Verified Tail Bounds for Randomized Programs

JOSEPH TASSAROTTI,  Carnegie Mellon University
ROBERT HARPER,  Carnegie Mellon University

The development of type systems and static analyses that automatically bound the complexity of programs is an active area of research. Yet because there are inevitably programs outside the scope of automated techniques, a variety of interactive formal methods have been developed for establishing resource bounds. A particularly difficult and important case is the complexity of randomized programs. Prior work has mostly focused on expected run time bounds. But, these programs are often known to offer much stronger performance guarantees: not only do they run efficiently on average, but the probability that they deviate substantially from their average case behavior is very small. These tail bounds are particularly important when it comes to the analysis of parallel algorithms or space usage. Despite their importance, tight tail bounds on performance are usually quite difficult to derive.

We mechanize a theorem by Karp, along with several extensions, that provide an easy to use “cookbook” method for verifying tail bounds, much like the traditional “Master Theorem” gives bounds for deterministic algorithms. We apply these results to several examples: the number of comparisons performed by QuickSort, the span of parallel QuickSort, the height of randomly generated binary search trees, and the number of rounds needed for a distributed leader election protocol. Because the constants involved in our symbolic bounds are concrete, we are able to use them to derive numerical probability bounds for various input sizes for these examples. Our work complements a growing body of semi-automatic and interactive techniques that have been developed for cases where automatic analyses of complexity are not yet feasible.

CCS Concepts:
• General and reference → Verification;
• Mathematics of computing → Probabilistic algorithms;

Additional Key Words and Phrases: Probabilistic Recurrences, Analysis of Parallel Algorithms

ACM Reference format:
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A long-standing goal of research in programming languages is to develop tools that ensure the quality and correctness of programs. Beyond knowing that programs produce correct output, we also want to know that they do so efficiently. To that end, a number of researchers have developed type systems and static analyses that measure and bound the resource usage of programs [Albert et al. 2011; Černý et al. 2013; Çiçek et al. 2017; Gulwani et al. 2009; Hoffmann et al. 2012, 2017; Knoop et al. 2011]. Despite enormous progress in this area, there are inevitably programs that fall outside the scope of these tools. When this happens, it is useful to have alternative interactive techniques that can be used to help verify bounds on program behavior [Atkey 2011; Danner et al. 2015]. Even though these interactive techniques are not fully automatic, we would still like to minimize the tedium and work involved in using them.

It is particularly difficult to automatically analyze the resource usage of randomized algorithms. Of course, part of the reason for using randomness is to improve performance, so it is especially important to be able to formally analyze the resource usage of such programs. Hence, in addition
to the development of tools for automatically analyzing some of these programs [Chakarov and Sankaranarayanan 2013; Chatterjee et al. 2017a,b], there is also an extensive line of work developing program logics for manual verification in important cases where automation does not apply. Some of these logics [Barthe et al. 2009, 2017] are focused on relational reasoning, which can be used to show one program is equivalent to some simpler program. Then, this simpler program can be analyzed further to draw conclusions about the behavior of the original program\(^1\). In other cases [Kaminski et al. 2016], the logic is used to establish a bound on the expected resource usage of the program. Finally, in some prior work [Eberl 2017b; van der Weegen and McKinna 2008] expected run time bounds have been established directly in a theorem prover, without the use of an intermediate program logic.

Beyond expectation bounds. Although expectation bounds are an important first step in the analysis of these algorithms, there are other stronger properties that often hold. For many randomized algorithms, we can establish tail bounds which bound the probability that the algorithm takes more than a given amount of time.

For example, it is well known that QuickSort performs \(O(n \log n)\) comparisons on average when sorting a list of length \(n\), and this fact has been verified in theorem provers before [Eberl 2017b; van der Weegen and McKinna 2008]. However, not only does it do \(O(n \log n)\) comparisons on average, but the probability that it takes more than \(O(n \log n)\) comparisons is vanishingly small for sufficiently large lists. To be precise, let \(W_n\) be the number of comparisons when sorting a list of length \(n\). Then, for any positive \(k\) there exists \(c_k\) such that

\[
\Pr [W_n > c_k n \log n] < \frac{1}{n^k}
\]

When we say that such \(c_k\) exist, we mean so in a constructive and practical sense: we can actually determine them and they are not absurdly large, so that one can derive interesting concrete bounds. For instance, when \(n = 10\) million, the probability that \(W_n\) is greater than \(8n \log_2 n\) is less than \(10^{-9}\).

These kinds of tail bounds hold for many other classical randomized algorithms, and are often stronger than asymptotic expectation bounds. In addition, these bounds are sometimes compositional in situations where expectation bounds are not. For example, in the analysis of parallel algorithms, an important cost measure is span, which is the longest sequential dependency in the computation. The span affects the degree to which adding more processors speeds up a parallel algorithm [Blelloch and Greiner 1995; Blumofe and Leiserson 1999]. Given two programs \(e_1\) and \(e_2\) with spans \(S_1\) and \(S_2\), the span of their parallel composition is \(c + \max(S_1, S_2)\), where \(c\) is some constant overhead. Now, if \(e_1\) and \(e_2\) are randomized, so that \(S_1\) and \(S_2\) are random variables, we would like to be able to combine bounds on \(S_1\) and \(S_2\) to get a bound on the span of their composition. However, with expectation bounds alone, we cannot because:

\[
E [c + \max(S_1, S_2)] \geq c + \max (E [S_1], E [S_2])
\]

To bound the expected span of the parallel composition, we would need an inequality in the opposite direction. In contrast, with the tail bound, if \(S_1\) and \(S_2\) are independent random variables, then

\[
\Pr [c + \max(S_1, S_2) > k] = 1 - (1 - \Pr [S_1 > k - c]) (1 - \Pr [S_2 > k - c])
\]

Similar problems with expectation bounds arise when analyzing the stack space usage of sequential programs and the heights of search structures like binary trees.

Despite this, there is a good reason for the stronger emphasis on expectation bounds rather than tail bounds in the field of formal methods: tight tail bounds on running time are usually

\(^1\)In some applications, such as cryptography and privacy, the equivalence between two programs is the essential property, and the second program does not need to be analyzed further.
Quite difficult to derive. One approach to obtaining these bounds is to find the expected value and the variance and then use Chebyshev’s inequality. As we have already pointed out, finding the expected value alone can be difficult, but the variance is often even harder. For example, Iliopoulos and Penman [2010] have given a careful pencil-and-paper derivation of QuickSort’s variance that would be similar to the kind of detail needed for mechanization, and their derivation takes 16 pages. Moreover, much of their argument consists of algebraic manipulations which are often difficult to do in existing interactive theorem provers. Other approaches involve the use of methods from analytic combinatorics [Flajolet and Sedgewick 2009] or the theory of concentration of measure [Dubhashi and Panconesi 2009]. Although these techniques are very effective, to be able to use them in a theorem prover one would first need to be able to mechanize the extensive body of results that they depend upon.

The need for “cookbook” methods. Let us contrast the difficulty described above with the (relative) ease of analyzing deterministic algorithms. For deterministic divide-and-conquer algorithms, the cost is often given by recurrences of the form

\[ W(x) = a(x) + \sum_{i=1}^{n} W(h_i(x)) \]

where the “toll” function \( a(x) \) represents the cost to process an input and divide it into subproblems of size \( h_1(x), \ldots, h_n(x) \) which are then solved recursively. Every undergraduate algorithms course covers “cookbook” techniques such as the Master Theorem [Bentley et al. 1980; Cormen et al. 2009] that can be used to straightforwardly derive asymptotic bounds on these kinds of recurrences. Moreover, these results can also be used to easily analyze the span of parallel divide-and-conquer algorithms, because in the deterministic case, recurrences of the form:

\[ S(x) = b(x) + \max_{i=1}^{n} S(h_i(x)) \]

reduce to \( S(x) = b(x) + S(\max_{i=1}^{n} (h_i(x))) \), because \( S \) is usually monotone.

What is nice about these methods is that they give a process for carrying out the analysis: find the toll function, bound the size of recursive problems, and then use the theorem. Even if the first two steps might require some ingenuity, the method at least suggests an approach to decomposing the problem.

Besides being easy to use, results like the Master Theorem do not have many mathematical prerequisites. This makes them ideal for use in interactive theorem provers. Indeed, Eberl [2017c] has recently mechanized the more advanced Akra and Bazzi [1998] recurrence theorem in Isabelle and has used it to verify the solution to a number of recurrence relations.

For randomized divide-and-conquer algorithms, the same recurrence relations arise, except the \( h_i(x) \) are random variables because the algorithms use randomness to divide the input into subproblems. Because of the similarity between deterministic and probabilistic recurrences, textbook authors sometimes give the following heuristic argument before presenting a formal analysis [Cormen et al. 2009, p. 175–177]: In an algorithm like QuickSort, the size of the sublists generated by the partitioning step can be extremely unbalanced in the worst case, but this happens very rarely. In fact, each sublist is unlikely to be much more than \( \frac{3}{4} \) the length of the original list. And, for a deterministic recurrence like \( W(n) = n + W(\frac{3}{4}n) + W(\frac{3}{4}n) \), the master theorem says the result will be \( O(n \log n) \). Thus, intuitively, we should expect the average running time of Quicksort to be something like \( O(n \log n) \).

