Fairness-Aware Programming

Aws Albarghouthi
University of Wisconsin–Madison
aws@cs.wisc.edu

Samuel Vinitsky
University of Wisconsin–Madison
vinitskys@cs.wisc.edu

ABSTRACT

Increasingly, programming tasks involve automating and deploying sensitive decision-making processes that may have adverse impacts on individuals or groups of people. The issue of fairness in automated decision-making has thus become a major problem, attracting interdisciplinary attention. In this work, we aim to make fairness a first-class concern in programming. Specifically, we propose *fairness-aware programming*, where programmers can state fairness expectations natively in their code, and have a runtime system monitor decision-making and report violations of fairness.

We present a rich and general specification language that allows a programmer to specify a range of fairness definitions from the literature, as well as others. As the decision-making program executes, the runtime maintains statistics on the decisions made and incrementally checks whether the fairness definitions have been violated, reporting such violations to the developer. The advantages of this approach are two fold: (*i*) Enabling declarative mathematical specifications of fairness in the programming language simplifies the process of checking fairness, as the programmer does not have to write ad hoc code for maintaining statistics. (*ii*) Compared to existing techniques for checking and ensuring fairness, our approach monitors a decision-making program in the wild, which may be running on a distribution that is unlike the dataset on which a classifier was trained and tested.

We describe an implementation of our proposed methodology as a library in the Python programming language and illustrate its use on case studies from the algorithmic fairness literature.

CCS CONCEPTS

• **Theory of computation → Program specifications**; • **Software and its engineering → Specification languages**;

KEYWORDS

Probabilistic specifications; Fairness; Assertion languages; Runtime monitoring; Runtime verification

ACM Reference Format:
Aws Albarghouthi and Samuel Vinitsky. 2019. Fairness-Aware Programming. In *FAT* '19: Conference on Fairness, Accountability, and Transparency, January 29–31, 2019, Atlanta, GA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3287560.3287588

1 INTRODUCTION

With algorithmic decision-making becoming the norm, the issue of algorithmic fairness has been identified as a key problem of interdisciplinary dimensions. Across the vast computer science community, the past few years have delivered a range of practical techniques aimed at addressing the fairness question.

In this work, we aim to make fairness a first-class concern in programming. Specifically, we propose *fairness-aware programming*, where developers can state fairness expectations natively in their code, and have a runtime system monitor decision-making and report violations of fairness. This programming-language-based approach to fairness yields two advantages:

- The developer can declaratively state their fairness expectations natively in their code and have them checked. As such, the developer need not write ad hoc code for collecting statistics and checking whether fairness definitions are violated—they simply state the fairness definitions that they expect to hold. We argue that embedding notions of fairness directly within the programming language reduces the barrier to investigating fairness of algorithmic decision-making.
- The past few years have witnessed numerous techniques for constructing fair classifiers [1, 4, 11, 13, 25], as well as for testing and auditing fairness of existing decision-making procedures [2, 8, 12, 23]. Generally, these techniques assume a provided dataset or distribution that is representative of the population subject to decision-making. Unfortunately, the data may be itself biased—e.g., due to historical discrimination—or it may simply be unrepresentative of the actual population. By monitoring actual algorithmic decisions as they are made, we can check whether fairness is violated in the wild, which, for instance, may exhibit different population characteristics than the data used for training and validation in a machine learning setting.

Fairness Specifications Consider a software developer who constructed a decision-making procedure following the best practices for ensuring certain notions of fairness relevant to their task. Once they have deployed their decision-making procedure, they want to ensure that their fairness assumptions still hold. To enable them to do so, we propose treating fairness definitions as first-class constructs of a programming language. Thus, the developer can specify, in their code, that a given procedure f (perhaps a learned classifier) satisfies some notion of fairness φ. As the procedure makes the decisions, the runtime system monitors the decisions, potentially inferring that φ is violated, therefore alerting the developer.

The approach we propose is analogous to the notion of *assertions*, which are ubiquitous in modern programming languages. For instance, the developer might assert that `x > 0`, indicating that they expect the value of `x` to be positive at a certain point in the program. If this expectation ends up being violated by some execution of the program, the program crashes and the developer is alerted that their assertion has been violated. We propose doing the same for fairness definitions. The difficulty, however, is that fairness definitions are typically probabilistic, and therefore detecting their violation cannot be done through a single execution as in traditional assertions. Instead, we have to monitor the decisions made by the procedure, and then, using statistical tools, infer that a fairness property does not hold with reasonably high confidence.

To enable this idea, we propose a rather general and rich language of specifications that can capture a range of fairness definitions from the literature, as well as others. As an example, our specification language can capture notions of *group fairness*, for example, that of Feldman et al. [11]:

```
@spec(pr(r|s) / pr(r|¬s) > 0.8)
def f(x,s):
  ...
```

In this example, the developer has specified (using the annotation `@spec`) that the hiring procedure `f` (whose return variable is `r`) has a selection rate for minorities (denoted by `s`) that is at least 0.8 that of non-minorities.

Generally, our specification language allows arbitrary conditional expectation expressions over events concerning the input and return arguments of some procedure. Further, we extend the language to be able to specify *hyperproperties* [5], which refer to two different executions of a procedure. This allows us to specify, for example, properties like individual fairness, where we have to compare between *pairs* of inputs to a procedure that are similar. Section 2 provides concrete examples.

Runtime Analysis To determine that a procedure f satisfies a fairness specification φ, we need to maintain statistics over the inputs and outputs of f as it is being applied. Specifically, we compile the specification φ into *runtime monitoring* code that executes every time f is applied, storing aggregate results of every probability event appearing in φ. For instance, in the example above, the monitoring code would maintain the number of times the procedure returned `true` for a minority applicant. Every time `f` is applied, the count is updated if needed. In the case of hyperproperties, like individual fairness, the runtime system has to remember all decisions made explicitly, so as to compare new decisions with past ones.

Assuming that inputs to the decision-making procedure are drawn from an unknown distribution of the underlying population, we can employ concentration of measure inequalities to estimate the probability of each event in φ and therefore whether φ is violated, with some prespecified probability of failure. If the runtime code detects a violation, we may ask it to completely halt applications of the decision-making program until further investigation.

