A Machine-checked Proof of the Average-case Complexity of Quick sort in Coq

James McKinna and Eelis van der Weegen

Department of Computer Science, University of Nijmegen, the Netherlands

Abstract. We describe a machine-checked proof of Quicksort's $O(n \log n)$ average-case complexity, developed using the Coq system. To represent the algorithm, a shallow embedding is used in which the algorithm is defined as a monadically expressed functional program, with both the counted operation and the monad turned into parameters. Profiling instrumentation is then transparently inserted into this otherwise straightforward algorithm definition by instantiating it with the right monads. The instrumentation is sufficient to support formal yet convenient derivation of key properties of the algorithm's behavior needed by the proof—a step usually omitted in existing proofs. Thus, we directly take advantage of the computational nature of the type theory on which Coq is based.

1 Introduction

Proofs of Quicksort's $O(n \log n)$ average-case complexity are included in many textbooks on computational complexity [4] [TODO: more]. This paper documents what the authors believe to be the first fully formal machine-checked version of such a proof, developed using the Coq proof assistant [16].

The formalization is based on the "paper proof" in [4], but adds a great deal of theory simply to handle all the details that are omitted in that original. In particular, the latter takes key assertions about the algorithm's behavior for granted. Indeed, it does not even formally define the algorithm in the first place. While this practice is common and perfectly reasonable for paper proofs intended for human consumption, it is a luxury we do not afford ourselves. Hence, a large part of this paper is devoted to the representations we use to define the algorithms, and how they support complexity proofs. These representations take the form of a shallow monadic embedding, and depend critically on the type theoretic nature of Coq's calculus [2].

Section 2 introduces the basic idea of the embedding, and demonstrates its use by briefly proving Quicksort's quadratic worst-case complexity with it. In section 3, the technique is expanded to handle the notion of the *expected value* of non-deterministic programs, which then lets us state the main theorem in section 4. Sections 5 and 6 detail the actual proof. Section 7 ends with conclusions and final remarks.

The Coq source files containing the entire formalization can be downloaded from `http://www.eelis.net/research/quicksort/`. The Coq version used is 8.2 beta 2.

2 A Shallow Monadic Embedding

Before the main theorem can even be formally stated, let alone proved, a formal definition of the algorithm is needed, expressed in a programming language with well-defined semantics, suitable for subsequent formal reasoning. Here, we exploit the computational nature of the type theory on which Coq is based. Rather than defining the syntax and semantics of a programming language from scratch (this would be a "deep" embedding) like we would have to do in, say, ZFC mathematics, we instead use the built-in lambda calculus as a functional programming language, and write the algorithm as a specially adorned lambda term. More specifically, we express the algorithm monadically, with both the monad and the counted operation (in our case, the elementwise comparison—the usual complexity metric for sorting algorithms) made into parameters. Expressed this way, a deterministic quicksort that simply selects the head of the input list as its pivot element, and uses two simple filter passes to partition the input list, looks as follows:

```
Variables (M : Monad) (T : Set) (le : T → T → M bool).
Definition gt (x y : T) : M bool := liftM negb (le x y).
Fixpoint filter (c : T → M bool) (l : list T) : M (list T) :=
  match l with
  | nil ⇒ ret nil
  | h :: t ⇒
    t' ← filter c t;
    b ← c h;
    ret (if b then h :: t' else t')
  end.
Program Fixpoint qs (l : list T){measure length l} : M (list T) :=
  match l with
  | nil ⇒ ret nil
  | pivot :: t ⇒
    lower ← filter (gt pivot) t >>= qs;
    upper ← filter (le pivot) t >>= qs;
    ret (lower ++ pivot :: upper)
  end.
```

Monad is a dependent record containing the (coercible) carrier of type $Set \to Set$, along with *bind* (infix: $>\!\!>\!\!=$) and *ret* (for "return") operations, and the three monad laws. For a general introduction to monadic programming and monad laws, see [17]. The notation $x \leftarrow y; z$, shorthand for $y >\!\!>\!\!= \lambda x \to z$ (where x may occur in z) is known as "do-notation".

We use Coq's **Program** facility [15] to cope with the fact that Quicksort's recursion is non-structural, by using the input list length as a measure. The proof obligations generated by **Program** for the above definition and measure are trivial enough for Coq to prove mostly by itself.