This raises a natural question: Is there a variant of the Master Theorem that can be used to justify this kind of heuristic argument? Moreover, because Equation 2 does not simplify to a version
of Equation 1 in the randomized setting, we ideally want something that can be used to analyze recurrences of both forms.

For the case where there is only a single recursive call (so that \( n = 1 \) above), Karp [1994] developed such a result. At a high-level, using Karp’s theorem involves two steps. First, bound the average size of the recursive subproblem by finding a function \( m \) such that \( E[h_1(x)] \leq m(x) \). Next, find a solution \( u \) to the deterministic recurrence relation

\[
u(x) \geq a(x) + u(m(x))
\]

Then the theorem says that for all positive integers \( w \),

\[
\Pr[W(x) > u(x) + wa(x)] \leq \left(\frac{m(x)}{x}\right)^w
\]

There are a few side conditions on the functions \( m \) and \( u \) which are usually easy to check. Although this method usually does not give the tightest possible bounds, they are often rather strong. Recently, Karp’s technique has been extended [Tassarotti 2017] to the more general case for \( n > 1 \) for both span and work recurrences.

Our Contribution. In this paper, we present a mechanization of Karp’s theorem and these extensions in Coq, and use it to develop verified tail bounds for (1) the number of comparisons in sequential QuickSort, (2) the span arising from comparisons in parallel QuickSort, (3) the height of a randomly generated binary search tree, and (4) the number of rounds needed in a distributed randomized leader election protocol. By using the Coq-Interval library [Martin-Dorel and Melquiond 2016] we are able to instantiate our bounds in Coq to establish numerical results such as the \( 10^{-9} \) probability bound for QuickSort quoted above. To our knowledge, this is the first time these kinds of bounds have been mechanized.

This work contributes to the on-going research in verified cost analysis by giving new techniques for the difficult case of randomized parallel and sequential algorithms. This complements prior work on automatic cost analysis by addressing cases that are, at present, beyond the scope of automatic techniques. We add to the body of semi-automatic and interactive tools that have been developed for cost analysis (e.g., [Atkey 2011; Carbonneaux et al. 2015; Danner et al. 2015], among others) and probabilistic verification (e.g., [Barthe et al. 2017, 2016; Kaminski et al. 2016; Morgan et al. 1996]). In particular, we consider high probability performance bounds, an important property beyond the feasible scope of existing techniques.

Our development is an instance of the kind of result called for by Gaboardi and Hsu [2015]: tools that make it possible to verify “Theory A”-style analysis of algorithms in a manner similar to the paper proofs and informal heuristics used in that community. In this respect, we believe that the right lemma (and mechanization) can be just as important as sophisticated automation techniques and program logics when it comes to making formal methods feasible.

We start by outlining the mechanization of probability theory that our work is based on (§2). Next, we discuss how we represent randomized algorithms in Coq in a monadic style and the reasoning techniques used to extract probabilistic recurrence relations from these encodings (§3). We then describe Karp’s theorem and its extensions in more detail (§4). To demonstrate how Karp’s result is used, we describe our verification of the four examples mentioned above (§5). Of course, formalization often requires changing parts of a paper proof, and our experience with Karp’s theorem was no different. We discuss the issues we encountered and what we had to change in §6. Finally, we compare our approach to related work (§7) and conclude by discussing possible extensions and improvements to our development (§8).
**The supplementary material contains our Coq development, as well as a hyperlinked outline that connects each statement in this paper to its occurrence in the mechanization.**

## 2 PROBABILITY PRELIMINARIES

We first need a set of basic results and definitions about probabilities and expectations to be able to even state Karp’s theorem. We had to decide whether to use a measure-theoretic formulation or restrict ourselves to discrete distributions. Although the Isabelle standard library has an extensive formalization of measure theoretic probability, we are not aware of a similarly complete set of results in Coq (we discuss existing libraries later in §7). Moreover, the applications we had in mind only involved discrete distributions, so we did not need the extra generality of the measure-theoretic approach. To keep things simple, we decided to develop a small library for discrete probability theory. Defining probability and expectation for discrete distributions still involves infinite series over countable sets, which can raise some subtle issues involving convergence. We use the Coquelicot real analysis library [Boldo et al. 2015] to deal with infinite series.

The definition of probability distributions is given in Figure 1. We represent them as a record type parameterized by a countable type. We use the ssreflect [Gonthier et al. 2016] library’s definition of countable types (countType), which consists of a type \( A \) equipped with a surjection from \( \mathbb{N} \) to \( A \). The distribution record consists of three fields: (1) a probability mass function \( \text{pmf} : A \rightarrow \mathbb{R} \) that assigns a probability to each element of \( A \), (2) a proof that \( \text{pmf} \ a \) is non-negative for all \( a \), and (3) a proof that the countable series that sums \( \text{pmf} \ a \) over all \( a \) converges and is equal to 1. The Coq coercion mechanism is used to implicitly coerce distributions into their \( \text{pmf} \) field. Events on a distribution are boolean valued predicates on the underlying type, and the probability of an event \( P \) with respect to a distribution \( d \) is the infinite sum of \( d \ a \) for all \( a \) satisfying \( P \).

Random variables on a distribution \( \text{rvar} \) are functions from the underlying countable space \( A \) to some other type \( B \). The expected value of a real-valued random variable is defined in the usual way as the infinite sum:

\[
\sum_{r \in \text{img}(X)} \Pr[X = r] \cdot r
\]

Because the underlying distribution is discrete, the image of the random variable is a countable set, so we can define such a sum. Of course, expectations of discrete random variables do not always exist, because the above series may not converge absolutely. In the Coquelicot library, \( \text{Series} \) is a total function, but its value is not the actual limit of the partial sums if the corresponding series does not converge. We separately define a predicate (omitted) that expresses the existence of the expectation.

Dealing with infinite series and issues of convergence can often be tedious. In actuality, many randomized algorithms only involve finite distributions, because they terminate in all cases, not merely with probability 1. For random variables defined on such distributions, the expectation always exists, because the series is actually just a finite sum. Therefore, for our mechanization of Karp’s theorem, we restrict our attention to these finite distributions. When the distribution is finite, we use the following lemma to convert from a Coquelicot Series into ssreflect’s \( \text{bigop} \) summation operator:

**Lemma** SeriesC_fin_big \( \{A : \text{finType}\} (a : A \rightarrow \mathbb{R}) : \\
\text{Series} (\text{countable_sum} a) = \text{\textbackslash big}[\text{Rplus}/0]_{(i \in A)} (a \ i) \).

This lets us use the considerable library developed in ssreflect for working with such sums, as in prior work that developed finite probability theory in Coq [Affeldt and Hagiwara 2012].
Record distrib (A: countType) := mkDistrib {
  pmf :> A -> R;
  pmf_pos : forall a, pmf a >= 0;
  pmf_sum1 : is_series (countable_sum pmf) 1
}.

Definition pr {A: countType} (d: distrib A) (P: A -> bool) :=
  Series (countable_sum (fun a => if P a then d a else 0)).

Record rvar (A: countType) (B: eqType) := mkRvar {
  rvar_dist : distrib A;
  rvar_fun :> A -> B;
}.

Definition pr_eq {A} {B: eqType} (X: rvar A B) (b: B) :=
  pr (rvar_dist X) (fun a => X a == b).

Definition Ex {A} (X: rrvar A) : Rbar :=
  Series (countable_sum (fun r => (pr_eq X r * r))).

Fig. 1. Basic definitions for discrete probability distributions and random variables.

3 MONADIC ENCODING

We represent sequential and parallel randomized algorithms in Coq using a monadic embedding.
Variants of this kind of representation have been used in many prior formalizations and domain
specific languages [Audebaud and Paulin-Mohring 2009; Barthe et al. 2009; Petcher and Morrisett
2015; Ramsey and Pfeffer 2002] and our development is heavily based on the ideas developed in this
prior work. In this section, we briefly describe some of the choices we made for our representation
and how we reason about programs expressed in this embedding.

3.1 Probabilistic Choice Monad

The definition of the basic probabilistic monad is given in Figure 2. The type $1\text{dist }A$ represents
probabilistic computations that result in a value of type $A$. The type is a Coq record type consisting
of (1) a list of pairs of real numbers and values of type $A$ (outcomes), (2) a proof that all the real
numbers in the list are non-negative (nonneg), and (3) a proof that the sum of the first components
of the list equals 1 (sum1). The coercion mechanism implicitly coerces an $1\text{dist }A$ into its list of
outcomes. The types of the nonneg and sum1 fields are set up so that they have a unique inhabitant:
any two proofs of these properties are equal. This ensures that two terms of type $1\text{dist }A$ will be
equal exactly when their outcome fields are equal.

The bind operation ($\text{dist }_\text{bind}$) is similar to the way bind is defined for the list non-determinism
monad: For each element $(r, a)$ in the list of outcomes for some $1\text{dist }A$, we apply the function
$f : A \rightarrow 1\text{dist }B$ to $a$, which returns a list of pairs of reals and values of type $B$. We then scale
the first component of elements of this list by $r$. Finally, all the results are appended together. This
represents the process of performing the computation represented by 1 to obtain a random element
of type $A$ (i.e., "sampling" from the distribution represented by 1), and then passing this to $f$. The
return operation ($\text{dist }_\text{ret}$) applied to a corresponds to the probabilistic computation that simply
Record mkDist (A: Type) := mkDist { 
  outcomes :> list (R * A);
  nonneg : all (fun r => Rle_dec 0 r) (map fst outcomes);
  sum1 : \big[Rplus/0\]_(a <- map fst outcomes) a == 1 
}. 