Python Implementation We have implemented a prototype of our proposed ideas in the Python programming language. Fairness specifications are defined as Python *decorators*, which are annotations that wrap Python functions and therefore affect their behavior. In our setting, we use them to intercept calls to decision-making functions and monitor the decisions made. To illustrate our approach, we consider two case studies drawn from the fairness literature and demonstrate how our approach may flag violations of fairness.

Contributions We summarize our contributions as follows:

- ***Fairness-aware programming:*** We propose a programming-language approach to fairness specification, where the developer declaratively states fairness requirements for sensitive decision-making procedures in their code.
- ***Incremental runtime analysis:*** We propose a runtime-checking technique that incrementally checks the provided fairness specifications every time a decision is made. The decision-making procedure is flagged when a fairness specification is falsified with high confidence—after witnessing enough decisions.
- ***Implementation & case studies:*** We describe an implementation of our proposed methodology as a library in the Python programming language—using Python decorators—and illustrate its use on case studies from the algorithmic fairness literature.

2 OVERVIEW & EXAMPLES

In this section, we provide a general overview of our fairness-aware programming approach and provide a set of examples to demonstrate its versatility.

We consider a programming language where the developer can annotate a procedure f with a specification φ, which is a probabilistic statement over the behavior of f. Here, f is assumed to be a procedure making a potentially sensitive decision, and φ is a fairness definition that we expect f to adhere to.

While our approach is generic, we adopt Python-like syntax in our examples, due to the popularity of the language for data-analysis tasks and the fact that we implemented our approach as a Python library. Given a Python procedure `f`, the fairness definition will be provided using a *decorator* of the form `@spec(`φ`)` that directly precedes `f`. Note that `f` may be a machine-learned classifier or manually written code. Throughout, we assume that the return value of `f` is stored in a local variable `r`.

Example A: Group Fairness Consider the scenario where a developer has trained a classifier `f` for deciding which job applicants to invite for a job interview. Following existing fairness techniques and data available to them, the developer ensured that `f` satisfies a form of group fairness where the selection rate from the minority group is very close to that of the majority group—assuming, for illustration, that we divide the population into minority and majority. The developer is aware that the fairness definition may not hold once the classifier is deployed, perhaps because the data used for training may exhibit historical biases or may be old and not representative of the current population distribution. As such, the developer annotates the procedure `f` as follows:

```
@spec(pr(r|s) / pr(r|¬s) > 0.8 ∧
      pr(r|s) / pr(r|¬s) < 1.2)
def f(x1,...,xn,s):
  ...
```

Here, `f` takes a set of attributes of the job applicant, `x1,...,xn`, along with a Boolean (sensitive) attribute `s` indicating whether the applicant is a minority. The fairness specification indicates that the selection rate of the minority applicants should be at most different from that of the majority applicants by a factor of 0.2.

Now, when `f` is deployed and is making interview decisions, the programming language incrementally improves an estimate for every probability expression `pr(...)` in the specification. This is performed automatically in the background using concentration inequalities, assuming job applications are independently and identically distributed following an unknown distribution D. The more decisions made by `f`, the more confident the runtime is about whether the specification holds. The developer can choose the level of confidence at which a violation of fairness is reported. For instance, the developer might configure the runtime to only report violations if the probability that the specification is violated is $\geqslant 0.9$.

To further ground our example in practice, the procedure `f` may be a wrapper for a saved PyTorch model [21], and `f` may be applied in real time to a stream of applications that are processed using, e.g., the `filter` operator in Apache Spark's [24] Python interface, which removes applications that do not return `true` on `f`. By annotating `f` with `@spec`, we ensure that it is monitored as it is being applied to the stream of job applications.

Example B: Robustness (individual fairness) The previous example considered selection rates amongst groups. Another important, albeit less studied, class of fairness definitions is individual fairness [9], where the goal is to ensure that similar individuals are treated similarly. Individual fairness can also be seen as a measure of *robustness* of a decision-maker: small changes to an individual's record should not change the decision. The following definition is an adaptation of the definition put forth by Bastani et al. [3], which was proposed for robustness of neural networks:

```
def sim(i,j): ... # Boolean function
@spec(pr(r^a ≠r^b|sim(x^a,x^b)) < 0.1)
def f(x):
  ...
```

Continuing our hiring scenario, the procedure `f` takes as input an application `x` and returns a Boolean decision. Notice that this specification here is slightly different from what we have considered thus far: it talks about two inputs and outputs of `f`, identified by a superscript. For a partial illustration, the expression

$$\mathit{pr}(\mathtt{sim}(\mathtt{x}^a,\mathtt{x}^b))$$

is interpreted as the probability that two applications drawn from the distribution are similar, as defined by the user-defined Boolean function `sim`. The full specification above says that the probability of making different decisions on two applicants $\mathtt{x}^a$ and $\mathtt{x}^b$ should be bounded by 0.1 (under the condition that the applicants are similar).

Notice that, in this setting, every time a decision is made, checking the specification involves comparing the decision to all previously made decisions and updating the statistics. This is because the specification is a so-called *hyperproperty* [5], since it compares multiple program executions.

Example C: Proxy Consider an algorithmic pricing scenario, where an e-commerce website presents custom prices depending on the user. Suppose that the algorithm uses browser type (e.g., Chrome, Firefox, etc.) as an attribute when deciding the price to present to a given user. And suppose that the developers of this algorithm, by examining their data, determined that it is safe to use browser type as it cannot be used as a proxy for minority status, ensuring that minorities receive fair pricing. To make sure that this is indeed true in practice, the developers can use the ϵ-proxy definition proposed by Datta et al. [6, 7]. This definition ensures that the *normalized mutual information* between the browser type and the sensitive attribute is bounded above by a small number ϵ.

```
@spec(1 - (en(b|s) + en(s|b))/en(b,s) < 0.05)
def f(x1,...,xn,b,s):
  ...
```

The pricing procedure `f` takes the browser type `b` and minority status `s`. The function `en` is *(conditional) entropy*; the expression on the left side of the inequality measures how strong a proxy is browser type for minority status. Note that we use `en` for brevity: it can be translated into an arithmetic expression over probabilities (`pr`).

