Verifying Algorithmic Versions of the Lovász Local Lemma

Rongen Lin, Hongjin Liang(✉), and Xinyu Feng

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China
relin@smail.nju.edu.cn {hongjin,xyfeng}@nju.edu.cn

Abstract. Algorithmic versions of the Lovász Local Lemma (ALLLs), or rather, the Moser-Tardos algorithm and its variants, are impactful in both theory and practice. In this paper, we take the first step towards the goal of formally verifying ALLLs by applying programming language techniques. We propose two proof recipes, called loop truncation and resampling-table-based coupling, for bridging the gap between Hoare-style program logics and ALLLs' original informal proofs. We formally verify six existing important results related to ALLLs, and propose a new result which generalizes several existing results. Our proof recipes can also be used to verify general properties of other probabilistic programs in addition to ALLLs.

1 Introduction

The Lovász Local Lemma [19, 57] (LLL) is a powerful tool in combinatorics. It guarantees the existence of a combinatorial object with certain properties in a probability space. It has also been helpful for proving the existence of solutions to numerous significant problems in computer science, such as the Boolean Satisfiability Problem and the Graph Coloring Problem, since these problems can be viewed as instances of the problem of finding some combinatorial objects.

Besides proving the solution's existence, we also want to *efficiently construct* a solution. To this end, people have devised algorithmic versions of the Lovász Local Lemma (ALLLs). The most notable one is the Moser-Tardos (MT) algorithm proposed by Moser and Tardos in their Gödel Prize-winning paper [50]. The algorithm searches the probability space for the desired combinatorial object iteratively, bringing us a constructive proof for LLL. It is efficient in that the expected total number of iterations is bounded. Since then, a huge number of works have emerged, some explore the power of the MT algorithm [53, 42, 31, 43, 1, 37], some find variants of the MT algorithm [31, 16, 34, 30, 25, 36, 29, 13], and some utilize the MT algorithm to solve problems in various areas of computer science [31, 43, 33, 9, 26, 55, 15, 14, 27, 23], including applications in real-world systems [2, 39].

Therefore it is of great importance to formally verify the (total) correctness of ALLLs, in particular, that the MT algorithm and its variants almost surely terminate (i.e. terminate with probability 1) and their expected iteration times

have certain upper bounds. Previous works (e.g. [50]) have given proofs for the correctness of ALLLs, though these proofs are rather informal. Therefore, a natural choice is to formally verify ALLLs by formalizing existing informal proofs.

However, we encounter a challenge when verifying ALLLs by following existing proofs. We propose *Proof Recipe 1* to circumvent this challenge, and propose *Proof Recipe 2* for completing the verification after applying *Proof Recipe 1*.

Challenge: Handling infinite execution traces. It is challenging to formulate some subgoals in ALLLs' existing informal proofs using distribution-based semantics, which is commonly used in the literature of probabilistic program verification. The reason is that, on the one hand, these subgoals are about complex properties of the algorithm's execution traces, and we have to take *infinite* traces into account until we prove their absence. On the other hand, distribution-based semantics can only describe certain simple properties of these infinite traces, e.g. their overall probability.

Proof Recipe 1. We propose a proof recipe called *loop truncation* to circumvent the above challenge. For a loop in an ALLL, we transform it to a set of arbitrarily truncated loops. Now we have a set of "truncated algorithms", which can only generate *finite* execution traces. Then, instead of directly verifying the original algorithm, we prove a common bound of the expected iteration times for all the truncated algorithms. The latter can be proved following existing proofs, and now we do not have to handle infinite traces when formulating the subgoals.

Proof Recipe 2. A crucial step commonly found in many proofs of ALLLs, is to prove *an inequality between probabilities involving two programs*. Specifically, for the original ALLL program C_1 and a property $\mathbf{p}$, one constructs a program C_2 and a property $\mathbf{q}$, and shows that the probability of $\mathbf{p}$ holding after C_1's execution is not greater than the probability of $\mathbf{q}$ holding after C_2's execution.

To prove this inequality, existing informal proofs introduce variants of C_1 and C_2, say C_1' and C_2', that use a new random source called *resampling table*. By assuming that C_1 and C_2 are respectively equivalent to C_1' and C_2', they reduce the original inequality to a similar inequality that involves C_1' and C_2', and prove the latter. We elaborate on these proofs in Sec. 2.1.

Following the above proof idea, we propose a proof recipe called *resampling-table-based coupling* to formally prove the aforementioned inequality. At the core of this proof recipe is a new measure-theoretic semantics for probabilistic programs, which we call a *resampling-table-based semantics*. This semantics formalizes the *resampling table* in existing proofs as a built-in structure. We formulate C_1' (C_2') by giving C_1 (C_2) this new semantics without changing its syntax, and express the equivalence between C_1 and C_1' (C_2 and C_2') as the equivalence between a classic probabilistic semantics and the new semantics. We prove the semantics equivalence once and for all, instead of repeatedly proving the equivalence between every pair of programs. Then it remains to prove the inequality involving C_1' and C_2', which is now an inequality on the new semantics.

Our proof recipe, resampling-table-based coupling, further reduces the problem to verifying the two programs C_1' and C_2' individually. The idea is to introduce an intermediate assertion specifying the resampling table as the common random source to bridge the two programs' unary verification. The unary verification can be done using a simple Hoare-style program logic.

Contributions. Using the above two proof recipes, we have successfully verified several ALLL-related results. In summary, we make the following contributions:

- We verify six important results from [50, 53, 42, 31] for the first time. They include all the three "probabilistic" results from Moser and Tardos's Gödel Prize-winning paper [50].
- We propose a proof recipe called *loop truncation*, which circumvents the challenge when verifying ALLLs with classic distribution-based semantics.
- We propose a proof recipe called *resampling-table-based coupling*. It expresses the informal proof idea of an important inequality in a formal and concise way, taking a perspective of semantics equivalence and Hoare-style reasoning.
- We propose a new result related to the Moser-Tardos algorithm, with results from [50, 53, 42] as its corollaries. The statement and the proof of this result are formal, and the proof is done by applying our proof recipes.

Our proof recipes can also be used to prove general properties (i.e. total correctness and inequalities between probabilities) of probabilistic programs *beyond* ALLLs (see Ex. 1 and Ex. 2). We also discuss the relationship between our proof recipes and existing formal proof methods for *positive almost sure termination* and *asynchronous coupling* in Sec. 7.

Outline. We review the original informal proof of the MT algorithm, and introduce the challenge and our main ideas in Sec. 2. We then give the mathematical preliminaries in Sec. 3, and define the programming language, including our new semantics, in Sec. 4. Then we introduce our two proof recipes in Sec. 5. By applying these recipes, we verify six existing important ALLL-related results and a new result in Sec. 6. We finally discuss related work in Sec. 7.

The technical report [46] contains the full formal details of this work, including all the definitions and all the proofs for lemmas, theorems and examples.

2 Informal Development

To formally verify the ALLL-related results, a natural choice is to follow their original informal proofs. Below we first provide a brief overview of the original informal proof of Moser and Tardos's seminal result [50], which serves as an example for understanding the ideas behind the original proofs of many ALLL-related results. We then explain the verification challenge and our proof recipes.

```
Independently sample X_1, ..., X_N
while ∃j ∈ [1, M]. η_j holds do
  Choose such an η_j
  for all X_i that η_j depends on do
    Resample X_i
Output the current values of X_1, ..., X_N
```

Fig. 1. The MT algorithm

```
succ := 1
for all η_j ∈ g_WT(wt) do
  for all X_i that η_j depends on do
    Resample X_i
  if η_j does not hold then succ := 0
Output succ
```

Fig. 2. The check(wt) algorithm

2.1 Moser and Tardos's Proof

The Moser-Tardos (MT) algorithm efficiently constructs a solution for the following problem. Given N program variables $X_1, \ldots, X_N$ and M events $\eta_1, \ldots, \eta_M$, where each variable is associated with some random distribution and each event depends on some of $X_1, \ldots, X_N$, we would like to construct an assignment of $X_1, \ldots, X_N$ such that none of the M events occurs. The Lovász Local Lemma [19, 57] provides the Erdős-Lovász condition which sufficiently ensures the existence of such assignments. The MT algorithm finds such an assignment as shown in Fig. 1. Here "(re-)sample X_i" means the following: sample from the random distribution with which X_i is associated, and assign the result to X_i.

Moser and Tardos prove that, under the Erdős-Lovász condition, the expectation of the total iteration number of the algorithm's outer loop is no more than a real number r_{EL}, and thus the algorithm almost surely terminates. (Here we do not expose the definitions of the Erdős-Lovász condition and r_{EL}, which can be found in Thm. 4.) In the remainder of this subsection, we sketch their proof.

Restatement of the proof goal. Moser and Tardos restate their proof goal using *execution logs*. For every execution of the algorithm, its execution log Λ is a sequence of events η_j, which are dynamically chosen at the beginning of the outer loop iterations. We write $\Lambda\langle i\rangle$ for the i-th element of Λ, which is the event chosen at the i-th iteration. We write $|\Lambda|$ for the length of Λ, so it specifies the total number of the outer loop iterations. If the loop does not terminate in an execution, then $|\Lambda| = \infty$. Now, Moser and Tardos restate their proof goal as

$$\mathbb{E}[|\Lambda|] \leq r_{\mathsf{EL}}. \tag{1}$$

That is, the expected length of the execution log has an upper bound r_{EL}, where the randomness of Λ comes from the randomness of the MT algorithm. From (1), Moser and Tardos conclude that the program almost surely terminates. The proof of (1) can be divided into three stages, which will be discussed in turn.

Stage 1. In this stage, Moser and Tardos rewrite $\mathbb{E}[|\Lambda|]$ by defining a special mathematical structure called *witness trees*. A witness tree wt is a tree with some special properties, where each node is labeled with an event from $\eta_1, \ldots, \eta_M$. One can construct a witness tree wt from an execution log Λ following some specific procedure, and we write $wt = f_{\mathsf{WT}}(\Lambda)$ for this. From the concrete definitions and properties of wt and f_{WT} (which we omit here), Moser and Tardos rewrite

$\mathbb{E}[|\Lambda|]$ as the infinite series in (2). It enumerates all witness trees wt, and sums the probabilities that wt can be constructed from some prefix of Λ (that is, there exists a sequence Λ' such that: Λ' is a prefix of Λ, and $wt = f_{\mathsf{WT}}(\Lambda')$ holds).

$$\mathbb{E}[|\Lambda|] = \sum_{wt} \Pr[wt = f_{\mathsf{WT}}(\text{some prefix of } \Lambda)] \tag{2}$$

Stage 2. Next, Moser and Tardos give an upper bound of the probability in (2). That is, for all witness trees wt, they prove that

$$\Pr[wt = f_{\mathsf{WT}}(\text{some prefix of } \Lambda)] \leq p(wt), \tag{3}$$

where $p(wt)$ is a specific real number related to wt, whose definition we omit. Instead of directly proving (3) (which is challenging), Moser and Tardos construct a program check(wt), which outputs either 0 or 1, and then prove the following:

(a) The check(wt) algorithm outputs 1 with probability $p(wt)$.
(b) $\Pr[wt = f_{\mathsf{WT}}(\text{some prefix of } \Lambda)] \leq \Pr[\text{check}(wt) \text{ outputs } 1]$.

(3) then follows from the above two properties. The proof of (a) is not difficult. What is really interesting is the proof of (b). To see this, we present the check(wt) algorithm in Fig. 2, where $g_{\mathsf{WT}}(wt)$ gives us an event sequence collecting the labels of wt's nodes in a certain order (in fact, a reversed BFS ordering of wt).

To prove (b), Moser and Tardos observe that whenever wt can be generated by the MT algorithm and check(wt) is run on the *same* random source, check(wt) outputs 1. They capture this observation by specifying the random sources using *resampling tables* (RT) and letting the algorithms explicitly use the tables.

Specifically, Moser and Tardos give an RT-MT algorithm[1], and assume that it is "equivalent" to the MT algorithm, i.e., the two algorithms produce the same distribution of execution logs. The idea of the RT-MT algorithm is to transfer the lazy samplings in the MT algorithm to eager ones: the RT-MT algorithm performs all the samplings ahead of time and stores the results in an table (the RT) so that it can interpret all subsequent samplings as *deterministic* table queries.

The RT-MT algorithm is shown in Fig. 3, where we highlight the difference with Fig. 1 in blue. At the beginning, the RT-MT algorithm randomly generates a resampling table RT, which has N rows and an infinite number of columns. For all $i \in [1, N]$, this step independently samples X_i an infinite number of times, and fills the i-th row of RT with these samples. Subsequently, every sampling step of the MT algorithm is replaced by a table-query step in the RT-MT algorithm. For instance, resampling X_i is replaced by reading the leftmost unread element from the i-th row of RT, and assigning the result to X_i.

