Verified Tail Bounds for Randomized Programs

Joseph Tassarotti[1] and Robert Harper[1]

Carnegie Mellon University, Pittsburgh, USA

Abstract. We mechanize a theorem by Karp, along with several extensions, that provide an easy to use "cookbook" method for verifying tail bounds of randomized algorithms, 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.

1 Introduction

Formal verification of randomized algorithms remains a challenging problem. In recent years, a number of specialized program logics [8, 10, 11, 37, 44] and automated techniques [21, 6, 20] have been developed to analyze these programs. In addition, a number of randomized algorithms have been verified directly in interactive theorem provers [28, 27, 55] without using intermediary program logics. Besides establishing correctness results, much of this work has focused on verifying the *expected* or *average* cost of randomized algorithms. Although expectation bounds are an important first step in cost analysis, 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 randomized 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 [28, 55]. However, not only does it do $O(n \log n)$ comparisons *on average*, but the probability that it does 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 is 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.

Despite this, there is a good reason for the prior emphasis on expectation bounds rather than tail bounds in the field of formal methods: tail bounds on running time are usually quite difficult to derive. Common approaches for dervving these bounds involve the use of methods from analytic combinatorics [31] or the theory of concentration of measure [26]. 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 mentioned 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

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

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 [24, 13] that can be used to straightforwardly derive asymptotic bounds on these kinds of recurrences. Moreover, these results can also be used to easily analyze recurrences for other types of resource use, such as the maximum stack depth or the span of parallel divide-and-conquer algorithms [15]. Recurrences for these kinds of resources have the form:

$$S(x) = b(x) + \max_{i=1}^{n} S(h_i(x)) \tag{2}$$

We will call recurrences of the form in Equation 1 "work recurrences" and those of the form in Equation 2 "span recurrences". Although Equation 2 does not fit the format of the Master Theorem directly, when S is monotone the recurrence simplifies to $S(x) = b(x) + S(\max_{i=1}^{n}(h_i(x)))$ and so can be analyzed using the Master Theorem.

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 [29] has recently mechanized the more advanced Akra and Bazzi [2] 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 [24, 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[1], 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 [38] 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 strong. Recently, Karp's technique has been extended [54] 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 [42] 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.

We start by outlining the mechanization of probability theory that our work is based on (§2). We then describe Karp's theorem and its extensions in more detail (§3). To demonstrate how Karp's result is used, we describe our verification of the examples mentioned above, with a focus on the sequential QuickSort analysis (§4). 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 §5. Finally, we compare our approach to related work (§6) and conclude by discussing possible extensions and improvements to our development (§7).

[1] This is because in general $E[\max(X_1, X_2)] \geq \max(E[X_1], E[X_2])$.

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

2.1 Discrete Probability

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 §6). 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 [17] 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` [33] library's definition of countable types (`countType`), which consists of a type `A` equipped with a surjection from `nat` to `A`. The distribution record consists of three fields: (1) a probability mass function (`pmf` : `A` $\to$ `R`) that assigns a probability to each element of `A`, (2) a proof that `pmf a` is non-negative for all `a`, and (3) a proof that the countable series that sums `pmf a` over all `a` converges and is equal to 1. The Coq coercion mechanism is used to implicitly coerce distributions into their `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 (`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 series $\sum_{r \in \mathsf{img}(X)} \Pr[X = r] \cdot r$. 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, `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. For random variables defined on such distributions, the expectation always exists, because the series is actually just a finite sum. For our mechanization of Karp's

```
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} (O: distrib A) (P: A -> bool) :=
  Series (countable_sum (fun a => if P a then O a else 0)).

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

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

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

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

theorem, we restrict our attention to these finite distributions. When the distribution is finite we can convert from a Coquelict `Series` to a simple sum.

2.2 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 [50, 3, 9, 48]. In this section, we briefly describe some of the choices we made for our representation.

The definition of the basic probabilistic monad is given in Figure 2. The type `ldist 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 `ldist 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 `ldist A` will be equal exactly when their outcome fields are equal.

The bind operation (`dist_bind`) is similar to the way bind is defined for the list non-determinism monad: For each element $(\mathtt{r}, \mathtt{a})$ in the list of outcomes for some $\mathtt{l} : \mathtt{ldist\ A}$, we apply the function $\mathtt{f} : \mathtt{A} \to \mathtt{ldist\ B}$ 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 `l` to obtain

a random element of type `A` (*i.e.,* "sampling" from the distribution represented by `l`), and then passing this to `f`. The return operation (`dist_ret`) applied to `a` corresponds to the probabilistic computation that simply 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 uses the `ssreflect` "big-op" operation to express the sum of the first components of each pair of the form $(\mathtt{r}, \mathtt{a})$ occurring in the `outcomes` of `l`. The corresponding distribution is finite because the list of outcomes is finite.