Example D: Recommendation For our final example, consider a movie recommendation system, where user data has been used to train a recommender that, given a user profile, recommends a single movie, for simplicity. A key problem with recommender systems is isolation of similar users from certain types of recommendations (e.g., left-leaning social media users tend to see only articles from left-leaning websites, and vice versa). Suppose that the recommender was constructed with the goal of ensuring that male users are not isolated from movies with a strong female lead. Then, the developer may add the following specification to their recommender code.

```
@spec(pr(femaleLead(r)|s = male) > 0.2)
def f(x1,...,xn,s):
  ...
```

The above specification ensures that for male users, the procedure recommends a movie with a female lead at least 20% of the time.

Combining Fairness Definitions In the aforedescribed examples, we considered a single fairness definition at a time. Our technique is not limited to that: A developer can simply combine two different fairness definitions, e.g., φ_1 and φ_2, by conjoining them, i.e., using

$$\mathtt{@spec}(\varphi_1 \wedge \varphi_2)$$

Analogously, the developer may want to specify that at least one of the two definition of fairness is not violated, in which case they can disjoin the two definitions using

$$\mathtt{@spec}(\varphi_1 \vee \varphi_2)$$

3 A FAIRNESS-AWARE LANGUAGE

In this section, we formalize a simple abstract programming language that includes a fairness specification language. Concretely, for the purpose of our discussion, we will limit programs in our language to a pair (f, φ), denoting a single decision-making procedure f and an associated fairness specification φ.

Decision-Making Procedures We define a decision-making procedure as a function

$$f : X \to [c, d]$$

from elements of some type X to real values in the non-empty range $[c, d] \subset \mathbb{R}$.[1] The procedure f is implemented using standard imperative programming constructs—assignments, conditionals, loops, etc. The details of how f is implemented are not important for our exposition, and we therefore omit them. The only assumption we make is that f is a pure function—i.e., its results are not dependent on global variables.[2]

We shall assume that elements of the type X are vectors, and use the variable $\boldsymbol{x}$ to denote the input arguments of procedure $f(\boldsymbol{x})$. When needed, we will expand $f(\boldsymbol{x})$ as $f(x_1, \ldots, x_n)$.

Fairness Specification Language The language of specifications φ is presented as a grammar in Figure 1. Intuitively, a specification φ is a Boolean combination of inequalities of the form ETerm $> c$, where $c \in \mathbb{R}$ and ETerm is an arithmetic expression over expectations.

An expectation $\mathbb{E}[\cdot]$ has one of two forms: $\mathbb{E}[E]$ or $\mathbb{E}[H]$. An *expression* E is a numerical expression over the input variable $\boldsymbol{x}$ and a special variable r which denotes the value of $f(\boldsymbol{x})$.

Example 3.1. The expectation

$$\mathbb{E}[r > 0]$$

denotes the probability that f returns a positive value. The expectation

$$\mathbb{E}[x_1 + x_2]$$

denotes the expected value of the sum of the first two arguments of $f(x_1, \ldots, x_n)$.

Note that conditional probabilities and expectations can be encoded in our grammar. For instance, a conditional probability $\mathbb{P}[E_1 \mid E_2]$ can be written as $\mathbb{E}[E_1 \wedge E_2]/\mathbb{E}[E_2]$.

The *hyperexpression* H is non-standard: it is a numerical expression over two annotated copies of program variables, $\{\boldsymbol{x}^a, r^a\}$ and $\{\boldsymbol{x}^b, r^b\}$. We assume that the expression is symmetric, meaning that if we swap the a and the b variables, we still get the same result. As an example, consider the following expectation:

$$\mathbb{E}[r^a \neq r^b]$$

which denotes the probability that two runs of the function result in different answers. Notice that the expression $r^a \neq r^b$ is symmetric, since it is always equal to $r^b \neq r^a$ for any values of r^a and r^b. Another example of a hyperexpression appears in our robustness example from Section 2:

$$\mathbb{E}[\boldsymbol{x}^a \approx \boldsymbol{x}^b]$$

[1]We model Boolean procedures by returning $\{0, 1\} \subset \mathbb{R}$ values.
[2]Our technique can easily handle randomized functions, but we assume deterministic functions for conciseness.

$$\begin{aligned}
\varphi \in \text{Spec} &:= \text{ETerm} > c \\
&\mid \text{Spec} \wedge \text{Spec} \\
&\mid \text{Spec} \vee \text{Spec} \\
\\
\text{ETerm} &:= \mathbb{E}[E] \\
&\mid \mathbb{E}[H] \\
&\mid c \in \mathbb{R} \\
&\mid \text{ETerm } \{+, -, \div, \times\} \text{ ETerm}
\end{aligned}$$

Figure 1: Grammar of a specification φ. Spec **and** ETerm **are non-terminals. The expressions E and H are defined in the text.**

where the operator $\approx$ is some symmetric Boolean function indicating whether the two inputs $\boldsymbol{x}^a$ and $\boldsymbol{x}^b$ are similar. Thus, this denotes the probability that two inputs to f are similar.

Semantics of Specifications We have thus far only provided an intuition for what it means for a specification to be satisfied or violated. We will now formalize this notion.

We assume that there is a probability distribution D over the domain X of the decision-making procedure f. This distribution characterizes the population from which inputs to f are drawn—e.g., population of job or loan applicants. Now, each expectation $\mathbb{E}[E]$ is interpreted such that $\boldsymbol{x} \sim D$ and $r = f(\boldsymbol{x})$.

In the case of an expectation over a hyperexpression, $\mathbb{E}[H]$, we consider two identical and independent copies of $\boldsymbol{x}$, called $\boldsymbol{x}^a$ and $\boldsymbol{x}^b$. Each expectation $\mathbb{E}[H]$ is interpreted such that $\boldsymbol{x}^a \sim D$ and $\boldsymbol{x}^b \sim D$, and where $r^a = f(\boldsymbol{x}^a)$ and $r^b = f(\boldsymbol{x}^b)$.

Now that we have defined interpretations of expectations, we say that a specification φ is *satisfied* by procedure f if it evaluates to *true* after plugging in the numerical values of the expectations. Otherwise, we say φ is *violated* by f.