By instantiating the above definitions with the right monad, we can transparently insert comparison-counting instrumentation into the algorithm, which will prove to be sufficient to let us reason about its complexity. But before we do so, let us note that if the above definitions are instead instantiated with the identity monad and an ordinary elementwise comparison on T, then the monadic scaffolding melts away, and the result is equivalent to an ordinary non-instrumented, non-monadic version, suitable for extraction and correctness proofs. This means that while we will instantiate the definitions with less trivial monads to support our complexity proofs, we can take some comfort in knowing that the object of those proofs is, in a very concrete sense, the actual Quicksort algorithm (as one would write it in a functional programming language), rather than some idealized model thereof.

For reasons that will become clear in later sections, we compose the monad we will instantiate the above definitions with using a monad transformer called *MMT* (for "monoid monad transformer"), which piggybacks a monoid onto an existing monad.

Variables $(monoid : Monoid)$ $(monad : Monad)$.

Let $C_{MMT} : Set \to Set := monad \circ prod\ monoid$.

Let $ret_{MMT}\ (T : Set) : T \to C_{MMT}\ T := ret \circ pair\ (monoid\_zero\ monoid)$.

Let $bind_{MMT}\ (A\ B : Set)\ (a : C_{MMT}\ A)\ (ab : A \to C_{MMT}\ B) : C_{MMT}\ B :=$
$x \leftarrow a; y \leftarrow ab\ (snd\ x); ret\ (monoid\_mult\ monoid\ (fst\ x)\ (fst\ y), snd\ y)$.

Definition $MMT : Monad := Build\_Monad\ C_{MMT}\ bind_{MMT}\ ret_{MMT}$.

(In the interest of brevity, we omit proofs of the monad laws for *MMT* and all other monads defined in this paper. These proofs can all be found in the Coq code.)

We use *MMT* to piggyback the additive monoid on $\mathbb{N}$ onto the identity monad, and lift elementwise comparison into the resulting monad, which we call *SP* (for "simply-profiled").

Definition $SP : Monad := MMT\ (\mathbb{N}, 0, +)\ IdMonad$.

Definition $le_{SP}\ (x\ y : \mathbb{N}) : SP\ bool := (1, le\ x\ y)$.

When instantiated with this monad and comparison operation, *qs* produces the comparison count as part of its result.

Definition $qs_{SP} := qs\ SP\ le_{SP}$.

Eval *compute* **in** $qs_{SP}\ (3 :: 1 :: 0 :: 4 :: 5 :: 2 :: nil)$.
$= (16, 0 :: 1 :: 2 :: 3 :: 4 :: 5 :: nil)$

Defining *cost* and *result* as the first and second projection, respectively, we have the identities

$\forall\ x, cost\ (ret_{SP}\ x) = 0$,
$\forall\ x\ f, cost\ (bind_{SP}\ x\ f) = cost\ x + cost\ (f\ (result\ x))$,
$\forall\ x\ y, cost\ (le_{SP}\ x\ y) = 1$.

This very modest amount of machinery is sufficient for a straightforward proof of Quicksort's quadratic worst-case complexity.

Proposition. $qs\_worst : \forall\ l, cost\ (qs_{SP}\ l) \leqslant (length\ l)^2$.

[TODO: get rid of period after "Proposition" above]

Proof. The proof is by induction matching qs's recursion. For an empty input list, we have $cost\ (qs_{SP}\ nil) = cost\ (ret\ nil) = 0 \leqslant (length\ l)^2$. For a non-empty input list $(pivot :: t)$, the cost decomposes into

$$\begin{aligned}
&cost\ (filter\ (le\ pivot)\ t) + cost\ (qs_{SP}\ (result\ (filter\ (lt\ pivot)\ t))) + \\
&cost\ (filter\ (gt\ pivot)\ t) + cost\ (qs_{SP}\ (result\ (filter\ (gt\ pivot)\ t))) + \\
&cost\ (ret\ (result\ (qs_{SP}\ (result\ (filter\ (lt\ pivot)\ t))) +\!\!+ \\
&\quad pivot :: result\ (qs_{SP}\ (result\ (filter\ (gt\ pivot)\ t))))).
\end{aligned}$$