```
Record ldist (A: Type) := mklDist {
  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.

3 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.

3.1 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 $size : I \to \mathbb{R}^{\geq 0}$ such that $size(z)$ 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:

$$W(z) = a(size(z)) + W(h(z)) \tag{3}$$

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 (admittedly, we did so in §1). 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. 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 $a : \mathbb{R} \to \mathbb{R}^{\geq 0}$ is required to be continuous and increasing[2] on (d, ∞), but equal to 0 on the interval $[0, d]$. In addition, it is required that $0 \leq size(h(z)) \leq size(z)$, *i.e.,* the size of the subproblem is not bigger than the original.

Then, assume there exists some continuous function $m : \mathbb{R} \to \mathbb{R}$ such that for all z, $E[size(h(z))] \leq m(size(z))$ and $0 \leq m(size(z)) \leq size(z))$. 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 $u : \mathbb{R} \to \mathbb{R}$. He assumes such a solution exists and derives the following tail bound for W in terms of u:

Theorem 1 ([38]). *For all z and integer w such that $size(z) > d$,*

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

[2] In fact, the assumptions in [38] are slightly stronger than this. But as we discuss in §5, we discovered that the weaker assumptions mentioned here are sufficient.

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 [29], 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.

3.2 Extension to Binary Work and Span Recurrences

Although Theorem 1 makes it easier to get strong tail bounds, it cannot be used in many cases because it only applies to programs with a single recursive call.

Tassarotti [54] 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 $g_1 : \mathbb{R} \to \mathbb{R}$ such that for all $z \in I$ and (z_1, z_2) in the support of $(h_1(z), h_2(z))$:

$$g_1\left(size(z_1)\right) + g_1\left(size(z_2)\right) \leq g_1\left(size(z)\right)$$

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:

$$E\left[\max\left(size(h_1(z)), size(h_2(z))\right)\right] \leq m(size(z))$$

For bounding span recurrences of the form:

$$S(z) \leq a(size(z)) + \max(S(h_1(z)), S(h_2(z))) \tag{4}$$

we assume once more that u is a solution to the recurrence $u(x) \geq a(x)+t(m(x))$. Then we have:

Theorem 2. *For all z and integer w such that $size(z) > d$ and $g_1(size(z)) > 1$,*

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

The difference between the bound above and the one in Theorem 1 is the additional factor $g_1(size(z))$. Generally speaking, $g_1(size(z))$ will be much smaller than $\left(\frac{m(size(z))}{size(z)}\right)^w$, so this factor is negligible for large inputs.

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

$$W(z) \leq a(size(z)) + W(h_1(z)) + W(h_2(z)) \tag{5}$$

we need a second "ranking" function g_2 with the same property that $g_2\,(size(z_1))+ g_2\,(size(z_2)) \leq g_2\,(size(z))$ for all z_1 and z_2 in the support of the joint distribution $(h_1(z), h_2(z))$ when $size(z) > d$. In the proof by Tassarotti [54], this second ranking function is used to transform the work recurrence into a span recurrence which is then bounded by Theorem 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 3. *For all z and integer w such that $size(z) > d$ and $g_1(size(z)) > 1$,*

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

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

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 §5, where we describe issues we encountered when trying to translate the paper proofs into Coq.