Similarly, Moser and Tardos give the RT-check(wt) algorithm as shown in Fig. 4, and assume that it is "equivalent" to check(wt), i.e., the two algorithms have the same output distribution.

[1] In [50], Moser and Tardos did *not* explicitly introduce new algorithms (RT-MT and RT-check). The algorithm here is a possible interpretation of their prose description.

Randomly generate an RT
Assign the first col. of RT to $X_1, \ldots, X_N$
while $\exists j \in [1, M]$. η_j holds **do**
  Choose such an η_j
  for all X_i that η_j depends on **do**
    Assign the next number of
    the i-th row of RT to X_i
Output the current values of $X_1, \ldots, X_N$

Fig. 3. The RT-MT algorithm

Randomly generate an RT
$succ := 1$
for all $\eta_j \in g_{\mathsf{WT}}(wt)$ **do**
  for all X_i that η_j depends on **do**
    Assign the next number of
    the i-th row of RT to X_i
  if η_j does not hold **then** $succ := 0$
Output $succ$

Fig. 4. The RT-check(wt) algorithm

Since the MT algorithm and check(wt) are "equivalent" to their RT-based counterparts respectively, to prove (b), we only need to show that,

(b') $\Pr[wt = f_{\mathsf{WT}}(\text{some prefix of } \Lambda \text{ of RT-MT})] \leq \Pr[\text{RT-check}(wt) \text{ outputs } 1]$.

Note that the first lines of the RT-MT algorithm and RT-check(wt) are the same, and all other parts of these two programs are non-probabilistic. Thus, we couple the random sources of the RT-MT algorithm and RT-check(wt), or rather, let the first lines of these two programs generate the same RT. Then it remains to prove that, for any RT, if wt can be generated from the RT-MT algorithm using this RT, then RT-check(wt) with the same RT must output 1.

The proof is based on the following observation. If wt can be generated from the RT-MT algorithm using RT, then *in retrospect* RT must have some crucial properties, and these properties will make RT-check(wt) output 1. More precisely, for all events η_j in wt, at the time η_j is chosen in the execution of the RT-MT algorithm, it must hold under the current assignment formed by some of RT's entries. Then, during the execution of RT-check(wt), when the program tests η_j, the test passes because the current assignment must be formed by (almost) the same entries of RT.

Stage 3. Finally, Moser and Tardos prove that,

$$\sum_{wt} p(wt) \leq r_{\mathsf{EL}}, \quad \text{if the Erdős-Lovász condition holds.} \tag{4}$$

It can be proved in a purely mathematical (i.e. program-independent) yet simple way, as pointed out by Srinivasan [58].

Combining all three stages above, Moser and Tardos obtain (1):

$$\begin{aligned} \mathbb{E}[|\Lambda|] &= \sum_{wt} \Pr[wt = f_{\mathsf{WT}}(\text{some prefix of } \Lambda)] && \text{Stage 1, (2)} \\ &\leq \sum_{wt} p(wt) && \text{Stage 2, (3)} \\ &\leq r_{\mathsf{EL}}. && \text{Stage 3, (4)} \end{aligned}$$

Two parts in Moser and Tardos's reasoning that need more careful formalization. First, Moser and Tardos restate their ultimate proof goal as (1) using $|\Lambda|$, the length of the execution log Λ. However, their restatement is ambiguous, since without defining the program semantics, it is unclear how programs are executed and generate execution logs. Similar ambiguity arises when stating those subgoals that also involve quantities related to Λ, e.g. (2) and (3).

Second, Moser and Tardos's original proof of *Stage 2* is far from rigorous. To prove (b), they assume that the MT algorithm and check(wt) are "equivalent" to their RT-based variants, but they did not strictly define and prove the "equivalences". Besides, they did not give a rigorous proof of (b') with these RT-based variants strictly defined.

In the next subsections, we show how we formally state and verify Moser and Tardos's result. We illustrate the proof path in Fig. 5, which is also explained below.

2.2 Stating Proof Goals Using Distribution-Based Semantics

To formally state Moser and Tardos's ultimate proof goal, we must formulate the program semantics and the expected total number of iterations (or equivalently, the expected length of the execution log Λ).

We use a classic distribution-based semantics as the formal program semantics. This semantics (and other equivalent semantics, e.g. the probabilistic wp-semantics [45, 48] and Kozen's "Semantics 2" [44]) is commonly used in the literature of probabilistic program verification (e.g. [45, 48, 3, 7, 21]). It interprets the execution result of a program C as a sub-distribution μ over states. For any state σ, this final state sub-distribution μ specifies the probability that the program C terminates at σ.

For specifying the expected total number of iterations, we introduce a fresh program variable cnt that records the number of iterations. Our code of the MT algorithm, $C_{\mathsf{MT}}(cnt)$, sets cnt to zero at the beginning, and increments it in each iteration of the outer loop. Consequently, when $C_{\mathsf{MT}}(cnt)$ terminates, the value of cnt is the total number of iterations.

Now, our proof goal can be stated as the following *total correctness* Hoare triple (assuming that the Erdős-Lovász condition holds on the probability space):

$$\vDash [\mathbf{true}]\, C_{\mathsf{MT}}(cnt)\, [\mathbb{E}[cnt] \leq r_{\mathsf{EL}}]\,. \tag{5}$$

Informally it says, the execution of $C_{\mathsf{MT}}(cnt)$ in the distribution-based semantics almost surely terminates (i.e., terminates with probability 1), and the expectation of the value of cnt (represented as $\mathbb{E}[cnt]$) at the final state sub-distribution is no greater than r_{EL}. The goal is shown on the top of Fig. 5.

For proving (5), we follow the original proof. That is, we formulate the subgoals in the three stages in Sec. 2.1 using distribution-based semantics, and then prove them. However, we encounter a challenge when formulating (2) and (3).

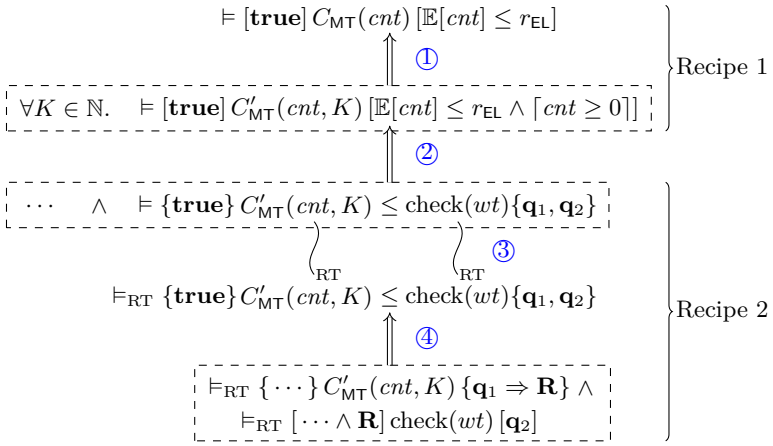

Fig. 5. Our proof path of Moser and Tardos's result, where $\mathbf{q}_1 = \mathsf{Gen}(wt, cnt, K)$ and $\mathbf{q}_2 = \mathsf{Succ}$

Challenge: Handling infinite execution traces. The problem arises when formulating the probability (6), which appears in both (2) and (3).

$$\Pr[wt = f_{\mathsf{WT}}(\text{some prefix of } \Lambda)] \tag{6}$$

Let μ be the final state sub-distribution of $C_{\mathsf{MT}}(cnt)$. Then, it is challenging to formulate (6) using μ. Note that (6) can be positive even when $C_{\mathsf{MT}}(cnt)$ never terminates. But if we simply define (6) as the probability of some event on μ, this probability must be 0 if $C_{\mathsf{MT}}(cnt)$ never terminates, since μ is now a null sub-distribution (which specifies that $C_{\mathsf{MT}}(cnt)$ terminates at σ with probability 0 for any σ). Other definition attempts using μ may also fail.

The difficulty in formulating (6) lies in the following facts. On the one hand, (6) is the total probability of $C_{\mathsf{MT}}(cnt)$'s possibly infinite execution traces on which $wt = f_{\mathsf{WT}}(\text{some prefix of } \Lambda)$ holds. This is a complex property that may involve only some of $C_{\mathsf{MT}}(cnt)$'s infinite traces. On the other hand, distribution-based semantics can only express certain simple properties of infinite traces, and thus cannot express (6). From μ, all we know about $C_{\mathsf{MT}}(cnt)$'s infinite traces is their overall probability $1 - |\mu|$, where $|\mu|$ is the weight of μ (see Sec. 3.1).

One should not simply rule out infinite traces by strengthening (2) and (3) to include almost sure termination of $C_{\mathsf{MT}}(cnt)$, since in Sec. 2.1 the termination has not been derived until the ultimate goal is fully proved (also, it is not easy to prove the termination alone, as discussed in Sec. 7).

2.3 Proof Recipe 1: Loop Truncation

We circumvent the aforementioned challenge by proposing *loop truncation*. Our idea is to do a code transformation on loops, so that the codes after transformation do not generate infinite traces. For the main loop in $C_{\mathsf{MT}}(cnt)$, our

transformation introduces a loop bound K whose value is an arbitrary natural number, and turns the original loop **while** (b) **do** C into a set of truncated loops $\{\,\textbf{while}\ (b \wedge \mathit{cnt} < K)\ \textbf{do}\ C \mid K \in \mathbb{N}\,\}$. Since we increment cnt in the loop body C, each truncated loop **while** $(b \wedge \mathit{cnt} < K)$ **do** C terminates in at most K rounds, and thus can only generate finite execution traces.

Soundness of this transformation can be captured by Lem. 1 below (we will show the more general form in Thm. 2 in Sec. 5.1). It says, the original loop guarantees almost sure termination and its expected total iteration number is bounded by r, as long as all the truncated loops terminate and their expected total iteration numbers have the same upper bound r. Here $\lceil \mathit{cnt} \geq 0 \rceil$ says, cnt, the number of iterations, is always non-negative after **while** $(b \wedge \mathit{cnt} < K)$ **do** C's execution. Without this condition the transformation is unsound.

Lemma 1. *For all P, b, C, r, if*

$$\forall K \in \mathbb{N}. \quad \vDash [P]\,\textbf{while}\ (b \wedge \mathit{cnt} < K)\ \textbf{do}\ C\,[\mathbb{E}[\mathit{cnt}] \leq r \wedge \lceil \mathit{cnt} \geq 0 \rceil]\,,$$

then $\vDash [P]\,\textbf{while}\ (b)\ \textbf{do}\ C\,[\mathbb{E}[\mathit{cnt}] \leq r]$.

Using this transformation, we can reduce (5) to proving the total correctness of $C'_{\mathsf{MT}}(\mathit{cnt}, K)$ for all K, where $C'_{\mathsf{MT}}(\mathit{cnt}, K)$ is the resulting code after transforming the main loop of $C_{\mathsf{MT}}(\mathit{cnt})$ to a truncated one. That is, we prove (7) for all K.

$$\vDash [\textbf{true}]\, C'_{\mathsf{MT}}(\mathit{cnt}, K)\, [\mathbb{E}[\mathit{cnt}] \leq r_{\mathsf{EL}} \wedge \lceil \mathit{cnt} \geq 0 \rceil] \tag{7}$$

We show this as Step ① in Fig. 5. The double arrow represents logical implication. Then we can prove (7) following Moser and Tardos's proof ideas explained in Sec. 2.1. We formulate subgoals (2) and (3) for $C'_{\mathsf{MT}}(\mathit{cnt}, K)$; however, we will not encounter the aforementioned challenge, since $C'_{\mathsf{MT}}(\mathit{cnt}, K)$ does not have infinite execution traces.

Serving as a proof method for PAST. Lem. 1 is itself a general proof method for positive almost sure termination (PAST) [11], whenever we use cnt to record the number of program steps. The PAST property says, the program terminates not only almost surely, but also within finite number of steps in expectation. We give an example in Ex. 1 in Sec. 5.1.

2.4 Proof Recipe 2: Resampling-Table-Based Coupling

Following the ideas in Sec. 2.1, we prove (7) in three stages. The most challenging part is proving (b) in *Stage 2*, which is an inequality between probabilities involving two programs.

