, y^j \in \texttt{LV}(c)$ whether there is a flow from x^i to y^j. This analysis is based on a relation $\texttt{flowsTo}(x^i, y^j, c)$ built over the sets $\texttt{RD}(l, c)$ for every location l. This relation is defined as:

Definition 8 (Feasible Data-Flow). *Given a program c and two labeled variables x^i, y^j in this program,* $\texttt{flowsTo}(x^i, y^j, c)$ *is*

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

This relation gives us an overapproximation of the *variable may-dependency* relation for direct dependencies (dependencies that do not go through other variables).

Edge Construction: The third component constructs an edge by computing a transitive closure (through other variables) of the flowsTo relation. There is a directed edge from x^i to y^j if and only if there is chain of of variables in the flowsTo relation between x^i and y^j. This is defined as follows:

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

We prove that the set $\texttt{E}_{\texttt{est}}(c)$ soundly approximates the set $\texttt{G}_{\texttt{trace}}(c)$.

Lemma 5.1 (Mapping from Egdes of $\texttt{G}_{\texttt{trace}}$ to $\texttt{G}_{\texttt{est}}$). *For every program c we have:*

$$\forall e = (v_1, v_2) \in \texttt{E}_{\texttt{trace}}(c) \; . \; \exists e' \in \texttt{E}_{\texttt{est}}(c) \; . \; e' = (v_1, v_2)$$

Example. Consider Fig. 3(c), the edge $l^6 \longrightarrow a^5$ is built by the $\texttt{flowsTo}(l^6, a^5, c)$ relation because a is used directly in the query expression $\chi[k] * a$ and we also have $a^5 \in \texttt{RD}(6, \texttt{twoRounds(k)})$ from the reaching definition analysis. The edge $x^3 \longrightarrow j^5$ represents the control flow from j^5 to x^3, which is soundly approximated by our flowsTo relation. The edge $l^6 \longrightarrow x^3$ is produced by the transitivity of $\texttt{flowsTo}(l^6, a^5, c)$ and $\texttt{flowsTo}(a^5, x^3, c)$.

5.1.3 *Weight Estimation.* The set $\texttt{W}_{\texttt{est}}(c)$ of weights for the estimated dependency graph is estimated from the Abstract Transition Graph $\texttt{absG}(c)$ of c using *reachability-bound analysis* [25]. Specifically, we estimate as the weight of a node with label l a symbolic upper bound on the execution times of the command with label l obtained by reachability-bound analysis. These symbolic upper bounds are expressions with the input variables as free variables, hence they correspond to the weight functions in the semantics-based dependency graphs. Our reachability-bound algorithm adapts to our setting ideas from previous work [36, 37, 43]. Specifically, it provides an upper bound on the number of times every command can be executed by using three steps.

(1) This step assigns to each edge $l \xrightarrow{dc} l' \in \mathsf{absE}(c)$ a *local bound* as follows. We look at the strongly connected components of $\mathsf{absG}(c)$. If the edge does not belong to any strongly connected components, then the local bound is 1, representing the fact that the edge is not in a loop and so it gets executed at most once. If the edge belongs to a strongly connected component and one of the variables x in dc decreases, then the local bound is x. Otherwise, if the edge belongs to a strongly connected component and there is a variable y that decreases in the difference constraint of some other edge, and if by removing this other edge, the original edge does not belong anymore to the strongly connected components of $\mathsf{absG}(c)$, then the local bound is y. Otherwise, the local bound is ∞. Notice that the output is either a symbolic constant in $\mathcal{SC}$ or a variable that is not an input variable.

(2) This step aims at determining the *reachability-bound* $\mathsf{TB}(e, c)$ of every edge $e \in \mathsf{absE}(c)$. Every bound is a symbolic expression from the set $\mathcal{A}_{in}$ built out of symbols in $\mathcal{SC}$ and the operations $+, *, \max$. For every edge, if the local bound of this edge computed at the previous step is a symbol in $\mathcal{SC}$ then this is already the reachability-bound. If instead the local bound of the edge is a variable y which is not an input variable, this step will eliminate it and replace it with a symbolic expression. In order to do this, this step will compute two quantities: first, it will recursively sum the reachability-bounds of all the edges whose difference constraint may increment the variable y, plus the corresponding increment; second, it will recursively sum the reachability-bounds of all the edges whose difference constraint may reset the variable y to a (symbolic) expression that doesn't depend on it, multiplied by the maximal value of this symbolic expression. The sum of these two quantities provides the symbolic expression that is an upper bound on the number of times the edge can be reached. To compute these two quantities we use two mutually recursive procedures.

Using the reachability-bound $\mathsf{TB}(e, c)$ for every edge $e = (l, dc, l') \in \mathsf{absE}(c)$ we can provide an upper bound on the visiting times of each vertex $x^l \in \mathsf{absV}(c)$, which is also the estimated weight of this vertex.

Definition 9 (Weight Estimation). *The estimated weight set $\mathsf{W}_{\mathsf{est}}(c)$ of program c is a set of pairs $\mathsf{W}_{\mathsf{est}}(c) \in \mathcal{P}(\mathcal{LV} \times \mathcal{SC})$. Each pair maps a vertex $x^l \in \mathsf{V}_{\mathsf{est}}(c) \subseteq \mathsf{LV}(c)$ to a symbolic expression over $\mathcal{SC}$, as follows.*

$$\mathsf{W}_{\mathsf{est}}(c) \triangleq \left\{(x^l, \hat{w}) \mid x^l \in \mathsf{V}_{\mathsf{est}}(c) \land \hat{w} = \sum \{\mathsf{TB}(e, c) | e \in \mathsf{absE}(c) \land e = (l, \_, \_)\}\right\}.$$

Notice that $\hat{w} \in \mathcal{SC}$ is an expression over symbols in $\mathcal{SC}$. In particular, it may contain the input variables and so it may effectively be used as a function of the input - and capture loop bounds in terms of these inputs.

Theorem 5.1 (Soundness of the Weight Estimation). *Let c be a program and $\mathsf{W}_{\mathsf{est}}(c)$ be its estimated weight set. Then, for every $(x^l, w) \in \mathsf{W}_{\mathsf{trace}}(c)$, we have a corresponding estimated weight $(x^l, \hat{w}) \in \mathsf{W}_{\mathsf{est}}(c)$ and for every possible $\tau_0 \in \mathcal{T}_0(c), v \in \mathbb{N}$, if $\langle \tau_0, \hat{w} \rangle \Downarrow_e v$, then v is an upper bound on $w(\tau_0)$. In symbols:*

$$\begin{array}{l} \forall c \in \mathcal{C}, x^l \in \mathsf{LV}(c), w \in \mathcal{T}_0(c) \to \mathbb{N}, \tau_0 \in \mathcal{T}_0(c), v \in \mathbb{N} \cup \{\infty\}, \\ \qquad (x^l, w) \in \mathsf{W}_{\mathsf{trace}}(c) \;. \implies \exists (x^l, \hat{w}) \in \mathsf{W}_{\mathsf{est}}(c) \land \left(\langle \tau_0, \hat{w} \rangle \Downarrow_e v \implies w(\tau_0) \leq v\right) \end{array}$$

Notice that in this theorem, the evaluation $\langle \tau_0, \hat{w} \rangle \Downarrow_e v$ is needed in order to obtain a concrete value v from the symbolic weight $\hat{w}$ by specifying a value for the input variables through τ_0.

