A Theory AB Toolbox

Marco Gaboardi[1] **and Justin Hsu**[2]

1 **University of Dundee and Harvard University**
m.gaboardi@dundee.ac.uk
2 **University of Pennsylvania**
justhsu@cis.upenn.edu

1 Introduction

One of the most unexpected discoveries in the study of algorithms is the *power of randomness*. For many tasks, random algorithms perform better than deterministic algorithms, both in theory and in practice. This is often the case even when the tasks seems purely determined, with no random input or source of noise. Therefore, it's not surprising that random algorithms have been widely adopted in practice; examples span a wide range of computer science, from machine learning to database technology to graphics.

A natural question is: Are these algorithms correct? Today, proofs of these requirements are accomplished by enlisting highly skilled humans to produce "*paper proofs*" by hand. However, even the most skilled humans are not perfect, and even if an algorithm is proved correct, who is to say that it is *implemented* correctly? We propose to use tried-and-true techniques from program verification—like type systems and more general static analyses—to check, and perhaps automatically derive, *formal* versions of the requisite *paper proofs*.

At first glance, the paper proofs we are trying to imitate are extremely diverse, conjuring up insights that seem totally mystifying. Faced with this problem, a standard approach is to use general-purpose theorem provers. These are extremely expressive, but there is a real cost: The proofs work at a low level of abstraction that bears little resemblance to the informal proof. As a result, the formal proofs are verbose and require substantial manual effort to construct. An expert who knows the formal proof perfectly clearly often labors to convince a theorem prover of this fact.

To avoid these drawbacks, we envision building a toolbox of verification techniques following an overarching principle: *take inspiration from human reasoning*. While some parts of a given paper proof may be genuinely novel, experts often rely on simple yet powerful abstractions for intuitive reasoning about particular properties; accordingly, paper proofs remain simplest, most concise, and most lightweight. As much as possible, we want to take advantage of these patterns used when humans reason about randomized algorithms.

Compared to carrying out proofs with general purpose theorem provers, we expect that narrowing the gap between informal and formal reasoning will lead to simpler, more intuitive proofs; humans familiar with the paper proof of properties should find it possible to use our verification framework with minor training. By restricting the type of reasoning, we also expect a higher degree of automation in our tools, reducing the verification burden even further.

As we hope to build everything around proof patterns, we must accept that there are facts about randomized algorithms that we will not be able to verify. Some theorems may use "ad hoc" tools—or at least, rare tools that are not worth the effort in formalizing. The focus of our toolbox—at least at first—is to provide a tool for composing mathematical facts rather than verifying everything from the ground up. Based on our survey of proofs we might hope to verify, we believe a small collection of primitive axioms and operations along with powerful composition patterns will already suffice to verify many interesting proofs.

Conference title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–9
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 What is correctness for randomized algorithms?

Randomized algorithms serve many functions, some mundane—movie recommendations, elevator scheduling; some important—driving directions, credit approval ratings; and some life-critical—robotic surgery, air traffic control, safety systems in cars.

These applications involve randomization in a variety of ways. For instance, an algorithm may only have access to "noisy" inputs, and the goal is to control the effect of the noise. Or the algorithm may introduce randomness to a deterministic problem; maybe the added noise helps smooth out worst-case inputs, leading to faster execution or smaller space usage. Or maybe the randomness is inherently part of the problem of interest.

With so many settings, correctness can mean many different things. For instance, we can consider a program correct if it returns an *optimal*, or alternatively we can consider it correct if it returns a output *quickly* that may not be optimal, but is close enough. There are probabilistic versions of typical metrics for deterministic algorithms, like expected accuracy or expected convergence rate, but there are also performance metrics specific to randomized algorithms, like generalization error in machine learning algorithms.

A common thread in all of these concepts is reasoning about *probabilities*, whether explicitly or implicitly. For example, an interesting guarantee might be "With at least 99% probability, the algorithm computes an answer within 0.01 of the true answer". A different accuracy and convergence rate guarantee could be "The expected value of the complexity of the randomized algorithm to reach an accuracy bound α is quadratic in $1/\alpha$".

Though there have been quite successful formalizations of probability theory [2], they typically model probabilities from a single point of view—say, as explicit distributions. In contrast, typical paper proofs treat probabilities in a multi-faceted way. The proof may work with random variables, when the reasoning looks a bit like symbolically manipulating algebraic formulas; say, two samples Z_1 and Z_2 from a normal distribution may be added together as $Z_1 + Z_2$. In other cases, proofs may work with probabilities of certain events directly; say, proving that $\Pr[X > 10] = 0.1$. Proofs may even involve looking at probabilities geometrically.

