Abstract Interpretation of Algorithmic Fairness

Becky Lytle

April 30, 2018

Abstract

Currently, many societally impactful decisions are made by machine learning algorithms. As these algorithms are given more power, we want to ensure that they are not making discriminatory or biased decisions. The past literature on algorithmic fairness has displayed weaknesses which probabilistic program analysis may be suited to address. My thesis looks into the topic of verifying fairness properties of decision-making algorithms using this type of program analysis. I will delve into the background behind algorithmic fairness and program analysis, and then I will go on to compare a variety of program analysis based methods for understanding the fairness of a decision-making algorithm. Ultimately, I will weigh in on which approach contributes most significantly to the field. I then go on to explain my independent work and how I have built off of algorithmic fairness research in order to aid an existing fairness-checking algorithm using an approach which analyzes the abstract syntax tree of a program.
## Contents

1 Introduction 3

2 Machine Learning 3

3 Algorithmic Fairness 4

4 Probabilistic Programming 5

5 Analysis of Probabilistic Programs 6
   5.1 Probabilistic symbolic execution 6
   5.2 Static analysis for probabilistic programs: inferring whole program properties from finitely many paths 7
   5.3 PSI: Exact symbolic inference for probabilistic programs 8

6 Convex Polyhedral Abstractions 8
   6.1 Abstract Interpretation 8
   6.2 Convex Polyhedra 8
   6.3 Dynamic Enforcement of Knowledge-based Security Policies using Probabilistic Abstract Interpretation 10
      6.3.1 Tracking Beliefs 10
      6.3.2 Belief Revision via Abstract Interpretation 13
      6.3.3 Abstract Semantics for $\mathbb{P}$ 13
      6.3.4 Discussion 14

7 Program Analysis of Fairness 14
   7.1 FairSquare: Probabilistic Verification of Program Fairness 14
   7.2 Algorithmic Transparency via Quantitative Input Influence: Theory and Experiments with Learning Systems 16
   7.3 Proxy Discrimination in Data-Driven Systems 17

8 Conclusion 18

9 Building on FairSquare 19
   9.1 My Work 19
      9.1.1 How FairSquare Works 19
      9.1.2 How I Modified the FairSquare System 21
      9.1.3 How This Aids FairSquare 22
   9.2 Examples 23
      9.2.1 Example 1: Successful 23
      9.2.2 Example 2: Inconclusive 23
   9.3 Testing This Approach 24
   9.4 Conclusion 25
   9.5 Reproducing These Results 25

10 Acknowledgements 26
1 Introduction

Algorithmic fairness has become increasingly important due to the power that decision-making algorithms have in our society. For example, algorithms have recently made decisions about hiring, welfare allocation, prison sentencing, and policing [1]. A big issue with this is that data often includes bias in it such as racism or sexism, and machine learning algorithms therefore classify based on this bias. If we’re streamlining the process of making decisions via machine learned classifiers, then we also want to make sure that we are improving the quality of these decisions. We therefore need a way to automatically check that an algorithm is not making biased decisions. For example, we want to verify that an algorithm is not sentencing people to longer prison sentences based on their race [1].

In the past, people have attempted to check algorithmic fairness using black-box techniques, focusing mostly on protected attributes, and using machine learning-based techniques, such as a regularization-inspired approach. The ML-based approaches affect the accuracy of the program and therefore there exists a significant trade off here between accuracy and preventing discrimination [3]. The black-box approaches often are not able to identify what is truly wrong with the ML-algorithm that is making the decision, and this leaves room for future, white-box approaches to step in and analyze decision-making algorithms themselves. Several papers in recent years have therefore chosen to solve this problem of algorithmic bias with varying types of program analysis. Program analysis has recently been used in relation to cyber-physical systems, programs and hardware, and in the field of differential privacy [1]. Through this literature, we see that program analysis is a powerful tool that allows us to reason about a variety of different topics. Program analysis has just recently become a popular way to reason about decision-making algorithms due to the deficiencies of past algorithmic fairness-based literature [5].

In this thesis, I will first give a background of machine learning, algorithmic fairness, and probabilistic programming in order to provide enough information to adequately understand the problem I will be discussing. I will then do a literature review of relevant articles, including papers on the analysis of probabilistic programs and program analysis of algorithmic fairness in particular. I aim to understand which approach to algorithmic bias is best out of the ones that I survey, and I will compare and contrast each article’s methods in order to come to a conclusion about how program analysis would best be used in order to verify algorithmic fairness.

2 Machine Learning

Machine learning allows computers to learn various things without someone specifically programming them [4]. There are two main types of machine learning: supervised and unsupervised. Unsupervised learning involves finding patterns that are hidden in unlabeled data [4]. By unlabeled, I mean to say that
this data is not categorized in any specific way, and we aim to find these hidden categorizations. In unsupervised learning, there is no explicit way to test one’s accuracy in categorizing the data [4]. Supervised learning assumes that we are given labeled data which we then use to create a model for future label prediction. For example, we may receive a data set in which each row is a person with different attributes (features) and their labels may say whether or not they are married. Therefore, we would then train a model with the goal of eventually being able to take a person’s features and predict whether they are married or not. For the purpose of this paper, I will talk mostly about supervised learning considering these are generally the types of algorithms that are currently being put in place to make societally impactful decisions.

3 Algorithmic Fairness

Algorithmic fairness has recently become popular in research contexts due to decision-making algorithms recently being given significant power in society [1]. There are a variety of examples demonstrating the bias that can make these algorithms dangerous. For example, predictions of recidivism have been associated with one’s race, and it has also been shown that one’s gender affects what job-related ads one will see online [5]. Clearly, algorithms have power in vastly different realms—predicting recidivism and online advertising—yet still have the ability to discriminate no matter what realm the algorithm may lie in.

Despite the prevalence of this problem, there are no rigorous definitions of fairness that are universally decided upon [1]. Some definitions of fairness rely on how discrimination has been described in legal texts, while others may be defined mathematically, such as group and individual fairness [5]. There has in fact been research done which addresses this specific problem. Sorelle Frielder, Carlos Scheidegger, and Suresh Venkatasubramanian published a paper which presents a specific mathematical setting in order to make the many definitions of fairness seen in various research papers compatible with one another [8]. This paper’s results imply that researchers in the future should reason about algorithmic fairness by explicitly stating their assumptions about the constructs and observations, where the constructs are unobserved, meaningful variables for the algorithm’s prediction, and the observations are the observed input for the algorithm [8].

Aside from the problem of actually defining fairness itself, there have been a wide variety of approaches to the problem of detecting or preventing bias in algorithms. Some research has looked into whether or not bias exists in the data and its predicted labels, without looking at the specific machine learning algorithm that created these labels. This is what is sometimes referred to as a black-box technique because they do not have access to the source which assigned predicted labels to the data. For example, Salvatore Ruggieri et al (2010) wanted to discover discrimination in a data set of historical decision records made either by algorithms or by humans. They base their ideas of fairness on civil rights law and formalize the process of finding discrimination in data sets by specifically
modelling contexts where discrimination may occur [13].