We first formally specify the inequality. To this end, we introduce the tuple $\vDash \{P\} C_1 \leq C_2 \{\mathbf{q}_1, \mathbf{q}_2\}$. Here P is a predicate specifying state distributions μ, while $\mathbf{q}_1$ and $\mathbf{q}_2$ are predicates over states σ. The tuple says that, the probability of $\mathbf{q}_1$ holding at the terminating states of C_1 is not greater than the probability

of $\mathbf{q}_2$ holding at the terminating states of C_2, where C_1 and C_2's executions start from the same μ satisfying P and use the distribution-based semantics. Then, we can formulate (b) for $C'_{\mathsf{MT}}(cnt, K)$ and check(wt) as follows.

$$\vDash \{\mathbf{true}\} C'_{\mathsf{MT}}(cnt, K) \leq \text{check}(wt) \{\mathsf{Gen}(wt, cnt, K),\ \mathsf{Succ}\} \tag{8}$$

Here $\mathsf{Gen}(wt, cnt, K)$ roughly says that wt can be generated and is well-formed with respect to cnt and K. The predicate Succ says that the output $succ$ is 1. See Step ② in Fig. 5.

Following Moser and Tardos's proof in Sec. 2.1, we introduce the RT-MT algorithm (now with a truncated loop) and the RT-check(wt) algorithm. We need to give strict definitions of these variants, and to prove that they are indeed equivalent to the original $C'_{\mathsf{MT}}(cnt, K)$ and check(wt) respectively.

Resampling-table-based semantics. Instead of introducing the RT-MT algorithm and the RT-check(wt) algorithm with explicit statements for generating the RT and accessing it, our approach is to *keep the program syntax unchanged but re-interpret the code using a new semantics*. Our RT is a built-in structure of the new semantics, and it is randomly generated before programs start execution.

More specifically, we re-interpret (8) using the novel *RT-based semantics*. In this semantics, we let a program execute with a resampling table RT, which stores all sampling results of the program in advance, and serves as an oracle for the sampling statements in the program. Each sampling statement is interpreted as a query to RT. So this semantics is deterministic given a specific RT.

Our RT-based semantics is equivalent to the classic distribution-based semantics explained in Sec. 2.2. By specifying and proving the semantics equivalence, we essentially show that all programs (including the MT algorithm and check(wt) in Sec. 2.1) are "equivalent" to their RT-based variants.

Based on the semantics equivalence, we can show the equivalence between $\vDash \{P\} C_1 \leq C_2 \{\mathbf{q}_1, \mathbf{q}_2\}$ and $\vDash_{\text{RT}} \{P\} C_1 \leq C_2 \{\mathbf{q}_1, \mathbf{q}_2\}$. The latter specifies the same relational property as the former but uses the RT-based semantics for execution. See Step ③ in Fig. 5.

Resampling-table-based coupling. Our proof recipe reduces the relational verification for $\vDash_{\text{RT}} \{P\} C_1 \leq C_2 \{\mathbf{q}_1, \mathbf{q}_2\}$ to unary verification of each of C_1 and C_2 in the RT-based semantics.

Specifically, we couple the random sources of C_1 and C_2, i.e. let them use the same RT in their executions. We prove: for all RT, if C_1 using RT terminates on a state satisfying $\mathbf{q}_1$, then C_2 using the same RT must also terminate on a state satisfying $\mathbf{q}_2$.

To prove this, we introduce an intermediate assertion $\mathbf{R}$ to describe what kind of RT can make $\mathbf{q}_1$ hold after the execution of C_1. Usually $\mathbf{R}$ specifies that "some entries in RT have some properties". With $\mathbf{R}$, we can split the goal into the following two subgoals:

– For all RT, if C_1 using RT terminates at a state satisfying $\mathbf{q}_1$, then *in retrospect* RT must satisfy $\mathbf{R}$. This is formulated as the Hoare-triple

$$\vDash_{\text{RT}} \{\cdots\}\, C_1\, \{\mathbf{q}_1 \Rightarrow \mathbf{R}\}\,. \tag{9}$$

The post-condition reflects this retrospective reasoning. We omit the pre-condition, which usually degenerates to a regular state assertion. Then we only need classical (non-probabilistic) Hoare-style proofs for the Hoare triple.

– Starting with any RT satisfying $\mathbf{R}$, the execution of C_2 must terminate at a final state satisfying $\mathbf{q}_2$, that is,

$$\vDash_{\mathrm{RT}} [\cdots \wedge \mathbf{R}]\, C_2\, [\mathbf{q}_2]\,. \tag{10}$$

Here $\mathbf{R}$ is in the precondition. We omit the rest parts of the precondition.

Note that the first subgoal (9) only needs to be *partial correctness*. It says, for any execution of C_1, if it terminates and the final state satisfies $\mathbf{q}_1$, RT must satisfy $\mathbf{R}$. Then the *total correctness* of C_2 (the second subgoal (10)) says, starting from the same RT, C_2 terminates at a final state satisfying $\mathbf{q}_2$. This way we can prove that the probability of $\mathbf{q}_1$ at the end of C_1 is not greater than the probability of $\mathbf{q}_2$ at the end of C_2. Step ④ in Fig. 5 shows this reduction of the relational reasoning to unary proofs of the two programs separately.

Our reasoning above benefits from a key novelty of our RT-based semantics with respect to existing random-source-based semantics (e.g. Kozen's "Semantics 1" [44] and those in [10, 17]). That is, our RT is an immutable structure that never changes during program execution. In particular, used samples are not popped out of RT. Therefore the assertion $\mathbf{R}$ derived from the post-condition of (9) must also hold over the RT at the beginning of the execution. So we can use it in the precondition in (10).

Finding such an $\mathbf{R}$ is not difficult in many cases, especially when verifying ALLLs. We give another example in Sec. 5.2.

3 Preliminaries

In this section, we review some fundamentals of probability theory in two stages. We first introduce some basics of discrete probability theory without mentioning their measure-theoretic extensions, serving as the foundation of our distribution-based semantics in Sec. 4.1. Then we turn to the measure-theoretic probability theory, which forms the basis of our RT-based semantics in Sec. 4.2.

3.1 Discrete Probability Theory

We use notations from [21, 3]. A (discrete) *sub-distribution* over a set A is defined as a function $\mu : A \to [0, 1]$ that satisfies the following two conditions: (1) the support of μ, denoted by $\mathit{supp}(\mu) = \{a \in A : \mu(a) > 0\}$, is countable; (2) $|\mu| \leq 1$, where $|\mu| = \sum_{a \in A} \mu(a)$ is μ's weight.

A sub-distribution μ is called a *distribution* if $|\mu| = 1$. We denote by $\mathbb{SD}_A$ all of the sub-distributions over A, and by $\mathbb{D}_A$ all of the distributions over A. We write $\Pr_{a\sim\mu}[E(a)]$, which is defined as $\sum_{a \in A : E(a)} \mu(a)$, for the probability of $E : A \to \mathrm{Prop}$ on the sub-distribution μ. We write $\mathbb{E}_{a\sim\mu}[V(a)]$, which is defined as $\sum_{a \in A} \mu(a) \cdot V(a)$, for the expected value of $V : A \to \mathbb{R}$ on μ.

$$(\mathit{Dsts})\ \mathcal{D} ::= (\kappa_1, \ldots, \kappa_N) \qquad (\mathit{Evts})\ \mathcal{E} ::= (\eta_1, \ldots, \eta_M)$$
$$(\mathit{Dst})\ \kappa \in \mathbb{D}_{\mathit{Real}} \qquad (\mathit{Evt})\ \eta \in \underbrace{\mathit{Real} \times \cdots \times \mathit{Real}}_{N\ \mathit{Real}\text{'s}} \to \{\text{true}, \text{false}\}$$
$$\text{vbl}(\eta, j) \ \text{ iff } \ \exists r_1, \ldots, r_N, r'.\ \eta(r_1, \ldots, r_N) \neq \eta(r_1, \ldots, r_{j-1}, r', r_{j+1}, \ldots, r_N)$$
$$\mathrm{P}(\eta) \triangleq \sum_{\substack{r_1 \in \mathit{supp}(\mathcal{D}[1]), \ldots, r_N \in \mathit{supp}(\mathcal{D}[N]) \\ \eta(r_1, \ldots, r_N) = \text{true}}} \prod_{i \in [1,N]} \mathcal{D}[i](r_i)$$
$$\Gamma(j) \triangleq \{k : \exists i.\ \text{vbl}(\mathcal{E}[j], i) \wedge \text{vbl}(\mathcal{E}[k], i))\} \setminus \{j\}$$

$$(\mathit{Expr})\ e ::= v \mid x \mid e_1 + e_2 \mid a[e] \mid e_1\langle e_2 \rangle \mid \mathsf{len}(e) \mid \mathsf{app}(e_1, e_2) \mid \ldots$$
$$(\mathit{Bexp})\ b ::= \text{true} \mid \text{false} \mid e_1 = e_2 \mid b_1 \wedge b_1 \mid \mathsf{hold}(e, e_1, \ldots, e_N) \mid \mathsf{vbl}(e_1, e_2) \mid \ldots$$
$$(\mathit{Stmt})\ C ::= \textbf{skip} \mid x := e \mid x := \mathsf{Sample}(e) \mid a[e_1] := e_2$$
$$\mid C_1; C_2 \mid \textbf{if}\ (b)\ \textbf{then}\ C_1\ \textbf{else}\ C_2 \mid \textbf{while}\ (b)\ \textbf{do}\ C \mid \ldots$$

Fig. 6. Syntax of the programming language

For an infinite sequence $\vec{\mu}$, we define $\lim \vec{\mu}$ as the sub-distribution μ such that $\lim_{n\to\infty} \sum_{a \in A} |\vec{\mu}[n](a) - \mu(a)| = 0$. One can prove that such a μ is unique if it exists, otherwise we leave $\lim \vec{\mu}$ undefined.

For $\mu \in \mathbb{SD}_A$ and function $f \in A \to \mathbb{SD}_B$, we define the *expected sub-distribution* $\mathbb{E}_{a\sim\mu}\{f(a)\} \in \mathbb{SD}_B$ as $\lambda b.\ \sum_{a \in A} \mu(a) \cdot f(a)(b)$.

3.2 Measure-Theoretic Probability Theory

A set of subsets of a set Ω, say $\mathcal{F}$, is a σ-algebra on Ω if it contains Ω and is closed under complement and countable union. A measurable space is defined as a pair $(\Omega, \mathcal{F})$, where $\mathcal{F}$ is a σ-algebra on Ω. We call Ω the *sample space*.

A function $\mathcal{M} : \mathcal{F} \to [0, \infty)$ is called a (finite) *measure* on measurable space $(\Omega, \mathcal{F})$ if it satisfies $\mathcal{M}(\varnothing) = 0$ and is countably additive. A *measure space* is defined as a triple $(\Omega, \mathcal{F}, \mathcal{M})$, where $\mathcal{M}$ is a measure on measurable space $(\Omega, \mathcal{F})$. $(\Omega, \mathcal{F}, \mathcal{M})$ is called a *probability space* if $\mathcal{M}(\Omega) = 1$.

A discrete distribution μ can be lifted to a measure-theoretic probability space $(\Omega, \mathcal{F}, \mathcal{M})$, where $\Omega = \mathit{supp}(\mu)$, $\mathcal{F} = \mathcal{P}(\mathit{supp}(\mu))$, and $\mathcal{M}(A) = \sum_{a \in A} \mu(a)$ for all $A \subseteq \mathit{supp}(\mu)$.

Let $\{(\Omega_i, \mathcal{F}_i, \mathcal{M}_i) : i \in I\}$ be a collection of probability spaces for some possibly infinite set I. We denote by $\prod_{i\in I}(\Omega_i, \mathcal{F}_i, \mathcal{M}_i)$ the *product probability space* of $\{(\Omega_i, \mathcal{F}_i, \mathcal{M}_i) : i \in I\}$, defined as $(\Omega, \mathcal{F}, \mathcal{M})$, where: (1) $\Omega = \prod_{i\in I} \Omega_i$, (2) $\mathcal{F}$ is the smallest σ-algebra containing all $\prod_{i\in I} A_i$ such that $A_i \in \mathcal{F}_i$ and $\{j : A_j \subsetneq \Omega_j\}$ is finite, and (3) $\mathcal{M}(\prod_{i\in I} A_i) = \prod_{j\in J} \mathcal{M}_j(A_j)$ when $A_i \in \mathcal{F}_i$ and $J = \{j : A_j \subsetneq \Omega_j\}$ is finite. The above $(\Omega, \mathcal{F}, \mathcal{M})$ exists and is unique (see [54]).

4 Two Semantics of the Language

In this section we define the programming language. We first define the language syntax, and then give two equivalent semantics in Sec. 4.1 and Sec. 4.2.

Global parameters. Throughout the paper, we assume four global parameters for programs: N, M, $\mathcal{D}$ and $\mathcal{E}$. They are viewed as meta-variables, and can be configured differently for different programs.