In our view, the main challenge of handling these proofs is organizing and structuring proofs so that the numerous theorems can be brought to bear in a usable and lightweight way.

3 The current state-of-the-art

We are certainly not the first to argue the fact that programming language research should develop more techniques for randomized algorithms.

In a closely related area, there is a large amount of work in analyzing and testing approximate algorithms, for instance a recent work is the one of Sampson et al. [16]. In most of these works the goal is to provide guarantees on the performance of randomized programs by carrying out first some dynamic analysis inferring a synthetic representation of the random process, and then some static analysis to simplify and test the results against some specification. While certainly of interest, this approach do not provide actual proofs.

There is also a large amount of works that have approached the verification of approximate algorithms, either by using full-blown formalization in a theorem prover [2], or ad-hoc verification techniques based on program analysis [7]. There has been recent success in proving correctness (for various definitions of correctness) of randomized algorithms in specific domains like cryptography [3, 5], differential privacy [13, 6], etc.

However, looking at these and other approaches in the literature, it seems that there are few tools that produce proofs remotely similar to paper proofs! Indeed, while attending the Approx 2014 [1] workshop co-located with PLDI 2014—devoted to the topic of verifying randomized computation—we were struck by the lack of attention to *proofs*.

We believe that part of the problem here is that all these approaches have been studied by programming languages researchers with programming languages tools for the programming languages community. Our goal is instead to provide a toolbox that is interesting and useful for people from the algorithm and machine learning community but based on techniques from the programming language community. In this sense, we want a toolbox combining interesting aspects of both theory A and theory B.

4 Our proposal: Take abstractions from paper proofs

As we have discussed, we envision tools built around common structures humans use when proving correctness properties. These patterns take a variety of shapes and forms, but fundamentally, they are *abstractions*: they are used to give structure to the key parts of an argument, in a flexible and concise form, while concealing the boilerplate, "boring" parts of the proofs. From this point of view, what makes these patterns interesting is that while there are the typical abstractions we know and love—modularity, inductive arguments, etc.—there are also patterns that are harder to recognize. For instance, they may involve reasoning about code in a non-modular way, or grouping random sampling in particular, non-local ways. We believe that understanding and capturing patterns from paper proofs—empirically, the best abstractions—is a key step to designing usable and powerful verification tools.

As a first step in designing our toolbox we will focus on accuracy. Here, we briefly discuss three such proof patterns and how they might be profitably incorporated into a verification framework. For concreteness we will focus mostly on language-based verification, though we expect these principles to be valuable in a wide variety of verification approaches.

4.1 Case study: The union bound

The union bound states that if event A happens except with probability p_A, and event B happens except with probability p_B, then both A and B happen except with probability at most $p_A + p_B$. In the context of randomized algorithms, A and B are often taken to be the event that some random noise is small. The probabilities p_A and p_B are often called *failure probabilities*—they are the probabilities that event A or B fails to happen.

Even though randomized algorithms naturally mix probabilistic and deterministic operations, the accuracy proof can be simplified by separating out the analysis of the random operations. A common first step when proving accuracy is to apply the union bound to argue that, except with probability p, all the random noise added is small. Assuming this fact, the rest of the analysis can then treat each sampling operation as returning small noise, and prove accuracy *without* explicitly reasoning about randomness.

In general, the reasoning naturally divides into two parts: (1) keep track of the total probability that the noise is too large, using the union bound to aggregate the failure probability of various sampling operations, and (2) carry out the accuracy analysis by viewing the original program as a *deterministic* program, where each sampling operation is assumed to return some small value.

For example, consider an algorithm that adds noise to each element of a list:[1]

Listing 1 Applying a union bound

```
noiselist (in : list R) :[b] { out : list R | dist(in, out) < T } =
  match in with
  | Nil -> return Nil
  | Cons x xs ->
    sample noisy <- addnoise x in
    sample rest <- noiselist xs in
    return (Cons noisy rest)
```

The noise function `addnoise` comes with an accuracy specification encoding the probability of adding noise more than T. The annotation in the first line state that the output list `out` has each element at most T away from the corresponding element in input list `in`. The annotation `[b]` indicates the *failure probability*: the accuracy assertion holds with probability at least $1 - b$.