Example. Consider again Fig. 3(c), the estimated weight for a^5 is k, and this is a sound estimation. For an arbitrary $\tau_0 \in \mathcal{T}_0(c)$, we know that $\langle \tau_0, k\rangle \Downarrow_e \rho(\tau_0)k$ and by the weight w_k for the vertex a^5 (as in Fig. 3(b)) we know $w_k(\tau_0) = \rho(\tau_0)k$.

5.2 Dependency Graph Estimation

We now have all the elements to construct the estimated dependency graph of a program.

Definition 10 (Estimated Dependency Graph). *Given a program c with the feasible data flow relation $\mathtt{flowsTo}(x^i, y^j, c)$ for every $x^i, y^j \in \mathsf{LV}(c)$, and reachability-bound, $\mathsf{TB}(e, c)$ for every edge $e \in \mathsf{absE}(c)$ on the abstract transition graph, the estimated dependency graph of c has four components:*

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

each of the four components is generated as follows.

$$\begin{array}{lcl}
\mathsf{V}_{\mathsf{est}} & := & \left\{x^l \;\middle|\; x^l \in \mathsf{LV}(c)\right\} \\
\mathsf{E}_{\mathsf{est}} & := & \left\{(x^i, y^j) \;\middle|\; \begin{array}{l} x^i, y^j \in \mathsf{V} \wedge \exists n \in \mathbb{N}, z_1^{r_1}, \dots, z_n^{r_n} \in \mathsf{LV}(c) \,.\, n \geq 0 \wedge \\ \mathtt{flowsTo}(y^j, z_1^{r_1}, c) \wedge \dots \wedge \mathtt{flowsTo}(z_n^{r_n}, x^i, c) \end{array}\right\} \\
\mathsf{W}_{\mathsf{est}} & := & \left\{(x^l, \hat{w}) \mid x^l \in \mathsf{V}_{\mathsf{est}}(c) \wedge \hat{w} = \sum\{\mathsf{TB}(e, c) | e \in \mathsf{absE}(c) \wedge e = (l, \_, \_)\}\right\} \\
\mathsf{Q}_{\mathsf{est}} & := & \left\{(x^l, 1) \;\middle|\; x^l \in \mathsf{LV}(c) \wedge x^l \in \mathsf{QV}(c)\right\} \cup \left\{(x^l, 0) \;\middle|\; x^l \in \mathsf{LV}(c) \wedge x^l \notin \mathsf{QV}(c)\right\}
\end{array}$$

The estimated dependency graph of a program provides an overapproximation of the dependencies between variables and upper bounds on the number of times each command is executed. The path-searching algorithm presented in the next section, will combine these two to provide a sound estimation of the adaptivity.

In Fig. 3(c) we present the estimated dependency graph for our running example. The overapproximation is tight in terms of both edges and weights.

5.3 Adaptivity Upper Bound Computation

We estimate the adaptivity upper bound, $\mathsf{A}_{\mathsf{est}}(c)$ for a program c as the maximum query length over all finite walks in its *estimated dependency graph*, $\mathsf{G}_{\mathsf{est}}(c)$.

Definition 11 (Estimated Adaptivity). *Given a program c and its estimated dependency graph $\mathsf{G}_{\mathsf{est}}(c)$ the estimated adaptivity for c is*

$$\mathsf{A}_{\mathsf{est}}(c) \triangleq \max\{\mathtt{len}^{\mathsf{q}}(k) \;\mid\; k \in \mathcal{WK}(\mathsf{G}_{\mathsf{est}}(c))\}.$$

Notice that different from a walk on $\mathsf{G}_{\mathsf{trace}}(c)$, a walk $k \in \mathcal{WK}(\mathsf{G}_{\mathsf{est}}(c))$ on the graph $\mathsf{G}_{\mathsf{est}}(c)$ does not rely on an initial trace. This because, similarly to what we did for the weights in $\mathsf{W}_{\mathsf{est}}(c)$ in the previous section, we use symbolic expressions over input variables. Similarly, the adaptivity bound $\mathsf{A}_{\mathsf{est}}(c)$ will also be a symbolic arithmetic expression over the input variables. With this symbolic expression we can prove the upper bound sound with respect to any initial trace.

Theorem 5.2 (Soundness of $\mathsf{A}_{\mathsf{est}}(c)$). *For every program c, its estimated adaptivity is a sound upper bound of its adaptivity.*

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

Symbolic expressions as used in the weight are great to express symbolic bounds but make the computation of a maximal walk harder. Specifically, one has to face two challenges. The first is non-termination. A naive traversing strategy leads to non-termination because the weight of each vertex in $\mathsf{G}_{\mathsf{est}}(c)$ is a symbolic expression containing input variables. We could try to use a depth first search strategy using the longest weighted path to approximate the finite walk with the weight as its visiting time. However, these approach would face the second challenge: approximation. It would consistently and considerably over-approximate the adaptivity.

To address these two challenges we design an algorithm AdaptBD combining Depth First Search and Breadth First Search. The idea of this algorithm is to reduce the task of computing the walk with the maximal query length to the task of computing local versions of the adaptivity on the maximal strongly connected components of the graph $\mathsf{G}_{\mathsf{est}}(c)$ and then compose them into the program adaptivity. The algorithm uses another algorithm $\mathsf{AdaptBD}_{\mathsf{SCC}}$ recursively, which finds the walk for a strong connected component of $\mathsf{G}_{\mathsf{est}}(c)$. The pseudocode of $\mathsf{AdaptBD}_{\mathsf{SCC}}$ is given as Algorithm 1

Algorithm 1 Adaptivity Bound Algorithm on An SCC ($\mathsf{AdaptBD}_{\mathsf{SCC}}(\mathsf{c}, \mathsf{SCC_i})$)

```
Require: The program $c$, A strong connected component of $\mathsf{G_{est}}(c)$: $\mathsf{SCC_i} = (\mathsf{V}_i, \mathsf{E}_i, \mathsf{W}_i, \mathsf{Q}_i)$
 1: init. $\mathsf{r_{SCC}}$: $\mathcal{A}_{\mathsf{in}}$ List with initial value 0.
 2: init. visited : $\{0, 1\}$ List with initial value 0; $\mathsf{r}$ : $\mathcal{A}_{\mathsf{in}}$ List, initial value $\infty$;
        flowcapacity: $\mathcal{A}_{\mathsf{in}}$ List, initial value $\infty$; querynum: INT List, initial value $\mathsf{Q}_i(v)$.
 3: if $|\mathsf{V}_i| = 1$ and $|\mathsf{E}_i| = 0$:    return $\mathsf{Q}(v)$
 4: def dfs(G, s, visited):
 5:     for every vertex $v$ connected by a directed edge from $s$:
 6:         if visited[$v$] = false:
 7:             flowcapacity[v] = min($\mathsf{W_i}$(v), flowcapacity[s]);    querynum[v] = querynum[s] + $\mathsf{Q_i}$(v);
 8:             r[v] = max(r[v], flowcapacity[v] $\times$ querynum[v]);
 9:             visited[$v$] = 1;    dfs(G, v, visited);
10:         else: #{There is a cycle finished}
11:             r[v] = max(r[v], r[s] + min($\mathsf{W_i}$(v), flowcapacity[s]) $*$ (querynum[s] + $\mathsf{Q_i}$(v)));
12:     return r[c]
13: for every vertex $v$ in $\mathsf{V}_i$:
14:     initialize the visited, r, flowcapacity, querynum with the same value at line:2.
15:     $\mathsf{r_{SCC}}$ = max($\mathsf{r_{SCC}}$, dfs($\mathsf{SCC_i}$, v, visited));
16: return $\mathsf{r_{SCC}}$
```