Fixpoint dist_bind {A B} (f: A -> ldist B) (l: ldist A) := 
  match l with
  | [::] => [::]
  | (r, x) :: l => map (fun py => (r * py.1, py.2)) (f x) ++ dist_bind f l
  end.

Definition dist_ret {A} (x: A) := [::(1, x)].

Fig. 2. Monadic representation of probability distributions as lists of pairs of probabilities and values. The definitions of bind and return shown here are just the parts defining the outcomes field of the resulting record, without the proofs that the results sum to 1 and are all non-negative.

returns a with probability 1. We use Coq’s notation mechanism to represent binding m in e by writing x ← m; e, and write mret a for returning a.

We convert probabilistic computations represented by terms of type ldist A into random variables (rvar) in which the probability of a is equal to:

\big[Rplus/0\]_(a <- outcomes l | snd a == r) fst a.

which sums the first component of each pair of the form (r, a) occurring in the outcomes of l. The corresponding distribution is finite because the list of outcomes is finite.

3.2 Reasoning about the representation

In the introduction, we sketched the “cookbook” method enabled by Karp’s theorem (which we describe more precisely in §4). That method requires three different kinds of reasoning about programs encoded in the monadic representation:

Equivalence to probabilistic recurrence. We first need to show the program’s costs match the format of a divide-and-conquer probabilistic recurrence. In some situations, the structure of the program almost exactly matches the form of the recurrence relation, while in other cases we need to prove it is equivalent to a computation that does satisfy the recurrence relation.

To establish these equivalences, we use various rules for equational reasoning about the probabilistic monad. A selection of these rules, which are fairly standard, is given in Figure 3. Sometimes these rules give us exact equalities, but in other cases we only get that the random variables are equal in distribution, which is defined as:

\text{Definition } eq_dist \ {A B C} \ {X: rvar A C} \ {Y: rvar B C} := \forall x, pr_eq X x = pr_eq Y x.

That is, for each x, the two random variables take the value x with equal probability. This relation is a congruence: it is an equivalence relation respected by the monadic operations.

\text{In traditional presentations of probability theory, two real-valued random variables } X \text{ and } Y \text{ are often defined to be equal in distribution if } \Pr \{X > x\} = \Pr \{Y > x\} \text{ for all } x. \text{ In the discrete case, this is equivalent to the definition we give, and we have mechanized a proof of this equivalence. For continuous distributions they are not the same.}
Lemma ldist_bind_ext \{A B: eqType\} m (f g: A -> ldist B):
(forall a, f a = g a) ->
(x <- m; f x) = (x <- m; g x).

Lemma ldist_bind_swap m1 m2 f:
\text{eq_dist} (\text{rvar_of_ldist} (a <- m1; b <- m2; f a b))
(\text{rvar_of_ldist} (b <- m2; a <- m1; f a b)).

Fig. 3. Selection of equational reasoning rules

Lemma Ex_mbind_ldist1 \{A\} (h: ldist A) (f: A -> ldist R) :
Ex (rvar_of_ldist (x <- h; f x)) =
\big[Rplus/\emptyset\big]_a (\text{pr_eq} (\text{rvar_of_ldist} h) a * Ex (rvar_of_ldist (f a))).

Lemma Ex_mret_simpl (x: R): Ex (rvar_of_ldist (mret x)) = x.

Fig. 4. Rules for calculating expectations.

Because we are interested in deriving tail bounds, the \emph{stochastic dominance} relation on random variables is also useful in our development. Given two real-valued random variables, \(X\) and \(Y\), we say that \(Y\) stochastically dominates \(X\) if:

\[ \forall r \in \mathbb{R}, \Pr [X > r] \leq \Pr [Y > r] \]

If \(X\) is stochastically dominated by \(Y\) then we can get tail bounds on \(X\) by deriving tail bounds for \(Y\).

\textbf{Bounding the expected size of recursive subproblems.} Once we have shown that the program obeys a probabilistic recurrence, we need to come up with a bound on the expected size of the recursive subproblems. We use the rules in Figure 4 to transform the expectations of monadic representations into various sums. The rule \texttt{Ex_mbind_ldist1} says that the expectation of sampling from \(h\) and then passing the result to \(f\) is equal to summing the expectation of \(f\) \(a\) times the probability that \(h\) takes the value \(a\) over all values of \(a\). Meanwhile, \texttt{Ex_mret_simpl} just says that the expected value of a computation that always returns \(x\) is simply \(x\).

We make heavy use of the \texttt{ssreflect bigop} lemmas to manipulate and bound the sums that result from applying these rewrite rules.

\textbf{Functional correctness.} To reduce the possibility that we have made a mistake in encoding the algorithm in the monadic style, we want to prove the correctness of the various examples (e.g. that QuickSort actually sorts the input). Also, to be able to carry out some of the previous two reasoning steps, we often need to show that the program maintains certain invariants. To do so, we have a small set of Hoare-style post-condition rules for proving facts about the output of programs. The basic rules are shown in Figure 5. The specification \texttt{mspec m P} says that all of the outputs of the randomized computation \(m\) satisfy the predicate \(P\). The rule \texttt{mspec_mret} says that the computation always returning \(x\) satisfies the post-condition \(P\) if \(P\ x\) holds. The \texttt{mspec_conseq} rule lets us weaken a post-condition. Finally, \texttt{mspec_mbind} is the analogue of the Hoare sequencing rule for combining two computations.

Verified Tail Bounds for Randomized Programs

\begin{verbatim}
Definition mspec {A: eqType} (m: ldist A) (P: A -> Prop) :=
  forall y, y \in (map snd (outcomes m)) -> P y.

Lemma mspec_mret {A: eqType} (x: A) (P: A -> Prop):
  P x -> mspec (mret x) P.

Lemma mspec_conseq {A: eqType} (m: ldist A) (P Q: A -> Prop):
  mspec m P -> (forall a, P a -> Q a) -> mspec m Q.

Lemma mspec_mbind {A B: eqType} (f: A -> ldist B) m (P: A -> Prop) (Q: B -> Prop):
  mspec m P -> (forall a, P a -> mspec (f a) Q) -> mspec (mbind f m) Q.
\end{verbatim}

Fig. 5. Selection of Hoare-style post-condition rules.

4 KARP’S THEOREM

Now that we have a formalization of the basic concepts of probability theory and a way to describe randomized algorithms in Coq, we can give a more careful explanation of Karp’s theorem and its extensions.

4.1 Unary Recurrences

The setting for Karp’s theorem is more general than the informal account we gave in the introduction. Specifically, he assumes that there is a set \( I \) of algorithm inputs, a function \( \text{size}: I \to \mathbb{R}^{\geq 0} \) such that \( \text{size}(z) \) is the “size” of input \( z \), and a family of random variables \( h(z) \) which correspond to the new problem that is passed to the recursive call of the algorithm. The random variable \( W(z) \), which represents the cost of the algorithm when run on input \( z \), is assumed to obey the following unary recurrence:

\[
W(z) = a(\text{size}(z)) + W(h(z))
\]  

(3)

Although the intent of this recurrence is clear, it requires some care to interpret: on the right hand side, \( h(z) \) is a random variable, but it is given as an argument to \( W \), which technically has \( I \) as a domain, not \( I \)-valued random variables. Instead, we should read this not as the composition \( W \circ h \) applied to \( z \), but rather as a specification for the process which first generates a random problem according to \( h(z) \) and then passes it to \( W \). In other words, this part of the recurrence is really describing a monadic process of the form:

\[ z' \leftarrow h(z); W(z') \]

Already, Equation 3 addresses a detail that is often glossed over in informal treatments of randomized algorithms. In informal accounts, one often speaks about a random variable \( W(n) \), which is meant to correspond to the number of steps taken by an algorithm when processing an instance of size \( n \). The issue is that usually, the exact distribution depends not just on the size of the problem but also the particular instance, so it is somewhat sloppy to regard \( W(n) \) as a random variable. For instance, when randomized QuickSort is run on a list containing duplicate elements, a (good) implementation will generally perform fewer total comparisons. Even if one tries to avoid this issue by, say, restricting only to lists that do not contain duplicates, one would still need to prove that the distribution depends on the size of the list alone. Admittedly, this is mostly harmless in informal treatments, but it is a detail that would otherwise have to be dealt with in a theorem prover.
We assume there is some constant $d$ that is the “cut-off” point for the recurrence: when the input’s size drops below $d$ no further recursive calls are made. The function $a : \mathbb{R} \rightarrow \mathbb{R}^+$ is required to be continuous and increasing on $(d, \infty)$, but equal to $0$ on the interval $[0, d]$. In addition, it is required that $0 \leq \text{size}(h(z)) \leq \text{size}(z)$, i.e., the size of the subproblem is not bigger than the original.