The union bound is carried out by the use of `sample`. In more detail, suppose we have shown that program e satisfies accuracy assertion A with probability failure probability p_A, and program e' satisfies accuracy assertion B with failure probability p_B, *assuming that A holds*. Then, our language concludes that the program `sample x <- e in e'` (which samples x from e and runs e') satisfies assertion B, but now with the failure probability given by the union bound: $p_A + p_B$.

Looking more closely, we can view the union bound as a kind of composition principle: to analyze the failure probability of a big algorithm, it's enough to analyze the failure probability of each of its component programs. This is a powerful principle that simplifies the reasoning on paper, and we should take advantage of this principle in verification as well.

4.2 Case study: Independence bounds

While a wide range of randomized algorithms can be proved accurate with just the union bound, some algorithms require a finer analysis based on *independence of random variables*. For instance, some accuracy results rely on *concentration around the mean*: the principle that, if we take a series of independent random draws from a distribution μ, the average of the draws is with high probability not too far from the mean of μ. For example, the accuracy proof of a random counter [8, 10] from the privacy literature uses a concentration bound for sums of independent draws from the Laplace distribution.

Arguments in terms of independence are problematic for program verification. They are often global, involving reasoning about collections of random draws that may be far apart in the actual code of the program. A priori, it may not be clear that the random draws are even independent. To provide some structure, we can imagine a two-stage approach.

First, we can separate random sampling from the main program and annotate it with where and how to apply the independence bounds. This makes it easier to verify independence for collections of random draws.

For example, suppose we want to take three independent samples from some distribution μ. Furthermore, say that there is a concentration bound stating that the sum of up to three independent draws from μ satisfies some property ϕ. In a first stage analysis we might write the following:

[1] We give pseudocode in a functional language to illustrate our ideas, but we don't restrict to any particular kind of language.

Listing 2 Applying independence bounds

```
sample n1 <- mu in
sample n2 <- mu in
sample n3 <- mu in

sum1 = boundsum n1 :[b] phi1 in
sum2 = boundsum (n1 , n2) :[b] phi2 in
sum3 = boundsum (n1 , n2 , n3) :[b] phi3 in
(sum1 , sum2 , sum3)
```

Each `boundsum` applies the concentration bound to a list of random samples. The returned values `[sum1, sum2, sum3]` represent the sums n_1, $n_1 + n_2$, and $n_1 + n_2 + n_3$ respectively, each tagged with a probabilistic assertion ϕ_1, ϕ_2, or ϕ_3 and failure probability b given by the concentration bound. For instance, the assertions might look like

$$\phi_1(x) := |x - c| < T_1$$
$$\phi_2(x) := |x - 2c| < T_2$$
$$\phi_3(x) := |x - 3c| < T_3,$$

where c is the mean of distribution μ.

The guarantees from concentration bounds—ϕ_i holds with probability at least $1 - \beta$—are probabilistic assertions, which can then be combined with the union bound. Indeed, the two principles are often used together in paper proofs, first applying concentration bounds relying on independence to arrive at some preliminary facts, then applying union bounds to combine the facts.

In the second stage, the samples, along with facts derived from applying the independence bounds, are fed into a program written in our union bound language. There, the samples can be used and the facts assumed as needed, tracking the failure probability. In the simple example above, after applying the independence bounds, we may want to reason about adding up the sums. The corresponding second stage program might look as follows:

Listing 3 Second stage program

```
sumup (s1 :[b] { x : R | phi1 } )
      (s2 :[b] { x : R | phi2 } )
      (s3 :[b] { x : R | phi3 } )
      :[3 * b] { out : R | dist(out , 6 * c) <  T1 + T2 + T3 } =
sample x <- s1 in
sample y <- s2 in
sample z <- s3 in
return (x + y + z)
```

Note that the inputs to this program carry the assertions derived in the first-stage program above. Also note that the `sample` operation combines the failure probability (b for each assertion) to give the final failure probability ($3b$).