This algorithm takes as input the program c and a $\mathsf{SCC_i}$ of $\mathsf{G_{est}}(c)$, and outputs an adaptivity bound for $\mathsf{SCC_i}$. If $\mathsf{SCC_i}$ contains only one vertex, x^l without any edge, $\mathsf{AdaptBD}_{\mathsf{SCC}}$ returns the query annotation of x^l as the adaptivity. If $\mathsf{SCC_i}$ contains at least one edge, $\mathsf{AdaptBD}_{\mathsf{SCC}}$:

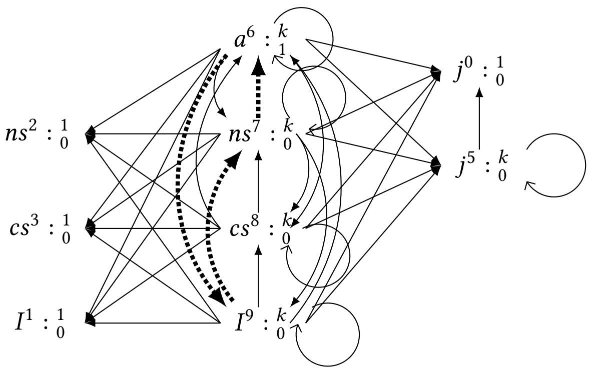
(1) first collects all the paths in $\mathsf{SCC_i}$;
(2) it then calculates the adaptivity of every path by the method `dfs`;
(3) in the end, it outputs the maximal adaptivity among all paths as the adaptivity of $\mathsf{SCC_i}$.

By the property of SCC, the paths collected in step 1 are all simple cycles with the same starting and ending vertex. Step 2 is the key step. It recursively computes the adaptivity upper bound on the fly of paths collected through a DFS procedure `dfs` (lines: 4-13). This procedure guarantees that the visiting times of each vertex is upper bounded by its weight, and addresses the approximation challenge, via two special lists parameters `flowcapacity` and `querynum` (lines:7-11).

`flowcapacity` is a list of symbolic expressions $\mathcal{A}_{in}$ which tracks the minimum weight when searching a path, and updates the weight when the path reaches a certain vertex. `querynum` is a list of integer initialized with the value of the query annotation $\mathsf{Q}_i(v)$ for every vertex. It tracks the total number of vertices with query annotation 1 along the path. The operation at line: 8 and line: 11 implements the operation `flowcapacity[v]` $\times$ `querynum[v]`. Because `flowcapacity[v]` is the minimum weight over this path, this guarantees that every vertex on the estimated walk is allowed to be visited at most `flowcapacity[v]`, and this walk is a valid finite walk. Then `querynum[v]` guarantees that `flowcapacity[v]` $\times$ `querynum[v]` computes an accurate query length because `querynum[v]` is only the number of the vertices with query annotation 1, giving a tight bound without losing the soundness.

```
multipleRounds(k) ≜
$[j \leftarrow k]^0$; $[I \leftarrow []]^1$;
$[ns \leftarrow 0]^2$; $[cs \leftarrow 0]^3$;
while $[j > 0]^4$ do
$\big($$[j \leftarrow j - 1]^5$;
$[a \leftarrow$ query$(I)]^6$;
$[ns \leftarrow$ updnscore$(ns, a)]^7$;
$[cs \leftarrow$ updcscore$(cs, a)]^8$;
$[I \leftarrow$ updI$(I, ns, cs)]^9$$\big)$
```

(a)



(b)

Fig. 7. (a) The simplified multiple rounds example (b) The estimated dependency graph from AdaptFun

Theorem 5.3 (Soundness of AdaptBD). *For every program c, we have*

$$\mathrm{AdaptBD}(\mathsf{G}_{\mathtt{est}}(c)) \geq \mathsf{A}_{\mathtt{est}}(c).$$

6 EXAMPLES

We illustrate here how our analysis work on two different examples. Our first example, Algorithm `multipleRounds` in Fig. 7(a), is a simplified form of the *monitor argument* by Rogers et al. [35]. The input k is the number of iterations. It uses a list I to track queries. Specifically at each iteration it updates I by using the result of a query which relies on I: query$(\chi[I])$. After k iterations, the algorithm returns the columns of the hidden database D which are not contained in the tracking list I. The functions updnscore(p, a), updcscore(p, a), update(I, ns, cs) simplify the computations of updating ns, cs and I. They depend on the result of the query but they do not perform queries themselves. Different from the code in the example `twoRounds(k)`, the query request, $[a \leftarrow$ query$(I)]^6$, in each loop iteration depends on the tracking list I, which in turn depends on all the queries from previous iterations. In this sense, all these k queries are fully adaptively chosen, and so the adaptivity is k. The estimated dependency graph $\mathsf{G}_{\mathtt{est}}$(`multipleRounds(k)`) is presented in Fig. 7(b) and we omitted the semantics-based dependency graph $\mathsf{G}_{\mathtt{trace}}$(`multipleRounds(k)`) because it has the same topology and only differs in weights. Our program analysis AdaptFun provides a tight upper bound for this example using AdaptBD(`multipleRounds(k)`). It first finds a path on the graph $\mathsf{G}_{\mathtt{est}}$(`multipleRounds(k)`) $a^6 : {}^{k}_{1} \to I^9 : {}^{k}_{0} \to ns^7 : {}^{k}_{0}$ with three weighted vertices. Then AdaptBD algorithm transforms this path into a walk $a^6 : {}^{k}_{1} \to I^9 : {}^{k}_{0} \to ns^7 : {}^{k}_{0} \to a^6 : {}^{k}_{1} \cdots$, where a^6, I^9, ns^7 are all visited k times respectively. So $\mathsf{A}_{\mathtt{est}}$(`multipleRounds(k)`) $= k$. We know for any initial trace τ_0, $\langle \tau_0, k \rangle \Downarrow_e \rho(\tau_0)k$, i.e., A(`multipleRounds(k)`)$(\tau_0) \leq \rho(\tau_0)k$ for any τ_0, and so what we have produced is a tight and sound bound.