Then, assume there exists some continuous function $m : \mathbb{R} \rightarrow \mathbb{R}$ such that for all $z$, $E[\text{size}(h(z))] \leq m(\text{size}(z))$ and $0 \leq m(\text{size}(z)) \leq \text{size}(z)$. Moreover, the function $m(x)/x$ must be non-decreasing. Karp then argues that if there exists a solution to the deterministic recurrence relation $\tau(x) = a(x) + \tau(m(x))$, there must be a continuous minimal solution $u : \mathbb{R} \rightarrow \mathbb{R}$. He assumes such a solution exists and derives the following tail bound for $W$ in terms of $u$:

**Theorem 4.1 ([Karp 1994])**. For all $z$ and integer $w$ such that $\text{size}(z) > d$,

$$\Pr[W(z) > u(\text{size}(z)) + w \cdot a(\text{size}(z))] \leq \left(\frac{m(\text{size}(z))}{\text{size}(z)}\right)^w$$

Because $u$ is the minimal solution to the deterministic recurrence, we can replace $u$ with any other solution $t$ in the above bound: if $W(z)$ is greater than the version with $t$, then by minimality of $u$, it must be bigger than the version with $u$. This means we do not need to find a closed form for the minimal solution $u$, because any solution will give us a bound.

It is important to note that $m$, $a$ and $u$ are all functions from $\mathbb{R}$ to $\mathbb{R}$. This means that we do not have to deal with subtle rounding issues that sometimes come up when attempting to formalize solutions to recurrences for algorithms. Eberl [2017c], in his formalization of the Akra-Bazzi theorem, has pointed out how important this can be. The trade-off is that establishing that the recurrence holds everywhere on the domain $\mathbb{R}$ can be harder, especially at the boundaries where the recurrence terminates.

### 4.2 Extension to Binary Work and Span Recurrences

Although Theorem 4.1 makes it easier to get strong tail bounds, it cannot be used in many cases because it only applies to programs that have a single recursive call.

Tassarotti [2017] describes an extension to cover the general case of work and span recurrences with $n > 1$ recursive calls. In our mechanization, we only handle the case where there are two recursive calls (so that $n = 2$) because this is sufficient for many examples. In this setting, we now have two random variables $h_1$ and $h_2$ giving the recursive subproblems. These variables are generally not independent: for QuickSort, $h_1$ would be the lower partition of the list and $h_2$ is the upper partition. However, it is assumed that there is some function $g_1 : \mathbb{R} \rightarrow \mathbb{R}$ such that for all $z \in I$ and $(z_1, z_2)$ in the support of $(h_1(z), h_2(z))$:

$$g_1(\text{size}(z_1)) + g_1(\text{size}(z_2)) \leq g_1(\text{size}(z))$$

Informally, we can think of this function $g_1$ as a kind of ranking function, and the above inequality is saying that the combined rank of the two subproblems is no bigger than that of the original problem. The function $m$ is now required to bound the expected value of the maximum size of the two subproblems:

$$E[\max(\text{size}(h_1(z)), \text{size}(h_2(z)))] \leq m(\text{size}(z))$$

For bounding span recurrences of the form:

$$S(z) \leq a(\text{size}(z)) + \max(S(h_1(z)), S(h_2(z)))$$

\[\text{(4)}\]

---

3In fact, the assumptions in [Karp 1994] are slightly stronger than this. But as we discuss in §6, we discovered that the weaker assumptions mentioned here are sufficient.
we assume once more that \( u \) is a solution to the recurrence \( u(x) \geq a(x) + t(m(x)) \). Then we have:

**Theorem 4.2.** For all \( z \) and integer \( w \) such that \( \text{size}(z) > d \) and \( g_1(\text{size}(z)) > 1 \),

\[
\Pr [S(z) > u(\text{size}(z)) + w \cdot a(\text{size}(z))] \leq g_1(\text{size}(z)) \cdot \left( \frac{m(\text{size}(z))}{\text{size}(z)} \right)^w
\]

The difference between the bound above and the one in Theorem 4.1 is the additional factor \( g_1(\text{size}(z)) \). Generally speaking, \( g_1(\text{size}(z)) \) will be much smaller than \( \left( \frac{m(\text{size}(z))}{\text{size}(z)} \right)^w \), so this factor is negligible for large inputs.

The bound for binary work recurrences is slightly different. Given the recurrence:

\[
W(z) \leq a(\text{size}(z)) + W(h_1(z)) + W(h_2(z))
\]

we need a second “ranking” function \( g_2 \) with the same property that \( g_2(\text{size}(z_1)) + g_2(\text{size}(z_2)) \leq g_2(\text{size}(z)) \) for all \( z_1 \) and \( z_2 \) in the support of the joint distribution \((h_1(z), h_2(z))\) when \( \text{size}(z) > d \). In the proof by Tassarotti [2017], this second ranking function is used to transform the work recurrence into a span recurrence which is then bounded by Theorem 4.2, and this bound is converted back to a bound on the original recurrence. From the perspective of the user of the theorem, we now need \( u \) to solve the deterministic recurrence \( u(x) \geq \frac{a(x)}{g_2(x)} + u(m(x)) \), and we obtain the following bound:

**Theorem 4.3.** For all \( z \) and integer \( w \) such that \( \text{size}(z) > d \) and \( g_1(\text{size}(z)) > 1 \),

\[
\Pr [W(z) > g_2(\text{size}(z)) \cdot u(\text{size}(z)) + w \cdot a(\text{size}(z))] \leq g_1(\text{size}(z)) \cdot \left( \frac{m(\text{size}(z))}{\text{size}(z)} \right)^w
\]

Observe that on the left side of the bound, we re-scale \( u \) by a factor of \( g_2(\text{size}(z)) \) because it was the solution to a recurrence in which we normalized everything by \( g_2 \).

Because \( m \) must bound the expected value of the maximum of the subproblem sizes, the reader might wonder whether these theorems are actually usable. After all, we motivated the need for tail bound results in the introduction by explaining that expectations of maxima are hard to derive. Admittedly, computing this expectation bound is the trickiest part of using these theorems, but it is still easier than bounding the span directly because the \( h_i \) are simpler than the full definition of \( S \).

The above results let us fairly easily obtain tail bounds for a wide variety of probabilistic recurrences arising in the analysis of randomized divide-and-conquer algorithms. In the next section, we demonstrate their use by verifying a series of examples. After showing how they are used, we return to the discussion of the results themselves in §6, where we describe issues we encountered when trying to translate the paper proofs into Coq.

### 5 Examples

We now apply the results developed in the previous sections to several examples.

#### 5.1 Sequential QuickSort

Our first example is bounding the number of comparisons performed by a sequential implementation of randomized QuickSort. To count the number of comparisons that the monadic implementation of the algorithm performs, we combine the probabilistic monad from the previous section with a version of the writer monad that increments a counter every time a comparison is done. This cost monad is defined by:

Definition cost A := (nat * A).

Definition cost_bind {A B} (f: A -> cost B) x :=
Fixpoint qs l : ldist (cost (list nat)) :=
  match l as l'
  return with
  | [:::] => mret ([:::])
  | [:::a] => mret ([::a])
  | (a :: b :: l') =>
    p <- draw_pivot (a :: b :: l');
    '(lower, middle, upper) <- partition p l;
    ls <- qs (lower);
    us <- qs (upper);
    mret (ls ++ middle ++ us)
end

Fig. 6. Simplified version of code for sequential QuickSort. In ssreflect, we write [:::] for the empty list and [:::a] for a list containing the single element a. Because randomized QuickSort is not structurally recursive, Coq’s termination checker will not accept the above definition; instead we have to define it by well-founded recursion on the size of the input list.

(x.1 + (f (x.2)).1, (f (x.2)).2).
Definition cost_ret {A} (x: A) := (0, x).

A computation of type cost A is just a pair of a nat, representing the count of the number of comparisons, and an underlying value of type A. The bind operation sums costs in the obvious way. We can then define a version of comparison in this monad:

Definition compare (x y: nat) :=
  (1, ltngtP x y).
where ltngtP is a function from the ssreflect library that returns whether x < y, x = y, or x > y.

The code\(^4\) for QuickSort is given in Figure 6 and has type list nat → ldist (cost (list nat)). This is the standard randomized functional version of QuickSort: For empty and singleton lists, qs simply returns the input. Otherwise, it selects an element uniformly at random from the list using draw_pivot. It then uses partition to split the list into three parts: elements smaller than the pivot, elements equal to the pivot, and elements larger than the pivot. Elements smaller and larger than the pivot are recursively sorted and then the results are joined together. Partition uses the compare operator defined above, which implicitly counts the comparisons it performs.

What is the probabilistic recurrence for this algorithm? In each round of the recursion, the algorithm performs \(n\) comparisons to partition a list of length \(n\). So, taking the size function to be the length of the list, we have the toll function \(a(x) = x\). There are two recursive calls, and we have to sum the comparisons performed by each to get the total, so we need to use Theorem 4.3.

The \(h_1\) and \(h_2\) functions giving the recursive subproblems correspond to the lower and upper sublists returned by partition. We now need to bound the expected value of the maximum of the sizes of these two lists. We first show:

\[
E\left[\max\left(size(h_1(l)), size(h_2(l))\right)\right] \leq \frac{1}{size(l)} \sum_{i=0}^{size(l)-1} \max(i, size(l) - i - 1)
\]

\(^4\)Technically, the actual definition in our development is different, because the Coq termination checker cannot determine that this definition always terminates. Instead, the real version defines the algorithm by well founded induction on the size of the input list.
To get some intuition for this inequality, imagine the input list \( l \) was already sorted. In this situation, if the pivot we draw is in position \( i \), then the sublist of elements less than \( i \) only contains elements to the left of \( i \) in \( l \) and the sublist of elements larger than \( i \) contains only elements to the right of \( i \) in \( l \). The size of each sublist is therefore at most \( i \) and \( size(l) - i - 1 \), respectively, which corresponds to the \( i \)th term in the sum above. The factor of \( \frac{1}{size(l)} \) is the probability of selecting each pivot index, because they are all equally likely. Of course, the input list is not actually sorted, but when we select pivot position \( i \), we can consider where its position would be in the final sorted list, and the result is just a re-ordering of the terms in the sum.