Example 3.2. To give a simple illustrative example, let the distribution D be the normal distribution centered at 0 with scale 1. Now consider the identity function $f(x) = x$ and the specification

$$\varphi \triangleq \mathbb{E}[r^a > 0 \wedge r^b > 0] > 0.5$$

As described above, we evaluate this specification over two variables drawn from D:

$$x^a \sim D$$
$$x^b \sim D$$

φ states that $\mathbb{E}[f(x^a) > 0 \wedge f(x^b) > 0] > 0.5$. This is not true, since f is the identity function and drawing two positive values has probability 0.25. Therefore, we say that φ is violated by f.

4 RUNTIME FAIRNESS ANALYSIS

In this section, we describe a program transformation that instruments the decision-making procedure f with additional functionality so as to detect a violation of a fairness property φ.

The setting we work in is one where f is being continuously applied to inputs $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots$, which are viewed as independent random variables following an unknown distribution D. At every step i—i.e., after making the ith decision—we want to check if the property φ is violated using the empirical observations we have made thus far. To achieve this, we summarize the instrumentation as follows:

- ***Empirical expectations***: For every expectation $\mathbb{E}[E]$, we maintain the sum of observed values of E, which we shall denote E^s. Thus, at any point i in the process, we can compute the empirical estimate of $\mathbb{E}[E]$, namely, E^s/i. Note that at every step, we simply update E^s by adding the new observation. Thus, we do not need to remember past observations—just their aggregate. With hyperexpressions, however, we need to remember past observations, as every new observation has to be compared to past observations.
- ***Concentration inequalities***: Once we have updated our estimates for every expectation, we can bound the estimates using a concentration of measure inequality, e.g., Hoeffding's inequality, which gives us an (ϵ, δ)-guarantee on the empirical expectations. The more observations we make, the tighter the bounds get. It is important to note, though, since we are working in an online setting, where we are continuously making statistical tests, we need to ensure that we handle the *multiple comparisons* problem.
- ***Checking violations***: Finally, to check violations of φ, we plug in the empirical estimates of expectations into φ and evaluate it using the uncertain (ϵ, δ) values we computed. Specifically, we evaluate φ by appropriately propagating errors across its operators (e.g., $+$, $\times$, $\wedge$, etc.). If we deduce that φ is violated with a probability greater than some threshold κ, then we halt the program and report a violation.

Program Instrumentation and Transformation We now precisely describe how we perform a transformation of a procedure f to monitor violations of φ. We shall denote the new, transformed procedure f_φ. We begin by demonstrating the case where there are no hyperexpressions in φ.

Algorithm 1 shows the instrumented procedure f_φ. The new procedure begins by calling $f(\boldsymbol{x})$ and storing the result in a local variable r. Then, it proceeds to increment a global variable n indicating the number of decisions made thus far. The loop considers every expression E in φ and updates a corresponding global variable E^s that holds the sum of all values of E witnessed so far—the notation $E(\boldsymbol{x}, r)$ is E evaluated on given values of $\boldsymbol{x}$ and r.

The function bound is treated as a black-box for now, as one can employ different well-known statistical techniques. bound provides an (ϵ, δ) estimate for the expected value of E, which we denote as $\overline{E}_{(\epsilon,\delta)}$. Specifically, $\overline{E}$ is the value E^s/n, and the two values $\epsilon, \delta \in \mathbb{R}^{>0}$ denote an additive error and a probability of failure, respectively, such that:

$$\mathbb{P}[|\overline{E} - \mathbb{E}[E]| \geqslant \epsilon] \leqslant \delta$$

In other words, our estimate $\overline{E}$ is within an ϵ from $\mathbb{E}[E]$ with a probability of $1 - \delta$.

Evaluation over Uncertain Quantities After computing a value $\overline{E}_{(\epsilon,\delta)}$ for every $E \in \varphi$, we plug those values into φ and check if it is violated and with what probability. This is implemented using the function uEval, which evaluates φ by propagating the additive errors and failure probabilities across arithmetic/Boolean operators.

Algorithm 1 Instrumenting f for monitoring φ violations

```
1:  procedure $f_\varphi(\boldsymbol{x})$
2:      $r \leftarrow f(\boldsymbol{x})$                                ▷ invoke $f$
3:      $n \leftarrow n + 1$                                ▷ update count
4:      for $E \in \varphi$ do
5:          $E^s \leftarrow E^s + E(\boldsymbol{x}, r)$                 ▷ increment sum of $E$
6:          $\overline{E}_{(\epsilon,\delta)} \leftarrow$ bound$(E^s, n)$              ▷ estimate $\mathbb{E}[E]$
7:      $\psi \leftarrow \varphi$ with every $\mathbb{E}[E]$ replaced by $\overline{E}_{(\epsilon,\delta)}$
8:      if uEval$(\psi)$ = $false_{\delta'}$ and $1 - \delta' \geqslant \kappa$ then
9:          Abort with $\varphi$-violation
10:     return $r$
```

$$\overline{E}_{(\epsilon,\delta)} \odot \overline{E}'_{(\epsilon',\delta')} = (\overline{E} \odot \overline{E}')_{(\epsilon_\odot, \delta+\delta')}$$

where $\epsilon_\odot = \max_{s \in S_\odot} |(\overline{E} \odot \overline{E}') - s|$ and
$S_\odot = \{(\overline{E} \oplus \epsilon) \odot (\overline{E}' \oplus' \epsilon') \mid \oplus, \oplus' \in \{+, -\}\}$

Figure 2: Evaluation of uncertain interval arithmetic data type, where $\odot \in \{+, -, \times, \div\}$. For $\odot = \div$, if $0 \in [\overline{E}' - \epsilon', \overline{E}' + \epsilon']$, then uEval is assumed to immediately return $\top$, denoting *unknown*.

$$\overline{E}_{(\epsilon,\delta)} > c = \left\{ \begin{array}{ll} false_\delta & \overline{E} + \epsilon \leqslant c \\ true_\delta & \overline{E} - \epsilon > c \\ \top & \text{otherwise} \end{array} \right\}$$
$$\psi_\delta \wedge \psi'_{\delta'} = (\psi \wedge \psi')_{\delta+\delta'}$$
$$\psi_\delta \vee \psi'_{\delta'} = (\psi \vee \psi')_{\delta+\delta'}$$

Figure 3: Evaluation of uncertain Boolean combinations of inequalities data type

We now discuss the implementation of uEval in detail: it takes a formula ψ, which is φ but with all expectations replaced by their uncertain $\overline{E}_{(\epsilon,\delta)}$ values. It then proceeds to evaluate ψ in two steps:

(1) simplifying arithmetic terms and appropriately propagating error (ϵ) and probability (δ) values, and
(2) simplifying Boolean connectives by checking inequalities and propagating probability values.