The *filter* costs are easily proved (by induction on t) to be $length\ t$ each. The cost of the final ret is 0 by definition. The induction hypothesis applies to the recursive qs_{SP} calls. Furthermore,

$$\begin{aligned}
&length\ (result\ (filter\ (le\ pivot)\ t)) + \\
&length\ (result\ (filter\ (gt\ pivot)\ t)) \leqslant length\ t
\end{aligned}$$

can easily be proved by induction on t, because the two predicates filtered on are mutually exclusive. Abstracting $filter\ (le\ pivot)\ t$ and its gt counterpart, this leaves

$$\begin{aligned}
&\forall\ (t\ flt\ flt' : list\ T), length\ flt + length\ flt' \leqslant length\ t \rightarrow \\
&\quad length\ t + (length\ flt)^2 + length\ t + (length\ flt')^2 + 0 \leqslant (S\ (length\ t))^2,
\end{aligned}$$

which is true by elementary arithmetic. □

We now expand the machinery in preparation of the average-case complexity proof.

3 Nondeterminism and Expected Values

The Quicksort algorithm used by the average-case proof we formalize differs from the one presented in the last section in two ways: it is nondeterministic, and uses a single three-way partition pass instead of two two-way filter passes. Combined, these two traits ensure that the $O(n \log n)$ average-case property holds not just averaged over all input lists, but for each individual input list as well; nondeterministic pivot selection avoids the pathological cases any deterministic pivot selection strategy inevitably suffers, while a single three-way partition pass avoids the pathological quadratic case that two two-way filter passes suffer for an input list consisting of elements all of the same value. This greatly simplifies things, because it means that the global bound will follow immediately if it is proved for an arbitrary input.

Hence, we define

```
Variables (T : Set) (M : Monad) (cmp : T → T → M comparison).
Fixpoint partition (pivot : T) (l : list T) : M (Partitioning T) :=
  match l with
  | nil ⇒ ret emptyPartitioning
  | h :: t ⇒
    b ← cmp h pivot;
    tt ← partition pivot t;
    ret (addToPartitioning b h tt)
  end.
Variable pick : ∀ (A : Set), ne_list A → M A.
Program Fixpoint qs (l : list T){measure length l} : M (list T) :=
  match l with
  | nil ⇒ ret nil
  | h :: t ⇒
    i ← pick [0 .. length t];
    let pivot := nth (h :: t) i in
    part ← partition pivot (remove (h :: t) i);
    low ← qs (part Lt);
    upp ← qs (part Gt);
    ret (low ++ pivot :: part Eq ++ upp)
  end.
```

Here, *comparison* is an enumeration with values Lt, Eq, and Gt. A *Partitioning* T is a function of type *comparison* $\to$ *list* T. An *ne_list* T is a non-empty list of T's. The functions *nth* and *remove* select and remove the nth element of a list, respectively.

Nondeterminism can now be achieved by instantiating these definitions with a suitable monad, along with a corresponding *pick* operation. A deterministic, non-instrumented version can still be obtained, simply by instantiating the definitions with the identity monad and a corresponding deterministic *pick* operation, like *head*.

Let us now consider what kind of nondeterminism monad would be suitable for reasoning about the expected value of a nondeterministic program. The list monad is commonly used to emulate nondeterministic computation. With the list monad, the program

$$x \leftarrow \mathit{pick}\ [0,1]; \mathbf{if}\ x = 0\ \mathbf{then}\ \mathit{ret}\ 0\ \mathbf{else}\ \mathit{pick}\ [1,2],$$

produces $[0,1,2]$ as its list of possible outcomes. Unfortunately, the information that 0 is a more likely outcome than 1 or 2, has been lost. Such relative probabilities are critical to the notion of an expected value: the expected value of the program above is $\mathit{avg}\ [0, \mathit{avg}\ [1,2]] = \frac{3}{4} \not\equiv 1 = \mathit{avg}\ [0,1,2]$. This makes list nondeterminism unsuitable for our purposes.

Using tree nondeterminism instead solves the problem.