Next we show that for all \( n > 1 \):

\[
\sum_{i=0}^{n-1} \max(i, n - i - 1) = \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{2} \right\rfloor \cdot \left\lfloor \frac{n}{2} \right\rfloor \leq \frac{3n^2}{4}
\]

The proof is by induction on \( n \). Fortunately, ssreflect contains many facts about binomial coefficients and rounded halving of natural numbers which we use to show this bound. We combine the two inequalities to conclude:

\[
E \left[ \max \left( size(h_1(l)), size(h_2(l)) \right) \right] \leq \frac{3}{4} \cdot size(l)
\]

The above bound is for the case when the list has at least 2 elements; otherwise the recursion is over so that the sublists have length 0. Hence we can take \( m \) to be:

\[
m(x) = \begin{cases} 0 & x < 4/3 \\ \frac{3x}{4} & x \geq 4/3 \end{cases}
\]

We use 4/3 as the cut-off point rather than 2 in the above definition because it makes the recurrence easier to solve.

To use Theorem 4.3, we need to come up with two “ranking” functions \( g_1 \) and \( g_2 \) such that \( g_1(size(h_1(z))) + g_1(size(h_2(z))) \leq g_1(size(z)) \) for each \( i \). Ideally, we want \( g_1 \) to be as small as possible, because it scales the final bound we derive, whereas for \( g_2 \) we want to pick something that makes it easy to solve the recurrence \( t(x) \geq a(x)/g_2(x) + t(m(x)) \). Like the derivation of the bound \( m \), these parts of the proof are not automatic and require some experimentation. Taking \( g_1 \) and \( g_2 \) to be:

\[
g_1(x) = x \\
g_2(x) = \begin{cases} \frac{1}{2} & x \leq 1 \\ \frac{x}{x-1} & 1 < x < 2 \\ x & x \geq 2 \end{cases}
\]

we can check that they satisfy the necessary conditions. The last step is to solve the resulting recurrence. We find that:

\[
t(x) = \begin{cases} 1 & x \leq 1 \\ \log_{4/3} x + 1 & x > 1 \end{cases}
\]

is a solution. Proving \( t \) satisfies the deterministic recurrence requires just under 60 lines in our mechanization.
Writing $T(x)$ for the total number of comparisons performed on input $x$, Theorem 4.3 now gives us:

$$\Pr \left[ T(x) > size(x) \cdot \log_{4/3}(size(x)) + 1 + w \cdot size(x) \right] \leq size(x) \cdot \left( \frac{3}{4} \right)^w$$

for $l$ such that $size(x) > 1$. More concisely, if we set $n = size(x)$, then this becomes:

$$\Pr \left[ T(x) > n \log_{4/3} n + 1 + wn \right] \leq n \cdot \left( \frac{3}{4} \right)^w$$

In Coq, this is rendered as:

```
Theorem bound x w:
  rsize x > 1 ->
  pr_gt (T x) (rsize x * (k * ln (rsize x) + 1) + INR w * rsize x)
  <= (rsize x) * (3/4)^w.
```

where $k = \frac{1}{\ln 4/3}$, $rsize$ returns the length of a list as a real number, and $INR : \text{nat} \to \mathbb{R}$ coerces its input into a real number.

To understand the significance of these bounds, observe that if $w$ is on the order of $c \cdot \log_{4/3} n$ for some constant $c$, the above becomes:

$$\Pr \left[ T(x) > (c + 1)n \log_{4/3} n + 1 \right] \leq n \cdot \left( \frac{3}{4} \right)^{c \log_{4/3} n} = \frac{1}{n^{c-1}}$$

so that when $c > 2$, the probability goes very quickly to 0 for lists of even moderate size.

We can now use the Coq-Interval library, which provides tactics for establishing numerical inequalities, to compute the value of this bound for particular choices of $n$. In particular, we can establish the claim from the introduction: when sorting a list with 10 million elements, the probability that QuickSort performs more than $8n \log_2 n$ comparisons is less than $10^{-9}$.

Remark concrete2:

```
forall l, rsize l = 10 ^ 7 ->
  pr_gt (T l) (10^7 * (8 * 1/(ln 2) * ln (10^7))) <= 1/(10^9).
```

### 5.2 Parallel Quicksort

Our next example is a bound on the span arising from comparisons in parallel QuickSort. As we mentioned in the introduction, the span of an algorithm is the longest sequential chain of computations that it performs. This measure is important, because it affects how much performance improvement we can expect by using more processors.

There are many cost models for parallelism, much like in the sequential setting where we have everything from the RAM model and multitape Turing machines to language based cost models. Here we will consider something like the cost model of the parallel language NESL [Blelloch 1995]. In NESL, the span of functional list operations like maps and filters is equal to the maximum span of applying the operation to each element of the list, plus some constant overhead.

Because these overheads are constant, it makes sense to count just the span arising from comparison operations when analyzing sorting algorithms\(^5\). This is no different from what we do when comparing sequential sorting algorithms by the number of comparisons they perform.

\(^5\)As always, when we want to understand how an algorithm will perform on a particular machine, we have to consider whether our cost model is realistic. To account for how parallel languages are implemented on some machines, there are other cost models in which filter operations actually have a $\log n$ span overhead. This can affect the asymptotic running time of algorithms like QuickSort, so a different analysis is needed when considering those models.
To track the work and span of a parallel computation, we modify the cost monad from the previous example:

Record cost A := mkCost {
  work : nat;
  span : nat;
  result : A;
}.

Definition cost_bind {A B} (f: A -> cost B) x :=
  mkCost (work x + work (f (result x))
           (span x + span (f (result x))
           (result (f (result x))).

Definition cost_ret {A} (x: A) :=
  mkCost 0 0 x.

Because bind represents sequential composition of code, we sum both the work and span to get the total cost.

In contrast, when we run computations in parallel, we add their work together to get the combined work, but only take the maximum of the spans. For example, we represent the cost of parallel execution of a pair of computations by:

Definition par2 {A B} (a: cost A) (b: cost B) : cost (A * B) :=
  {| result := (result a, result b) ;
    work := work a + work b;
    span := max (span a) (span b) |}.

The \(n\)-ary parallel composition can be defined analogously. Similarly, parallel maps and filters are defined to have the sum of the work of applying the operation to each element of the list, whereas the span is the maximum.

The code for parallel QuickSort using these operations is shown in Figure 7. As in sequential QuickSort, a pivot element is randomly selected from the list. However, rather than making a single sequential pass through the list for partitioning, the parallel version runs three filter passes in parallel, which find the sublists less than, equal to, and greater than the pivot. Then the lower and upper sublists are sorted by parallel recursive calls, and the final results are combined with the middle partition.

Of course, the parallel version does more total comparisons, because each of the three filters compares the pivot against every element of the list. But, all of these happen in parallel, so the span arising from each step of the recursion is just 1. Thus, we know the following definition of \(a\) bounds the span for partitioning:

\[
a(x) = \begin{cases} 
0 & x \leq 1 \\
 x - 1 & 1 < x < 2 \\
 1 & x \geq 2 
\end{cases}
\]

The definition of \(a\) on the interval \((1, 2)\) is chosen so that it is continuous on the interval \((1, \infty)\), as required by Theorem 4.2. Of course, the \(h_i\) are the same as in sequential QuickSort, so we can re-use the same bound \(m(x) = \frac{3x}{4}\), with essentially the same proof. The resulting recurrence,
Definition partition (n: nat) (l: list nat) :=
  par3 (parfilter (fun x => ltc x n) l)
    (parfilter (fun x => eqc x n) l)
    (parfilter (fun x => gtc x n) l);

Fixpoint qs l : ldist_cost (list nat) :=
  match l as l' with
  | ::[] => mret ([::])
  | ::a => mret ([::a])
  | a :: b :: l' =>
    p <- draw_pivot (a :: b :: l');
    '(lower, middle, upper) <- partition p l;
    '(ls, us) <- par2 (qs lower) (qs upper);
    mret (ls ++ (middle) ++ us)
  end

Fig. 7. Simplified version of code for parallel QuickSort.

t(x) = a(x) + t(m(x)) is then the same as for the sequential recurrence\(^6\), so we once again have the solution \( t(x) = \log_{4/3}(x) + 1 \). We also re-use \( g_1(x) = x \) from the sequential analysis, and then Theorem 4.2 implies:

\[
\Pr\left[ T(x) > \log_{4/3} n + 1 + w \right] \leq n \cdot \left( \frac{3}{4} \right)^w
\]

5.3 Height of Binary Search Tree

Our third example is the average height of a binary search tree generated by inserting the elements of a list in random order. When elements are inserted this way, the tree will have logarithmic height with very high probability, as opposed to the worst-case possibility of linear height. Of course, we cannot always assume that elements are inserted in this random order, so data structures like treaps [Seidel and Aragon 1996] and randomized binary search trees [Martínez and Roura 1998] use randomness in a way that mimics the effects of random insertion order.