The result of uEval(ψ) is one of the following: $true_{\delta'}$, $false_{\delta'}$, or unknown ($\top$). If the result is $false_{\delta'}$ and $1 - \delta' \geqslant \kappa$, then we know that φ is violated with a probability of at least the set threshold κ.

Figure 2 defines how uEval simplifies arithmetic expressions. Effectively, uEval implements arithmetic over intervals of the form $[\overline{E} - \epsilon, \overline{E} + \epsilon]$. For instance, consider the rule for addition. If we are given $\overline{E}_{(\epsilon,\delta)}$ and $\overline{E}'_{(\epsilon',\delta')}$, this implies that the value of $E+E'$ is $\overline{E}+\overline{E}'$, within an $\epsilon_+ = \epsilon+\epsilon'$ additive error and with a $1-(\delta+\delta')$ probability (following the union bound). Intuitively, as we perform arithmetic on uncertain values, the intervals and failure probabilities both increase.

After simplifying all arithmetic expressions, we proceed to simplify inequalities and Boolean connectives, as shown in the rules in Figure 3. The first rule shows how to check if an inequality of the form $\overline{E}_{(\epsilon,\delta)} > c$ holds. If the full interval $[\overline{E} - \epsilon, \overline{E} + \epsilon]$ is $\leqslant c$, then we know that the inequality is *false* with a failure probability of δ. Similarly, if $[\overline{E}-\epsilon, \overline{E}+\epsilon]$ is $> c$, then we know that the inequality is *true* with a failure probability of δ. Otherwise, we cannot make a conclusive statement based on the data we have, and we therefore propagate the special Boolean value $\top$, which stands for *unknown* and has a failure probability of 0. Technically, since $\top$ can take any Boolean value, we have $\top \wedge \psi = \mathit{false}$ and $\top \vee \psi = \psi$.

After applying the simplification rules to exhaustion, we arrive at a value of the form ψ_δ, where $\psi \in \{\mathit{true}, \mathit{false}, \top\}$. If $\psi = \mathit{false}$ and $1-\delta \geqslant \kappa$, then we know that the specification φ is violated with a probability at least that of the threshold κ.

Example 4.1. Consider the following φ:

$$\frac{\mathbb{E}[E]}{\mathbb{E}[E']} > 0.1$$

Suppose we have the following estimates of the two expectations: $10_{(4,0.1)}$ for $\mathbb{E}[E]$ and $20_{(4,0.1)}$ for $\mathbb{E}[E']$. Then, applying the rules for division, we get

$$\frac{10_{(4,0.1)}}{20_{(4,0.1)}} = 0.5_{(0.375,0.2)}$$

The value 0.5 is the result of dividing the two empirical expectations: 10/20. The failure probability 0.2 is the result of taking the union bound of the two failure probabilities, 0.1 + 0.1. The error 0.375 is the largest deviation from 0.5 possible, which occurs by dividing $(10+4)/(20-4)$.

Now using the rule for evaluating inequalities, we know that the inequality above holds with probability at least $1-0.2 = 0.8$, since the lower bound $0.5 - 0.375$ is > 0.1.

Instrumentation for Hyperexpressions Thus far, we have not discussed how to monitor hyperexpressions. Algorithm 2 shows an extension of Algorithm 1 that additionally tracks the empirical mean of hyperexpressions, as shown in the second for loop. For every hyperrexpression H, the variable H^s contains the aggregate. Recall, however, that hyperexpressions are over two copies of the variables. Therefore, unlike expressions, every time we invoke $f(\boldsymbol{x})$ and receive a value r, we need to update H^s with all valuations of H^s on $(\boldsymbol{x}, r)$ together with every previously seen pair of values $(\boldsymbol{x}', r')$, which we maintain in a list called *hist*. Since H is symmetric, we only evaluate it in one direction—i.e., $H(\boldsymbol{x}', r', \boldsymbol{x}, r)$ and not $H(\boldsymbol{x}, r, \boldsymbol{x}', r')$. The empirical mean of H is:

$$\overline{H} = \frac{H^s}{\binom{n}{2}}$$

since every two decisions made are compared to each other once.

Defining bound **with Concentration Inequalities** At this point, we have shown how to maintain all the required statistics in f_φ. All that is left to do is apply a concentration of measure inequality to determine (ϵ, δ) values for the empirical means. There are two issues here: (*i*) we need a concentration inequality that handles hyperexpressions and (*ii*) the multiple comparisons problem.

To handle hyperexpressions, we can apply Hoeffding's inequality for *U-statistics* [14], which applies to cases where we are computing the mean of an k-ary symmetric function over a sequence of random variables. In our case, hyperexpressions are 2-ary symmetric functions. We therefore use the following instantiation:

$$\mathbb{P}[|\overline{H} - \mathbb{E}[H]| \geqslant \epsilon] \leqslant \delta$$

where $\delta = 2e^{-0.5n\epsilon^2}$. The same inequality applies to expressions (although we can technically get tighter bounds there).

To deal with the multiple comparisons problem, we apply the standard trick of union bounding the probability δ; that is, we take the failure probability to be $\sum_{n=1}^{\infty} 2e^{-0.5n\epsilon^2}$. This has a closed form by setting ϵ appropriately as a function of n.