As our second example, we want to show a program where AdaptFun is not precise because of its path-insensitive nature. One such program is `multiRoundsOdd(k)` presented in Fig. 8(a). For this program, AdaptFun over-approximate the adaptivity. The problem witnessed by this example occurs when the control flow is more sophisticated than what the static analysis can handle. multiRoundsOdd(k) has adaptivity $1 + k$ and a while loop with two paths. In each iteration, the queries $[y \leftarrow$ query$(\chi[x])]^5$ and $[p \leftarrow$ query$(\chi[x])]^6$ are based on the results of previous queries stored in x, similarly to Example 6. The difference is that only the query answer from $[y \leftarrow$ query$(\chi[x])]^5$ in the first branch is used in the query request command at line 7, $[x \leftarrow$ query$(\chi(\ln(y)))]^7$. However, this branch is only executed in even iterations ($j = 0, 2, \cdots$). From the semantics-based dependency graph in Figure 8(b), the weight function $\lambda\tau_0 \,.\, \rho(\tau_0)\frac{k}{2}$ for the vertex y^5 counts the number of times $[y \leftarrow$ query$(\chi[x])]^5$ is evaluated during the program execution under an initial trace τ_0, i.e., half of the initial value of k from τ_0. However, AdaptFun fails to realize that all the odd iterations only execute the first branch and that only even iterations

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

(a)

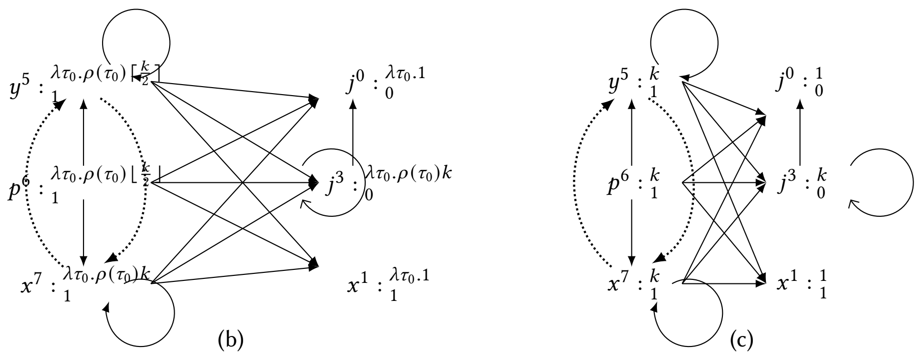

$y^5 : \overset{\lambda\tau_0.\rho(\tau_0)\lceil\frac{k}{2}\rceil}{1}$
$p^6 : \overset{\lambda\tau_0.\rho(\tau_0)\lfloor\frac{k}{2}\rfloor}{1}$
$x^7 : \overset{\lambda\tau_0.\rho(\tau_0)k}{1}$
$j^0 : \overset{\lambda\tau_0.1}{0}$
$j^3 : \overset{\lambda\tau_0.\rho(\tau_0)k}{0}$
$x^1 : \overset{\lambda\tau_0.1}{1}$
(b)
$y^5 : \overset{k}{1}$
$p^6 : \overset{k}{1}$
$x^7 : \overset{k}{1}$
$j^0 : \overset{1}{0}$
$j^3 : \overset{k}{0}$
$x^1 : \overset{1}{1}$
(c)

Fig. 8. (a) The multiple rounds odd example (b) The semantics-based dependency graph (c) The estimated dependency graph from AdaptFun.

execute the second branch. So it considers both branches for every iteration when estimating the adaptivity. In this sense, the weight estimated for y^5 and p^6 are both k as in Figure 8(c). As a result, AdaptFun computes $y^5 \longrightarrow x^7 \longrightarrow y^5 \longrightarrow \cdots \longrightarrow x^7$ as the walk of maximal query length in Figure 8(c) where each vertex is visited k times, and so the estimated adaptivity is $1 + 2 * k$, instead of $1 + k$.

7 IMPLEMENTATION

Set Up. We implemented AdaptFun as a tool that takes a labeled command as input and outputs two upper bounds on the program adaptivity and the number of query requests respectively. This implementation consists of an abstract control flow graph generation, edge estimation (as presented in Section 5.1.2), and weight estimation (as presented in Section 5.1.3) in Ocaml, and the adaptivity computation algorithm shown in Section 5.3 in Python. The OCaml program takes the labeled command as input and outputs the program-based dependency graph and the abstract transition graph, which we feed into the Python program and the Python program provides the adaptivity upper bound and the query number as the final output.

Accuracy and Performance. We first evaluate our implementation in terms of its accuracy and performance in estimating the adaptivity. To do this we consider 23 example programs with the evaluation results shown in Table 1.

In this table, the first column is the name of each program, the second column is the program adaptivity, the third column is the output of the AdaptFun implementation, which consists of two expressions: the first one is the upper bound for adaptivity and the second one is the upper bound for the total number of query requests in the program. And the last column is the performance evaluation w.r.t. the program size in terms of lines of code. The performance evaluation contains three parts. The first part is the running time of the Ocaml code, which parses the program and generates the $\mathsf{G}_{\mathsf{est}}(c)$. The second and third parts are the running times of the reachability bound analysis algorithm and the adaptivity computation algorithm, AdaptBD(c).

The examples show a wide range of values for their adaptivity. The first two programs twoRounds(k), mR(k) are the same as Fig. 3(a) and Fig. 7(a). AdaptFun computes tight adaptivity bound for the first three examples. The third example is a logistic regression algorithm with gradient decent (short for LRGD). For the fourth program mROdd(k), AdaptFun outputs an over-approximated upper bound $2 + \max(1, 2k)$ because of the path insensitive nature of the weight estimation algorithm, as

Table 1. Accuracy and Performance Evaluation of AdaptFun implementation

Program c	*adaptivity*	AdaptFun		L.O.C	running time (second)		
		$\mathsf{A}_{\mathsf{est}}$	query#		graph	weight	AdaptBD
twoRounds(k)	2	2	$k+1$	8	0.0005	0.0017	0.0003
mR(k)	k	k	k	10	0.0012	0.0017	0.0002
1RGD(k, r)	k	k	$2k$	10	0.0015	0.0072	0.0002
mROdd(k)	$1+k$	$2+\max(1,2k)$	$1+3k$	10	0.0015	0.0061	0.0002
mRSingle(k)	$1+k$	$1+\max(1,k)$	$1+k$	9	0.0011	0.0075	0.0002
seqRV()	4	4	4	4	0.0011	0.0003	0.0001
ifVD()	2	2	3	5	0.0010	0.0005	0.0001
ifCD()	3	3	4	5	0.0005	0.0003	0.0001
loop(k)	$1+k/2$	$1+\max(1,k/2)$	$1+k/2$	7	0.0021	0.0015	0.0001
loopRV(k)	$1+2k$	$1+2k$	$2+3k$	9	0.0016	0.0056	0.0001
loopVCD(k)	$1+2Q_m$	$Q_m+\max(1,2Q_m)$	$2+2Q_m$	6	0.0016	0.0007	0.0001
loopMPVCD(k)	$2+Q_m$	$2+Q_m$	$2+2Q_m$	9	0.0017	0.0043	0.0001
loop2VD(k)	$2+k^2$	$3+k^2$	$1+k+k^2$	10	0.0018	0.0126	0.0001
loop2RV(k)	$1+k+k^2$	$2+k+k^2$	$2+k+k^2$	10	0.0017	0.0186	0.0001
loop2MV(k)	$1+2k$	$1+\max(1,2k)$	$1+k+k^2$	10	0.0016	0.0071	0.0001
loop2MPRV(k)	$1+k+k^2$	$3+k+k^2$	$2+2k+k^2$	10	0.019	0.0999	0.0002
loopM(k)	$1+k$	$2+\max(1,2k)$	$1+3k$	9	0.0017	0.0062	0.0001
loopM2(k)	$1+k$	$2+k$	$1+3k$	9	0.0017	0.0062	0.0001
loop2R(k)	$1+3k$	$1+3k$	$1+3k+k^2$	11	0.019	0.2669	0.0007
mR(k, N)	k	k	k	27	0.0026	85.9017	0.0004
mRCom(k)	$2k$	$2k$	$2k$	46	0.0036	5104	0.0013
seqCom(k)	12	12	326	502	0.0426	1.2743	0.0223
tRCom(k)	2	$*$	$1+5k+2k^2$	42	0.0026	$*$	$*$
jumbo(k)	$\max(20, 8+k^2)$	$*$	$44+k+k^2$	71	0.0035	$*$	$*$
big(k)	$22+k+k*k$	$*$	$121+11k+4k^2$	214	0.0175	$*$	$*$

discussed in Section 6. The 5th program, mRSingle(k) is the example program in Fig. 6. AdaptFun outputs $1+\max(1,k)$, which is tight with respect to our adaptvitiy definition in Def. 7. However, because Def. 7 is a loose definition of mRSingle(k)'s actual adaptivity rounds, the result is still an over-approximation.