Other research has looked at ways to modify the machine learning model itself in order to prevent algorithmic bias in general. These methods are ML-based and do not rely on outside techniques such as program analysis, which I will delve into further later. Yahav Bechavod and Katrina Ligett (2017) approach the problem of algorithmic fairness with a regularization-inspired approach. In machine learning, regularization is a way to prevent a model from over-fitting its training data. Bechavod and Ligett focus on a fairness metric called Equal Opportunity, which looks similar to individual fairness as it has been defined in other papers [1]. Their regularization approach to algorithmic fairness appears to work well in practice, but they also say that their approach cannot offer any theoretic guarantees, which is clearly a weakness because the goal should be to guarantee that some fairness property will be achieved [3].

Another paper by Benjamin Fish, Jeremy Kun, and Adam Lelkes (2016) looks into three specific machine learning models: logistic regression, boosting, and support vector machines. They attempt to achieve algorithmic fairness by altering the decision boundaries created by these models for specific protected groups [7]. Though this approach works well for removing discrimination, they find that there is a trade-off here with accuracy and discrimination [7].

A lot of research into algorithmic fairness has specifically looked at protected features themselves and how much they affect a given machine learning model [5]. However, this is not the only way in which discrimination can occur. For example, there may be other attributes that are correlated with protected attributes, and these need to be analyzed as well [5]. This focus on protected attributes leaves room for future research to widen the scope of what fairness truly means and how it can be measured.

Overall, there has been a wide variety of approaches to this problem, including black box approaches, ML-based approaches, and approaches focused solely on protected attributes. However, there is still opportunity for future work in this field. White box approaches and approaches which focus on attributes other than protected attributes are needed. Probabilistic program analysis is something that addresses these needs directly, considering it analyzes the machine-learning program itself (a white-box method) and allows a more general formalization of what fairness means.

4 Probabilistic Programming

Probabilistic programs are characterized as operating over distributions rather than discrete values [12]. We are able to think of machine learning algorithms in this way; the input for these algorithms is data which is drawn from an unknown probability distribution, which is essentially what supervised learning algorithms attempt to model. In probabilistic program research, the behavior of a particular program is reasoned about with respect to the distribution of its input [12].

We know that inputs determine the behavior of a probabilistic program due
to research done by Dexter Kozen (1981). In his research, he proves that all programs are determined by how they behave on inputs that are fixed. Specifically, he proves that if two programs agree whenever they are given a fixed input, then they are equivalent [12]. By agree, Kozen means that two programs produce the same outcome. Programs being discrete in nature is something essential that allowed this theorem to be proven [12]. Overall, this finding allows us to reason about programs in a specific way, where inputs determine what paths will be taken in a program.

There are different types of program analysis techniques. Symbolic execution is one of these—in this approach, one determines which inputs cause different parts of a program to be run [16]. Symbolic execution is a type of abstract interpretation, which is a way to understand and approximate the semantics of programs [16]. Approaches like these are used and at times, modified, in order to use program analysis for a further goal. As mentioned previously, program analysis has been used in many fields, such as with differential privacy and cyber-physical systems [1]. Clearly it is a powerful tool that applies to many different disciplines, and I will delve further into how these techniques have been used in research settings.

5 Analysis of Probabilistic Programs

5.1 Probabilistic symbolic execution

Jaco Geldenhuys, Matthew Dwyer, and Willem Visser (2012) wanted to research further into symbolic execution due to the rise of automated decision-making programs, as this technique has been used to reason about these programs in a variety of settings, such as determining worst-case performance bounds of a program. Specifically, they approach their research with the goal of calculating the probabilities of executing different paths in a program. They extend an already existing symbolic execution system called Symbolic Pathfinder in order to implement their techniques [10].

In order to count the number of solutions to path conditions, Geldenhuys et al utilize computational algebra techniques [10]. Path conditions are conditions on input symbols (symbols are used as input in symbolic execution) [16]. These conditions are satisfiable only if a path is feasible, which means that the path is taken when the program is executed [16]. This approach allows them to get exact results for path probabilities. Calculating the probability for a specific path involves counting the number of solutions and then dividing that by the total set of values that the input can take [10]. They note that this approach only works if the inputs are uniformly distributed, and that they assume this of the programs they analyze for the sake of their research.

They optimize their techniques using PC slicing and count memoization [10]. Path condition (PC) slicing involves optimizing when two path conditions are distinct and can therefore be reasoned about in different ways. Count memoization refers to the saving of counts in memory so that they do not need to be
calculated more often than necessary.

Geldenhuys et al evaluate their techniques on a variety of different programs in order to show the unique benefits that generating exact probabilities of program paths has to offer. They note that their techniques would allow finding a bug in a program by looking at the least likely paths, considering these paths are often the hardest to test in practice [10]. Therefore, Geldenhuys et al’s research would be well suited for those who are developing unit tests for programs. Overall, we see through this paper that program analysis aids in the calculation of path probabilities and therefore is a powerful tool for reasoning about program behavior.

5.2 Static analysis for probabilistic programs: inferring whole program properties from finitely many paths

In a paper by Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani (2013), an approach for the static analysis of probabilistic programs which behave in different ways based on uncertain data is developed. Static analysis involves analyzing a program without actually executing it [14]. If data is uncertain, it means that it is drawn from a probability distribution. They give examples of such programs, which include programs used in risk analysis and medical decision-making. They are specifically interested in the values of program queries, which are correctness properties of programs that investigate the probabilities of program variable assertions [14]. Sankaranarayanan et al’s approach to this problem guarantees bounds on the values of program queries.

Their static analysis technique has two main components. First, they strategically choose an adequate set of paths to do the analysis on. This is important because in order to reason about a probabilistic program’s behavior, one must choose a finite set of its paths to analyze [14]. This is not the case for static analysis in the non-probabilistic case. Second, they compute path probability bounds; these bounds describe the probability that an execution of a program will take a certain path [14]. However, they do not achieve exact results. They do this using a combination of symbolic execution and probabilistic volume-bound computations along the chosen paths.

Sankaranarayanan et al state that their approach only works for programs with linear assignments and conditionals, so this is a weakness in terms of how powerful their approach may be when applied to a given problem. Their approach also does not work with data drawn from unknown distributions, which is another reason why this research may not apply in practice to a wide variety of problems, considering data distribution information is not always given.

They then compare their approach to relevant research in the literature. For example, they bring up the work by Geldenhuys et al that I mentioned in section 5.1. Sankaranarayanan et al state that their approach is superior to Geldenhuys et al because it infers whole program bounds, which the competing work does not do. Overall, this approach seems to be powerful and has potential to be important but so far does not seem to work for a wide enough array of problems,
considering its inability to work with unknown data distributions and non-linear assignments.

5.3 PSI: Exact symbolic inference for probabilistic programs

Timon Gehr, Sasa Misailovic, and Martin Vechev (2016) approach the analysis of probabilistic programs with the goal of developing exact inference techniques for these programs. They state that the literature surrounding probabilistic inference largely uses approximate inference techniques, which perform more efficiently but also de-prioritize accuracy. Therefore, they wanted to see how effective exact inference of probabilistic programs would be in practice.

Gehr et al contribute a symbolic analysis system for exact inference in a probabilistic program called PSI [9]. They evaluate their system and determine that it performs better than other, similar approaches. For example, they compared PSI against Mathematica and found that PSI was able to compute a precise result for programs that Mathematica was unable to compute [9]. Therefore, in terms of precision, PSI seems to perform very well and appears to be a useful tool in practice.