Algorithm 2 Instrumenting f for monitoring φ violations including hyperexpressions

```
1:  procedure $f_\varphi(\boldsymbol{x})$
2:      $r \leftarrow f(\boldsymbol{x})$
3:      $n \leftarrow n + 1$
4:      for $E \in \varphi$ do
5:          $E^s \leftarrow E^s + E(\boldsymbol{x}, r)$
6:          $\overline{E}_{(\epsilon,\delta)} \leftarrow \mathsf{bound}(E^s, n)$
7:      for $H \in \varphi$ do
8:          $H^s \leftarrow H^s + \sum_{(\boldsymbol{x}',r') \in hist} H(\boldsymbol{x}', r', \boldsymbol{x}, r)$
9:          $\overline{H}_{(\epsilon,\delta)} \leftarrow \mathsf{bound}(H^s, n)$
10:     Append $(\boldsymbol{x}, r)$ to list $hist$
11:     $\psi \leftarrow \varphi$ with every $\mathbb{E}[Y]$ replaced by $\overline{Y}_{(\epsilon,\delta)}$
12:     if $\mathsf{uEval}(\psi) = \mathit{false}_{\delta'}$ and $1 - \delta' \geqslant \kappa$ then
13:         Abort with $\varphi$-violation
14:     return $r$
```

Practical Choice of δ Values In practice, since we are given the threshold κ that we want to achieve, we can conservatively pick values for δ across expressions such that if $\mathsf{uEval}(\psi)$ returns $\mathit{true}_{\delta'}$, then we know that $1-\delta' \geqslant \kappa$.

Example 4.2. For example, suppose we have φ of the form

$$\frac{\mathbb{E}[E_1]}{\mathbb{E}[E_2]} > c$$

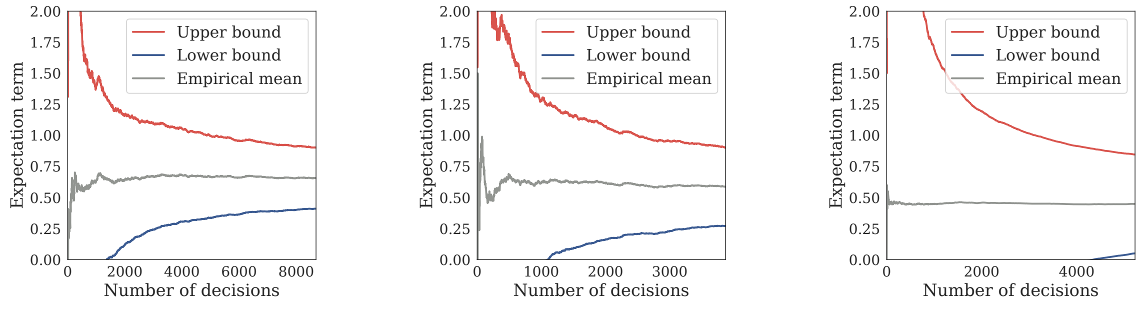
and a threshold of $\kappa = 0.9$. We can divide the failure probability of 0.1 on the two expectations. If we bound $\mathbb{E}[E_1]$ with $\delta_1 = 0.05$ and $\mathbb{E}[E_2]$ with $\delta_2 = 0.05$, then, using the structure of φ, we know that the simplification rules will result in a value of the form $\psi_{0.1}$, and therefore $1 - 0.1 \geqslant 0.9$.

5 CASE STUDIES

In this section, we describe a prototype implementation of fairness-aware programming and two case studies designed to exhibit the various features and nuances of our approach.

Implementation We have implemented a prototype of our approach as a library of *decorators* in the Python programming language. Decorators in Python can be seen as function annotations that wrap the function and can modify its behavior when running. In our case, a decorator contains a fairness specification φ, which, every time f is invoked, remembers the decisions it makes and performs some statistical reasoning to check whether φ is violated.

Our implementation optimizes the estimates for conditional probabilities. The naïve approach is to transform a conditional expectation $\mathbb{E}[A \mid B]$ into $\mathbb{E}[A \wedge B]/\mathbb{E}[B]$. However, the (ϵ, δ) bounds



(a) Group fairness **(b) Group fairness with modified classifier** **(c) Individual fairness**

Figure 4: Simulations from our case studies

on the estimates of the two expectations compound after applying the division operation—as formalized in Figure 2. To work around this loss of precision, we employ the standard approach of rejection “sampling”, where we discard invocations of f that do not satisfy B when computing $\mathbb{E}[A]$, thus resulting in a finer bound on $\mathbb{E}[A \mid B]$ This improves precision and therefore the number of observations needed to discover a violation of φ.[3]

Case Study A: Group Fairness For our case studies, we consider classifiers trained and used in the FairSquare tool [2].[4] These classifiers were trained on the Adult income dataset,[5] which has been a popular benchmark for fairness techniques. The task there is to predict someone’s income as high (true) or low (false). Unfairness can therefore be exhibited in scenarios where this is used to, for instance, assign salaries automatically.

FairSquare provides the classifiers as Python functions, which we annotate with fairness definitions. Additionally, FairSquare provides a generative probabilistic population model, which we used to simulate a stream of inputs to the classifier.

For our first case study, we consider a group fairness property of the following form

```
@spec(pr(r|sex=f ∧ q) / pr(r|sex=m ∧ q) > 0.9)
```

This property specifies that the rate at which qualified females receive a high salary is at least 0.9 that of qualified males. The Boolean qualification attribute (`q`) is used to limit the fairness constraint to a subset of the population, e.g., those that meet some minimum requirement for consideration for high salaries.

We annotated one of the FairSquare classifiers[6] with this property and simulated it using the provided probability distribution. We fixed a threshold κ of 0.85, meaning that violations should only be reported with confidence $\geqslant 0.85$. We automatically fix the probability δ for each expectation, so that only the error ϵ varies, decreasing as a function of the number of observations made (see Appendix for more details). Figure 4a shows the value of the term `pr(r| sex=f ∧ q) / pr(r|sex=m ∧ q)` as a function of the number of observations. The figure shows the empirical mean, as well as its upper and lower bounds (i.e., empirical mean +/- ϵ). As shown, the empirical mean converges to about 0.7, and after about 8000 observations, we conclude a violation of the fairness property, since the upper bound (red) goes below 0.9, indicating that the selection rate for qualified females is less than 0.9 that of qualified males, with a probability $\geqslant 0.85$.

To further illustrate, we manually modified the classifier to favor males, reducing the ratio `pr(r|sex=f ∧ q) / pr(r|sex=m ∧ q)` to about 0.6. The simulation results are shown in Figure 4b, where about 3800 observations are made before deducing that the fairness definition is violated. Notice that for the modified classifier we required a smaller number of observations to declare a violation; this is because the selection ratio (0.6) is much lower in than 0.9, and so a larger error ϵ suffices to show a violation with the same probability of $\geqslant 0.85$.