4 Examples

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

4.1 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 §2.2 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 :=
  (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[3] for QuickSort is given in Figure 3. 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.

```
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. 3. 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, the actual definition in our development defines it by well-founded recursion on the size of the input.

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 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

[3] The definition in our development is actually defined by well-founded induction on the size of the input, because the Coq termination checker cannot determine that this definition always terminates.

expected value of the maximum of the sizes of these two lists. We first show:

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

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 ith 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 by induction on n that:

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

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 define m to be $m(x) = 0$ for $x < 4/3$ and $m(x) = \frac{3x}{4}$ otherwise. We use $4/3$ as the cut-off point rather than 2 because it makes the recurrence easier to solve.

To use Theorem 3, we need to come up with two "ranking" functions g_1 and g_2 such that $g_i(size(h_1(z))) + g_i(size(h_2(z))) \leq g_i(size(z))$ 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. We define the following choices for the parameters of Theorem 3:

$$g_1(x) = x \qquad g_2(x) = \begin{cases} \frac{1}{2} & x \leq 1 \\ \frac{x}{x-1} & 1 < x < 2 \\ x & x \geq 2 \end{cases} \qquad t(x) = \begin{cases} 1 & x \leq 1 \\ \log_{\frac{4}{3}} x + 1 & x > 1 \end{cases}$$

We can check g_1 and g_2 satisfy the necessary conditions, and that t is a solution to the resulting deterministic recurrence relation.

Writing $T(x)$ for the total number of comparisons performed on input x, Theorem 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` : `nat` $\to$ `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).
```

4.2 Other Examples

We have mechanized the analysis of three other examples using Karp's theorem. A discussion of these examples is given in the appendix of the full version of this paper submitted as supplementary material. Here we give a brief description of the examples:

1. Parallel QuickSort: using Theorem 2 we show that the longest chain of sequential dependencies from comparisons in a parallel version of QuickSort is $O(\log(n))$ with high probability.
2. Binary search tree: we analyze the height of a binary search tree which is generated by inserting a set of elements under a random permutation. We show the height is $O(\log(n))$ with high probability using Theorem 2.

3. Randomized leader election: we consider a protocol for distributed leader election that has been analyzed by several authors [30, 49]. The protocol consists of stages called "rounds". At the beginning of a round, each active node generates a random bit. If the bit is 1, the node remains "active" and sends a message to all the other nodes; otherwise, if the bit is 0 it becomes inactive and stops trying to become the leader. If every active node generates a 0 within a round, no messages are sent and instead of becoming inactive, those nodes try again in the next round. When there is only one active node remaining, it is deemed the leader. We use Theorem 1 to show that with high probability at most $O(\log n)$ rounds are needed.

5 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 sketch of the paper proof. Recall that Theorem 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,

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

The first thing one would naturally try to prove this is to proceed by induction on the size of z. However, immediately one realizes that 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

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

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 $K_r^i(z)$ for $i \in \mathbb{N}$, the details of which are not important. By induction on i, it is shown that $K_r^i(z) \leq D_r(size(z))$ for all r and z. Finally, it is stated that $\Pr[W(z) > r] \leq \sup_i K_r^i(z)$, hence one concludes that $\Pr[W(z) > r] \leq D_r(size(z))$. By setting $r = u(size(z)) + w \cdot a(size(z)))$, we get the desired bound.

Termination assumption. The first problem we had was that we were unable to prove that $\Pr[W(z) > r] \leq \sup_i K_r^i(z)$. In the original paper proof, this inequality is simply stated without further justification. Young [56] 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 any continuous and invertible solution t to the recurrence relation. In this way, we avoid the need to prove the existence, continuity, and invertibility of the minimal solution. In fact, rather than assuming t is invertible, we merely 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 $u^{-1}(r - a(x)\left\lceil \frac{r-u(x)}{a(x)} \right\rceil)$. 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 end, we found it necessary to add an explicit assumption that t' is non-zero everywhere.

Unneeded assumptions. In the original paper proof, the toll function 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 [38] for the particular case where $a(x) = 0$ for $x \leq d$ and 1 otherwise.

However, after finishing the mechanization of Theorem 1, we suspected that the assumptions on a could be weakened, avoiding the need for the additional lemma. We changed the assumptions to only require that a was monotone and continuous on the interval (d, ∞). In turn, we require the function t which solves

the deterministic recurrence to be strictly increasing on the interval (d, ∞). 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 [39] 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 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 [54] instead.

6 Related Work

6.1 Verfication of Randomized Algorithms and Mechanized Probability Theory

Audebaud and Paulin-Mohring [3] developed a different monadic encoding for reasoning about randomized algorithms in Coq that can represent randomized algorithms that do not necessarily terminate. It would be interesting to try to generalize our version of Karp's theorem and apply them to programs expressed using this monad.