6 Convex Polyhedral Abstractions

6.1 Abstract Interpretation

Abstract interpretation is a specific method of program analysis in which the semantics of computer programs are approximated [2]. In order to understand more about these semantics, programs are partially executed such that all relevant calculations are not actually performed [2]. Semantics of a program are a mathematical description of all the potential run-time behaviors that it could have [2]. The term concrete semantics refers to the most specific semantics possible, such that the real execution of the program is described. Abstract semantics are then reasoned about over some chosen domain in order to further understand the program’s behavior.

There are a variety of different abstract domains over which one can do abstract interpretation. Non-relational domains are those which do not take the relationship between program variables into consideration, whereas relational domains do consider these relationships [2]. Some examples of relational domains are linear equalities and convex polyhedra [2]. I will explain the latter in depth below, as it presents significant benefits in terms expressivity and precision.

6.2 Convex Polyhedra

I will introduce the concept of a convex polyhedra as described by Hicks et al in a paper using abstract interpretation to protect a user’s private information online.
Definition 6.1. A convex polyhedron \( C = (B, V) \) consists of a set of linear inequalities \( B = \{\beta_1, ..., \beta_m\} \) over dimensions \( V \). The variable \( C \) represents the set of all convex polyhedra. A polyhedron \( C \) describes a set of states, \( \gamma_C(C) \), where \( \sigma \models \beta \) means that the state \( \sigma \) satisfies the inequality \( \beta \). For the equation below, let \( \text{fv}(\beta) \) be the set of variables that occur in \( \beta \).

\[
\gamma_C(C) = \sigma : \text{fv}(\sigma) = V, \forall \beta \in B, \sigma \models \beta
\]  

A state can be understood as the state of a program at a certain point, including the values of all variables involved in the program [11]. Also, for future reference, let \( \text{fv}((B, V)) \) represent the set of variables \( V \) of a polyhedron.

When we have a state \( \sigma \) and an ordering on the variables in \( \text{fv}(\sigma) \), \( \sigma \) can be understood as a point in \( N \)-dimensional space, where \( N \) is the size of \( \text{fv}(\sigma) \) [11]. Then, the set \( \gamma_C(C) \) can be observed as integer-valued points in an \( N \)-dimension polyhedron [11]. In this way, state and point can be used interchangeably.

In practice, the polyhedral domain is often used in abstract interpretation in order to reason about programs and prove various properties about them [15]. This domain is a fully relational numerical domain, which means that it can feasibly encode all of the possible linear constraints between variables in a program. Therefore, it is more expressive than weakly relational domains, which are not able to represent all possible linear constraints. Examples of weakly relational domains include octagons and pentagons [15].

Although there are clear benefits to using polyhedra in abstract interpretation, there are also significant drawbacks. For example, using the polyhedral domain requires in the worst-case exponential space and time complexity [15]. Therefore, scalability is an issue in terms of using this domain. Due to this problem, there has been research into methods which address this issue with varying types of approximation methods. One such paper by Gagandeep Singh, Markus Puschel, and Martin Vechev looks into the decomposition of polyhedra as a way to decrease the complexity of this abstract domain while maintaining a level of precision.

First, Singh et al note that the set of program variables split into specific subsets. This partitioning ensures that there only exist linear constraints between variables in the same subset [15]. Using this knowledge, Singh et al decompose a polyhedron into smaller polyhedra; this therefore reduces the complexity of the polyhedral domain. A problem that they face when performing this decomposition is that these partitions change dynamically and non-trivially as the program states change [15]. Therefore, we see that there is still ample opportunity for research in this field, as the problem of the polyhedral domain’s complexity has not been completely resolved.

Overall, using the polyhedral domain in abstract interpretation is an incredibly powerful tool for reasoning about programs. There have been a wide variety of applications of this method, in the realms of computer security, algorithmic fairness, and others. I will give an overview of its use in computer security in order to demonstrate its general use and to set up a related method used in a paper by Albarghouthi et al (2017) which reasons about fairness as a program property and uses abstract interpretation to prove the fairness of an algorithm.
6.3 Dynamic Enforcement of Knowledge-based Security Policies using Probabilistic Abstract Interpretation

Social media sites like Facebook and Twitter receive personal data from us, such as our gender, birth date, and location. Once these websites have our data, there is little we can do to stop outside sources from obtaining and using this data [11]. It has been shown that one can be identified based on very minimal personal data, making seemingly harmless data such as one’s birth date more powerful and dangerous than one had previously thought [11]. Due to this security-related problem, there has been research done that looks into how we can best keep our data safe and remain somewhat anonymous online once our data is being used elsewhere.

A paper by Michael Hicks, Piotr Mardziel, Stephen Magill, and Midhakar Srivatsa explores the concept of knowledge-based security policies which are used when determining whether to answer online queries about a user’s secret data, such as their birth date. As mentioned previously, anonymization is one main reason behind withholding certain data from queriers online.

I will describe Hicks et al’s problem setup before delving into their overall approach. In this paper’s model, a user $U$ has an agent which responds to a querier $Q$’s query. This query involves $U$’s (secret) personal data. For each $Q$, the agent represents $Q$’s belief of what $U$’s secret values are using a probability distribution. The agent only answers a query if it determines that releasing the required information will not increase $Q$’s ability to figure out other secret data above some pre-determined threshold [11].

Hicks et al say that in order to implement this specific model, they need to develop an algorithm to check whether a query is safe or not, where safeness is defined as a query not violating a knowledge-based policy. Additionally, they state the need for a method which revises a querier’s belief based on the answer that is given to the query. While these can both be implemented using probabilistic programming languages, these languages are either inefficient or too unsound for security purposes [11]. Hicks et al also remarks that while exact inference methods are not flexible in terms of large state spaces, approximate inference methods also have flaws; specifically, they are not precise enough for this specific problem.

Therefore, they introduce an abstract interpretation based method that is able to do approximate inference but is not too imprecise. Specifically, the probabilities of a querier’s belief are always more than the true probability distributions, so it is clear that this approach achieves a significant level of precision. The revolutionary part of their approach is the new abstract domain they use, which they call a probabilistic polyhedra. This polyhedral domain is used to represent beliefs.

6.3.1 Tracking Beliefs

Hicks et al utilize a method from Clarkson et al for revising a querier’s belief of what the values of the user’s secret variables are. First, they describe
the programming language which is used for queries. This programming language includes variables, integers, rationals, arithmetic operations, relational operations, arithmetic expressions, boolean expressions, and statements [11]. A *computation* is defined by a statement $S$, which can be understood as a relation between states. In other words, if you have an input state $\sigma$, and you run the program, you will receive an output state $\sigma'$ [11]. A *state* can be understood as a map from a variable to an integer.