One observation we can make here is the number of decisions we needed to establish a fairness violation is 3800–8000.[7] One may argue that these numbers are too large. However, consider the case of algorithmic decisions being made at the scale of a large corporation, which may receive thousands of job applications per day, or algorithmic decisions made during crowdsourcing, where a huge number of decisions of who to pay and how much to pay can be made in an instant. In those scenarios, a few thousand decisions is a small number. At a more technical level, we could imagine a deployment where the developer can see in real-time the graphs in Figure 4. As such, the developer may decide to investigate unfairness as soon as the empirical mean has stabilized, before the lower/upper bounds are tight enough to conclusively establish a violation. Further, in our implementation, we have employed a vanilla concentration of measure inequality (Hoeffding’s), but there is room for improvement, e.g., by modifying the parameters of the inequality or investigating other inequalities. We leave the investigation of the merits of different statistical tests for future work.

Case Study B: Individual Fairness For our next case study, we consider the same classifier from case study A but with a different fairness specification involving hyperexpressions. The following definition specifies that the probability that two individuals who are similar (`sim`) receive different outcomes is less than 0.05.

[3]This optimization is only applied to conditional probabilities over expressions, but not hyperexpressions, as rejection sampling does not easily translate to that setting.
[4]Available at https://github.com/sedrews/fairsquare (commit bd27437)
[5]https://archive.ics.uci.edu/ml/datasets/Adult/
[6]`M_BN_F_SVM_V6_Q.fr`
[7]While Figure 4 shows single simulations; these are representative of the average number of observations needed to discover a violation.

```
@spec(pr(r^a ≠r^b|sim(x^a,x^b)) < 0.05)
```

x is the input to the classifier, and similarity is a function of the education level—two individuals are similar if their education levels differ by at most 2.

Figure 4c shows the estimated value of the term `pr(r^a ≠r^b|sim(x^a,x^b))`. The empirical estimate (gray line) converges to about 0.45, the probability that a pair of similar individuals are given different salaries. After about 5000 observations, we deduce that `pr(r^a ≠r^b|sim(x^a,x^b))` is greater than 0.05, as indicated by the lower bound (blue), which goes above 0.05.

Recall that estimating an expectation $\mathbb{E}[H]$ is more involved due to the fact that, after every decision, we update the sum H^s with $\sum_{(\boldsymbol{x}',r')\in hist} H(\boldsymbol{x}',r',\boldsymbol{x},r)$, which compares the current observation $(\boldsymbol{x},r)$ with all previous observations as recorded in the list *hist*. Therefore, the time it takes to update H^s increases linearly in the number of observations. In our case study, we have found that time taken to compute the sum is not prohibitive. For instance, if we keep the simulation running past the point it discovers a violation, we observe that the time it takes to compute[8] the sum is ~ 0.015s when $|hist| = 10^4$ and ~ 0.6 when $|hist| = 10^6$. Of course, the time taken is also a function of the complexity of evaluating H, which in our case is simple. Our prototype implementation in Python is not optimal, as Python is an interpreted language. We envision a number of optimizations if $|hist|$ grows very large or H is complex: For instance, compiling f_φ into low-level code, or parallelizing the evaluation of the sum using *reducers*.

6 DISCUSSION

Fairness Definitions Our foremost design goal for the specification language is to provide a flexible set of operators that can capture most existing definitions from the literature. However, there are definitions that we cannot immediately capture in the language and instrumentation in its current form.

Consider, for instance, the work of Datta et al. [6, 7], where they consider influence of a proxy. To demonstrate that a proxy is *influential* on the decision, they intervene on the distribution D by varying the values of variables under test independently of the others. This cannot be currently defined in our specification language, as we cannot consider hypothetical inputs to the procedure f, only the ones we have witnessed in the course of execution. To handle such property, we have to modify the specification language to be able to invoke the procedure f. We made the decision to restrict the language to be only a monitor of decisions and not a tester that intervenes. The same problem holds for other causal definitions of fairness [12, 16, 17].

Another prominent property that we cannot capture is individual fairness, as formalized by Dwork et al. [9]. There, one considers randomized classifiers and measures that distance between output distributions for two similar inputs. Just like with the property discussed above, to be able to measure distance between distributions for two distinct inputs, the monitoring code has to be able to repeatedly invoke a randomized f to get an accurate picture of its output distribution for each input.

[8] 1.3 GHz Intel Core i5; 8 GB RAM; Mac OS X 10.12.6

Population Distribution Like many other works in the algorithmic fairness space, we make the assumption that the underlying population distribution D of the decision-making procedure f is fixed and inputs are i.i.d. This, however, may not be always true. For instance, the decisions made may affect the population (e.g., in a setting like giving loans), a problem that has been recently explored by Liu et al. [20]. In most cases, impacts of decision-making are delayed, and therefore our work can potentially catch unfairness over a small time-scale where the population distribution is constant. Even when the underlying distribution is constantly shifting, one could imagine incorporating that fact into the runtime, e.g., by maintaining a sliding window where older, less-representative observations are discarded.

7 RELATED WORK

The algorithmic fairness literature has been rapidly expanding in breadth and depth. In this section, we focus on the most related works from the fairness literature and relevant works from software engineering and verification.

Enforcing and Checking Fairness We focus on two types of work on algorithmic fairness: (*i*) works on enforcing fairness and (*ii*) works developing techniques for checking fairness.

We have shown that our language-based approach can capture a range of properties from the literature. These include forms of group fairness, which have appeared in a various forms in the literature, e.g., [9, 11, 15]. Some notions of fairness, like equalized odds [13], work exclusively in the context of supervised learning, where we are ensuring fairness with respect to some labeled data. In this work, we consider the setting in which we are observing decisions as they are being made, on unseen data, and therefore we are not targeting notions of fairness tied to supervised learning. In Section 6, we discussed some of the notions of fairness that our language cannot capture.

In the context of enforcing fairness, much of the work in the machine-learning community has been focused on modifying the learning algorithm to take fairness into account, e.g., by incorporating fairness into the objective function [10], eliminating proxies to sensitive attributes by preprocessing the data [11, 25], or applying a post-processing step to make a classifier fair [13]. Works in security and verification have studied approaches that syntactically modify unfair programs to make them fair [1, 7].