As defined at the top of Fig. 6, $\mathcal{D}$ and $\mathcal{E}$ represent the "N distributions" and "M events" in ALLL's setting (see Sec. 2.1) respectively. Each event η_j in $\mathcal{E}$ takes N reals as input, and outputs a boolean value. Each κ_i in $\mathcal{D}$ is a distribution over reals, and is associated with the i-th argument of every η_j in $\mathcal{E}$.

Fig. 6 also gives important notations related to $\mathcal{D}$ and $\mathcal{E}$, which are used in the statements and the formal proofs of ALLL-related results. $\mathrm{vbl}(\eta, j)$ holds iff the event η depends on its j-th argument.[2] $\mathrm{P}(\eta)$ is the probability of the event η occurring, given that its N arguments are independently distributed according to $\mathcal{D}[1], \ldots, \mathcal{D}[N]$ respectively. $\Gamma(j)$ is the index set of events that depend on some argument that $\mathcal{E}[j]$ also depends on, except $\mathcal{E}[j]$ itself.

Syntax of the programming language. As shown at the bottom of Fig. 6, we use customized program statements, expressions and boolean expressions to formulate ALLLs' code. We write $x := \mathsf{Sample}(e)$ to sample from the distribution $\mathcal{D}[e]$ and store the result in the program variable x. The boolean expression $\mathsf{hold}(e, e_1, \ldots, e_N)$ tests if the event $\mathcal{E}[e]$ holds with arguments $e_1, \ldots, e_N$. Moreover, $\mathsf{vbl}(e_1, e_2)$ tests if the event $\mathcal{E}[e_1]$ depends on its e_2-th argument.

We use arrays to formulate the N variables $X_1, \ldots, X_N$ in ALLLs. We use $a[e]$ to represent the element of array a with index e, and use $a[e_1] := e_2$ for the in-place update.

We use lists to formulate the execution logs in ALLLs. To access and manipulate the execution log, we introduce list-related expressions. We use $e_1\langle e_2\rangle$ for the e_2-th element of list e_1, use $\mathsf{len}(e)$ for the length of list e, and use $\mathsf{app}(e_1, e_2)$ for appending an element e_2 to list e_1.

Using the syntax in Fig. 6, we can formulate the code of the MT algorithm, $C_{\mathsf{MT}}(cnt)$, in Fig. 12 in Sec. 6.

States and state distributions. As defined below, a state σ maps each program variable in $PVar$ to some value v. For simplicity, we view each array element as a program variable. A value v is either a real r or a list Λ of natural numbers.

$$(State) \quad \sigma \in PVar \to Val \qquad (DState) \quad \mu \in \mathbb{D}_{State}$$

State distributions μ are used to specify that, with probability $\mu(\sigma)$, the program state before or after the execution of a program is exactly σ. We write $[\![e]\!]_\sigma$ and $[\![b]\!]_\sigma$ for the evaluation of e and b in a state σ.

Below we give two equivalent probabilistic semantics of our language, a classic distribution-based semantics and an RT-based semantics. We use n for natural numbers and p, r for reals. Throughout this paper, we assume that the program's execution does not get stuck, and the evaluation of expressions does not abort.

[2] The name "vbl" is short for "variables". Moser and Tardos [50] used $\mathrm{vbl}(\eta)$ as the minimal set of variables (i.e. arguments of the event) that determine η.

r_{10}	r_{11}	r_{12}	r_{13}	$\cdots$
r_{20}	r_{21}	r_{22}	r_{23}	$\cdots$

Fig. 7. A resampling table RT with $N = 2$

4.1 Distribution-Based Semantics

Following [21, 3], we first define the semantic function $[\![C]\!](\sigma) \in \mathbb{SD}_{State}$. Here $[\![C]\!](\sigma)(\sigma')$ represents the probability of C's execution from σ finally reaching σ'. For example, for the sampling operation $x := \mathsf{Sample}(e)$ that samples from the distribution $\mathcal{D}[i]$ and gets r as the result, the probability is $\mathcal{D}[i](r)$. That is,

$$[\![x := \mathsf{Sample}(e)]\!](\sigma)(\sigma') = \begin{cases} \mathcal{D}[i](r) & \text{if } [\![e]\!]_\sigma = i \in [1, N] \text{ and } \sigma' = \sigma\{x \rightsquigarrow r\} \\ 0 & \text{otherwise} \end{cases}.$$

We give the full definition in [46]. We further define $[\![C]\!](\mu) \in \mathbb{SD}_{State}$ (where $\mu \in DState$) by lifting $[\![C]\!](\sigma)$, using the expected sub-distribution in Sec. 3.1:

$$[\![C]\!](\mu) \triangleq \mathbb{E}_{\sigma\sim\mu}\{[\![C]\!](\sigma)\}.$$

4.2 Resampling-Table-Based Semantics

Informally, in our new RT-based semantics, a program first randomly generates a resampling table (RT); with this table, the program then starts its deterministic execution. Below we first give the definition of an RT, and specify how the semantics "generates" an RT. Then we define an RT-based operational semantics, which describes the deterministic execution of the program with a certain RT. Finally, we combine all the above definitions into the RT-based semantic functions $[\![C]\!]_{\mathrm{RT}}(\sigma)$ and $[\![C]\!]_{\mathrm{RT}}(\mu)$.

The resampling table is defined as follows.

$$\begin{array}{lll} (RTable) \quad RT & \in & [1, N] \times Nat \to Real \quad \text{where generable}(RT) \\ \text{generable}(RT) & \text{iff} & \forall i, j.\ RT[i][j] \in supp(\mathcal{D}[i]) \end{array}$$

A resampling table RT is a matrix with size $N \times \infty$. An example of such table is shown in Fig. 7, where $N = 2$ and $RT[i][j] = r_{ij}$ for $i \in [1, 2]$ and $j \in Nat$. Intuitively, as described in Sec. 2.1, the i-th row of RT stores the ahead-of-time samples from the distribution $\mathcal{D}[i]$. Additionally, we require that generable(RT) holds. That is, every entry in the i-th row of RT must be able to be sampled from the distribution $\mathcal{D}[i]$. This accords with the intuition of the RT.

We specify how the semantics "generates" an RT. To this end, we define the probability space of all (generable) RTs as $(\Omega, \mathcal{F}, \mathcal{M})$, and thus $\mathcal{M}(\{RT \mid \cdots\})$ represents the probability of some RT from set $\{RT \mid \cdots\}$ being generated. The definition is shown below:

$$(\Omega, \mathcal{F}, \mathcal{M}) \triangleq \prod_{(i,j)\in[1,N]\times Nat}(\Omega_{i,j}, \mathcal{F}_{i,j}, \mathcal{M}_{i,j}),$$

$$\frac{[\![e]\!]_\sigma = v}{RT \vdash (x := e, \sigma, \iota) \to (\mathbf{skip}, \sigma\{x \rightsquigarrow v\}, \iota)}$$

$$\frac{[\![e]\!]_\sigma = i \in [1, N] \qquad \iota' = (\iota[1], \ldots, \iota[i-1], \iota[i]+1, \iota[i+1], \ldots, \iota[N])}{RT \vdash (x := \mathsf{Sample}(e), \sigma, \iota) \to (\mathbf{skip}, \sigma\{x \rightsquigarrow RT[i][\iota[i]]\}, \iota')}$$

Fig. 8. RT-based operational semantics

where $(\Omega_{i,j}, \mathcal{F}_{i,j}, \mathcal{M}_{i,j})$ is lifted from the discrete distribution $\mathcal{D}[i]$ (see Sec. 3.2). Note that $\Omega = \mathit{RTable}$, i.e., the sample space is indeed the set of all RTs.

Below we explain our construction of $(\Omega, \mathcal{F}, \mathcal{M})$. The probability space of all RTs is the *infinite product* of probability spaces of all entries, since an RT is generated by filling all of its entries by an infinite number of independent samples. For the entry in row i and (arbitrary) column j, its probability space is lifted from $\mathcal{D}[i]$, from which the entry is sampled.

We then define the RT-based operational semantics, with selected semantics rules shown in Fig. 8. The definition is almost standard, except that it interprets sampling operations to table queries. Recall that, when the program performs a sampling from the distribution $\mathcal{D}[i]$, it reads the leftmost unread entry in the i-th row of RT as the result. To keep track of these entries, we maintain the heads ι in the program configuration to record their column numbers.

$$(\mathit{Heads}) \quad \iota \quad ::= \quad (n_1, \ldots, n_N)$$

ι is an N-tuple. Its i-th component, $\iota[i]$, represents the column number of the leftmost unread entry in the i-th row of RT. Now, $RT \vdash (C, \sigma, \iota) \to^* (C', \sigma', \iota')$ says that, starting from the program state σ, with the leftmost unread entries of RT initially specified by ι, C deterministically executes to C' using RT, where the result state is σ' and finally the leftmost unread entries in RT are specified by ι'. When the program performs a sampling from $\mathcal{D}[i]$, it takes $RT[i][\iota[i]]$ as the result and increments $\iota[i]$. In other program steps, ι remains unchanged.

Now the RT-based semantic functions are defined below, where $\iota_{\mathsf{init}} = (0, \ldots, 0)$ represents the initial positions of heads.

$$[\![C]\!]_{\mathrm{RT}}(\sigma) \triangleq \lambda\sigma'.\, \mathcal{M}(\{RT \mid RT \vdash (C, \sigma, \iota_{\mathsf{init}}) \to^* (\mathbf{skip}, \sigma', \_)\})$$
$$[\![C]\!]_{\mathrm{RT}}(\mu) \triangleq \mathbb{E}_{\sigma\sim\mu}\{[\![C]\!]_{\mathrm{RT}}(\sigma)\}$$

Informally, the probability of C's execution from σ finally reaching σ', say $[\![C]\!]_{\mathrm{RT}}(\sigma)(\sigma')$, is the probability of some RT, which satisfies the following property, being generated: starting from σ, C's execution using RT finally reaches σ'. This property is formally stated as $RT \vdash (C, \sigma, \iota_{\mathsf{init}}) \to^* (\mathbf{skip}, \sigma', \_)$, with the help of the operational semantics.

Lem. 2 shows that the RT-based semantics is indeed well-defined.

Lemma 2. *For all* $C, \sigma, \sigma', \iota$, $\{RT \mid RT \vdash (C, \sigma, \iota) \to^* (\mathbf{skip}, \sigma', \_)\} \in \mathcal{F}$.

To conclude this subsection, we give the following theorem, which states the equivalence between the distribution-based semantics defined in Sec. 4.1 and the RT-based semantics.

$$
\begin{array}{ll}
(\mathit{Assn}) & \mathbf{p}, \mathbf{q}, \mathbf{r} ::= b \mid \neg \mathbf{q} \mid \mathbf{q}_1 \wedge \mathbf{q}_2 \mid \mathbf{q}_1 \vee \mathbf{q}_2 \mid \forall X.\, \mathbf{q} \mid \exists X.\, \mathbf{q} \mid \ldots \\
(\mathit{PExp}) & \xi ::= r \mid \mathbb{E}[e] \mid \Pr[\mathbf{q}] \mid \xi_1 + \xi_2 \mid \xi_1 - \xi_2 \mid \ldots \\
(\mathit{PAssn}) & P, Q, R ::= \lceil \mathbf{q} \rceil \mid \xi_1 = \xi_2 \mid \neg Q \mid Q_1 \wedge Q_2 \mid \forall X.\, Q \mid \exists X.\, Q \mid \ldots
\end{array}
$$

$$
\begin{array}{ll}
[\![r]\!]_\mu \triangleq r & \mu \vDash \lceil \mathbf{q} \rceil \quad \text{iff } \forall \sigma.\ \sigma \in \mathit{supp}(\mu) \Longrightarrow \sigma \vDash \mathbf{q} \\
[\![\mathbb{E}[e]]\!]_\mu \triangleq \mathbb{E}_{\sigma \sim \mu}[[\![e]\!]_\sigma] & \mu \vDash \xi_1 = \xi_2 \ \text{ iff } \ [\![\xi_1]\!]_\mu = [\![\xi_2]\!]_\mu \\
[\![\Pr[\mathbf{q}]]\!]_\mu \triangleq \Pr_{\sigma \sim \mu}[\sigma \vDash \mathbf{q}] & \mu\{X \rightsquigarrow v\} \triangleq \mathbb{E}_{\sigma \sim \mu}\{\delta(\sigma\{X \rightsquigarrow v\})\} \\
[\![\xi_1 + \xi_2]\!]_\mu \triangleq [\![\xi_1]\!]_\mu + [\![\xi_2]\!]_\mu & \mu \vDash \exists X.\, Q \ \text{ iff } \ \exists v.\ \mu\{X \rightsquigarrow v\} \vDash Q
\end{array}
$$

Fig. 9. Assertions over states and state distributions

Theorem 1 (Semantics Equivalence). *For all C and μ, $[\![C]\!](\mu) = [\![C]\!]_{\mathrm{RT}}(\mu)$.*

5 Proof Recipes

Our ultimate proof goals are formulated as total correctness Hoare triples $\vDash [P]C[Q]$ using the distribution-based semantics of Sec. 4.1.