Next, Hicks et al introduce a probabilistic semantics for tracking beliefs. By definition of a knowledge-based policy, a user $U$ must be able to quantitatively measure what a querier $Q$ can learn from their query. In order to achieve this, $U$ keeps track of a distribution $\delta$; this distribution represents $Q$’s belief of the possible values of $U$’s secrets and is a map from states to positive real numbers. Note that the *mass* of $\delta$, otherwise written as $||\delta||$, is the sum of the probabilities assigned to specific states: $\sum_\sigma \delta(\sigma)$ [11]. Also, the *support* of $\delta$ is the set of states which have non-zero probability [11]. The probabilistic semantics used for tracking beliefs is defined as a set of specific operations on distributions, as shown below.

\[
\begin{align*}
\llbracket \text{skip} \rrbracket \delta &= \delta \\
\llbracket x := E \rrbracket \delta &= \delta[x \rightarrow E] \\
\llbracket \text{if } B \text{ then } S_1 \text{ else } S_2 \rrbracket \delta &= \llbracket S_1 \rrbracket (\delta \land B) + \llbracket S_2 \rrbracket (\delta \land \neg B) \\
\llbracket \text{pif } Q \text{ then } S_1 \text{ else } S_2 \rrbracket \delta &= \llbracket S_1 \rrbracket (q \cdot \delta) + \llbracket S_2 \rrbracket ((1 - q) \cdot \delta) \\
\llbracket S_1; S_2 \rrbracket \delta &= \llbracket S_2 \rrbracket (\llbracket S_1 \rrbracket \delta) \\
\llbracket \text{while } B \text{ do } S \rrbracket \delta &= \text{lfp}(\lambda f : \text{Dist} \rightarrow \text{Dist}.\lambda \delta. f(\llbracket S \rrbracket (\delta \land B)) + (\delta \land \neg B)) 
\end{align*}
\]

where ...
\[ \delta[x \rightarrow E] = \lambda \sigma \sum_{\tau \mid x \rightarrow [E] \tau = \sigma} \delta(\tau) \] (8)

\[ \delta_1 + \delta_2 = \lambda \sigma \delta_1(\sigma) + \delta_2(\sigma) \] (9)

\[ \delta \land B = \lambda \sigma. \text{if } [B] \sigma \text{ then } \delta(\sigma) \text{ else } 0 \] (10)

\[ p \cdot \delta = \lambda \sigma. p \cdot \delta(\sigma) \] (11)

\[ ||\delta|| = \sum_{\sigma} \delta(\sigma) \] (12)

\[ \text{normal}(\sigma) = \frac{1}{||\delta||} \cdot \delta \] (13)

\[ \delta \land B = \text{normal}(\sigma \land B) \] (14)

\[ \delta_1 \times \delta_2 = \lambda (\sigma_1, \sigma_2) \cdot \delta_1(\sigma_1) \cdot \delta_2(\sigma_2) \] (15)

\[ \sigma = \lambda \tau. \text{if } \sigma = \tau \text{ then } 1 \text{ else } 0 \] (16)

\[ \sigma \mid x \in \text{Var}_V. \sigma(x) \] (17)

\[ \delta \mid = \lambda \sigma_V \in \text{State}_V. \sum_{\tau : \tau \mid V = \sigma_V} \delta(\tau) \] (18)

\[ f_x(\delta) = \delta \mid (fv(\delta) - x) \] (19)

\[ \text{support}(\delta) = \sigma : \delta(\sigma) > 0 \] (20)

Operation 2, skip, is simply the identity operation on distributions. Operation 6 is simple as well; it represents executing \( S_1 \) with distribution \( \delta \) and then giving that as input to \( S_2 \).

The semantics of assignment is represented by operation 3. This operation is defined below in equation 21. In plain English, when assigning expression \( E \) to \( x \), you get a distribution where state \( \sigma \)'s probability is the sum of the probabilities of all states \( \tau \) that are equivalent to \( \sigma \) when \( x \rightarrow \tau \) \[11\].

\[ \delta[x \rightarrow E] = \lambda \sigma \sum_{\tau \mid x \rightarrow [E] \tau} \delta(\tau) \] (21)

The semantics of conditionals, represented by operations 4 and 5, can be understood as the following. We take the sum of the distributions of the two branches \( S_1 \) and \( S_2 \), where the first branch’s distribution depends on \( B \) being true and vise versa for the second branch’s distribution \[11\]. This sum becomes the resulting distribution. This explanation applies to both a regular conditional (operation 4) and a probabilistic conditional (operation 5), the only difference being that probabilities \( q \) and \( 1 - q \) must be reasoned about differently than a Boolean value \( B \).

Operation 7 represents the semantics of a while loop. One while-loop iteration can simply be thought of as if \( B \) then \( S \) else \( \text{skip} \). Therefore, the semantics of an entire while loop is the composition of the fixed point of the resulting distributions from each of the iterations of a while-loop \[11\]. In order to implement this, the loop body can be run until the mass of \( \delta \land B \) is zero \[11\].
Now that we’ve precisely defined these semantics, we can begin to understand how we may revise a belief about possible valuation of a user’s secrets. Hicks et al describe steps to do this, based on Clarkson et al’s approach to belief revision. To set up this approach, I will first define some relevant notation [11]. Let the values of a user U’s secret variables $H$ be given by the hidden state $\sigma_H$. The querier Q’s initial belief of the values of $H$ is a distribution $\delta_H$. A query can be represented as a program $S$; this program takes both $H$ and $L$ (known, non-secret variables) as input. The program outputs final values of $L$.

First, Hicks et al says that $S$ is evaluated probabilistically using $\delta_H$ in order to produce $\delta'$, an output distribution. Next, using the actual secret values $\sigma_H$, $S$ is evaluated to produce an output state $\hat{\sigma}_L$. Lastly, the querier’s belief is revised based on $\hat{\sigma}_L$.

### 6.3.2 Belief Revision via Abstract Interpretation

How does a system know whether or not to reveal the answer to query? If we have a query $S$, the user’s system will only provide $\sigma_L$, the output of $S$, if the system determines that the querier is not likely to figure out the secret state $\sigma_H$ beyond some identified threshold $t$ [11].

Hicks et al introduce a new, revolutionary approach which enforces the threshold described above. Though it would be possible to evaluate queries using a probabilistic programming language, Hicks et al claim that existing approaches based on these languages cannot ensure security guarantees; on the other hand, they also claim that exact inference is too inefficient. Hicks’ approach therefore is based on abstract interpretation, where the time complexity depends on the query itself instead of the input space. They introduce two abstract domains, one being a probabilistic polyhedron. A probabilistic polyhedron is defined as a convex polyhedron with data about the probabilities of its points [11]. In their implementation, they actually extend this to powersets of polyhedra in order to retain precision, despite adding to the execution time.

Recall the definition of convex polyhedra as described previously in section 6.2. Hicks et al extends this standard definition of convex polyhedra to probabilistic polyhedra in order to use polyhedra to represent sets of distributions over program states; they define probabilistic polyhedra as follows.

**Definition 6.2.** A probabilistic polyhedron $P$ is a tuple $(C, s^{\text{min}}, s^{\text{max}}, p^{\text{min}}, p^{\text{max}}, m^{\text{min}}, m^{\text{max}})$, with $\mathbb{P}$ as the set of probabilistic polyhedra [11]. First, we define $s^{\text{min}}$ and $s^{\text{max}}$ as the number of support points in polyhedron $C$. Then, we define $p^{\text{min}}, p^{\text{max}}$ as lower and upper bounds on the probability mass per support point. Lastly, $m^{\text{min}}$ and $m^{\text{max}}$ are bounds on the total probability mass. Therefore, $P$ represents the set of distributions which satisfy these constraints.