In the context of checking fairness, a number of systems have been proposed. For example, the work on Themis [12] uses a notion of causal fairness and generates tests to check fairness of a given decision-making procedure. The work on FairTest [23] provides a comprehensive framework for investigating fairness in data-driven pipelines.

Runtime & Statistical Verification Our work is reminiscent of statistical verification techniques. Most closely related is the work of Sampson et al. [22], where probabilistic assertions are added to approximate (noisy) programs to check that they return answers within reasonable bounds. There are numerous differences between our techniques: First, we consider an online checking of assertions, as opposed to an offline testing phase by sampling from a known distribution. Second, we consider a much richer class of properties, where we have arithmetic and Boolean operations over expectations, as opposed to a single event of interest.

There is a rich body of work on various forms of runtime verification, which involves monitoring properties at runtime. Little work in that area, however, has targeted probabilistic properties. Lee et al. [18, 19] proposed applying statistical hypothesis testing at runtime to detect violations of a certain form of temporal properties in real-time systems. Like the work of Sampson et al. [22], they are restricted to a single probability expression and do not consider hyperexpressions. Our work is also distinguished by embedding the probabilistic properties to be checked into the language, as opposed to an external specification mechanism.

8 CONCLUSION

We proposed fairness-aware programming, where fairness definitions can be declaratively specified in decision-making code and checked at runtime. We argued that embedding notions of fairness directly within the programming language reduces the barrier to investigating fairness of algorithmic decision-making. We demonstrated our approach by implementing it in Python and applying it to example classifiers from the literature.

Acknowledgements We thank Philip Thomas and Yuriy Brun for insightful discussions about this work. We would also like to thank the FAT* reviewers for their feedback. This work is supported by NSF grant #1704117.

REFERENCES

[1] Aws Albarghouthi, Loris D'Antoni, and Samuel Drews. 2017. Repairing decision-making programs under uncertainty. In *International Conference on Computer Aided Verification*. Springer, 181–200.

[2] Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V Nori. 2017. FairSquare: probabilistic verification of program fairness. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 80.

[3] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi. 2016. Measuring neural net robustness with constraints. In *Advances in neural information processing systems*. 2613–2621.

[4] Toon Calders and Sicco Verwer. 2010. Three naive Bayes approaches for discrimination-free classification. *Data Mining and Knowledge Discovery* 21, 2 (2010), 277–292.

[5] Michael R Clarkson and Fred B Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210.

[6] Anupam Datta, Matt Fredrikson, Gihyuk Ko, Piotr Mardziel, and Shayak Sen. 2017. Proxy non-discrimination in data-driven systems. *arXiv preprint arXiv:1707.08120* (2017).

[7] Anupam Datta, Matthew Fredrikson, Gihyuk Ko, Piotr Mardziel, and Shayak Sen. 2017. Use privacy in data-driven systems: Theory and experiments with machine learnt programs. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1193–1210.

[8] Anupam Datta, Shayak Sen, and Yair Zick. 2016. Algorithmic Transparency via Quantitative Input Influence. In *Proceedings of 37th IEEE Symposium on Security and Privacy*.

[9] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard S. Zemel. 2012. Fairness through awareness. In *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*. 214–226. https://doi.org/10.1145/2090236.2090255

[10] Cynthia Dwork, Nicole Immorlica, Adam Tauman Kalai, and Mark DM Leiserson. 2018. Decoupled classifiers for group-fair and efficient machine learning. In *Conference on Fairness, Accountability and Transparency*. 119–133.

[11] Michael Feldman, Sorelle A. Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. 2015. Certifying and Removing Disparate Impact. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*. 259–268. https://doi.org/10.1145/2783258.2783311

[12] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. 2017. Fairness testing: testing software for discrimination. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 498–510.

[13] Moritz Hardt, Eric Price, and Nathan Srebro. 2016. Equality of Opportunity in Supervised Learning. *CoRR* abs/1610.02413 (2016). http://arxiv.org/abs/1610.02413

[14] Wassily Hoeffding. 1963. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association* 58, 301 (1963), 13–30.

[15] Faisal Kamiran and Toon Calders. 2009. Classifying without discriminating. In *Computer, Control and Communication, 2009. IC4 2009. 2nd International Conference on*. IEEE, 1–6.

[16] Niki Kilbertus, Mateo Rojas Carulla, Giambattista Parascandolo, Moritz Hardt, Dominik Janzing, and Bernhard Schölkopf. 2017. Avoiding discrimination through causal reasoning. In *Advances in Neural Information Processing Systems*. 656–666.

[17] Matt J Kusner, Joshua Loftus, Chris Russell, and Ricardo Silva. 2017. Counterfactual fairness. In *Advances in Neural Information Processing Systems*. 4066–4076.

[18] Insup Lee, Oleg Sokolsky, et al. 2005. RT-MaC: Runtime monitoring and checking of quantitative and probabilistic properties. *Departmental Papers (CIS)* (2005), 179.

[19] Insup Lee, Oleg Sokolsky, John Regehr, et al. 2007. Statistical runtime checking of probabilistic properties. In *International Workshop on Runtime Verification*. Springer, 164–175.

[20] Lydia T. Liu, Sarah Dean, Esther Rolf, Max Simchowitz, and Moritz Hardt. 2018. Delayed Impact of Fair Machine Learning. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. 3156–3164. http://proceedings.mlr.press/v80/liu18c.html

[21] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[22] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S McKinley, Dan Grossman, and Luis Ceze. 2014. Expressing and verifying probabilistic assertions. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 112–122.

[23] F. TramÃĺr, V. Atlidakis, R. Geambasu, D. Hsu, J. Hubaux, M. Humbert, A. Juels, and H. Lin. 2017. FairTest: Discovering Unwarranted Associations in Data-Driven Applications. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*. 401–416. https://doi.org/10.1109/EuroSP.2017.29

[24] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113

[25] Richard S. Zemel, Yu Wu, Kevin Swersky, Toniann Pitassi, and Cynthia Dwork. 2013. Learning Fair Representations. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*. 325–333. http://jmlr.org/proceedings/papers/v28/zemel13.html