The next thirteen programs in Table 1 from 6th to 19th row are handcrafted programs based on benchmarks from the work by Gulwani et al. [24]. They all have small sizes but complex structures in order to test the programs under different situations including data, control dependency, the multiple paths nested loop with related counters, etc. The names of these programs obey the convention that, if means there is an if control in the program; loop means there is while loop and loop2 represents two levels nested loop in the program; C denotes Control; D for Dependency; V for Variable; M for Multiple; P for Path and R for Related.

Our algorithm computes tight bounds for the adaptivity of programs from row six, seqRV() to row fourteen, loop2MPRV(k) and over-approximate the *adaptivity* for the examples in row fifteen and sixteen because of the path-insensitive nature of the algorithm computing the weights.

The last six programs are synthesized programs composed of the previous programs in order to test the performance limitation when the input program is large. From the evaluation results, the performance bottleneck is the reachability bound analysis algorithm used to compute the weights.

The reachability-bound analysis algorithm presented in Section 5.1.3 time out after 5 minutes when evaluating the examples tRCom(k), jumbo and big(k).

Alternative Implementations. With our second evaluation we show the need of the components of AdaptFun. For this, we implement three alternative versions of AdaptFun where different components are omitted or replaced with components with weaker guarantees. We show the results of the comparison in Table 2. We use the same set of example programs as in Table 1 and present their estimated adaptivity rounds, query numbers and the running time. For each alternative implementation, we marked the result in red if it is imprecise w.r.t. the true value.

1. In column AdaptFun-I, the implementation replaces the reachability bound analysis algorithm in Section 5.1.3 with a light reachability bound analysis algorithm and computes the *adaptivity* for jumboS, jumbo and big effectively. The results show that the alternative one computes tight

Table 2. Accuracy Evaluation of AdaptFun Alternative Implementations

Program c	AdaptFun-I		AdaptFun-II		AdaptFun-III		running time
	$\mathsf{A}_{\mathsf{est}}$	query#	$\mathsf{A}_{\mathsf{est}}$	query#	$\mathsf{A}_{\mathsf{est}}$	query#	AdaptFun-I
twoRound(k)	2	$k+1$	2	$k+1$	2	2	0.0010
mR(k)	$\max(1,k)$	k	$\max(1,k)$	k	1	1	0.0016
lRGD(k, r)	$\max(1,k)$	$2k$	$\max(1,k)$	$2k$	1	2	0.0019
mROdd(k)	$2+\max(1,2k)$	$1+3k$	$2+\max(1,2k)$	$1+3k$	4	4	0.0019
mRSingle(k)	$1+\max(1,k)$	$1+k$	$1+\max(1,k)$	k	2	2	0.0015
seqRV()	4	4	4	4	4	4	0.0001
ifVD()	2	3	2	3	2	3	0.00012
ifCD()	3	4	2	4	3	4	0.0007
loop(k)	$1+\max(1,\frac{k}{2})$	$1+\frac{k}{2}$	$1+\max(1,\frac{k}{2})$	$1+\frac{k}{2}$	2	2	0.0023
loopRV(k)	$1+\max(1,2k)$	$2+3k$	$1+\max(1,2k)$	$2+3k$	4	4	0.0019
loopVCD(k)	$Q_m+\max(1,2Q_m)$	$2+2Q_m$	2	$2+2Q_m$	3	4	0.0019
loopMPVCD(k)	$2+Q_m$	$2+2Q_m$	2	$2+2Q_m$	3	4	0.0020
loop2VD(k)	$3+k^2$	$2+k+k^2$	$3+k^2$	$2+k+k^2$	4	4	0.0021
loop2RV(k)	$2+k+k^2$	$2+k+k^2$	$2+k+k^2$	$2+k+k^2$	4	4	0.0021
loop2MV(k)	$1+\max(1,2k)$	$1+k+k^2$	$1+\max(1,2k)$	$1+k+k^2$	3	3	0.0019
loop2MPRV(k)	$3+k+k^2$	$2+2k+k^2$	$3+k+k^2$	$2+2k+k^2$	5	5	0.0194
loopM(k)	$2+\max(1,2k)$	$1+3k$	$2+\max(1,2k)$	$1+3k$	4	4	0.0021
loopM2(k)	$2+k$	$1+3k$	$2+k$	$1+3k$	3	4	0.0021
loop2R(k)	$2+3k+k^2$	$1+k+k^2$	$2+3k+k^2$	$1+k+k^2$	4	2	0.0199
mR(k, N)	k	k	k	k	2	6	0.0033
mRCom(k)	$2k$	$2k$	$2k$	$2k$	1	2	0.0052
seqCom(k)	12	326	12	326	12	326	0.0652
tRCom(k)	2	$1+5k+2k^2$	$*$	$*$	2	8	0.0034
jumbo(k)	$\max(20,6+k+k^2)$	$44+k+k^2$	$*$	$*$	14	46	0.0123
big(k)	$28+k+k^2$	$121+11k+4k^2$	$*$	$*$	14	136	0.0181

bounds for all the examples from line:1 to line:14 and over-approximates the *adaptivity* for 15^{th} and 16^{th} due to path-insensitivity similar to the AdaptFun. For the 17^{th} example (loop2R), AdaptFun gives a tight upper bound while the alternative one gives a loose bound, so we keep both.

2. In AdaptFun-II, we remove the control flow analysis from Definition 8. The estimated data dependency only considers the data flow. However, the adaptivity should consider both the data and control flow, so results here are unsound. For example, program `ifCD`, `loopMPVCD(k)` and `loopVCD(k)` all have control dependency. However, the alternative implementation II in Table 2 estimates the adapvitiy rounds of 2 because it fails to identify the dependency through control flow.

3. In AdaptFun-III, we remove the reachability bound estimation from Section 5.1.3. We assign the weight 1 to every labeled variable on the estimated dependency graph. The adaptivity estimation results are all integers and does not depend on any program inputs. When instantiating the input variable with varied values, this estimated result is either greater or smaller than the true adaptivity.

In the last column, we present only the performance of the AdaptFun-I to compare the light reachability-bound analysis implementation with the original reachability-bound analysis. The performance of AdaptFun-II is similar to AdaptFun because they are based on the same reachability-bound analysis implementation. For the same reason, the AdaptFun-III has slightly better running time than AdaptFun-I. Overall, AdaptFun gives accurate estimated adaptivity upper bounds.

Effectiveness Evaluation. With our last evaluation we want to show the effectiveness of AdaptFun in terms of reducing the generalization error of real-world data analysis programs.