Before showing the definition of $\vDash [P]C[Q]$, we first define assertions in Fig. 9, following the assertion language in [21]. We write $\mathbf{p}, \mathbf{q}, \mathbf{r}$ for non-probabilistic assertions on program states, and P, Q, R for probabilistic assertions on state distributions. The assertion $\lceil \mathbf{q} \rceil$ holds on the distribution μ iff $\mathbf{q}$ holds on all states in the support of μ. We write **true** as a shorthand for $\lceil \mathrm{true} \rceil$. The expression $\Pr[\mathbf{q}]$ represents the probability that $\mathbf{q}$ holds, and $\mathbb{E}[e]$ represents the expected value of e. The assertion $\exists X.\, Q$ holds on μ, if Q holds on μ' obtained by assigning some constant v to X in all states in μ (here δ gives the Dirac distribution).

Then, $\vDash [P]C[Q]$ says that, starting from a state distribution satisfying P, C's execution terminates with probability 1, and thus the sub-distribution of the result states is actually a state distribution, which satisfies Q. We show the definition in Def. 1.

Definition 1 (Total Correctness). *For all P, C, Q, $\vDash [P]C[Q]$ holds iff*

$$\forall \mu.\ \ \mu \vDash P \implies |[\![C]\!](\mu)| = 1 \wedge [\![C]\!](\mu) \vDash Q.$$

In the following subsections, we formalize our two proof recipes, loop truncation and RT-based coupling.

5.1 Loop Truncation

We have explained a specialized form of loop truncation in Lem. 1 in Sec. 2.3. Below we show the more general theorem (Thm. 2).

Theorem 2 (Loop Truncation). *For all $P, b, C, \mathbf{E}, Q, e$ and r, if*

$$\forall K \in \mathbb{N}.\quad \vDash [P]\ \mathbf{E}[\textbf{while}\ (b \wedge e < K)\ \textbf{do}\ C]\ [Q \wedge \mathbb{E}[e] \le r \wedge \lceil e \ge 0 \rceil]\,,$$

modbf$(\mathbf{E}, e)$ *and* t-**closed**(Q)*, then* $\vDash [P]\ \mathbf{E}[\textbf{while}\ (b)\ \textbf{do}\ C]\ [Q]$.

Here $\mathbf{E}$ is a program context, and $\mathbf{E}[\textbf{while } (b) \textbf{ do } C]$ fills the hole in $\mathbf{E}$ with the loop $\textbf{while } (b) \textbf{ do } C$.

$$\begin{aligned}(Ctx)\ \ \mathbf{E} ::= {} & [\,] \mid C; \mathbf{E} \mid \mathbf{E}; C \mid \textbf{while } (b) \textbf{ do } \mathbf{E} \\ & \mid \textbf{if } (b) \textbf{ then } C \textbf{ else } \mathbf{E} \mid \textbf{if } (b) \textbf{ then } \mathbf{E} \textbf{ else } C\end{aligned}$$

Thm. 2 says that, to prove total correctness of $\mathbf{E}[\textbf{while } (b) \textbf{ do } C]$, we transform the code to $\mathbf{E}[\textbf{while } (b \wedge e < K) \textbf{ do } C]$ with a specific e. How to choose e is application-dependent. Usually we choose as e the loop counter incremented in the loop body, such as cnt in $C_{\mathsf{MT}}(cnt)$ (see Sec. 2.2 and Fig. 12). With an inappropriate e, the first premise of the theorem may be invalid or still hard to prove, though how e is chosen does not affect the validity of the theorem.

In addition to e, the first premise also asks users to find a common bound r (a real number) that can bound $\mathbb{E}[e]$ at the end of $\mathbf{E}[\textbf{while } (b \wedge e < K) \textbf{ do } C]$ for all K. Usually the postcondition Q can help us find such an r. Besides the upper bound r, we require that evaluating e at the end of $\mathbf{E}[\textbf{while } (b \wedge e < K) \textbf{ do } C]$ must result in a *non-negative* real number. These two bounds are crucial for ensuring almost sure termination of $\mathbf{E}[\textbf{while } (b) \textbf{ do } C]$.

The second premise, $\textbf{modbf}(\mathbf{E}, e)$, rules out those contexts $\mathbf{E}$ that make $\mathbb{E}[e] \leq r$ hold at the end of $\mathbf{E}[\textbf{while } (b \wedge e < K) \textbf{ do } C]$ vacuously, e.g. those that modify the program variables in e at the end of the context and make $e = r$ hold. $\textbf{modbf}(\mathbf{E}, e)$ syntactically restricts $\mathbf{E}$ such that the variables in e can be modified in $\mathbf{E}$ only before the code in the hole of $\mathbf{E}$ is executed. For example, $\textbf{modbf}(C'; [\,], e)$ holds for any C' and e, since only C', which is executed before the hole, can modify the variables in e in the context. Similarly, $\textbf{modbf}([\,], e)$ holds. We give the definition of $\textbf{modbf}(\mathbf{E}, e)$ in [46].

The third premise, $t\text{-}\textbf{closed}(Q)$, is for deriving the postcondition Q of $\mathbf{E}[\textbf{while } (b) \textbf{ do } C]$ from the same Q of $\mathbf{E}[\textbf{while } (b \wedge e < K) \textbf{ do } C]$. We say an assertion Q is t-closed [3], denoted by $t\text{-}\textbf{closed}(Q)$, if for all infinite state distribution sequences $\vec{\mu}$, if Q holds on $\vec{\mu}[i]$ for each i and $\lim \vec{\mu} = \mu$, then Q holds on μ. Many assertions are t-closed. For example, we can prove that $t\text{-}\textbf{closed}(\mathbb{E}[e] \leq r \wedge \lceil e \geq 0 \rceil)$ always holds for any e and any r.

Since $\textbf{modbf}([\,], e)$ and $t\text{-}\textbf{closed}(\mathbb{E}[e] \leq r \wedge \lceil e \geq 0 \rceil)$ both hold, Lem. 1 can be derived from Thm. 2.

Proof Sketch of Thm. 2. Due to the space limit, below we only show the case of $\mathbf{E} = [\,]$. We prove 1) almost sure termination and 2) the establishment of the postcondition Q, respectively.

For 1), assuming that $\textbf{while } (b) \textbf{ do } C$ terminates with probability $p < 1$, we derive a contradiction. From the premise we know $\textbf{while } (b \wedge e < K) \textbf{ do } C$ almost surely terminates, so it terminates in a state where $e \geq K$ with probability at least $1 - p$. Thus, by the semantics of $\mathbb{E}[e]$ (and since the value of e is non-negative), we know $\mathbb{E}[e] \geq (1-p)K$ holds at the end of $\textbf{while } (b \wedge e < K) \textbf{ do } C$. Therefore, we can find a sufficiently large K such that $\mathbb{E}[e] \geq (1-p)K > r$, which contradicts the premise.

For 2), the key is proving that, for all $\mu \models P$,

$$[\![\textbf{while } (b) \textbf{ do } C]\!](\mu) = \lim_{K \to \infty} [\![\textbf{while } (b \wedge e < K) \textbf{ do } C]\!](\mu).$$

Then, we can establish Q for **while** (b) **do** C, from t-**closed**(Q) and that Q is the postcondition for each **while** $(b \wedge e < K)$ **do** C. □

We apply Thm. 2 for the verification of the MT algorithm and its variants in Sec. 6. Here we show another example beyond ALLLs, which is taken from [41] (with slight modifications).

Example 1. Let $N = 1$ and $\mathcal{D}[1] = \{(0, \frac{1}{2}), (1, \frac{1}{2})\}$. The code C_{flip} is defined as **while** $(y = 1)$ **do** $\{\ y := \mathsf{Sample}(1); cnt := cnt + 1;\}$. We prove:

$$\vDash [\lceil cnt = 0 \wedge y = 1 \rceil]\ C_{\text{flip}}\ [\mathbb{E}[cnt] \leq 2]. \tag{11}$$

Here C_{flip} repeatedly flips a fair coin by sampling from $\mathcal{D}[1]$, until it gets heads $(y = 0)$. We use cnt to record the number of coin flips. Then our proof goal (11) says that C_{flip} almost surely terminates, and it flips at most twice in expectation.

To prove (11), by Thm. 2 (or Lem. 1), we only need to prove that, for all $K \in \mathbb{N}$, $\vDash [\lceil cnt = 0 \wedge y = 1 \rceil]\ C'_{\text{flip}}(K)\ [\mathbb{E}[cnt] \leq 2 \wedge \lceil cnt \geq 0 \rceil]$, where $C'_{\text{flip}}(K)$ is defined as **while** $(y = 1 \wedge cnt < K)$ **do** $\{y := \mathsf{Sample}(1); cnt := cnt + 1;\}$. We adapt the program logic Ellora [3] to complete the proof.

5.2 Resampling-Table-Based Coupling

As informally explained in Sec. 2.4, our RT-based coupling is for proving the relational tuple $\vDash \{P\}C_1 \leq C_2\{\mathbf{q}_1, \mathbf{q}_2\}$, an intermediate proof goal that appears in ALLLs' verification. We show the formal definition of $\vDash \{P\}C_1 \leq C_2\{\mathbf{q}_1, \mathbf{q}_2\}$ in Def. 2. Note that in this definition we neither require nor assume the termination of C_1 and C_2's executions.

Definition 2 (Inequality between Probabilities). *For all* $P, C_1, C_2, \mathbf{q}_1, \mathbf{q}_2$, $\vDash \{P\}C_1 \leq C_2\{\mathbf{q}_1, \mathbf{q}_2\}$ *holds iff*

$$\forall \mu.\ \ \mu \vDash P \implies \Pr_{\sigma \sim [\![C_1]\!](\mu)}[\sigma \vDash \mathbf{q}_1] \leq \Pr_{\sigma \sim [\![C_2]\!](\mu)}[\sigma \vDash \mathbf{q}_2].$$

Our RT-based coupling reduces the verification of the relational tuple to proving unary properties of C_1 and C_2's executions in the RT-based semantics respectively (i.e. the subgoals (9) and (10) in Sec. 2.4). We show the formal theorem in Thm. 3.

Theorem 3 (RT-Based Coupling). *For all* $\mathbf{p}, C_1, C_2, \mathbf{q}_1, \mathbf{R}, \mathbf{q}_2$, *if*

- **RTonly**$(\mathbf{R})$;
- $\vDash_{\text{RT}} \{\mathbf{p} \wedge \mathsf{hdinit}\}C_1\{\mathbf{q}_1 \Rightarrow \mathbf{R}\}$;
- $\vDash_{\text{RT}} [\mathbf{p} \wedge \mathbf{R} \wedge \mathsf{hdinit}]C_2[\mathbf{q}_2]$;

then $\vDash \{\lceil \mathbf{p} \rceil\}C_1 \leq C_2\{\mathbf{q}_1, \mathbf{q}_2\}$.