Our implementation is based on a generic definition of search trees in the Coq standard library. This tree datatype is parameterized by a type of “auxiliary information” which is stored in each node in addition to the key. For the implementations of balanced trees in the Coq standard library, this auxiliary information is the data used to maintain the invariants needed for balancing (e.g., the color of a node for Red-Black trees). The tree data type has two constructors: Leaf and Node \( a \ t1 \ x \ tr \), where \( a \) is auxiliary information, \( t1 \) and \( tr \) are the left and right subtrees of a node, and \( x \) is the key stored in the node.

Because our goal for this example is to study the height of non-balancing trees, we do not need additional information, so the type of our auxiliary information is just unit. The code defining insertion and the height of a tree is shown in Figure 8. We consider the height of a tree with a single node to be 0. The function add_list_random l t inserts the elements of list \( l \) into the tree \( t \) by repeatedly removing a random element from the list and inserting it into the tree, until the list is empty.

\(^6\)Observe that the result of dividing the toll function for sequential QuickSort by the \( g_2 \) function used there gives the toll function for parallel QuickSort.
Fixpoint add (x: nat) (t: tree) :=
  match t with
  | Leaf => Node tt Leaf x Leaf
  | Node _ tl v tr =>
    match (Nat.compare x v) with
    | Eq => t
    | Lt => Node tt (add x tl) v tr
    | Gt => Node tt tl v (add x tr)
    end
  end.

Fixpoint height (t: tree) :=
  match t with
  | Leaf => 0
  | Node _ Leaf v Leaf => 0
  | Node _ tl v tr =>
    1 + (max (height tl) (height tr))
  end.

Fixpoint add_list_random (l: list nat) (t: tree) :=
  match l with
  | [] => mret t
  | [::a] => mret (add a t)
  | (a :: b :: l') =>
    p <- draw_next (a :: b :: l');
    add_list_random (rem p (a :: b :: l')) (add p t)
  end.

Fig. 8. Code for binary search tree insertion and adding a list of elements in random order.

This random process of generating a tree does not match the “divide-and-conquer” format of the Karp-style theorems: there are no random \( h_1 \) and \( h_2 \) which divide the input into subproblems that are then processed recursively. Rather, at each step exactly 1 element is removed and inserted. However, it is well known [Martínez and Roura 1998] that this process is actually equivalent to one in which the divide-and-conquer nature is explicit and similar to that of QuickSort. Observe that, when the tree \( t \) is a leaf, and element \( p \) is selected first from \( l \) for insertion, all of the remaining elements of \( l \) that are less than \( p \) will be inserted in the left subtree of the root node containing \( p \), and all of the larger elements will be in the right subtree. We can express this recursive version in Coq as:

Fixpoint rand_tree_rec l :=
  match l with
  | [] => mret Leaf
  | [::a] => mret (Node tt Leaf a Leaf)
  | (a :: l) => p <- draw_next a l;
    tl <- rand_tree_rec [seq i <- (a :: l) | (i < p)];
    tr <- rand_tree_rec [seq i <- (a :: l) | (i > p)];
  end.
In this format, we can apply Theorem 4.2 to analyze the height of the resulting tree. The first step in our proof, therefore, is to prove that these two constructions yield equal distributions. We handle only the case when the list \( l \) has no duplicate elements, which simplifies the proof:

**Lemma alr_rt_perm l:**

\[
\text{uniq } l \rightarrow \\
\text{eq_dist (rvar_of_ldist (add_list_random l Leaf))} \\
\text{(rvar_of_ldist (rand_tree_rec l))}.
\]

From there, we can bound the height of trees generated by \( \text{rand_tree_rec} \) and convert them to bounds on \( \text{add_list_random} \). Because the height of a non-singleton tree is 1 more than the maximum of the heights of its children, we will use Theorem 4.2, and we can re-use the exact same choice of \( a, m, \) and \( \varrho \) as we did for the parallel span of QuickSort. Letting \( T(l) \) be the height of the tree generated from a list \( l \) of length \( n \) containing no duplicates, we obtain:

\[
\Pr[T(l) > \log_{4/3} n + 1 + w] \leq n \cdot \left(\frac{3}{4}\right)^w
\]

Using this bound, we can verify in Coq that a tree generated from a list with 10 billion elements has height greater than 300 with probability less than \( 10^{-15} \):

**Remark concrete:**

\[
\text{forall } l, \text{uniq } l \rightarrow \text{rsize } l = 10^{10} \rightarrow \\
\text{pr_gt (T l) 300 <= 1/(10^{15})}.
\]

Before moving to the final example, let us acknowledge that these first three examples are very similar: the same bound on the \( h_1 \) and \( h_2 \) functions was re-used, and even the final deterministic recurrence ended up being the same. This is not that surprising, because binary search trees and randomized QuickSort are known to be deeply related.

However, we see it as a benefit of the theorems from §4 that we are able to easily re-use some of these intermediate results without having to first prove an explicit connection between the height of the tree and the span of the parallel sort. This is important because when we consider minor variants of QuickSort and binary search trees, the connection between them becomes less clear. For example, with parallel QuickSort under a cost model where filtering has a \( O(\log n) \) overhead, the connection between the span and the binary search tree height is less obvious, yet we can still re-use aspects of the arguments above. In fact, in our Coq formalization we have verified the solution to the corresponding recurrence relation for span under this cost model without much additional work.

### 5.4 Randomized Leader Election

Our last example is a randomized leader election protocol. The set-up is that there are \( n \) distributed nodes in a fully connected network, and they want to designate one of them as a "leader" that will be used for coordination of tasks.\(^7\) We consider a protocol that has been analyzed by several authors [Fill et al. 1996; Prodinger 1993]. The protocol consists of stages called "rounds". At the beginning of a round, each node that wants to try to become the leader generates a random bit. If the bit is 1, the node is said to remain "active" and sends a message to all the other nodes indicating

\(^7\)Versions of this problem are a well-studied subject in the theory of distributed computing [Lynch 1996].
its continuing intention to become the leader. Otherwise, if the bit is 0 it becomes inactive and stops trying to become the leader. If, within a round, only a single node gets a 1 bit, it becomes the leader. Otherwise, if multiple nodes generate a 1 bit, they each try again in the next round. Of course, it is possible that every active node will generate a 0 within a round: in that case, no messages are sent within the round, and instead of becoming inactive, those nodes try again in the next round, so as to avoid the possibility of having no leader elected.

Code modeling the outcome of this protocol is shown in Figure 9. With the description of the protocol given above, it is possible for the protocol to never terminate because every active node could keep drawing a 1 bit and never become inactive. Therefore, to ensure termination, the leader_elect function takes an argument rounds which is a limit on the number of rounds simulated by the code. The remaining number of active nodes is represented by the argument players. If either rounds goes to 0, or the number of players drops below 2, the function terminates. If not, the process of each active node generating a bit and updating their status is simulated with binomial, which returns the number of nodes that generated a 1. If the result of binomial is 0, then the process repeats recursively with the same value of players, but the number of remaining rounds is decreased by 1. Otherwise, the output of binomial is the new number of active players in the recursive call.

As the name binomial suggests, the number of nodes that generate a 1 in a round is given by the binomial distribution, but the actual number of nodes that proceed to the next round is not quite the same because of the special case where every node generates a 0. Recursive random processes in which sub-problem size distributions are very nearly equal to a binomial distribution are known as binomial splitting processes. These arise in the analysis of many algorithms and data structures, including tries, radix sort, and random number generation, among others. (See [Fuchs et al. 2014] for an overview and references.)

Although the number of rounds needed for this protocol is unbounded, intuitively we expect that for \( n \) initially active nodes, not much more than \( \log n \) rounds should be needed because the number of active players very nearly halves on average. Because there is only a single recursive call in each round, we can use Theorem 4.1 to derive a tail bound that confirms this intuitive understanding.

Even though the recursive process here is rather different from the previous three examples, the recurrence relation solutions we developed there end up applying here as well. We define the size of an input to the protocol to be 0 if the number of rounds is 0, and otherwise it is the number of active nodes. Because each recursive call corresponds to 1 round, we can use the same choice of function \( a \) as we did for the tree height example. The recursive problem size \( h \) is the binomial process with the special case for all 0 bits. In expectation, it is not quite half the input size, but an easy inductive argument shows that once more

\[
m(x) = \begin{cases} 
0 & x < 4/3 \\
\frac{3x}{4} & x \geq 4/3 
\end{cases}
\]

suffices. Putting it all together, we show that if \( T(x) \) is the difference between the input rounds and the final remaining rounds, and we start with \( n \) initial players, then

\[
\Pr \left[ T(x) > \log_{4/3} n + 1 + w \right] \leq \left( \frac{3}{4} \right)^w
\]

Notice that unlike the previous bounds, we do not scale by a factor of \( n \) on the right side of the inequality, because Theorem 4.1 does not require this. The above bound implies, for instance, that with 512 players, the probability that the protocol takes more than 64 rounds is less than \( 10^{-5} \).
Fixpoint binomial (n: nat) : ldist nat :=
  match n with
  | 0 => mret 0
  | S n' => b <- flip;
       rest <- binomial n';
       if b then
         mret (S rest)
       else
         mret rest
  end.

Fixpoint leader_elect (rounds: nat) (players: nat) : ldist (nat * nat) :=
  match rounds with
  | 0 => mret (O, players)
  | S rounds' =>
    match players with
    | O => mret (rounds, O)
    | 1 => mret (rounds, S O)
    | S (S _) =>
      (surv <- binomial players;
       (if surv == O then
         (* no one survived, current players repeat in next round *)
         leader_elect rounds' players
       else
         leader_elect rounds' surv))
  end
end.