To do this we consider a set of benchmarks including nine real-world data analyses from our examples in Table 1 instantiated with different parameters, four programs implementing the algorithms from [30] and four data analysis programs from sklearn [8] benchmark.

We use acronyms for these programs to have a better presentation of our evaluation in a single table. We use 3(4)DeLRGD for 3(4)DegreeLRGD, RQ for repeatedQuery, nDPair for nDimPairwise, DT, DTOVR for decisionTree(OVR), LR, LROVR for logisticRegression(OVR). Similarly, we shorten the mechanisms: GS for Gaussian, DS for Data Split, TS for ThresholdOut.

For each program, we show in Table 3 the generalization errors when running without a mechanism and running with different mechanisms over uniformly generated training data.

Table 3. Evaluation of Data Analyses Generalization Error Using AdaptFun

Program c	AdaptFun		rmse with mechanism(k,m,n=10)				rmse with mechanism(k,m,n=1000)			
	A_{est}	query#	None	DS	GS	TS	None	DS	GS	TS
`twoRounds`	2	$k+1$	0.0006	0.0015	0.0004	**0.001**	0.050	**0.028**	0.031	0.040
`mR`	k	k	0.16	0.1545	0.1087	**0.1035**	0.066	0.050	**0.036**	0.064
`mROdd`	k	$2 \times k$	0.9375	0.999	0.7427	**0.4016**	0.211	0.220	**0.059**	0.171
`mRSingle`	k	k	0.8074	0.3600	0.7506	**0.4036**	0.761	0.758	**0.509**	0.593
`lRGD`	k	$2 \times k$	0.12	0.116	0.10	**0.06**	0.216	0.209	**0.014**	0.210
`3DimLRGD`	k	$3 \times k$	0.1159	0.10	0.1079	**0.092**	0.1966	0.1901	**0.1751**	0.1810
`4DimLRGD`	k	$4 \times k$	0.61	0.1080	0.30	**0.1399**	0.1112	0.1032	**0.0961**	0.1000
`3DeLRGD`	k	$3 \times k$	0.10	0.10	0.06	**0.04**	0.1096	0.1056	**0.0098**	0.1004
`4DeLRGD`	k	$4 \times k$	0.076	0.0984	0.0719	**0.064**	0.1084	0.1058	**0.1052**	0.1055
`DT`	k	k	0.0948	0.0948	0.0447	**0.0383**	1.465	1.283	**1.379**	1.414
`LR`	k	k	0.1316	0.1219	0.0893	**0.0632**	0.152	0.001	**0.001**	0.002
`DTOVR`	$k \times m$	$k \times m$	0.14142	0.1241	**0.0864**	0.1167	0.055	0.053	**0.007**	0.036
`LROVR`	$k \times m$	$k \times m$	0.0957	0.0917	**0.0815**	0.1092	1.000	1.000	**0.999**	1.002
`RQ` [30]	$m \times 2^m$	$m \times 2^m$	0.75	0.89	**6.01**	1.89	239.0	21.5	**18.557**	141.974
`nDPair` [30]	m	$m \times n$	0.34	0.10	0.19	**0.27**	0.0999	0.0999	0.0970	**0.0999**
`bestArm` [30]	n	$m \times n$	1.82	1.46	2.67	**0.81**	2.0452	1.3955	3.4147	**1.2871**
`lilUCB` [30]	n	$m \times n$	3.07	3.11	3.19	**1.79**	3.0174	3.137	3.5245	**2.3865**

The average generalization error of each program is measured by root-mean-square error. The root-mean-square error without any mechanism is shown in column "None". we mark in bold the rmse produced by the mechanism chosen by AdaptFun using the following heuristics. According to Theorem 2.2, Theorem 2.3 and Theorem 2.1. we compare the value of $\sqrt{\texttt{query\#}}$ and $A_{est}\sqrt{\log(\texttt{query\#})}$ and choose DataSplit if $\sqrt{\texttt{query\#}} \gg A_{est}\sqrt{\log(\texttt{query\#})}$ and Gaussian mechanism if $\sqrt{\texttt{query\#}} \ll A_{est}\sqrt{\log(\texttt{query\#})}$ and Thresholdout mechanism if these two quantities are close. It is worth stressing that these are rough heuristics since we are omitting the constants hidden in these theorems. We leave the problem to identify the best heuristic to future work.

We first evaluate example programs from Table 1 as the baseline evaluation to show that our tool can choose the mechanism that performs best in reducing the generalization error as we claimed. `twoRound` is trained to classify a uniformly generated data set of size n and k features and a randomly selected label from $\{0, 1\}$. With only two adaptive rounds, by instantiating $k = 1000$ and $n = 1000$, this algorithm overfitted and produced the generalization error 0.050. Since $2*\sqrt{\log(1000)} \ll \sqrt{1000}$, AdaptFun chooses the DataSplit mechanism. Evaluation result(**0.028**) shows that we choose the best. Then by instantiating $k = 10$, we see that $10 * \sqrt{\log(10)}$ is close to $\sqrt{10}$ and our tool chooses ThresholdOut, while Gaussian and ThresholdOut mechanism both perform well.

The 2nd to 5th programs represent different algorithms for fitting a two-dimensional data set of size n into a linear relation. The number of iterations is k and our tool identifies this as the adaptivity for these programs. By instantiating $k = 1000$ and $n = 1000$, we have $1000 * \sqrt{\log(1000)} \gg \sqrt{1000}$ which means that our tool chooses the Gaussian mechanism. When $k = 10$ and $n = 10$ instead, our tool chooses the ThresholdOut mechanism. The evaluation results in Table 3 show again that the chosen mechanism performs best among other mechanisms in reducing the generalization error.

`3DimLRGD` and `4DimLRGD` are the generalizations of LRGD into 3D and 4D data set of size n. The number of iterations (i.e., the fitting depth) is still k, and AdaptFun estimates correctly the adaptivity rounds k for these two programs. Instantiating k and n with 1000 and 1000, our tool chooses the Gaussian mechanism which still reduces rmse the most. When we instantiate k and n with 10 and 10, the ThresholdOut mechanism recommended by our tool also performs very well in Table 3.

We have four data analyses which are the implementation of algorithms from [30] within the *Guess and Check* [35] framework. Program `lilUCB` and `bestArm` are the algorithms to solve the best arm problem in the stochastic multi-armed bandit (MAB) setting with m total number of arms in order to identify the longest arm through sampling at most n times. Our tool identifies the number of sampling times n is the adaptivity rounds. Program `repeatedQuery` and `nDimPairwise` are the algorithms for ranking a 1-demensional dataset of size n via querying the data m times. Our tool identifies the adaptvitiy w.r.t. the sampling times for these two programs as m and $m \times 2^m$.

By instantiating $n = 1000$ and $m = 1000$, these programs produce large generalization errors. In this situation, $1000 * \sqrt{\log(1000^2)}$ and $\sqrt{1000^2}$ are close to each other, our tool chooses the Thresholdout mechanism. Evaluation results in Table 3 shows that both Thresholdout and Gaussian mechanism can reduce the rmse the most. When we look at a smaller n and m ($n, m = 10$), because the query number is $m \times n$ and our A_{est} is either m or n in these examples, our heuristic still chooses ThresholdOut mechanism in three of the four examples in the *Guess and Check* framework, `lilUCB`, `nDimPairwise` and `bestArm` and our recommendation performs well. The only exception is the `repeatedQuery` example, whose A_{est} and query number are both quite big as $m \times 2^m$, and out tool then chooses Gaussian mechanism, which does not perform well(6.01).