Barthe et al. [9] develop a probabilistic variant of Benton's relational Hoare logic [14] called pRHL to do relational reasoning about pairs of randomized programs. Extensions to and applications of pRHL for reasoning about probabilistic programs have been developed in a series of papers [10, 7, 11], and this kind of relational reasoning has been implemented in the EasyCrypt tool [5]. There are many other formal logics for reasoning about probabilistic programs (*e.g.,* [51, 8, 46, 40]). Kaminski et al. [37] 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 [34] in Isabelle.

van der Weegen and McKinna [55] mechanized a proof of the average number of comparisons performed by QuickSort in Coq, which used monad transformers to elegantly separate reasoning about correctness and cost while still being able to extract efficient code. Eberl [28] has recently mechanized a similar result, as well as bounds on the expected depth and height of binary search trees [27]. Eberl [29] has also mechanized the Akra-Bazzi theorem, a generalization of the Master Theorem for reasoning about deterministic divide and conquer recurrences.

More generally, multiple large developments of probability theory have been carried out in several theorem provers, including large amounts of measure theory [36, 35], the Central Limit Theorem [4], Lévy and Hoeffding's inequalities [25], and information theory [1], to name just some of these results.

6.2 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; see [45, 47, 31, 26] for expository accounts of both simple and more advanced techniques. Different "cookbook" methods like Karp's also exist: Bazzi and Mitter [12] develop a variant of the Akra-Bazzi master theorem for deriving asymptotic expectation bounds for work recurrences. Roura [52] presents a master theorem that also applies to recurrences like that of the expected work for QuickSort.

Chaudhuri and Dubhashi [23] extend the results of Karp [38] for unary probabilistic recurrence relations by weakening some of the assumptions of Theorem 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 be less beneficial for mechanization if we do not have a pre-existing library of results.

7 Conclusion

We have described our mechanization of theorems by Karp [38] and Tassarotti [54] 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.

In future work, it would be interesting to try to automate the inference of the a, g_1, and g_2 functions used when applying Karp's theorem. The resulting deterministic recurrence could also probably be solved automatically, since more complex recurrences have been analyzed automatically in related work (*e.g.,* [22]). 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 [19, 18].

It should also be possible to extend the applicability of our mechanization by handling arbitrary probability distributions instead of finite ones. Moreover, it may be possible to use tools like the probabilistic relational Hoare logic of Barthe et al. [9] to prove suitable refinements between imperative randomized algorithms and the functional versions we have analyzed here. This would allow one to derive corresponding tail bounds on the imperative versions.

Acknowledgments. The authors thank Jean-Baptiste Tristan, Jan Hoffmann, Justin Hsu, Guy Blelloch, Carlo Angiuli, 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

1. Affeldt, R., Hagiwara, M.: Formalization of Shannon's theorems in SSReflect-Coq. In: ITP. pp. 233–249 (2012)
2. Akra, M., Bazzi, L.: On the solution of linear recurrence equations. Comp. Opt. and Appl. 10(2), 195–210 (1998)
3. Audebaud, P., Paulin-Mohring, C.: Proofs of randomized algorithms in Coq. Sci. Comput. Program. 74(8), 568–589 (2009)
4. Avigad, J., Hölzl, J., Serafin, L.: A formally verified proof of the Central Limit Theorem. CoRR abs/1405.7012 (2014), http://arxiv.org/abs/1405.7012
5. Barthe, G., Crespo, J.M., Grégoire, B., Kunz, C., Béguelin, S.Z.: Computer-aided cryptographic proofs. In: ITP. pp. 11–27 (2012)
6. Barthe, G., Espitau, T., Fioriti, L.M.F., Hsu, J.: Synthesizing probabilistic invariants via Doob's decomposition. In: CAV. pp. 43–61 (2016)
7. Barthe, G., Espitau, T., Grégoire, B., Hsu, J., Strub, P.: Proving uniformity and independence by self-composition and coupling. In: LPAR (2017)
8. Barthe, G., Gaboardi, M., Grégoire, B., Hsu, J., Strub, P.: A program logic for union bounds. In: ICALP. pp. 107:1–107:15 (2016)
9. Barthe, G., Grégoire, B., Béguelin, S.Z.: Formal certification of code-based cryptographic proofs. In: POPL. pp. 90–101 (2009)
10. Barthe, G., Grégoire, B., Béguelin, S.Z.: Probabilistic relational hoare logics for computer-aided security proofs. In: MPC. pp. 1–6 (2012)
11. Barthe, G., Grégoire, B., Hsu, J., Strub, P.: Coupling proofs are probabilistic product programs. In: POPL. pp. 161–174 (2017)
12. Bazzi, L., Mitter, S.K.: The solution of linear probabilistic recurrence relations. Algorithmica 36(1), 41–57 (2003)
13. Bentley, J.L., Haken, D., Saxe, J.B.: A general method for solving divide-and-conquer recurrences. SIGACT News 12(3), 36–44 (Sep 1980)
14. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: POPL (2004)
15. Blelloch, G., Greiner, J.: Parallelism in sequential functional languages. In: Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture. pp. 226–237 (1995)
16. Blelloch, G.E.: Nesl: A nested data-parallel language (version 3.1). Tech. Rep. CMU-CS-95-170, Carnegie Mellon University (1995), https://www.cs.cmu.edu/~guyb/papers/Nesl3.1.pdf
17. Boldo, S., Lelay, C., Melquiond, G.: Coquelicot: A user-friendly library of real analysis for Coq. Mathematics in Computer Science 9(1), 41–62 (2015)