Fig. 9. Model of leader election protocol.

6 CHANGES NEEDED FOR MECHANIZATION

Anyone who has mechanized something based on a paper proof has probably encountered issues that make it harder than just “translating” the steps of the proof into the formal system. Even when the paper proof is correct, there are inevitably parts of the argument that are more difficult to mechanize than they appear on paper, and this can require changing the strategy of the proof.

Our experience mechanizing Karp’s theorem and its extensions was no different. In this section we describe obstacles that arose in our attempt to mechanize the proof.

Overview of proof: To put the following discussion in context, we need to give a little more detail about how the paper proof goes. Recall that Theorem 4.1 says that if we have a probabilistic recurrence $W$ with a corresponding deterministic recurrence solved by $u$, then for all $z$ and integer $w$,

$$\Pr [W(z) > u(size(z)) + w \cdot a(size(z))] \leq \left( \frac{m(size(z))}{size(z)} \right)^w$$

The first thing one would naturally try to prove this is to proceed by induction on the size of $z$, or perhaps the size of $u(size(z)) + w \cdot a(size(z))$. However, immediately one realizes that the
The induction hypothesis needs to be strengthened: the bound above is only shown at each integer $w$, so there are “gaps” in between where we do not have an appropriately tight intermediate bound. To address this, Karp defines a function $D_r$ which “interpolates” the bound $\left(\frac{m(size(z))}{size(z)}\right)^w$ to fill in these gaps. This function $D_r$ is somewhat complicated, and is defined in a piecewise manner as follows:

1. If $r \leq 0$ and $x > 0$, $D_r(x) = 1$
2. If $r > 0$:
   a. If $x \leq d$ then $D_r(x) = 0$
   b. If $x > d$ and $u(x) \geq r$ then $D_r(x) = 1$
   c. If $x > d$ and $u(x) < r$ then

   $$D_r(x) = \left(\frac{m(x)}{x}\right)^{\frac{r-u(x)}{a(x)}} \cdot \frac{x}{u^{-1}(r - a(x))^{\frac{r-u(x)}{a(x)}}}$$

This definition is intricate, especially the last case. However, if we set $D_0$ defined even if the denominator is $t$, which “interpolates” the bound $\frac{m(size(z))}{size(z)}$ to fill in these gaps. This function $D_r$ is somewhat complicated, and is defined in a piecewise manner as follows:

1. If $r \leq 0$ and $x > 0$, $D_r(x) = 1$
2. If $r > 0$:
   a. If $x \leq d$ then $D_r(x) = 0$
   b. If $x > d$ and $u(x) \geq r$ then $D_r(x) = 1$
   c. If $x > d$ and $u(x) < r$ then

   $$D_r(x) = \left(\frac{m(x)}{x}\right)^{\frac{r-u(x)}{a(x)}} \cdot \frac{x}{u^{-1}(r - a(x))^{\frac{r-u(x)}{a(x)}}}$$

This definition is intricate, especially the last case. However, if we set $r = u(size(z)) + w \cdot a(size(z))$, then $D_r(size(z))$ simplifies to $\left(\frac{m(size(z))}{size(z)}\right)^w$, confirming the intuition that this is some kind of interpolation.

Karp’s proof proceeds by recursively defining a sequence of functions $K_i^r(z)$ for $i \in \mathbb{N}$, the details of which are not important. By induction on $i$, it is shown that $K_i^r(z) \leq D_r(size(z))$ for all $r$ and $z$. Finally, it is stated that $\Pr[W(z) > r] \leq \sup_i K_i^r(z)$, hence one concludes that $\Pr[W(z) > r] \leq D_r(size(z))$. Finally one sets $r = u(size(z)) + w \cdot a(size(z))$ to get the desired bound.

**Termination assumption.** The first problem we had was that we were unable to prove that $\Pr[W(z) > r] \leq \sup_i K_i^r(z)$. In the original paper proof, this inequality is simply stated without further justification. Young [2016] has suggested that in fact one may need stronger assumptions on $W$ or $h$ to be able to conclude this and suggests two alternatives. Either $W$ can be assumed to be a minimal solution to the probabilistic recurrence, or one can assume that the recurrence terminates with probability 1, that is $\Pr[h^n(z) > d] \to 0$ as $n \to \infty$. In the end, we chose to make the latter assumption, because it is easy to show for most examples.

**Existence of a minimal solution.** Karp argues that if there is a solution to the deterministic recurrence relation, there must be a minimal solution $u$. The results in the theorem are then stated in terms of $u$. It seemed to us more efficient to simply state the results in terms of the assumed solution to the recurrence relation. After all, to use the theorem, one will have to exhibit some solution $t$ to the recurrence anyway, and unless one computes a closed form for the minimal solution $u$, the final bounds will have to be stated in terms of $t$. In this way, we avoid the need to prove the existence, continuity, and invertibility of the minimal solution.

Instead we assume that there is some monotone function $t$ which solves the recurrence and is continuous. And, rather than assuming $t$ is invertible, we assume that there exists a function $t'$ such that $t'(t(x)) = x$ for $x > d$ and $t(t'(x)) = x$ for $x > t(d)$. The definition of $D$ is then changed to replace occurrences of $u$ with $t$.

**Division by zero.** The original piecewise definition of $D$ above involves division by $u^{-1}(r - a(x))^{\frac{r-u(x)}{a(x)}}$. However, it is not clear that this is always non-zero on the domain considered, and this is not explicitly discussed in the paper proof. Because we replace the $u^{-1}$ function with a user-supplied function $t'$, it is even less clear whether $t'$ would be non-zero.

In the Coq standard library, division of real numbers is a total function, so the expression is defined even if the denominator is 0. This means that we do not have to exhibit a proof that the
$t'(\cdots)$ term is non-zero when defining $D$. The catch is that most of the lemmas about division in the standard library have a premise requiring that the denominator is non-negative. In the end, we found it necessary to add an explicit assumption that $t'$ is non-zero everywhere.

**Unneeded assumptions.** In the original paper proof, $a$ is assumed to be everywhere continuous and strictly increasing on $[d, \infty)$. This rules out recurrences like $W(z) = 1 + W(h(x))$ which show up in examples such as the leader election protocol. For that reason, there is actually an additional result in Karp [1994] for the particular case where $a(x) = 0$ for $x \leq d$ and 1 otherwise. Originally, we intended to mechanize this extra lemma as well.

However, after finishing the mechanization of Theorem 4.1, we suspected that these restrictive assumptions on $a$ could be weakened, avoiding the need for the additional lemma. We instead changed the assumptions to only require that $a$ was monotone and continuous on the interval $(d, \infty)$. In turn, we require that the function $t$ which solves the deterministic recurrence is strictly increasing on the interval $(d, \infty)$. Surprisingly, our prior proof script worked mostly unchanged: most of the changes actually ended up deleting helper lemmas we had needed under the original assumptions. This is not because our proof scripts were highly automated or robust, but because the original proof really was not exploiting these stronger assumptions. Checking this carefully with respect to the original paper proof would have been rather tedious, but was straightforward with a mechanized version.

**Extending to the binary case.** In a technical report, Karpinski and Zimmermann [1991] claimed to extend Karp’s result to work and span recurrences with multiple recursive calls, so we initially tried to verify their result. The argument is fundamentally like Karp’s original proof, so many steps were described briefly because they were intended to be similar to the corresponding parts of the proof of Theorem 4.1. However, at a crucial point, we were unable to mechanize one of the elided steps. It was at this point that we mechanized the results from Tassarotti [2017] instead.

7 RELATED WORK

7.1 Verification of Randomized Algorithms and Mechanized Probability Theory

We have remarked that our work is based on a small library of results from probability theory that we developed. Randomized algorithms are represented using a monadic encoding of lists of outcomes, which can be interpreted as a distribution. Audebaud and Paulin-Mohring [2009] also do probabilistic reasoning about randomized algorithms in Coq without developing a library based on measure theoretic results. They also use a monadic encoding, but model distributions on a type $A$ as functions $A \rightarrow [0, 1] \rightarrow [0, 1]$ and axiomatize necessary properties of the unit interval. In particular, the $\omega$-CPO structure of $[0, 1]$ is used to represent randomized algorithms that do not necessarily terminate. They develop an extensive set of rules for reasoning about programs written in this monadic style, and use their development for several examples, including proving termination of a kind of random walk.

The combination of reasoning about probability and complexity is also needed for the formal verification of cryptographic protocols. The CertiCrypt project [Barthe et al. 2009] uses the probabilistic monad of Audebaud and Paulin-Mohring [2009] to represent cryptographic algorithms. To model costs of computations, they combine this monad with one that counts steps as we did for some of the examples in §5. They develop a probabilistic variant of Benton’s relational Hoare logic [Benton 2004] called pRHL to do relational reasoning about pairs of programs. The complexity considerations involved in the analysis of cryptographic protocols usually have to do with broad complexity classes (e.g., considering adversaries that can take polynomial time) rather than precise bounds, so the quantitative reasoning involved is different from what we have considered in this...
Extensions to and applications of pRHL for reasoning about probabilistic programs have been developed in a series of papers [Barthe et al. 2017, 2012, 2017], and this kind of relational reasoning has been implemented in the EasyCrypt tool [Barthe et al. 2012]. There are many other formal logics for reasoning about probabilistic programs (e.g., [Barthe et al. 2016; Kozen 1983; Morgan et al. 1996; Ramshaw 1979]). Kaminski et al. [2016] present a weakest-precondition logic that can be used to establish expected running time. As an example, they prove a bound on the expected number of comparisons used by QuickSort. The soundness of their logic was later mechanized by Hölzl [2016] in Isabelle.