For the four data analysis programs, from sklearn [8] benchmark, the first two programs, `decisionTree` and `logisticRegression` are the implementations of the decision tree, logistic regression classifiers with $O(n)$ fitting depth, and the grid search hyperparameter selects an algorithm for each classifier with hyperparameter space of constant size c. The next two programs `decisionTreeOVR` and `logisticRegressionOVR` are the implementations of the decision tree, logistic regression, equipped with one v.s. rest model with k fitting depth for classifying the dataset with m classes. They are evaluated to classify a uniformly generated data set with m randomly selected classes label. For these programs, we translate them manually into our syntax and evaluate their adaptivity. By adaptivity analysis, our tool identified that the fitting depth is indeed the adaptivity rounds of these programs. Programs `decisionTreeOVR` and `logisticRegressionOVR` have $k * m$ adaptivity rounds, by instantiating $k = 1000$ and $m = 1000$, AdaptFun chooses Gaussian mechanism, although it is not optimal for the decision tree example. While when we have a smaller k and m ($k, m = 10$), the choice of our tool is reliable.

In summary, our experimental result shows that most of the time the knowledge of the adaptivity and the number of queries of a data analysis allow us to choose the best mechanism to reduce the generalization error. We expect that this situation can be further improved by using more precise heuristic than the ones we use here.

8 RELATED WORK

Dependency Definitions and Analysis. There is a vast literature on dependency definitions and dependency analysis. We consider a semantics definition of dependencies which consider (intraprocedural) data and control dependency [9, 13, 34]. Our definition is inspired by classical works on traditional dependency analysis [15] and noninterference [23]. Formally, our definition is similar to the one by Cousot [12], which also identifies dependencies by considering differences in two execution traces. However, Cousot excludes some forms of implicit dependencies, e.g. the ones generated by empty observations, which instead we consider. Common tools to study dependencies are dependency graphs [20]. We use here a semantics-based approach to dependency graph similar, for example, to works by Austin and Sohi [5], Hammer et al. [26] and [27]. Our approach shares some similarities with the use of dependency graphs in works analyzing dependencies between events, e.g. in event programming. Memon [33] uses an event-flow graph, representing all the possible event interactions, where vertices are GUI event edges represent pairs of events that can be performed immediately one after the other. In a similar way, we use edges to track the may-dependence between variables looking at all the possible interactions. Arlt et al. [4] use a weighted edges indicating a dependency between two events, e.g. one event possibly reads data written by the other event, with the weight showing the intensity of the dependency (the quantity of data involved). We also use weights but on vertices and with different meaning, they are functions describing the number of times the vertices can be visited given an initial state. Differently from all these previous works, we use a dependency graph with quantitative information needed to identify the length of chain of dependencies. Our weight estimation is inspired by works in complexity analysis and WCET. Specifically, it is inspired by works on reachability-bound analysis using program abstraction and invariant inference [24, 25, 38] and work on invariant inference through cost equations and ranking functions [2, 3, 11, 21].

Generalization in Adaptive Data Analysis. Starting from the works by Dwork et al. [18] and Hardt and Ullman [28], several works have designed methods that ensure generalization for adaptive data analyses [7, 16, 17, 19, 31, 35, 39, 40]. Several of these works drew inspiration from differential privacy, a notion of formal data privacy. 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. [42]. 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 [41] 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. [32] 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. [6] and Aguirre et al. [1] 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.

9 CONCLUSION AND FUTURE WORKS

We presented AdaptFun, a program analysis useful to provide an upper bound on the adaptivity of a data analysis, as well as on 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. We showed the applicability of our approach by implementing and experimentally evaluating our program analysis.

As future work, we plan to investigate the potential integration of AdaptFun in an adaptive data analysis framework like Guess and check by Rogers at al. [35]. As we discussed, this framework is designed to support adaptive data analyses with limited generalization error. As our experiments show, this framework could benefit from the information provided by AdaptFun to provide more precise estimate and improved confidence intervals. Another direction we will explore is to make the uppper bounds provided by AdaptFun more precise by integrating our algorithm with a path-sensitive approach.

REFERENCES

[1] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, Shin-ya Katsumata, and Tetsuya Sato. 2021. Higher-order probabilistic adversarial computations: Categorical semantics and program logics. *CoRR* abs/2107.01155 (2021). arXiv:2107.01155 https://arxiv.org/abs/2107.01155

[2] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. 2008. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5079)*, María Alpuente and Germán Vidal (Eds.). Springer, 221–237. https://doi.org/10.1007/978-3-540-69166-2_15

[3] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. 2010. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6337)*, Radhia Cousot and Matthieu Martel (Eds.). Springer, 117–133. https://doi.org/10.1007/978-3-642-15769-1_8

[4] Stephan Arlt, Andreas Podelski, Cristiano Bertolini, Martin Schäf, Ishan Banerjee, and Atif M. Memon. 2012. Light-weight Static Analysis for GUI Testing. In *23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27-30, 2012*. IEEE Computer Society, 301–310. https://doi.org/10.1109/ISSRE.2012.25

[5] Todd M. Austin and Gurindar S. Sohi. 1992. Dynamic Dependency Analysis of Ordinary Programs. In *Proceedings of the 19th Annual International Symposium on Computer Architecture. Gold Coast, Australia, May 1992*, Allan Gottlieb (Ed.). ACM, 342–351. https://doi.org/10.1145/139669.140395

[6] Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, and Pierre-Yves Strub. 2021. Mechanized Proofs of Adversarial Complexity and Application to Universal Composability. *IACR Cryptol. ePrint Arch.* 2021 (2021), 156. https://eprint.iacr.org/2021/156

[7] Raef Bassily, Kobbi Nissim, Adam D. Smith, Thomas Steinke, Uri Stemmer, and Jonathan R. Ullman. 2016. Algorithmic stability for adaptive data analysis. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, Daniel Wichs and Yishay Mansour (Eds.). ACM, 1046–1059. https://doi.org/10.1145/2897518.2897566

[8] Sklearn Benchmark. 2023. https://github.com/scikit-learn/scikit-learn/tree/main/examples

[9] Gianfranco Bilardi and Keshav Pingali. 1996. A framework for generalized control dependence. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*. 291–300.

[10] Guy Blanc. 2023. Subsampling Suffices for Adaptive Data Analysis. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, Orlando, FL, USA, June 20-23, 2023*, Barna Saha and Rocco A. Servedio (Eds.). ACM, 999–1012. https://doi.org/10.1145/3564246.3585226

[11] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2016. Analyzing Runtime and Size Complexity of Integer Programs. *ACM Trans. Program. Lang. Syst.* 38, 4 (2016), 13:1–13:50. http://dl.acm.org/citation.cfm?id=2866575

[12] Patrick Cousot. 2019. Abstract Semantic Dependency. In *Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11822)*, Bor-Yuh Evan Chang (Ed.). Springer, 389–410. https://doi.org/10.1007/978-3-030-32304-2_19

[13] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (1991), 451–490. https://doi.org/10.1145/115372.115320

[14] Yuval Dagan and Gil Kur. 2022. A bounded-noise mechanism for differential privacy. In *Conference on Learning Theory, 2-5 July 2022, London, UK (Proceedings of Machine Learning Research, Vol. 178)*, Po-Ling Loh and Maxim Raginsky (Eds.). PMLR, 625–661. https://proceedings.mlr.press/v178/dagan22a.html

[15] Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (1977), 504–513. https://doi.org/10.1145/359636.359712

[16] Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Roth. 2015. Generalization in Adaptive Data Analysis and Holdout Reuse. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett (Eds.). 2350–2358. https://proceedings.neurips.cc/paper/2015/hash/bad5f33780c42f2588878a9d07405083-Abstract.html

[17] Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Roth. 2015. The reusable holdout: Preserving validity in adaptive data analysis. *Science* 349, 6248 (2015), 636–638.