4.3 Case study: Martingale reasoning

Our third and final pattern involves *martingales*. Formally, a martingale is a sequence of random variables $X_1, X_2, \dots$. We allow X_i to depend on all previous random variables $X_1, \dots, X_{i-1}$. The martingale property requires that this sequence is somehow "memoryless": the information at time step i is captured by X_{i-1}, rather than the whole sequence

$X_1, \dots, X_{i=1}$. More formally, for any concrete values $x_1, \dots, x_{i-1}$, we require

$$\mathbb{E}[X_i \mid X_1 = x_1, \dots X_{i-1} = x_i] = x_i$$

A canonical example is a *random walk*. A person starts at a point on a line (call it 0), and at each time step, flips a fair coin. If heads, she goes to the left one unit; if tails, she goes to the right unit. If we let X_i be her position at time i, then $X_1, \dots, X_i$ is a martingale sequence.

A particularly powerful fact about martingales is the following theorem. We will present it informally, since the details are somewhat technical to spell out.

▶ **Theorem 1** (Optional Stopping Theorem, Informal). *Let $X_0, X_1, X_2, \dots$ be a martingale sequence, and let $\tau \in \mathbb{N}$ be a* stopping time*—a random variable that depends only on the martingale values before τ. For instance, we should be able to decide if $\tau = 3$ given just X_0, X_1, X_2, X_3. Then,*

$$\mathbb{E}[X_\tau] = \mathbb{E}[X_0].$$

Note that on the left, the time τ may be random as well, since it depends on the random sequence.

We think the optional stopping theorem can be a powerful tool for verifying probabilistic programs with loops. For instance, consider the following loop:

Listing 4 A loop

```
X = Y = 0;
while (|X| < 10) do
  Y = X;
  sample f <- flip(-1, 1);
  X = X + f;
end
```

The optional sampling theorem provides a powerful and flexible way to reason about this loop, which involves probabilistic sampling. Specifically, we can think of the iteration when this loop terminates as a stopping time—this iteration depends only on the previous iterations. If we know that X evolves as a martingale, then we can conclude that

$$\mathbb{E}[X_{fin}] = \mathbb{E}[X] = 0.$$

Since we know hat X_{fin} is 10 or -10, this immediately lets us conclude that $\Pr[X_{fin} = 10] = \Pr[X_{fin} = -10]$, something that is awkward to prove via other means.

We have outlined a simple example, just to give an idea of what verification via martingales may look like. There is a wide variety of proofs that can be carried out with similar martingale arguments, just by choosing the correct martingale and applying the optional stopping theorem. Picking the proper martingale to deduce a particular fact is something of a black art, and not something we would expect could be handled automatically. In many respects, finding the right martingale is similar to specifying a loop invariant—some human insight seems to be required.

That being said, martingales are a powerful technique for proving facts about probabilistic programs. For instance, it's know that many independence bounds (as described in Section 4.2) are actually consequences of optional stopping. Rather than building independence bounds into our toolbox (i.e., assuming them), martingales may allow us to actually *derive* these principles directly.

5 Looking ahead

The three case studies above are meant to give just a taste of the kind of toolbox we have in mind. Of course, there are many possible extensions; we briefly sketch a few directions below.

5.1 More tools in the toolbox

The most natural extension is simply adding more advanced tools.

Refining the union bound

The union bound in Section 4.1 is used extremely frequently, but it is a somewhat loose bound. There are several known refinements that give sharper bounds on the failure probability under certain assumptions. One promising candidate is the Lovász Local Lemma [12], which gives a better bound when each event is independent of most of the other events.

Advanced concentration bounds

While the concentration bounds in Section 4.2 require independence, there are more general bounds that relax this requirement. For instance, the Azuma-Hoeffding bound [9] gives concentration for martingale sequences $X_0, X_1, \ldots$. This is a weaker requirement than full independence, but can be trickier to prove.

5.2 Combining the tools

So far, we have proposed tools that focus on individual reasoning principles in a somewhat independent fashion. However, there is no reason that these techniques must be used in isolation. Obviously, there are some rules governing how, say, independence bounds can be mixed with martingale reasoning, but in paper proofs, different principles can be flexibly mixed and matched. In our view, understanding how to formalize and flexibly combine different tools is an important direction for handling more complex proofs.

In Section 4.2, we have a proposed a very small step in this direction: mixing union bounds and independence bounds. However, our proposal leaves much to be desired. For instance, it requires that independence bounds must be applied first, then union bounds. For many algorithms, this may require coding the algorithm in an unnatural way to conform to this proof structure. An ideal solution would avoid this unnecessary burden.

5.3 Beyond accuracy

While accuracy is certainly an important property of randomized algorithms, there are a host of other properties that could be handled by formal verification.