Inductive $ne\_tree\ (T : Set) : Set :=$
$\quad | \ Leaf : T \to ne\_tree\ T$
$\quad | \ Node : ne\_list\ (ne\_tree\ T) \to ne\_tree\ T.$

Definition $ret_{ne\_tree}\{A : Set\} : A \to C\ A := Leaf.$

Fixpoint $bind_{ne\_tree}\ (A\ B : Set)\ (m : C\ A)\ (k : A \to C\ B) : C\ B :=$
$\quad$ **match** m **with**
$\quad | \ Leaf\ a \Rightarrow k\ a$
$\quad | \ Node\ ts \Rightarrow Node\ (ne\_list.map\ (\lambda x \Rightarrow bind_{ne\_tree}\ x\ k)\ ts)$
$\quad$ **end**.

Definition $M_{ne\_tree} : Monad := Build\_Monad\ ne\_tree\ bind_{ne\_tree}\ ret_{ne\_tree}.$

Definition $pick_{ne\_tree}\ (T : Set) : ne\_list\ T \to M_{ne\_tree}\ T$
$\quad := Node\ \circ\ ne\_list.map\ Leaf.$

(We use non-empty trees because are not interested in computations that produce no values at all, and using potentially empty trees would complicate the definition of a tree's average value below.)

With this monad and pick operation, the same program now produces the tree $Node\ [Leaf\ 0, Node\ [Leaf\ 1, Leaf\ 2]]$, which preserves the relative probabilities. The expected value now coincides with the weighted average of these trees:

Definition $ne\_tree.avg : ne\_tree\ \mathbb{R} \to \mathbb{R} := ne\_tree.fold\ id\ ne\_list.avg.$

Relative probabilities are also the reason we use an n-ary choice primitive rather than a binary one, because correctly emulating (that is, without skewing the relative probabilities) an n-ary choice by a sequence of binary choices is only possible when n is a power of two.

Because we are often interested in the expected value of a measure of the output of a program that does not return numbers, we further define

Definition $expec\ (T : Set)\ (f : T \to \mathbb{N}) : ne\_tree\ T \to \mathbb{R}$
$\quad := ne\_tree.avg\ \circ\ ne\_tree.map\ f.$

Thus, given a program P of type $M_{ne\_tree}\ (list\ bool)$, $expec\ length\ P$ denotes the expected length of the result list, if we interpret values of type $M_{ne\_tree}\ T$ as nondeterministically computed values of type T.

To form the monad with which we will instantiate *qs* for the main theorem, we now piggyback the additive monoid on $\mathbb{N}$ onto M_{ne_tree} using MMT, and call the result NDP (for "non-deterministically profiled"):

Definition $M_{NDP} : Monad := MMT\ (\mathbb{N}, 0, +)\ M_{ne\_tree}.$

Definition $le_{NDP}\ (x\ y : T) : M\ bool := ret_{ne\_tree}\ (1, le\ x\ y).$

Definition $qs_{NDP} := qs\ M_{NDP}\ le_{NDP}\ (lift\ pick_{ne\_tree}).$

When working with computations in a monad formed by transforming M_{ne_tree} using MMT, we are often interested in the expected value of a function of the piggybacked monoid. For this, we introduce *monoid_expec*.

Definition *monoid_expec* $(m : Monoid)$ $(f : m \to \mathbb{N})\{A : Set\}$
$: MMT\ m\ M_{ne\_tree}\ A \to \mathbb{R} := expec\ (f \circ fst)$.

Thus, we have $\forall\ t, expec\ cost\ t = monoid\_expec\ id\ t$.

Like *expec*, *monoid_expec* gives rise to several identities. One identity that we would like to highlight is

$monoid\_expec\_plus : \forall\ (m : Monoid)\ (h : m \to \mathbb{N})\ (A\ B : Set)$
$(f : MMT\ m\ M_{ne\_tree}\ A)\ (g : A \to MMT\ m\ M_{ne\_tree}\ B) :$
$(\forall\ x\ y \in f \to monoid\_expec\ h\ (g\ (snd\ x)) = monoid\_expec\ h\ (g\ (snd\ y))),$
$monoid\_expec\ h\ (f \gg\!= g) =$
$monoid\_expec\ h\ f + monoid\_expec\ h\ (g\ (snd\ (ne\_tree.head\ f)))),$

which states that if one transforms M_{ne_tree} using a monoid m, then for a monoid homomorphism h from m to the additive monoid on $\mathbb{N}$, *monoid_expec* h distributes over *bind*, provided that the expected monoid value of the right hand side does not depend on the computed value of the left hand side. Since *id* is a monoid homomorphism, *monoid_expec_plus* applies to *NDP* and *expec cost*. In section 5, we will use it with another monoid and homomorphism.