### 6.3.3 Abstract Semantics for $\mathbb{P}$

Hicks et al provides abstract implementations of the operations described in the concrete semantics of section 6.6.2, which are needed in order to execute a program in this abstract probabilistic polyhedral domain. The only difference
with these new semantics is that they operate over abstract operations, rather than operations over distributions [11]. Hicks et al claims that all of the abstract operations over-approximate the concrete ones, which is necessary information to prove the overall precision of their method. The abstract operations include assignment, sum, product, scalar product, normalization, and other relevant operations. Hicks et al explain that when using the polyhedral domain, one can implement the threshold test and can therefore check if a query violates knowledge-based security policies.

6.3.4 Discussion

The probabilistic polyhedral domain is a state of the art approach which Hicks et al introduce to the field of abstract interpretation. Clearly it has a variety of benefits, as it utilizes approximate inference methods in order to keep the complexity low while also maintaining a level of precision. Despite these benefits, Hicks et al explain that there are many ways in which this specific approach can be improved. One such way is that their approach does not handle nominal valued variables as well as it handles integer valued variables. Therefore, their abstract interpretation method may not work perfectly on all types of programs. This is definitely something to look into in the future when trying to figure out how best to use the probabilistic polyhedral domain.

Overall, this paper demonstrates the use of convex polyhedra and in this case, a probabilistic polyhedral domain. This knowledge of how this domain functions will be necessary when looking to understand the paper by Albarghouthi et al which utilizes a similar method in order to prove fairness properties of a program.

7 Program Analysis of Fairness

Several researchers have gone on to use probabilistic program analysis in a variety of ways in order to reason about algorithmic fairness. I will delve into a few of these different approaches and will evaluate their usage.

7.1 FairSquare: Probabilistic Verification of Program Fairness

Albarghouthi et al (2017) approach algorithmic fairness in a highly quantitative way—they aim to prove fairness by imagining fairness as a program property. They choose to focus on two notable definitions of fairness: individual fairness and group fairness. Individual fairness says that similar inputs must have similar outputs, while group fairness states that a specific subset of inputs must have a similar output when compared to the output of the entire set of inputs [1].

They first consider probabilistic symbolic execution methods as described by Sankaranarayanan et al (2013) and Gehr et al (2016)—I reviewed these above in
sections 5.2 and 5.3. These articles use approaches that provide exact guarantees of a program using approximate techniques [1]. Albarghouthi et al wanted to focus on exact probabilistic verification techniques considering their aim was to construct proofs of fairness for a given program. Therefore, Albarghouthi et al design a new approach considering the existing approaches do not work well with the fairness properties that they consider [1]. This approach relies on weighted volume computation, which has roots in abstract interpretation over the polyhedral domain.

The first step in Albarghouthi et al’s methodology is to encode a machine learning algorithm logically as formulas in real arithmetic [1]. This involves defining our program according to a specific set of semantics. These semantics define a program $P$ as a sequence of statements $S$, which is described below.

$$S := V \leftarrow E \mid V \sim \text{Dist} \mid \text{if } B \text{ then } S \text{ else } S \mid S; S \quad (22)$$

$S$ can be an assignment statement, a probabilistic assignment, a conditional statement, or a sequence of statements. An assignment statement is represented by $V \leftarrow E$, where $E$ is an arithmetic expression over variables in $V$. A probabilistic assignment is represented by $V \sim \text{Dist}$, and is made by sampling a probability distribution from Dist.

Not only do they encode the machine learning algorithm, but they also encode a population model using these semantics, where this model is a probabilistic program that describes a model of the population (from which the input is drawn).

After defining a program and a population model using these semantics, they define the fairness properties of these programs which they wish to verify. Afterward, they compute the weighted volume of this logical encoding. This is equivalent to calculating the probability of picking an assignment that satisfies their chosen fairness properties [1]. This weighted volume computation problem is difficult to solve efficiently; Albarghouthi et al therefore propose a new symbolic-volume-computation algorithm and implement it in a new tool called FairSquare. This algorithm converges to the exact value of the volume and divides the weighted volume into rectangular regions in order to solve the problem efficiently [1]. This volume can be thought of as a polyhedra, and the process of turning this into rectangular regions can be thought of as decomposing the polyhedra in order to improve the efficiency of this process.

They evaluate their approach on various data sets and machine learning classifiers—they are able to detect unfairness as defined by violating either the definition of group fairness or individual fairness. They also compare FairSquare to similar tools in this field, such as those designed by Sankaranarayanan et al (2013) and Gehr et al (2016), and their tool is able to verify fairness properties that other tools cannot.

Overall, Albarghouthi et al’s approach is fairly revolutionary in this field considering they not only produce a way to automatically check the fairness of a decision-making algorithm, but they introduce a new algorithm for weighted volume computation problems as well. This new algorithm allows their tool to
run efficiently, and their verification of fairness properties does not require any background information about which features are protected and which are not, which is a strength of their approach, considering other approaches may require this information.

7.2 Algorithmic Transparency via Quantitative Input Influence: Theory and Experiments with Learning Systems

Anupam Datta, Shyak Sen, and Yair Zick (2016) approach machine learning algorithms with a belief that they need to be made more transparent. These algorithms are often understood as a black-box solution in which data goes in and a decision is made, yet no one truly knows what features of the data were most influential toward the final decision [6]. Datta et al introduce a variety of Quantitative Input Influence (QII) measures in order to show the degree to which inputs affect the outputs of these algorithms. They believe that boosting the interpretability of these models can help with the problem of detecting algorithmic bias. Therefore, their approach to algorithmic bias is somewhat of a diagnostic one, as their tools may give us an insight into the inner-workings of a machine learning algorithm, which may then allow someone to (subjectively) decide if an algorithm is behaving in an unfair way [6].

Datta et al aim to be able to produce transparency queries of different types. For example, a personalized transparency query may show us the influence of various features on an algorithm’s decision on one person. On the other hand, an aggregate transparency query may give us insight into how an algorithm makes a decision about an entire group of people. To formalize the idea of these queries, Datta et al states that a transparency query measures how an input influences a quantity of interest. A quantity of interest is a property of the behavior of the algorithm, depending on the given input distribution [6]. An example of this may be the conditional probability of an outcome for a specific group of people.

They create causal QII measures in order to model the difference in the quantity of interest when the algorithm operates over two different, yet related input distributions. For example, we would look at two correlated features in a data set, such as weight-lifting ability and Gender, and use their QII measures to determine which of these significantly affected the outcome of an algorithm (if not both) [6].

These QII measures help us reason about which inputs are influential, and Datta et al argue that this is helpful as a diagnostic tool for discrimination detection. However, they offer an example in which someone is analyzing data that includes SAT scores—they state that SAT scores can often be a proxy for other, protected attributes such as race. They say that in this case, a human expert would have to (subjectively) decide if it was discrimination for SAT scores to be highly influential in a model. I think that although their method helps people understand machine learning models further, I do not know if it is
powerful enough to help reason about discrimination on its own.

7.3 Proxy Discrimination in Data-Driven Systems

This paper by Anupam Datta and his colleagues seems to build upon the QII paper, as they shift their focus to proxy discrimination instead of focusing solely on the influence of protected features on a machine learning system’s outcome. In a data set, there may be correlated features in which one feature is a protected attribute and the other is not; if the latter feature is influential in a system’s outcome, then this would be labeled as proxy discrimination [5].