We apply Thm. 3 for verifying ALLLs, which we will explain in Sec. 6. Below we explain Thm. 3 in four aspects: (1) requiring $\lceil \mathbf{p} \rceil$ as the precondition in the relational tuple; (2) the assertions $\mathbf{R}$, hdinit and the requirement **RTonly**$(\mathbf{R})$; (3) the RT-based unary triples $\vDash_{\text{RT}}$; and (4) its proof ideas. We also show another example beyond ALLLs, and briefly discuss an extension of Thm. 3 at the end.

$$
\begin{array}{lll}
(\mathit{RTExpr}) & E & ::= e \mid \mathsf{RT}[E_1][E_2] \mid \mathsf{hd}_1 \mid \ldots \mid \mathsf{hd}_N \mid E_1 + E_2 \mid \ldots \\
(\mathit{RTBexp}) & B & ::= b \mid E_1 = E_2 \mid E_1 < E_2 \mid \ldots \\
(\mathit{RTAssn}) & \mathbf{P}, \mathbf{Q}, \mathbf{R} & ::= \mathbf{q} \mid B \mid \neg \mathbf{Q} \mid \mathbf{Q}_1 \wedge \mathbf{Q}_2 \mid \mathbf{Q}_1 \vee \mathbf{Q}_2 \mid \forall X.\, \mathbf{Q} \mid \exists X.\, \mathbf{Q} \mid \ldots
\end{array}
$$

$$
(\sigma, RT, \iota) \vDash \mathbf{q} \;\text{ iff }\; \sigma \vDash \mathbf{q} \qquad [\![\mathsf{hd}_n]\!]_{(\sigma,RT,\iota)} \triangleq \iota[n]
$$

$$
[\![\mathsf{RT}[E_1][E_2]]\!]_{(\sigma,RT,\iota)} \triangleq RT[i][j], \text{ if } [\![E_1]\!]_{(\sigma,RT,\iota)} = i, [\![E_2]\!]_{(\sigma,RT,\iota)} = j
$$

$$
\mathsf{hdinit} \triangleq \textstyle\bigwedge_{i \in [1,N]} . \, \mathsf{hd}_i = 0
$$

$$
\mathbf{RTonly}(\mathbf{R}) \;\text{ iff }\; \forall \sigma, RT, \iota.\ (\sigma, RT, \iota) \vDash \mathbf{R} \implies \forall \sigma', \iota'.\ (\sigma', RT, \iota') \vDash \mathbf{R}
$$

Fig. 10. Non-probabilistic assertions on RT-extended states

Lifting state assertions as preconditions. The relational tuples we prove are in a restricted form, namely that the precondition P is in the form of $\lceil \mathbf{p} \rceil$, where $\mathbf{p}$ is an assertion over states. Recall that $\lceil \mathbf{p} \rceil$ holds over μ iff $\mathbf{p}$ holds over any σ such that $\sigma \in \mathit{supp}(\mu)$ (see Fig. 9). Therefore the precondition $\lceil \mathbf{p} \rceil$ says we are only interested in the executions of C_1 and C_2 with the initial states satisfying $\mathbf{p}$. So we can fill the omitted part of the two subgoals (9) and (10) with $\mathbf{p}$, and turn them into classical (*deterministic*) Hoare triples $\vDash_{\mathrm{RT}} \{\mathbf{p}\}\, C_1\, \{\mathbf{q}_1 \Rightarrow \mathbf{R}\}$ and $\vDash_{\mathrm{RT}} [\mathbf{p} \wedge \mathbf{R}]\, C_2\, [\mathbf{q}_2]$.

Assertions over RT-extended states. Thm. 3 requires us to find an "intermediate assertion" $\mathbf{R}$ that describes (and *only* describes) the (non-probabilistic) properties of the resampling table RT. Since we need explicit reasoning about RT, the assertions used in the classical reasoning of $\vDash_{\mathrm{RT}}$ actually specify RT and the heads ι as well as the states σ.

In Fig. 10, we define non-probabilistic assertions $\mathbf{P}, \mathbf{Q}, \mathbf{R}$ over the extended states (σ, RT, ι). Besides using $\mathbf{q}$ to describe σ in the extended states, we introduce RT-expressions to specify RT and ι. We use $\mathsf{RT}[E_1][E_2]$ to represent the entry at row E_1 and column E_2 of RT, and use hd_n to represent the n-th head $\iota[n]$, where $n \in [1, N]$.

The assertion hdinit (defined as a shorthand in Fig. 10) says that all of the heads ι point to the first column of RT. It specifies the initial heads before program execution, so it appears in the preconditions of the two $\vDash_{\mathrm{RT}}$ triples in Thm. 3.

The requirement $\mathbf{RTonly}(\mathbf{R})$ (defined in Fig. 10) says that changing σ and/or ι in the extended state does not affect whether $\mathbf{R}$ holds. That is, $\mathbf{R}$ describes RT only. One can check that $\mathbf{RTonly}(\mathbf{R})$ holds if $\mathbf{R}$ does not syntactically contain any free variables and hd_n's.

RT-based unary triples. Now we can define the RT-based unary triples, $\vDash_{\mathrm{RT}} [\mathbf{P}]C[\mathbf{Q}]$ and $\vDash_{\mathrm{RT}} \{\mathbf{P}\}C\{\mathbf{Q}\}$. They are standard Hoare triples for total correctness and partial correctness respectively, using the RT-based operational semantics (in Fig. 8 of Sec. 4.2) for program execution.

Definition 3 (Total Correctness in RT-Based Operational Semantics). *For all* $\mathbf{P}, C, \mathbf{Q}$, $\vDash_{\mathrm{RT}} [\mathbf{P}]C[\mathbf{Q}]$ *holds iff*

```
1  L := []; d := 1;
2  bad := 0;
3  while (d ≤ k) do
4      if (¬findkey(L, x[d])) then
5          y := Sample(1);
6          if (findval(L, y)) then bad := 1;
7          L := app(L, (x[d], y));
8      d := d + 1
```

Fig. 11. The code $C^{\mathsf{bad}}_{\mathrm{PRF}}$ in Ex. 2

$$\forall \sigma, RT, \iota.\ \ (\sigma, RT, \iota) \vDash \mathbf{P} \implies \\ \exists \sigma', \iota'.\ RT \vdash (C, \sigma, \iota) \to^* (\mathbf{skip}, \sigma', \iota') \land (\sigma', RT, \iota') \vDash \mathbf{Q}.$$

Definition 4 (Partial Correctness in RT-Based Operational Semantics). *For all* $\mathbf{P}, C, \mathbf{Q}$, $\vDash_{\mathrm{RT}} \{\mathbf{P}\} C \{\mathbf{Q}\}$ *holds iff*

$$\forall \sigma, RT, \iota, \sigma', \iota'.\ \ (\sigma, RT, \iota) \vDash \mathbf{P} \land RT \vdash (C, \sigma, \iota) \to^* (\mathbf{skip}, \sigma', \iota') \\ \implies (\sigma', RT, \iota') \vDash \mathbf{Q}.$$

For total correctness, Def. 3 says there exists a terminating execution of (C, σ, ι) under RT. This essentially ensures the absence of non-terminating executions, because the RT-based operational semantics is *deterministic*.

We can use a classical Hoare-style program logic to prove the $\vDash_{\mathrm{RT}}$ triples. We show the logic in [46].

Proof ideas of the theorem. To prove Thm. 3, we need to bridge two gaps between the $\vDash_{\mathrm{RT}}$ triples in the premises and the $\vDash$ tuple in the conclusion. First, the $\vDash_{\mathrm{RT}}$ triples use the RT-based semantics, while the $\vDash$ tuple uses the distribution-based semantics. Second, the $\vDash_{\mathrm{RT}}$ triples are unary, while the $\vDash$ tuple is relational.

The key to bridging the gaps is reduction through the following RT-based tuple as an intermediate form, which is the counterpart of Def. 2 in the RT-based semantics.

Definition 5 (Inequality between Pr. in RT-Based Semantics). *For all* $P, C_1, C_2, \mathbf{q}_1, \mathbf{q}_2$, $\vDash_{\mathrm{RT}} \{P\} C_1 \leq C_2 \{\mathbf{q}_1, \mathbf{q}_2\}$ *holds iff*

$$\forall \mu.\ \ \mu \vDash P \implies \Pr_{\sigma \sim [\![C_1]\!]_{\mathrm{RT}}(\mu)}[\sigma \vDash \mathbf{q}_1] \leq \Pr_{\sigma \sim [\![C_2]\!]_{\mathrm{RT}}(\mu)}[\sigma \vDash \mathbf{q}_2].$$

Lemma 3 shows the equivalence between the two relational tuples, which follows from the semantics equivalence (Thm. 1). This lemma bridges the first gap, and is interesting in its own right.

Lemma 3. *For all* $P, C_1, C_2, \mathbf{q}_1, \mathbf{q}_2$,

$$\vDash \{P\} C_1 \leq C_2 \{\mathbf{q}_1, \mathbf{q}_2\} \iff \vDash_{\mathrm{RT}} \{P\} C_1 \leq C_2 \{\mathbf{q}_1, \mathbf{q}_2\}.$$

Our "intermediate assertion" $\mathbf{R}$ allows us to split the $\vDash_{\mathrm{RT}}$ relational tuple into two unary $\vDash_{\mathrm{RT}}$ triples, bridging the second gap.

Example 2. This example is adapted from an intermediate goal in [6]'s proof of the PRP/PRF switching lemma.[3] Let $k \geq 1$. For any $n_1, \ldots, n_k$, we prove that

$$\begin{aligned} &\vDash \{\lceil \mathsf{inp} \rceil\}\, C_{\mathrm{PRF}}^{\mathsf{bad}} \leq C_{\mathrm{PRF}} \\ &\{bad = 1,\ \ \exists X_1, X_2, Y.\ X_1 \neq X_2 \wedge \mathsf{find}(L, (X_1, Y)) \wedge \mathsf{find}(L, (X_2, Y))\}. \end{aligned} \tag{12}$$

We show the code of $C_{\mathrm{PRF}}^{\mathsf{bad}}$ in Fig. 11, and the code of C_{PRF} results from removing lines 2 and 6 from the figure. The assertion inp says that $n_1, \ldots, n_k$ are the inputs stored in $x[1], \ldots, x[k]$, which is defined as $\bigwedge_{i \in [1,k]} . \, x[i] = n_i$.

By extending the programming language, we implement a map in the program variable L, which stores some key-value pairs. One can insert a pair into the map by writing $\mathsf{app}(L, (e_1, e_2))$, and query for the existence of a key, a value or a pair by writing $\mathsf{findkey}(L, e)$, $\mathsf{findval}(L, e)$ or $\mathsf{find}(L, (e_1, e_2))$.

$C_{\mathrm{PRF}}^{\mathsf{bad}}$ and C_{PRF} do the following: for $n = x[1], \ldots, x[k]$, the programs check if n has been inserted in L as a key; if not, they sample a value y from $\mathcal{D}[1]$, and then insert the key-value pair (n, y) into L; if y has been inserted in L as a value, $C_{\mathrm{PRF}}^{\mathsf{bad}}$ marks bad.

(12) then says that, the probability of $C_{\mathrm{PRF}}^{\mathsf{bad}}$ terminating with $bad = 1$ is no more than the probability of C_{PRF} terminating with two key-value pairs with the same value left in L.

To prove (12), we apply Thm. 3. We take $\mathbf{R} = \mathsf{coll}$, where

$$\mathsf{coll} \ \triangleq\ \bigvee_{0 \leq i < j < |\{n_1, \ldots, n_k\}|} . \, \mathsf{RT}[1][i] = \mathsf{RT}[1][j].$$

coll says that, there exist two identical entries in the first row of RT, which are picked as samples in the executions of both $C_{\mathrm{PRF}}^{\mathsf{bad}}$ and C_{PRF}. Therefore coll specifies the kind of RT that can make $bad = 1$ hold after the execution of $C_{\mathrm{PRF}}^{\mathsf{bad}}$.

We can check that $\mathbf{RTonly}(\mathsf{coll})$ holds. Then, by applying Thm. 3, it remains to prove the following two unary $\vDash_{\mathrm{RT}}$ triples.

$\vDash_{\mathrm{RT}} \{\mathsf{inp} \wedge \mathsf{hdinit}\}\, C_{\mathrm{PRF}}^{\mathsf{bad}}\, \{bad = 1 \Rightarrow \mathbf{R}\}$

$\vDash_{\mathrm{RT}} [\mathsf{inp} \wedge \mathbf{R} \wedge \mathsf{hdinit}]\, C_{\mathrm{PRF}}\, [\exists X_1, X_2, Y.\ X_1 \neq X_2 \wedge \mathsf{find}(L, (X_1, Y)) \wedge \mathsf{find}(L, (X_2, Y))]$

We prove them using a simple Hoare-style program logic.

An extension of RT-based coupling. In [46], we give another relational proof recipe that extends Thm. 3. It asks users to provide two intermediate assertions $\mathbf{R}_1$ and $\mathbf{R}_2$ for splitting the $\vDash_{\mathrm{RT}}$ relational tuple, and provides more flexibility for reasoning about inequalities between probabilities.

6 Case Studies

We show the usefulness of our proof recipes (Thm. 2 and Thm. 3) by verifying several representative existing results about ALLLs and a new result about the MT algorithm. Below we first give a brief survey of several important research lines on ALLLs. Then we summarize the existing ALLL-related results that we have verified, and show how we verify Theorem 1.2 of [50] as an example. Finally, we explain our new result about the MT algorithm.