18. Carbonneaux, Q., Hoffmann, J., Reps, T., Shao, Z.: Automated resource analysis with Coq proof objects. In: CAV (2017)
19. Carbonneaux, Q., Hoffmann, J., Shao, Z.: Compositional certified resource bounds. In: POPL. pp. 467–478 (2015)
20. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: CAV. pp. 511–526 (2013)
21. Chatterjee, K., Fu, H., Murhekar, A.: Automated recurrence analysis for almost-linear expected-runtime bounds (2017)
22. Chatterjee, K., Novotný, P., Zikelic, D.: Stochastic invariants for probabilistic termination. In: POPL. pp. 145–160 (2017)
23. Chaudhuri, S., Dubhashi, D.P.: Probabilistic recurrence relations revisited. Theor. Comput. Sci. 181(1), 45–56 (1997)
24. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms (3. ed.). MIT Press (2009), http://mitpress.mit.edu/books/introduction-algorithms
25. Daumas, M., Lester, D., Martin-Dorel, É., Truffert, A.: Improved bound for stochastic formal correctness of numerical algorithms. Innovations in Systems and Software Engineering 6(3), 173–179 (Sep 2010)
26. Dubhashi, D.P., Panconesi, A.: Concentration of Measure for the Analysis of Randomized Algorithms. Cambridge University Press (2009), http://www.cambridge.org/gb/knowledge/isbn/item2327542/
27. Eberl, M.: Expected shape of random binary search trees. Archive of Formal Proofs 2017 (2017), https://www.isa-afp.org/entries/Random_BSTs.shtml
28. Eberl, M.: The number of comparisons in quicksort. Archive of Formal Proofs 2017 (2017), https://www.isa-afp.org/entries/Quick_Sort_Cost.shtml
29. Eberl, M.: Proving divide and conquer complexities in Isabelle/HOL. J. Autom. Reasoning 58(4), 483–508 (2017)
30. Fill, J.A., Mahmoud, H.M., Szpankowski, W.: On the distribution for the duration of a randomized leader election algorithm. Ann. Appl. Probab. 6(4), 1260–1283 (11 1996)
31. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press (2009)
32. Fuchs, M., Hwang, H., Zacharovas, V.: An analytic approach to the asymptotic variance of trie statistics and related structures. Theor. Comput. Sci. 527, 1–36 (2014), https://doi.org/10.1016/j.tcs.2014.01.024
33. Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, Inria Saclay Ile de France (2016), https://hal.inria.fr/inria-00258384
34. Hölzl, J.: Formalising semantics for expected running time of probabilistic programs. In: ITP. pp. 475–482 (2016)
35. Hölzl, J., Heller, A.: Three chapters of measure theory in Isabelle/HOL. In: ITP. pp. 135–151 (2011)
36. Hurd, J.: Formal Verification of Probabilistic Algorithms. Ph.D. thesis, Cambridge University (May 2003)
37. Kaminski, B.L., Katoen, J., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected run-times of probabilistic programs. In: ESOP. pp. 364–389 (2016)
38. Karp, R.M.: Probabilistic recurrence relations. J. ACM 41(6), 1136–1150 (1994)
39. Karpinski, M., Zimmermann, W.: Probabilistic recurrence relations for parallel divide-and-conquer algorithms. Tech. Rep. TR-91-067, International Computer