Datta et al offer examples of proxy discrimination in action. They describe a situation where a person’s zip code is correlated to their race. In this case, a bank could use zip code in their machine learning model in order to evaluate loan eligibility; though they are not directly accessing a protected attribute, they are engaging in proxy discrimination [5].

In this paper, they present a formal definition of proxy use in machine-learning programs. In order to do this, they first decompose a program. Decomposition involves rewriting a program $p$ as two new programs $p_1$ and $p_2$ that can be combined using substitution to represent the original program [5]. Using this decomposition, they are able to prove whether or not a data-driven system engages in proxy use. Similar to the paper by Albarghouthi et al (2017), they reason about a program in a language-independent way, where a program is made up of functions that evaluate arithmetic expressions. These expressions are made up of real numbers, variables, basic arithmetic, and if-then statements [5].

After detecting proxy use in a system, they introduce a method for removing this proxy use in order to ensure that the outcome is not discriminatory. They repair these programs over the expression-based language described previously, so this method is language-independent as well [5]. This repair involves locating expressions in a model that facilitate proxy use and simplifying them.

They evaluate their methods by testing them on various data sets. For one data set, when building a decision tree, spouse’s gender is located as a facilitator of proxy discrimination, and therefore, they remove it from the inputs given to the model.

Datta et al’s approach to algorithmic fairness is significant in this field, as they mention that many previous papers relating to discriminatory algorithmic practices usually only focus on protected features rather than on proxies. They also rigorously define proxy discrimination and offer many examples where other, similar papers’ methods would not be sufficient to identify the proxy use in action. Also, their ability to automatically repair algorithms that exhibit proxy use is significant as well, especially considering the paper by Albarghouthi et al presents the idea of algorithm repair as a problem they aim to research in the future.

They state that a problem with their approach is that they require knowledge of what the protected attributes are in the input data set; sometimes, this information may not be available, which would render their method ineffective.
Additionally, they comment on the fact that their detection and repair method enumerates through all possible subprograms within a program, and in the worst case, this would be exponentially large.

Overall, the evaluation of this method demonstrates that it can be very useful in situations that it would have been difficult to detect proxy use. Therefore, in situations where data about which features are protected is not lacking, this method is useful. Therefore, on the whole, Datta et al contribute an important approach to this field.

8 Conclusion

Some past literature in algorithmic fairness has been deficient for the following reasons. Some approaches are unable to balance algorithm accuracy with fairness, while others only focus on protected attributes and ignore all other features. Additionally, much of the past research take a black-box approach, which does not allow thorough analysis of the algorithm being used. Therefore, past literature has paved the way for probabilistic program analysis-based approaches to fairness, considering program analysis directly addresses some of these problems that I have mentioned.

We have seen that probabilistic program analysis is a powerful tool outside of algorithmic fairness, as it has been used to reason about different paths in a program in a variety of different settings and has been shown to have practical applications. After reviewing this literature, I presented three recent approaches to algorithmic fairness using program analysis. First, I reviewed Albarghouthi et al’s approach, which is novel in its ability to efficiently and automatically determine if a program adheres to specific, rigorously-defined fairness metrics; it also uses an approach based in the polyhedral domain of abstract interpretation, which has been shown to have benefits in terms of expressivity. Then, we saw Datta et al’s Quantitative Input Influence approach, which is a black-box method, yet it utilizes program analysis techniques. This approach is somewhat lacking, as it focuses solely on protected features as a way to reason about fairness. Lastly, I reviewed Datta et al’s more recent paper about proxy discrimination. This paper contributes an important approach to the literature, considering it not only focuses on protected features, but considers proxy features as well, which is something that has been largely ignored in past research. However, this approach requires knowledge of the domain, which may not always be available.

Overall, I believe that Albarghouthi et al’s approach has the most promise in terms of practicality and contribution to the field. Not only is their approach revolutionary, but they implement it in a software which automatically checks programs for fairness properties. They also evaluate their tool, FairSquare, by comparing against other papers’ similar approaches, further demonstrating its use. Because of this paper’s overall significance, I based my independent work on its approach.
9 Building on FairSquare

My initial aim was to look into the problem of verifying program correctness for machine learning programs to check fairness. As I’ve previously stated, algorithms are continuing to make more socially impactful decisions every year, and it is imperative that these algorithms do not replicate the bias and discrimination that are prevalent in many human-made decisions. I have devoted time to understanding the state of the art research in both probabilistic program analysis in general and algorithmic fairness of programs. One paper in particular that I have endeavored to understand in depth is “FairSquare: Probabilistic Verification of Program Fairness” by Albarghouthi et al. As I mentioned before, this approach to program fairness verification represents a significant contribution to the field of algorithmic fairness, and I intended to build upon their codebase in order to potentially aid FairSquare’s functionality.

I hypothesized that I would be able to improve upon FairSquare if I were to analyze a population model and its associated decision-making program in order to understand more information about each node in the abstract syntax tree (AST) of these programs. Specifically, I’d want to know whether each node was influenced by a protected variable (e.g., race) or not.

9.1 My Work

9.1.1 How FairSquare Works

The input to the FairSquare system is a program and its associated population model. First, this program and its model are parsed into an abstract syntax tree using Python’s AST library and a specifically defined Encoder class (code provided below, in Figure 9.3). Each part of the program – conditional statements, assignments, etc. – are treated as nodes. For example, given the code below in Figure 9.1, the abstract syntax tree in Figure 9.2 would be generated.

```python
def program():
    SensitiveAttribute(y)
    x = y + 5
    if (x > 10):
        z = a
    return z
```

Figure 9.1 Example Program
Once the program and its population model are parsed, the AST is then analyzed in the way described in Section 7.1 in order to determine if the program is fair or not. Therefore, this fairness checking happens solely post-parsing, once the python programs have been logically encoded as formulas in real arithmetic.

Below I have included some code snippets from FairSquare’s parsing process. I have written descriptive comments, hopefully to aid understanding of what the code is doing.

Albarghouthi et al’s Encoder class, defined below, builds on the Encoder class defined by Python’s AST library. It visits each node in the abstract syntax tree of a program by calling node-specific visitor functions, such as visit_Assign for assignment nodes in a program. I’ve included a snippet of this function below to give an idea of what is happening during the parsing process for this type of node.

```python
class Encoder(ast.NodeVisitor):
    ...
    def __init__(self):
        self.vdist = {}
        self.sensitiveAttribute = None
        self.qualified = None
        self.fairnessTarget = None
        self.model = None  # formula in real arithmetic
        self.program = None  # formula in real arithmetic
    ...
    def visit_Assign(self, node, d):
        assert(len(node.targets) == 1)
        lhs = node.targets[0].id  # left hand side of assignment
        rhs = node.value  # right hand side of assignment

        if isCall(rhs):  # if the rhs is a call expression
            return self.probAssign(lhs, rhs, d)
```

Figure 9.2 Abstract Syntax Tree Example

Once the program and its population model are parsed, the AST is then analyzed in the way described in Section 7.1 in order to determine if the program is fair or not. Therefore, this fairness checking happens solely post-parsing, once the python programs have been logically encoded as formulas in real arithmetic.