[3] In [6], $C_{\mathrm{PRF}}^{\mathsf{bad}}$ and C_{PRF} are defined using procedure calls. We adapt the code here.

```
1   d := 1; while (d≤N) do {a:=Sample(d); x[d]:=a; d:=d+1; }
2   flag := 0; cnt := 0; lst := [];
3   while (flag = 0) do
4       z := 0; h := 1;
5       while (h ≤ M) do
6           if (hold(h, x[1], ..., x[N])) then z := h;
7           h := h + 1;
8       if (z = 0) then flag := 1;
9       else
10          cnt := cnt + 1; lst := app(lst, z); d := 1;
11          while (d ≤ N) do
12              if (vbl(z, d)) then {a := Sample(d); x[d] := a; }
13              d := d + 1;
```

Fig. 12. The code of the MT algorithm, $C_{\mathsf{MT}}(cnt)$

Research lines of ALLLs. The MT algorithm is first proposed in [50], where the expected iteration number of the algorithm is bounded under the Erdős-Lovász condition [19, 57] and the Erdős-Spencer condition [20]. Following [50], some works [53, 42, 31, 1, 43, 37] further analyze the termination property and the iteration times of the MT algorithm under other conditions. Besides analyzing the iteration times of the MT algorithm, a number of works (including [50]) also analyze other sequential ALLLs [31, 34, 36, 29], explore properties of output distributions of ALLLs [31, 35, 29, 32], or design parallel and distributed ALLLs [50, 16, 30, 25, 13]. However, the proofs in all these works are relatively informal.

Existing results we verify. As listed below, we verify *six* representative results that cover the aforementioned research lines.

First, we verify the termination and the expected iteration times of the MT algorithm, under the Erdős-Lovász condition [19, 57], the cluster expansion condition [8], the Shearer's condition [56], and the Erdős-Spencer condition [20]. These four results are proposed and informally proved in Theorem 1.2 of [50], Theorem 1.4 of [53], Theorem 4 of [42] and Theorem 6.1 of [50].

Second, we verify (the second part of) Theorem 2.2 of [31] that estimates the output distribution of the MT algorithm under the Erdős-Lovász condition. This result can also be viewed as estimating the output distribution of a sequential ALLL that only executes on core events (see Theorem 3.3 of [31]).

Finally, we verify the termination and a tail bound of the iteration times of a parallelizable version of the MT algorithm, under the Erdős-Lovász condition with ϵ-slack. This variant and the tail bound are given in Theorem 1.3 of [50].

It is worth noting that we verify all the three "probabilistic" results from Moser and Tardos's Gödel Prize-winning paper [50].[4]

Verifying Theorem 1.2 of [50]. As an example, we explain in more detail how we verify Theorem 1.2 of [50], which we informally described in Sec. 2.

[4] In [50], Moser and Tardos propose four results, three related to the MT algorithm and its probabilistic variants, and one related to a deterministic variant.

Fig. 12 shows $C_{\mathsf{MT}}(cnt)$, the code of the MT algorithm that we verify. It first does independent samplings and stores the results in $x[1], \ldots, x[N]$ (line 1), where d and a are temporal variables. For the main loop (lines 3-13), we introduce *flag* to indicate whether a required assignment is found, *cnt* to record the number of iteration times, and *lst* to collect the indexes of the events in the execution log. They are initialized at line 2. In the main loop (lines 3-13), we use z to represent the index of the chosen event, which is an event that holds under the current $x[1], \ldots, x[N]$ (lines 4-7). If no such event exists, the code marks *flag* (line 8) and exits the loop (line 3). Otherwise, it resamples from $\mathcal{D}[d]$ for every d such that $\mathsf{vbl}(z, d)$ holds, and updates the corresponding $x[d]$ (lines 10-13).

Having defined the code of the MT algorithm, Moser and Tardos's result (Theorem 1.2 of [50]) is formally stated in Thm. 4. Note that $N, M, \mathcal{D}$ and $\mathcal{E}$ are global parameters and thus not fixed in Thm. 4, and r_{EL} is parametrized by M.

Theorem 4. *For all reals $\alpha_1, \ldots, \alpha_M \in (0, 1)$, if the Erdős-Lovász condition [19, 57] holds, i.e. $\forall i \in [1, M]$. $\mathrm{P}(\mathcal{E}[i]) \leq \alpha_i \prod_{j \in \Gamma(i)} (1 - \alpha_j)$, and let $r_{\mathsf{EL}} = \sum_{i \in [1,M]} \alpha_i (1 - \alpha_i)^{-1}$, then $\vDash [\mathbf{true}]\, C_{\mathsf{MT}}(cnt)\, [\mathbb{E}[cnt] \leq r_{\mathsf{EL}}]$.*

Proof Sketch. Our proof follows the path in Fig. 5. Due to the space limit, here we only explain our construction of $\mathbf{R}$, used in the two RT-triples at the bottom of Fig. 5. Let $\Lambda = g_{\mathsf{WT}}(wt)$. Then,

$$\mathbf{R} \triangleq \forall l \in [1, |\Lambda|].\ \forall V_1, \ldots V_N.\ \mathsf{RTAssign}(V_1, \ldots V_N, l, \Lambda) \Rightarrow \mathsf{hold}(\Lambda\langle l\rangle, V_1, \ldots, V_N),$$
$$\text{where } \mathsf{RTAssign}(V_1, \ldots, V_N, l, \Lambda) \triangleq \forall i \in [1, N].\ \mathsf{vbl}(\Lambda\langle l\rangle, i) \Rightarrow V_i = \mathsf{RT}[i][\mathsf{ve}(i, \Lambda, l-1)],$$
$$\mathsf{ve}(i, \Lambda, l) \triangleq \textstyle\sum_{l' \in [1,l]} [\mathsf{vbl}(\Lambda\langle l'\rangle, i)].$$

Informally $\mathbf{R}$ says that, every event in wt (denoted by $\Lambda\langle l\rangle$) must hold under any assignment of $V_1, \ldots, V_N$ satisfying $\mathsf{RTAssign}$. $\mathsf{RTAssign}$ says, the assignment contains the "relevant" entries of RT which make the event $\Lambda\langle l\rangle$ hold when it is chosen in the execution of $C'_{\mathsf{MT}}(cnt, K)$. For each such entry, its row number i corresponds to a variable that the event depends on (i.e. $\mathsf{vbl}(\Lambda\langle l\rangle, i)$ holds), and its column number is computed by $\mathsf{ve}(i, g_{\mathsf{WT}}(wt), l-1)$. Note our $\mathbf{R}$ only talks about the RT (and the wt), not about the actual execution of $C'_{\mathsf{MT}}(cnt, K)$.

We prove the remaining intermediate proof goals in Fig. 5 by adapting the program logic Ellora [3] (for proving $\vDash$ triples) and using a classical Hoare-style logic (for proving $\vDash_{\mathrm{RT}}$ triples). □

Our new result. Thm. 4 shows the MT algorithm's total correctness with r_{EL} as the upper bound of expected iteration times, under the Erdős-Lovász condition. There are many works [53, 42, 1, 43, 37] that informally study similar properties of the MT algorithm under other conditions. Most of these results use similar ideas with Moser and Tardos to analyze the algorithm, except that they introduce other witness-tree-like structures for analysis and derive various bounds. Like [50], they generate their witness-tree-like structures ds from prefixes of the execution log, enumerate the events in ds in some specific order, and bound a sum over all such structures to get their final upper bounds.

We unify these results to a general one. Our new result enables that, when proving the expected iteration number of the MT algorithm, without doing the complete proof following Moser and Tardos's idea, one only needs to instantiate the required witness-tree-like structures and prove some relevant mathematical side conditions. We show that Theorem 1.2 of [50], Theorem 1.4 of [53] and Theorem 4 of [42] are corollaries of our new result. We give details of our new result and proofs in [46].

7 Related Work

(Positive) almost sure termination. Existing proof methods for almost sure termination (AST) can be roughly classified into the following two categories: "direct" methods [48, 11, 12, 24, 49, 38, 47], which prove termination by constructing probabilistic ranking functions, and "indirect" methods [41, 52, 51, 40], which infer finite bounds on the expected runtime and then imply the termination.

However, these methods may not apply to ALLLs' termination. To construct the structures (e.g. ranking supermatingales [12, 24] and upper ω-invariants [41]) required by these methods, we need to understand what occurs during *each iteration* of the algorithm's outer loop, which is, however, not yet well understood. For example, [50] only analyzes the properties of the *entire* MT algorithm (e.g. (2)), not of each individual iteration.

In Sec. 2.3, we emphasize Lem. 1 as a general proof method for positive almost sure termination (PAST) [11]. Lem. 1 also serves as a *fallback plan* for proving (P-)AST. Informally, a part of existing methods [12, 24, 49, 41] provide stronger premises than Lem. 1's. These premises are easier to prove in most scenarios, except for ALLLs. For most programs, one can still apply these existing methods; for programs like ALLLs, one should take a step back and apply Lem. 1.

Asynchronous coupling. In Sec. 2.4, we apply the RT-based coupling proof recipe to (8), which involves $C'_{\mathsf{MT}}(cnt, K)$ and check(wt). Existing probabilistic relational program logics [4, 5, 6] support couplings, but none of them can prove (8). Specifically, these works only provide proof rules for *synchronous* couplings. Their rules say that, when the two programs sample from the same distribution synchronously, we can reason as if the two sampling statements return the same value. But, it may *not* be possible to synchronize the sampling statements in $C'_{\mathsf{MT}}(cnt, K)$ and check(wt) for the following reason. Given an execution log's prefix Λ and the corresponding witness tree $wt = f_{\mathsf{WT}}(\Lambda)$, $C'_{\mathsf{MT}}(cnt, K)$ resamples the variables that η_j depends on for every event η_j in Λ, and check(wt) does similar resamplings but its events are taken from the sequence $g_{\mathsf{WT}}(wt)$. However, $g_{\mathsf{WT}}(wt)$ can be different from Λ, since the construction of wt (i.e. $f_{\mathsf{WT}}(\Lambda)$) may drop some events in Λ and lose some ordering information of Λ, which $g_{\mathsf{WT}}(wt)$ cannot recover.

Recently [28] proposes a probabilistic relational program logic that supports *asynchronous* coupling. They introduce *presampling tapes*, a new kind of ghost state, which store the sampling results ahead of time. Our work is developed

independently, with a more focused goal of verifying ALLLs. Technically, our RTs look similar to their tapes, but there are two key differences as follows.

First, we give an RT-based operational semantics, where all the samples (which could be infinitely many) are generated at once and stored in the *RT before programs start execution*, and the *RT* is immutable during the program execution. By contrast, sample values are added into their tapes *one at a time* and *on demand* by ghost operations in the logical reasoning, and are popped out at sampling statements. We think their approach is more flexible, but ours is more suitable for complicated examples like ALLLs. In particular, as we explain at the end of Sec. 2.4, we can use an intermediate assertion $\mathbf{R}$ to specify *the whole sampling history*. $\mathbf{R}$ can be derived as the post-condition of the unary reasoning of one program, and then used as the pre-condition of the other, thanks to the immutability of *RT*. With dynamically changing tapes, they would need ghost variables to track the popped samples, and write complicated assertions to describe the correspondence between the tapes used by the two programs. We give a more detailed comparison in [46].

Second, the two works have different focuses. We mainly focus on verifying ALLLs, so we verify almost sure termination as well as a restricted form of relational properties (like (8)). Their work verifies contextual refinement, but does not verify termination.

Other related works. [22] proposes the *guard strengthening* proof rule for verifying lower bounds of expected values at the end of while loops. This rule introduces a loop with strengthened loop guard, which is similar to the truncated one in the premise of our loop truncation (Lem. 1 and Thm. 2). However, these two methods have different focuses. Their rule focuses on proving *lower bounds*, while our loop truncation focuses on proving general total correctness and PAST. The PAST is about an *upper bound* of the expected runtime.

We have discussed other related works in Sec. 2.2 and Sec. 2.4, including: the semantics that are equivalent to the distribution-based semantics [45, 48, 44], and the semantics with explicit random sources [44, 10, 17]. In the future, we would like to test our proof recipes with more applications, such as the other ALLL-related results mentioned in Sec. 6. We also plan to mechanize our work in a proof assistant like Coq, as [18] has mechanized the classical (i.e. non-constructive) proof of the Lovász Local Lemma in Isabelle/HOL. Mechanizing our work requires a measure-theoretic library that supports infinite product of measure spaces, which, to the best of our knowledge, is still lacking for Coq.