[18] Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Leon Roth. 2015. Preserving Statistical Validity in Adaptive Data Analysis. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, Rocco A. Servedio and Ronitt Rubinfeld (Eds.). ACM, 117–126. https://doi.org/10.1145/2746539.2746580

[19] Vitaly Feldman and Thomas Steinke. 2017. Generalization for Adaptively-chosen Estimators via Stable Median. In *Proceedings of the 30th Conference on Learning Theory, COLT 2017, Amsterdam, The Netherlands, 7-10 July 2017 (Proceedings of Machine Learning Research, Vol. 65)*, Satyen Kale and Ohad Shamir (Eds.). PMLR, 728–757. http://proceedings.mlr.press/v65/feldman17a.html

[20] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (1987), 319–349. https://doi.org/10.1145/24039.24041

[21] Antonio Flores-Montoya and Reiner Hähnle. 2014. Resource Analysis of Complex Programs with Cost Equations. In *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8858)*, Jacques Garrigue (Ed.). Springer, 275–295. https://doi.org/10.1007/978-3-319-12736-1_15

[22] Andrew Gelman and Eric Loken. 2014. The Statistical Crisis in Science. *Am Sci* 102, 6 (2014), 460.

[23] Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982.* IEEE Computer Society, 11–20. https://doi.org/10.1109/SP.1982.10014

[24] Sumit Gulwani, Sagar Jain, and Eric Koskinen. 2009. Control-flow refinement and progress invariants for bound analysis. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 375–385. https://doi.org/10.1145/1542476.1542518

[25] Sumit Gulwani and Florian Zuleger. 2010. The reachability-bound problem. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 292–304. https://doi.org/10.1145/1806596.1806630

[26] Christian Hammer, Martin Grimme, and Jens Krinke. 2006. Dynamic path conditions in dependence graphs. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2006, Charleston, South Carolina, USA, January 9-10, 2006*, John Hatcliff and Frank Tip (Eds.). ACM, 58–67. https://doi.org/10.1145/1111542.1111552

[27] Christian Hammer, Martin Grimme, and Jens Krinke. 2006. Dynamic path conditions in dependence graphs. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2006, Charleston, South Carolina, USA, January 9-10, 2006*, John Hatcliff and Frank Tip (Eds.). ACM, 58–67. https://doi.org/10.1145/1111542.1111552

[28] Moritz Hardt and Jonathan R. Ullman. 2014. Preventing False Discovery in Interactive Data Analysis Is Hard. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014.* IEEE Computer Society, 454–463. https://doi.org/10.1109/FOCS.2014.55

[29] John PA Ioannidis. 2005. Why most published research findings are false. *PLoS medicine* 2, 8 (2005), e124.

[30] Kevin G. Jamieson. 2015. The Analysis of Adaptive Data Collection Methods for Machine Learning.

[31] Christopher Jung, Katrina Ligett, Seth Neel, Aaron Roth, Saeed Sharifi-Malvajerdi, and Moshe Shenfeld. 2020. A New Analysis of Differential Privacy's Generalization Guarantees. 151 (2020), 31:1–31:17. https://doi.org/10.4230/LIPIcs.ITCS.2020.31

[32] Elisabet Lobo-Vesga, Alejandro Russo, and Marco Gaboardi. 2021. A Programming Language for Data Privacy with Accuracy Estimations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43, 2 (2021), 1–42.

[33] Atif M. Memon. 2007. An event-flow model of GUI-based applications for testing. *Softw. Test. Verification Reliab.* 17, 3 (2007), 137–157. https://doi.org/10.1002/stvr.364

[34] Lori L. Pollock and Mary Lou Soffa. 1989. An Incremental Version of Iterative Data Flow Analysis. *IEEE Trans. Software Eng.* 15, 12 (1989), 1537–1549. https://doi.org/10.1109/32.58766

[35] Ryan Rogers, Aaron Roth, Adam D. Smith, Nathan Srebro, Om Thakkar, and Blake E. Woodworth. 2020. Guaranteed Validity for Empirical Approaches to Adaptive Data Analysis. In *The 23rd International Conference on Artificial Intelligence and Statistics, AISTATS 2020, 26-28 August 2020, Online [Palermo, Sicily, Italy] (Proceedings of Machine Learning Research, Vol. 108)*, Silvia Chiappa and Roberto Calandra (Eds.). PMLR, 2830–2840. http://proceedings.mlr.press/v108/rogers20a.html

[36] Moritz Sinn, Florian Zuleger, and Helmut Veith. 2014. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 745–761. https://doi.org/10.1007/978-3-319-08867-9_50

[37] Moritz Sinn, Florian Zuleger, and Helmut Veith. 2017. Complexity and resource bound analysis of imperative programs using difference constraints. *Journal of automated reasoning* 59, 1 (2017), 3–45.

[38] Moritz Sinn, Florian Zuleger, and Helmut Veith. 2017. Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints. *J. Autom. Reason.* 59, 1 (2017), 3–45. https://doi.org/10.1007/s10817-016-9402-4

[39] Thomas Steinke and Lydia Zakynthinou. 2020. Reasoning About Generalization via Conditional Mutual Information. In *Conference on Learning Theory, COLT 2020, 9-12 July 2020, Virtual Event [Graz, Austria] (Proceedings of Machine*

Learning Research, Vol. 125), Jacob D. Abernethy and Shivani Agarwal (Eds.). PMLR, 3437–3452. http://proceedings.mlr.press/v125/steinke20a.html

[40] Jonathan R. Ullman, Adam D. Smith, Kobbi Nissim, Uri Stemmer, and Thomas Steinke. 2018. The Limits of Post-Selection Generalization. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 6402–6411. https://proceedings.neurips.cc/paper/2018/hash/77ee3bc58ce560b86c2b59363281e914-Abstract.html

[41] Daniel Winograd-Cort, Andreas Haeberlen, Aaron Roth, and Benjamin C. Pierce. 2017. A framework for adaptive differential privacy. *Proc. ACM Program. Lang.* 1, ICFP (2017), 10:1–10:29. https://doi.org/10.1145/3110254

[42] Samuel Yeom, Irene Giacomelli, Matt Fredrikson, and Somesh Jha. 2018. Privacy Risk in Machine Learning: Analyzing the Connection to Overfitting. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. IEEE Computer Society, 268–282. https://doi.org/10.1109/CSF.2018.00027

[43] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. 2011. Bound Analysis of Imperative Programs with the Size-Change Abstraction. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6887)*, Eran Yahav (Ed.). Springer, 280–297. https://doi.org/10.1007/978-3-642-23702-7_22

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009