Incentive and game-theoretic properties

In game theory settings, the inputs to an (often randomized) algorithm are controlled by rational agents that may seek to manipulate the output of the algorithm but changing their input. *Incentive properties* guarantee that agents can't gain much by this kind of manipulation.

From a program verification point of view, incentive properties are *relational program properties*. They reason about the relation between two runs of the same program: one where agents report honestly, and one where agents manipulate their inputs. As we observed in recent work [4], basic incentive properties can be fruitfully handled with relational program verification techniques.

However, the current state of verification for incentive properties is stuck on the same obstacle: More complex incentive properties depend on accuracy guarantees, which the tools we have are ill-equipped to handle. There are several examples for algorithms which compute equilibria—in a nutshell, the goal is to coordinate players to actions where no player has incentive to deviate (a kind of *stability* condition), but players may have incentive to manipulate which equilibrium is chosen. For instance, Rogers and Roth [15] propose an algorithm for computing Nash equilibrium that depends on the accuracy of randomized counters; Kearns, et al. [14] propose an algorithm for computing so-called coarse correlated equilibrium, where the incentive properties depend on the accuracy of a sophisticated learning algorithm.

Randomized computational complexity

The computational complexity of randomized algorithms is a deep area of study. At the most basic level, we might want to verify the complexity of a program that has probabilistic running time. The typical notion here is *expected running time*, the average time the program takes. The problem is most interesting (and most difficult) when the program runs a loop with a probabilistic guard. In such cases, the program may not even always terminate!

The wide available literature on the subject and the restricted availability of domain specific tools to build complexity proofs makes this area particularly attractive. A higher level question is whether there is a reasonably small class of patterns that we can target to verify a large number of examples.

The probabilistic method

Finally, our toolbox can be viewed as capturing reasoning about probabilistic quantities, not just quantities related to algorithms. For general mathematical proofs, a powerful tool is the *probabilistic method*, invented by Erdős [11]. This principle shows the (non-constructive) existence of an object satisfying a specific property P by first constructing a *random* instance of the property, then showing the probability P holds is strictly positive. The tools of probability theory are powerful and broadly applicable to general mathematics; to the extent that our toolbox can capture probabilistic reasoning, we should be able to model general mathematical proofs.

6 Conclusion

As our field of programming languages (and more generally, formal verification) matures, it is crucial to export our tools and techniques to other fields, so that all kinds of researchers can hear what formal verification has to say, as it were. By interacting with other fields, our field stands to be greatly enriched by exchanging ideas and considering new, interesting properties to verify. Our proposal to verify properties of randomized algorithms is a step in this program, but just a step. We hope there are many more steps to come.

References

1 First SIGPLAN Workshop on Probabilistic and Approximate Computing. Co-located with PLDI 2014, Edinburgh, Scotland., 2014.

2 Philippe Audebaud and Christine Paulin-Mohring. Proofs of randomized algorithms in coq. *Science of Computer Programming*, 74(8):568–589, 2009.

3 Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. pages 193–206, 2014.

4 Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. Higher-order approximate relational refinement types for mechanism design and differential privacy. 2015.

5 Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. pages 71–90, 2011.

6 Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. pages 97–110, 2012.

7 Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. pages 169–180, 2012.

8 T.-H. Hubert Chan, Elaine Shi, and Dawn Song. Private and continual release of statistics. 14(3):26, 2011.

9 Devdatt P Dubhashi and Alessandro Panconesi. *Concentration of measure for the analysis of randomized algorithms*. 2009.

10 Cynthia Dwork, Moni Naor, Toniann Pitassi, and Guy N. Rothblum. Differential privacy under continual observation. pages 715–724, 2010.

11 Paul Erdős. Graph theory and probability. *Canadian Journal of Mathematics*, 11:34–38, 1959.

12 Paul Erdos and László Lovász. Problems and results on 3-chromatic hypergraphs and some related questions. *Infinite and finite sets*, 10:609–627, 1975.

13 Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C Pierce. Linear dependent types for differential privacy. pages 357–370, 2013.

14 Michael Kearns, Mallesh Pai, Aaron Roth, and Jonathan Ullman. Mechanism design in large games: Incentives and privacy. pages 403–410, 2014.

15 Ryan M Rogers and Aaron Roth. Asymptotically truthful equilibrium selection in large congestion games. pages 771–782, 2014.

16 Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S McKinley, Dan Grossman, and Luis Ceze. Expressing and verifying probabilistic assertions. pages 112–122, 2014.