Large developments of probability theory have been carried out in some theorem provers. Hurd [2003] developed a substantial amount of measure theory in HOL and used it to verify the correctness of the Miller-Rabin primality test, among other examples. Hölzl and Heller [2011] have developed a library for measure theory in Isabelle, which was later the foundation for a mechanized proof of the Central Limit Theorem by Avigad et al. [2014]. Daumas et al. [2010] describe a mechanization of Lévy and Hoeffding’s inequalities in PVS, with the eventual goal of verifying tail bounds for floating point algorithms. Affeldt and Hagiwara [2012] developed a library for finite probability theory in Coq, which they used to formalize results from information theory.

Broadly speaking, most of the above work which has been mechanized and applied to reasoning about randomized algorithms has been focused on their functional correctness, rather than on achieving tight bounds on cost measures. In contrast, van der Weegen and McKinna [2008] mechanized a proof of the average number of comparisons performed by QuickSort in Coq. Eberl [2017b] has recently mechanized a similar result, as well as bounds on the expected depth and height of binary search trees [Eberl 2017a]. The latter proof about expected heights uses a trick described in Cormen et al. [2009] that involves actually bounding the expected value of what they call the “exponential height” of a tree: a tree of height \( h \) has exponential height \( e^h \). This transformation lets one partially work around the issues involving the fact that expectation does not commute with maxima that we described in §1. We do not know how generally applicable this trick is.

### 7.2 Automated Analysis of Randomized Algorithms

A number of static analysis tools have been developed for randomized algorithms. PRISM [Kwiatkowska et al. 2011] is a model checker that can be used to analyze Markov chains and other probabilistic models and establish bounds on the probability of certain events (e.g., an assertion failure). Chakarov and Sankaranarayanan [2013] describe a program analysis that tries to discover martingales and then uses the Azuma-Hoeffding inequality to derive probability bounds from these martingales. They use this to prove almost-sure termination and establish numerical bounds on the probability that certain variables fall outside a given range. As applications, they analyze models of classical control systems (e.g., robot navigation with dead-reckoning, balancing an inverted pendulum) and prove that the state of the system remains in some envelope with high probability. These bounds are different from the symbolic high probability performance bounds we derive here. This martingale approach has been extended in subsequent work [Barthe et al. 2016; Chatterjee et al. 2017b].

Flajolet et al. [1991] describe a system that automates the average case analysis of certain algorithms and combinatorial structures by automatically applying techniques from analytic combinatorics using a computer algebra system. Chatterjee et al. [2017a] give an algorithm for trying to automatically solve recurrences for expected work of various algorithms such as QuickSort and QuickSelect.
7.3 Techniques for Bounds on Randomized Algorithms

There are a vast number of tools and results that have been developed for analyzing properties of randomized algorithms. The textbooks by Mitzenmacher and Upfal [2005] and Motwani and Raghavan [1995] give a good introduction. The latter also includes a simplified account of some of the results from Karp [1994].

The techniques of analytic combinatorics have been applied to many randomized algorithms. Flajolet and Sedgewick [2009] give a very extensive account of these methods and the problems they have been applied to. At a very high level, the approach involves deriving the generating function of some combinatorial class of objects and then using results from complex analysis to study the asymptotic behavior of the coefficients of the generating function. Although these methods can give very tight bounds for a large number of important algorithms, to be able to use them in a theorem prover, we would need to mechanize a substantial amount of complex analysis. (There is some promising work in mechanizing complex analysis, such as the recent proof of Cauchy’s residue theorem in Isabelle by Li and Paulson [2016].)

Other approaches rely on advanced theorems from probability theory. For example, concentration inequalities like the Azuma-Hoeffding inequality can be used to achieve bounds on the number of comparisons performed by QuickSort that are much tighter than those we have derived [McDiarmid and Hayward 1996]. Branching process theory can be used to study binary search trees [Devroye 1986]. The contraction method [Rösler and Rüschendorf 2001] can be used to show that a randomized divide-and-conquer algorithm converges to some limiting distribution. Once more, these techniques achieve better bounds but at the trade-off of being harder to apply and requiring more prerequisites.

Different “cookbook” methods like Karp’s also exist. Bazzi and Mitter [2003] develop a variant of the Akra-Bazzi master theorem for deriving asymptotic expectation bounds for work recurrences. Roura [2001] presents a master theorem that also applies to recurrences like that of the expected work for QuickSort.

Chaudhuri and Dubhashi [1997] extend the results of Karp [1994] for unary probabilistic recurrence relations by weakening some of the assumptions of Theorem 4.1. Their proof uses only “standard” techniques from probability theory like Markov’s inequality and Chernoff bounds, so they argue that it is easier to understand. Of course, this approach may not be so beneficial for mechanization if we do not have a pre-existing library of results such as Chernoff bounds. However unfamiliar or opaque Karp’s proof may be, the lack of prerequisites makes it easier to just begin mechanizing it directly. That said, it would be interesting to try to mechanize their proof using, say, the more extensive probability libraries that have been developed for Isabelle.

8 CONCLUSION

We have described our mechanization of theorems by Karp [1994] and Tassarotti [2017] that make it easier to obtain tail bounds for various probabilistic recurrence relations arising in the study of randomized algorithms. To demonstrate the use of these results, we have explained our verification of four example applications. Moreover, we have shown that these results can be used to obtain concrete numerical bounds, fully checked in Coq, for input sizes of practical significance. To our knowledge, this is the first mechanization of these kinds of tail bounds in a theorem prover. We now describe some possibilities for future work:

Automation. We suspect that many of the steps involved in applying the theorems from §4 to examples could be automated. The $a$, $g_1$, and $g_2$ functions can most likely be inferred automatically. In some cases, it may even be possible to infer a bound on the expected sizes of the recursive subproblems. The resulting recurrence could also probably be solved automatically. We believe
this is plausible, because these recurrences are already considerably simpler than the kinds of recurrences that are analyzed automatically for expectation bounds as in Chatterjee et al. [2017b]. If these analyses are done as part of external tools, it would be useful to be able to produce proof certificates that could be checked using the Coq development we describe here, as in some other resource analysis tools [Carbonneaux et al. 2017, 2015].

Even in cases where recurrences cannot be solved automatically, it might be possible to develop tools to identify them. Danner et al. [2015] have developed a denotational semantics for deterministic programs that extracts recurrences, and it would be interesting to adapt this to the probabilistic setting.

We could also develop better tactics for automatically simplifying monadic expressions using some of the rules described in §3. Tactics for these purposes have been developed in some of the related work using the probabilistic choice monad [Audebaud and Paulin-Mohring 2009; Barthe et al. 2009; Petcher and Morrisett 2015].

**Non-Finite Probability Spaces.** We have defined probability distributions for countable spaces, but our formalizations of the theorems from §4 are specialized to the case for finite probability spaces. This suffices to handle the analysis of algorithms that terminate in a bounded number of steps for inputs of a given size, but not those which merely terminate with probability 1. Similarly, the probabilistic choice monad described in §3 only handled computations that always terminate. For that reason, we had to consider a version of the distributed leader election protocol which only runs for a bounded number of rounds.

We believe Karp’s theorem holds in the more general setting of discrete or even continuous spaces. If we mechanized the proof in this more general setting and used the probabilistic choice monad closer to that of Audebaud and Paulin-Mohring [2009] to encode non-terminating probabilistic computations, we could more naturally reason about these and some other examples.

**Imperative Algorithms.** The examples we have discussed have been purely functional algorithms and data structures. However, the tail bound theorems also apply to imperative algorithms, so long as they can be shown to obey an appropriate probabilistic recurrence. Of course, showing that these probabilistic recurrences hold can be more difficult in the imperative case.

The probabilistic relational Hoare logic developed by Barthe et al. [2009] may be useful in this regard. To analyze something like imperative QuickSort, one could first use pRHL to prove a probabilistic refinement relation between the imperative version and the functional version, then show that the functional version obeys the recurrence, and finally use Theorem 4.3 to derive a tail bound. In other cases, it may be easier to use a unary Hoare logic to directly establish the recurrence for the imperative version without using an intermediate functional formulation.

**Additional Analytic Tools.** We have argued that one benefit of Karp’s theorem and its extensions is that it produces fairly strong bounds with minimal mathematical prerequisites, which is useful in the setting of formal verification where we often lack relevant mathematical libraries. That said, it would certainly be useful to develop some of the mathematical results needed by other methods. In particular, results like the Azuma-Hoeffding inequality would be especially useful to mechanize, as it is the foundation of many recent static analysis tools for randomized programs.

**ACKNOWLEDGMENTS**

The authors thank Jean-Baptiste Tristan, Jan Hoffmann, Guy Blelloch, Carlo Angiuli, and Daniel Gratzer for feedback and discussions about this work.

This research was conducted with U.S. Government support under and awarded by DoD, Air Force Office of Scientific Research, National Defense Science and Engineering Graduate (NDSEG)
Fellowship, 32 CFR 168a. This work was also supported by a gift from Oracle Labs. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these organizations.

REFERENCES