Below I have included some code snippets from FairSquare’s parsing process. I have written descriptive comments, hopefully to aid understanding of what the code is doing.

Albarghouthi et al’s Encoder class, defined below, builds on the Encoder class defined by Python’s AST library. It visits each node in the abstract syntax tree of a program by calling node-specific visitor functions, such as visit_Assign for assignment nodes in a program. I’ve included a snippet of this function below to give an idea of what is happening during the parsing process for this type of node.

```python
class Encoder(ast.NodeVisitor):
    ...
    def __init__(self):
        self.vdist = {}
        self.sensitiveAttribute = None
        self.qualified = None
        self.fairnessTarget = None
        self.model = None  # formula in real arithmetic
        self.program = None  # formula in real arithmetic
    ...
    def visit_Assign(self, node, d):
        assert(len(node.targets) == 1)
        lhs = node.targets[0].id  # left hand side of assignment
        rhs = node.value  # right hand side of assignment

        if isCall(rhs):  # if the rhs is a call expression
            return self.probAssign(lhs, rhs, d)
```
9.1.2 How I Modified The FairSquare System

At a high level, I wanted to find the set of variables in a program that were affected by the initial protected attribute(s). By affected, I mean something such as “experience = race - 5” where race is the protected attribute, and experience is affected by the value of the variable race at this point in the program. Specifically, I wanted to find variables affected via either conditional statements or assignments. These are the two types of nodes in the programs that Albarghouthi et al tested their system on which were relevant in terms of my goals. I therefore edited the parsing section of the FairSquare system—specifically, the Encoder class—in order to achieve this.

Below, I include a snippet of what I added to the function visit_Assign in order to achieve my goals.

```python
if sum([1 for x in self.affector if x in self.protected]) > 0:
    self.protected += [lhs]

for node in ast.walk(rhs):
    if isName(node):
        if node.id in self.protected and lhs not in self.protected:
            self.protected += [lhs]
```

The attribute self.affector is initialized as None when creating an Encoder object. However, self.affector is changed in the visit_If function to include all variables in the guard of a conditional statement, so that when visiting the body of a conditional statement, self.affector includes variables that would affect this body of code being reached. For example, given the code below, self.affector would be set equal to a list including the variables x and y when visiting the assignment node z = 10.
if x > 5 and y < 10:
    z = 10

**Figure 9.5** Example Conditional Statement

After we’ve analyzed an entire conditional statement node (including the body), self.affector is reset to what it was before entering this if statement (for example, an empty list).

The attribute self.protected is initialized as a list with the original sensitive attributes, as specified by the input program. We add a variable v to this if we find that v is affected by something in self.protected.

In the above code which I added to visit_Assign, the left-hand side of an assignment \((lhs)\) is added to self.protected in two cases. In the first case, we add lhs to self.protected if a protected variable is in self.affector. In the second case, we add lhs to self.protected if anything in the right-hand side of an assignment is a protected variable.

Below, I have annotated the program from Figure 9.1 to include information about the attributes I have introduced to the Encoder class.

```python
def program():
    SensitiveAttribute(y)  #self.protected = [y]
    x = y + 5             #self.protected = [x, y]
    if (x > 10):
        #self.affector = [x]
        z = a               #self.protected = [x, y, z]
        #self.affector = [ ]

    return z
```

**Figure 9.6** Annotated Example Program

### 9.1.3 How This Aids FairSquare

By strategically checking conditional statement and assignment nodes, I am able to gather a comprehensive list of all variables in a program that are affected in some way by our initial protected attribute(s). Each program that FairSquare evaluates has a certain *fairness target*, which may look something like \( t \leq 0 \) – therefore, if \( t \) is not in the list of sensitive attributes, then we know that our fairness target is not affected at all by a protected attribute, and must therefore be met. In this case, my approach would deem this program to be *fair*. In the case that our fairness target is in the list of sensitive attributes once the parsing is complete, my approach is unable to reliably determine if the program is fair or not, and therefore, the label returned is simply “inconclusive.”

My approach adds a potential benefit to FairSquare’s existing system; if a program is deemed fair post-parsing, we know that we do not need to run the FairSquare fairness analysis on it. Current techniques for analyzing fairness, including the FairSquare approach, are very computationally expensive. Therefore, the fact that my approach is able to correctly classify a certain subset of programs is significant, as we save both time and computational power.
9.2 Examples

9.2.1 Example 1: Successful

My approach would be able to rule that the sample program below is fair, as we know that the hiring decision, presumably included in the fairness target, will never possibly be influenced by $x$ (the protected attribute) in any way. In this case, we know that we would not even need to run FairSquare in order to determine if the hiring decision will be fair or not. I have annotated the program below to include information about the `self.protected` attribute.

```plaintext
SensitiveAttribute(x)  #self.protected = [x]
if (x = 0) {
    y = 1;  #self.protected = [x, y]
} else {
    y = 0;
}
if (z > 5) {
    hired = 1;
} else {
    hired = 0;
}
```

Figure 9.7 Example Successful Result

9.2.2 Example 2: Inconclusive

In another example below, we see that our technique would mark the hiring decision as being affected by our protected variable. In this scenario however, we cannot actually determine the fairness of the program just by looking at it and would definitely need to run FairSquare to determine the fairness result of this program.

```plaintext
SensitiveAttribute(x)  #self.protected = [x]
if (x = 0) {
    y = 0;  #self.protected = [x, y]
} else {
    y = 0;
}
if (y == 0) {
    hired = 1;  #self.protected = [x, y, hired]
} else {
    hired = 0;
}
```

Figure 9.8 Example Inconclusive Result
9.3 Testing This Approach

I ran FairSquare’s trivial benchmarks from the paper that terminate quickly and recorded both my approach’s result and FairSquare’s result.

<table>
<thead>
<tr>
<th>id</th>
<th>My Result</th>
<th>FairSquare Result</th>
<th>Sensitive Vars</th>
<th>My Approach</th>
<th>FairSquare</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Inconclusive</td>
<td>Unfair</td>
<td>5</td>
<td>0.20s</td>
<td>6.76s</td>
</tr>
<tr>
<td>2</td>
<td>Inconclusive</td>
<td>Unfair</td>
<td>4</td>
<td>0.03s</td>
<td>3.80s</td>
</tr>
<tr>
<td>3</td>
<td>Inconclusive</td>
<td>Fair</td>
<td>8</td>
<td>0.05s</td>
<td>9.68s</td>
</tr>
<tr>
<td>4</td>
<td>Inconclusive</td>
<td>Unfair</td>
<td>6</td>
<td>0.03s</td>
<td>5.23s</td>
</tr>
<tr>
<td>5</td>
<td>Inconclusive</td>
<td>Unfair</td>
<td>8</td>
<td>0.03s</td>
<td>10.35s</td>
</tr>
<tr>
<td>6</td>
<td>Inconclusive</td>
<td>Fair</td>
<td>3</td>
<td>0.06s</td>
<td>9.83s</td>
</tr>
<tr>
<td>7</td>
<td>Fair</td>
<td>Fair</td>
<td>1</td>
<td>0.10s</td>
<td>9.25s</td>
</tr>
<tr>
<td>8</td>
<td>Fair</td>
<td>Fair</td>
<td>1</td>
<td>0.02s</td>
<td>3.12s</td>
</tr>
<tr>
<td>9</td>
<td>Fair</td>
<td>Fair</td>
<td>1</td>
<td>0.05s</td>
<td>6.22s</td>
</tr>
<tr>
<td>10</td>
<td>Fair</td>
<td>Fair</td>
<td>1</td>
<td>0.06s</td>
<td>5.94s</td>
</tr>
<tr>
<td>11</td>
<td>Fair</td>
<td>Fair</td>
<td>1</td>
<td>0.04s</td>
<td>11.15s</td>
</tr>
<tr>
<td>12</td>
<td>Inconclusive</td>
<td>Fair</td>
<td>2</td>
<td>0.03s</td>
<td>5.11s</td>
</tr>
<tr>
<td>13</td>
<td>Inconclusive</td>
<td>Fair</td>
<td>2</td>
<td>0.02s</td>
<td>4.38s</td>
</tr>
<tr>
<td>14</td>
<td>Inconclusive</td>
<td>Fair</td>
<td>2</td>
<td>0.03s</td>
<td>6.39s</td>
</tr>
<tr>
<td>15</td>
<td>Inconclusive</td>
<td>Fair</td>
<td>2</td>
<td>0.03s</td>
<td>10.35s</td>
</tr>
</tbody>
</table>