Science Institute (ICSI) (1991), https://www.icsi.berkeley.edu/ftp/global/pub/techreports/1991/tr-91-067.pdf
40. Kozen, D.: A probabilistic PDL. In: STOC. pp. 291–297 (1983)
41. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
42. Martin-Dorel, É., Melquiond, G.: Proving tight bounds on univariate expressions with elementary functions in Coq. J. Autom. Reasoning 57(3), 187–217 (2016)
43. Martínez, C., Roura, S.: Randomized binary search trees. J. ACM 45(2), 288–323 (1998)
44. McIver, A., Morgan, C., Kaminski, B.L., Katoen, J.: A new proof rule for almost-sure termination. PACMPL 2(POPL), 33:1–33:28 (2018), http://doi.acm.org/10.1145/3158121
45. Mitzenmacher, M., Upfal, E.: Probability and computing - randomized algorithms and probabilistic analysis. Cambridge University Press (2005)
46. Morgan, C., McIver, A., Seidel, K.: Probabilistic predicate transformers. ACM Trans. Program. Lang. Syst. 18(3), 325–353 (1996)
47. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press (1995)
48. Petcher, A., Morrisett, G.: The foundational cryptography framework. In: POST. pp. 53–72 (2015)
49. Prodinger, H.: How to select a loser. Discrete Mathematics 120(1), 149 – 159 (1993)
50. Ramsey, N., Pfeffer, A.: Stochastic lambda calculus and monads of probability distributions. In: POPL. pp. 154–165 (2002)
51. Ramshaw, L.H.: Formalizing the Analysis of Algorithms. Ph.D. thesis, Stanford University (1979)
52. Roura, S.: Improved master theorems for divide-and-conquer recurrences. J. ACM 48(2), 170–205 (2001)
53. Seidel, R., Aragon, C.R.: Randomized search trees. Algorithmica 16(4/5), 464–497 (1996)
54. Tassarotti, J.: Probabilistic recurrence relations for work and span of parallel algorithms. CoRR abs/1704.02061 (2017), http://arxiv.org/abs/1704.02061
55. van der Weegen, E., McKinna, J.: A machine-checked proof of the average-case complexity of quicksort in Coq. In: TYPES. pp. 256–271 (2008)
56. Young, N.: Answer to: Understanding proof of theorem 3.3 in Karp's probabilistic recurrence relations. Theoretical Computer Science Stack Exchange (2016), http://cstheory.stackexchange.com/q/37144

A Appendix

A.1 Parallel Quicksort

In this example, we bound the span arising from comparisons in parallel QuickSort. 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 [16]. 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[4]. This is no different from what we do when comparing sequential sorting algorithms by the number of comparisons they perform.

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 4. 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

[4] 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.

sublists are sorted by parallel recursive calls, and the final results are combined with the middle partition.

```
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' return 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. 4. Simplified version of code for parallel QuickSort.

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 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, $t(x) = a(x)+t(m(x))$ is then the *same* as for the sequential recurrence[5], 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 2 implies:

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

[5] 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.

A.2 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 [53] and randomized binary search trees [43] 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 tl x tr`, where `a` is auxiliary information, `tl` 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 5. 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.

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 [43] 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)];
                  mret (Node tt tl p tr)
  end.
```

In this format, we can apply Theorem 2 to analyze the height of the resulting tree. The first step in our proof, therefore, is to prove that these two constructions

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

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:
  uniq l ->
  eq_dist (rvar_of_ldist (add_list_random l Leaf))
          (rvar_of_ldist (rand_tree_rec l)).
```

From there, we can bound the height of trees generated by `rand_tree_rec` and convert them to bounds on `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 2, and we can re-use the exact same choice of a, m, and g_1 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:

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

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:
  forall l, uniq l -> rsize l = 10 ^ 10 ->
      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 §3 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.

A.3 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[6].

[6] Versions of this problem are a well-studied subject in the theory of distributed computing [41].

We consider a protocol that has been analyzed by several authors [30, 49]. 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 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 6. 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 [32] 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 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

```
Fixpoint binomial (n: nat) : ldist nat :=
  match n with
   | O => mret O
   | 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
  | O => 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. 6. Model of leader election protocol.

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 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}.