Acknowledgments. We thank anonymous referees for their suggestions and comments on earlier versions of this paper. This work is supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62232015.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Achlioptas, D., Gouleakis, T.: Algorithmic improvements of the Lovász local lemma via cluster expansion. In: FSTTCS 2012. pp. 16–23 (2012). https://doi.org/10.4230/LIPIcs.FSTTCS.2012.16
2. Anderson, E., Phillips, C., Sicker, D., Grunwald, D.: Optimization decomposition for scheduling and system configuration in wireless networks. IEEE/ACM Trans. Netw. **22**(1), 271–284 (2014). https://doi.org/10.1109/TNET.2013.2289980
3. Barthe, G., Espitau, T., Gaboardi, M., Grégoire, B., Hsu, J., Strub, P.Y.: An assertion-based program logic for probabilistic programs. In: ESOP 2018. pp. 117–144 (2018). https://doi.org/10.1007/978-3-319-89884-1_5
4. Barthe, G., Espitau, T., Grégoire, B., Hsu, J., Stefanesco, L., Strub, P.Y.: Relational reasoning via probabilistic coupling. In: LPAR 2015. p. 387–401 (2015). https://doi.org/10.1007/978-3-662-48899-7_27
5. Barthe, G., Gaboardi, M., Grégoire, B., Hsu, J., Strub, P.Y.: Proving differential privacy via probabilistic couplings. In: LICS 2016. p. 749–758 (2016). https://doi.org/10.1145/2933575.2934554
6. Barthe, G., Grégoire, B., Zanella Béguelin, S.: Formal certification of code-based cryptographic proofs. In: POPL 2009. p. 90–101 (2009). https://doi.org/10.1145/1480881.1480894
7. Batz, K., Kaminski, B.L., Katoen, J.P., Matheja, C.: Relatively complete verification of probabilistic programs: an expressive language for expectation-based reasoning. Proc. ACM Program. Lang. **5**(POPL), 1–30 (2021). https://doi.org/10.1145/3434320
8. Bissacot, R., Fernández, R., Procacci, A., Scoppola, B.: An improvement of the Lovász local lemma via cluster expansion. Comb. Probab. Comput. **20**(5), 709–719 (2011). https://doi.org/10.1017/S0963548311000253
9. Boissonnat, J., Dyer, R., Ghosh, A.: A probabilistic approach to reducing algebraic complexity of delaunay triangulations. In: ESA 2015. pp. 595–606 (2015). https://doi.org/10.1007/978-3-662-48350-3_50
10. Borgström, J., Dal Lago, U., Gordon, A.D., Szymczak, M.: A lambda-calculus foundation for universal probabilistic programming. In: ICFP 2016. pp. 33–46 (2016). https://doi.org/10.1145/2951913.2951942
11. Bournez, O., Garnier, F.: Proving positive almost-sure termination. In: RTA 2005. pp. 323–337 (2005). https://doi.org/10.1007/978-3-540-32033-3_24
12. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: CAV 2013. pp. 511–526 (2013). https://doi.org/10.1007/978-3-642-39799-8_34
13. Chang, Y.J., He, Q., Li, W., Pettie, S., Uitto, J.: Distributed edge coloring and a special case of the constructive Lovász local lemma. ACM Trans. Algorithms **16**(1), 8:1–8:51 (2020). https://doi.org/10.1145/3365004
14. Chen, A., Harris, D.G., Srinivasan, A.: Partial resampling to approximate covering integer programs. Random Struct. Algorithms **58**(1), 68–93 (2021). https://doi.org/10.1002/rsa.20964
15. Cheng, K., Haeupler, B., Li, X., Shahrasbi, A., Wu, K.: Synchronization strings: Highly efficient deterministic constructions over small alphabets. In: SODA 2019. pp. 2185–2204 (2019). https://doi.org/10.1137/1.9781611975482.132
16. Chung, K.M., Pettie, S., Su, H.H.: Distributed algorithms for the Lovász local lemma and graph coloring. In: PODC 2014. p. 134–143 (2014). https://doi.org/10.1145/2611462.2611465

17. Culpepper, R., Cobb, A.: Contextual equivalence for probabilistic programs with continuous random variables and scoring. In: ESOP 2017. pp. 368–392 (2017). https://doi.org/10.1007/978-3-662-54434-1_14
18. Edmonds, C., Paulson, L.C.: Formal probabilistic methods for combinatorial structures using the Lovász local lemma. In: CPP 2024. pp. 132–146 (2024). https://doi.org/10.1145/3636501.3636946
19. Erdős, P., Lovász, L.: Problems and results on 3-chromatic hypergraphs and some related questions. Infinite and finite sets **10**(2), 609–627 (1975)
20. Erdős, P., Spencer, J.: Lopsided Lovász local lemma and latin transversals. Discrete Applied Mathematics **30**(151-154), 10–1016 (1991). https://doi.org/10.1016/0166-218X(91)90040-4
21. Fan, W., Liang, H., Feng, X., Jiang, H.: A program logic for concurrent randomized programs in the oblivious adversary model. to appear in ESOP 2025
22. Feng, S., Chen, M., Su, H., Kaminski, B.L., Katoen, J., Zhan, N.: Lower bounds for possibly divergent probabilistic programs. Proc. ACM Program. Lang. **7**(OOPSLA1), 696–726 (2023). https://doi.org/10.1145/3586051
23. Fernández, M., Livieratos, J., Martín, S.: Bounds and constructions of parent identifying schemes via the algorithmic version of the Lovász local lemma. IEEE Trans. Inf. Theory **69**(11), 7049–7069 (2023). https://doi.org/10.1109/TIT.2023.3282452
24. Ferrer Fioriti, L.M., Hermanns, H.: Probabilistic termination: Soundness, completeness, and compositionality. In: POPL 2015. p. 489–501 (2015). https://doi.org/10.1145/2676726.2677001
25. Fischer, M., Ghaffari, M.: Sublogarithmic distributed algorithms for Lovász local lemma, and the complexity hierarchy. In: DISC 2017. pp. 18:1–18:16 (2017). https://doi.org/10.4230/LIPIcs.DISC.2017.18
26. Gebauer, H., Szabó, T., Tardos, G.: The local lemma is asymptotically tight for SAT. J. ACM **63**(5), 43:1–43:32 (2016). https://doi.org/10.1145/2975386
27. Graf, A., Harris, D.G., Haxell, P.: Algorithms for weighted independent transversals and strong colouring. ACM Trans. Algorithms **18**(1), 1:1–1:16 (2022). https://doi.org/10.1145/3474057
28. Gregersen, S.O., Aguirre, A., Haselwarter, P.G., Tassarotti, J., Birkedal, L.: Asynchronous probabilistic couplings in higher-order separation logic. Proc. ACM Program. Lang. **8**(POPL), 753–784 (2024). https://doi.org/10.1145/3632868
29. Guo, H., Jerrum, M., Liu, J.: Uniform sampling through the Lovász local lemma. J. ACM **66**(3), 18:1–18:31 (2019). https://doi.org/10.1145/3310131
30. Haeupler, B., Harris, D.G.: Parallel algorithms and concentration bounds for the Lovász local lemma via witness dags. ACM Trans. Algorithms **13**(4), 53:1–53:25 (2017). https://doi.org/10.1145/3147211
31. Haeupler, B., Saha, B., Srinivasan, A.: New constructive aspects of the Lovász local lemma. J. ACM **58**(6), 28:1–28:28 (2011). https://doi.org/10.1145/2049697.2049702
32. Harris, D.G.: New bounds for the Moser-Tardos distribution. Random Struct. Algorithms **57**(1), 97–131 (2020). https://doi.org/10.1002/rsa.20914
33. Harris, D.G., Srinivasan, A.: Constraint satisfaction, packet routing, and the Lovász local lemma. In: STOC 13. p. 685–694 (2013). https://doi.org/10.1145/2488608.2488696
34. Harris, D.G., Srinivasan, A.: A constructive algorithm for the Lovász local lemma on permutations. In: SODA 2014. pp. 907–925 (2014). https://doi.org/10.1137/1.9781611973402.68

35. Harris, D.G., Srinivasan, A.: Algorithmic and enumerative aspects of the Moser-Tardos distribution. ACM Trans. Algorithms **13**(3), 33:1–33:40 (2017). https://doi.org/10.1145/3039869
36. Harris, D.G., Srinivasan, A.: The Moser-Tardos framework with partial resampling. J. ACM **66**(5), 36:1–36:45 (2019). https://doi.org/10.1145/3342222
37. He, K., Li, Q., Sun, X.: Moser-Tardos algorithm: Beyond Shearer's bound. In: SODA 2023. pp. 3362–3387 (2023). https://doi.org/10.1137/1.9781611977554.CH129
38. Huang, M., Fu, H., Chatterjee, K., Goharshady, A.K.: Modular verification for almost-sure termination of probabilistic programs. Proc. ACM Program. Lang. **3**(OOPSLA), 129:1–129:29 (2019). https://doi.org/10.1145/3360555
39. Jiang, N., Gu, Y., Xue, Y.: Learning Markov random fields for combinatorial structures via sampling through Lovász local lemma. In: AAAI 2023. pp. 4016–4024 (2023). https://doi.org/10.1609/AAAI.V37I4.25516
40. Kaminski, B.L.: Advanced weakest precondition calculi for probabilistic programs. Ph.D. thesis, RWTH Aachen University, Germany (2019). https://doi.org/10.18154/RWTH-2019-01829
41. Kaminski, B.L., Katoen, J., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected run-times of probabilistic programs. In: ESOP 2016. pp. 364–389 (2016). https://doi.org/10.1007/978-3-662-49498-1_15
42. Kolipaka, K.B.R., Szegedy, M.: Moser and Tardos meet Lovász. In: STOC 2011. p. 235–244 (2011). https://doi.org/10.1145/1993636.1993669
43. Kolipaka, K.B.R., Szegedy, M., Xu, Y.: A sharper local lemma with improved applications. In: APPROX-RANDOM 2012. pp. 603–614 (2012). https://doi.org/10.1007/978-3-642-32512-0_51
44. Kozen, D.: Semantics of probabilistic programs. J. Comput. Syst. Sci. **22**(3), 328–350 (1981). https://doi.org/10.1016/0022-0000(81)90036-2
45. Kozen, D.: A probabilistic PDL. J. Comput. Syst. Sci. **30**(2), 162–178 (1985). https://doi.org/10.1016/0022-0000(85)90012-1
46. Lin, R., Liang, H., Feng, X.: Verifying algorithmic versions of the Lovász local lemma – technical report. https://plax-lab.github.io/publications/alll/alll-tr.pdf (2024)
47. Majumdar, R., Sathiyanarayana, V.R.: Positive almost-sure termination: Complexity and proof rules. Proc. ACM Program. Lang. **8**(POPL), 1089–1117 (2024). https://doi.org/10.1145/3632879
48. McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Springer (2005). https://doi.org/10.1007/B138392
49. McIver, A., Morgan, C., Kaminski, B.L., Katoen, J.: A new proof rule for almost-sure termination. Proc. ACM Program. Lang. **2**(POPL), 33:1–33:28 (2018). https://doi.org/10.1145/3158121
50. Moser, R.A., Tardos, G.: A constructive proof of the general Lovász local lemma. J. ACM **57**(2), 11:1–11:15 (2010). https://doi.org/10.1145/1667053.1667060
51. Ngo, V.C., Carbonneaux, Q., Hoffmann, J.: Bounded expectations: Resource analysis for probabilistic programs. In: PLDI 2018. p. 496–512 (2018). https://doi.org/10.1145/3192366.3192394
52. Olmedo, F., Kaminski, B.L., Katoen, J.P., Matheja, C.: Reasoning about recursive probabilistic programs. In: LICS 2016. p. 672–681 (2016). https://doi.org/10.1145/2933575.2935317
53. Pegden, W.: An extension of the Moser-Tardos algorithmic local lemma. SIAM J. Discret. Math. **28**(2), 911–917 (2014). https://doi.org/10.1137/110828290

54. Saeki, S.: A proof of the existence of infinite product probability measures. The American Mathematical Monthly **103**(8), 682–683 (1996). https://doi.org/10.1080/00029890.1996.12004804
55. Sarkar, K., Colbourn, C.J., Bonis, A.D., Vaccaro, U.: Partial covering arrays: Algorithms and asymptotics. Theory Comput. Syst. **62**(6), 1470–1489 (2018). https://doi.org/10.1007/S00224-017-9782-9
56. Shearer, J.B.: On a problem of Spencer. Combinatorica **5**, 241–245 (1985). https://doi.org/10.1007/BF02579368
57. Spencer, J.: Asymptotic lower bounds for Ramsey functions. Discrete Mathematics **20**, 69–76 (1977). https://doi.org/10.1016/0012-365X(77)90044-9
58. Srinivasan, A.: Progress on algorithmic versions of the Lovász local lemma. https://www.ias.edu/sites/default/files/video/Aravind.pdf (2013)