While my approach has many inconclusive results, every time it marks a program as Fair, FairSquare’s result agrees. This is what was predicted for my approach—it only gets true positives. It is clear that my approach is beneficial in terms of how quick it is; while these trivial programs take several seconds to prove as fair, my approach labels them as fair in under a second.

I then ran FairSquare’s nontrivial (not vacuously true) benchmarks (not included in the paper) that terminate quickly; results are seen below.

<table>
<thead>
<tr>
<th>id</th>
<th>My Result</th>
<th>FairSquare Result</th>
<th>Sensitive Vars</th>
<th>My Approach</th>
<th>FairSquare</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Inconclusive</td>
<td>Unfair</td>
<td>5</td>
<td>0.20s</td>
<td>10.95s</td>
</tr>
<tr>
<td>2</td>
<td>Inconclusive</td>
<td>Unfair</td>
<td>4</td>
<td>0.03s</td>
<td>6.55s</td>
</tr>
<tr>
<td>3</td>
<td>Inconclusive</td>
<td>Unfair</td>
<td>6</td>
<td>0.03s</td>
<td>7.91s</td>
</tr>
<tr>
<td>4</td>
<td>Fair</td>
<td>Fair</td>
<td>1</td>
<td>0.02s</td>
<td>4.39s</td>
</tr>
<tr>
<td>5</td>
<td>Fair</td>
<td>Fair</td>
<td>1</td>
<td>0.05s</td>
<td>8.11s</td>
</tr>
<tr>
<td>6</td>
<td>Fair</td>
<td>Fair</td>
<td>1</td>
<td>0.04s</td>
<td>9.91s</td>
</tr>
<tr>
<td>7</td>
<td>Inconclusive</td>
<td>Fair</td>
<td>2</td>
<td>0.03s</td>
<td>6.52s</td>
</tr>
<tr>
<td>8</td>
<td>Inconclusive</td>
<td>Fair</td>
<td>2</td>
<td>0.02s</td>
<td>4.82s</td>
</tr>
<tr>
<td>9</td>
<td>Inconclusive</td>
<td>Fair</td>
<td>2</td>
<td>0.02s</td>
<td>6.12s</td>
</tr>
<tr>
<td>10</td>
<td>Inconclusive</td>
<td>Fair</td>
<td>2</td>
<td>0.03s</td>
<td>17.85s</td>
</tr>
</tbody>
</table>

Similarly, for these benchmarks, my approach has several inconclusive results but again, only has true positives. Also, similarly to before, my approach
demonstrates a clear time-related benefit. Specifically, on average, my approach takes 0.6% of the time that FairSquare takes to classify programs as fair or unfair.

There are other benchmarks that the FairSquare team provides which take a very long time to run; though I have not had time to run these, I imagine my approach will provide a significant benefit in this situation specifically.

9.4 Conclusion

As I have stated previously, FairSquare’s system represents a revolutionary and influential piece of technology in the field of algorithmic fairness and probabilistic program analysis. Aside from the fact that it introduces a new weighted volume computation algorithm, it is also able to check fairness properties that other prevalent tools in the field are not able to detect. However, despite these significant strides, the FairSquare system is still fairly heavyweight. Many of their public benchmarks take an extremely long time to run; they recommend using a timeout for these. Therefore, there is a clear need for an approach which achieves at least some of what FairSquare does, perhaps through a less heavyweight process.

As shown in Section 9.3, My approach is able to reliably and accurately label a specific subset of programs as fair in 0.6% of the time that the FairSquare system takes to label a program. Though I only ran the benchmarks which terminate quickly for my results section, one could imagine that this time decrease would be especially helpful when running benchmarks that do not terminate quickly. In these cases, one would be able to label a specific subset of programs as fair without ever having to use FairSquare’s entire system.

The combination of my approach with FairSquare’s is an even stronger fairness-checking system than FairSquare alone. Therefore, my work represents a nontrivial addition to the field of algorithmic fairness through program analysis.

9.5 Reproducing These Results

Running my code is exactly the same as running FairSquare’s benchmarks, which is described on their github repo at https://github.com/sedrews/fairsquare. I will include their instructions below.

run.sh automates running batches of benchmarks. Usage:

```
./run.sh <benchmark-list> <results-directory> [ <timeout> ]
```

<benchmark-list> is a file where each line is a relative path, from this directory, to an .fr to be run

<results-directory> is the name of a non-existing directory where the output files will be stored
<timeout> is an optional argument that specifies a number of seconds after which to kill each individual benchmark (60 to 1800 recommended)

There are three existing benchmark lists in .txt files:

fast.txt contains the benchmarks from the paper that terminate quickly (within approximately one minute); all use vacuously true qualification

fastQ.txt contains the non-trivial qualification benchmarks (not included in the paper) that terminate quickly

all.txt contains all of the files from the qual/ and noqual/ directories (this will take a LONG time to run, in general; be sure to use a timeout)

10 Acknowledgements

I would like to express my gratitude toward the several people who were integral to making the completion of this thesis possible.

To Kristopher Micinski, my advisor, for always being supportive and helpful, offering your own insight at all turns, and for continuing to learn with me through this process.

To all my past coworkers at Bluecore, and to Beneah Kombe specifically, for building me up and giving me the confidence necessary to believe in myself and my intelligence in regards to computer science.

To my parents and my friends, for always eagerly nodding along during my conversations with them about this paper; thank you for always making me feel heard, despite not completely knowing what a polyhedral abstraction is!

To my fellow CS major and close friend, Maddy Hodges; you have taught me an unimaginable amount about computer science (and about life in general)! Thank you for constantly reminding me about the joy of learning–I would not be here without you.

And finally, to the Haverford Computer Science Department, for fostering an environment in which I felt eager to grow and learn as a scholar.
References


https://www.cc.gatech.edu/~harrold/6340/cs6340_fall2009/Slides/SymExClass-09.pdf