4 The Statement

Now that we have a solid definition of nondeterministic, instrumented quicksort, as well as the means to express the expected values of nondeterministic programs, the last thing needed before the main theorem can be stated, is the notion of big-O complexity. The definition we use is a standard textbook definition, except that we work with a separate measure function parameter:

Definition *measured_bigO* $(X : Set)$
$(m : X \to \mathbb{N})\ (f : X \to \mathbb{R})\ (g : \mathbb{N} \to \mathbb{R}) : Prop$
$:= \exists\ c, \exists\ n, \forall\ x, n \leqslant m\ x \to f\ x \leqslant c * g\ (m\ x)$.
Notation "**over** $m,\ f = O\ (g)$" := $(measured\_bigO\ m\ f\ g)$.

We now state the main theorem.

Theorem *qs_avg* : **over** *length*,
$expec\ cost \circ qs_{NDP} = O\ (\lambda n \Rightarrow n * \log_2\ n)$.

Thanks to the property discussed at the start of the previous section, the above follows as a corollary from the (slightly stronger) statement that

Theorem *qs_expec_cost* :
$\forall\ l, expec\ cost\ (qs_{NDP}\ l) \leqslant 2 * length\ l * S\ (\log_2 (length\ l))$,

the proof of which is described in the remainder of this paper.

5 Reduction to Pairwise Comparison Counts

The main idea in the proof is to reduce *qs_expec_cost* to a statement about the expected number of comparisons between specific pairs of elements from the input list.

If $X \equiv X_{I_0} \ldots X_{I_{n-1}}$ is the input list, with I a permutation of $[0 \ldots n-1]$ such that $X_0 \ldots X_n$ is sorted, then the expected number of comparisons between any X_i and X_j with $i < j$ is at most $2/S(j-i)$. In other words, the expected number of comparisons between two list elements is bounded by a simple function of the number of list elements that separate the two in the sort order. We prove this fact in the next section, but first show how *qs_expec_cost* follows from it.

Combined with the observation that the total expected number of comparisons equals the sum of the expected numbers of comparisons for each (i, j) in $IJ := \{(i, j) \in [0, \mathit{length}\ l) \mid i < j\}$, the property described above suggests breaking up the inequality into

$$\mathit{expec\ cost}\ (\mathit{qs}_{NDP}\ l) \leqslant \sum_{(i,j) \in IJ} \frac{2}{S(j-i)} \leqslant 2 * \mathit{length}\ l * S\ (\log_2 (\mathit{length}\ l)).$$

Proving the right inequality requires a bit of analysis involving the harmonic series. This part of the proof could be fairly directly transcribed from the paper proof, using the existing real-number theory in the Coq standard library, with few complications and additions. We refer the interested reader to the paper proof.

The left inequality is the challenging one. To bring it closer to the index summation, we first rewrite

$$\mathit{expec\ cost}\ (\mathit{qs}_{NDP}\ l) = \mathit{expec\ cost}\ (\mathit{qs}_{NDP}\ (\mathit{map}\ (\mathit{nth}\ (\mathit{sort}\ l))\ \mathit{li})),$$

where *sort* may be any sorting function (including *qs* itself), and where *li* is a permutation of $[0 \ldots n-1]$ such that $\mathit{map}\ (\mathit{nth}\ (\mathit{sort}\ l))\ \mathit{li} = l$ (such an *li* can easily be proven to exist).

Next, we introduce a specialized monad and comparison operation that go one step further in focusing specifically on these indices.

Definition $\mathit{Monoid}_U : \mathit{Monoid} := (\mathbb{N} * \mathbb{N}, \mathit{nil}, +\!\!+)$.

Definition $U : \mathit{Monad} := \mathit{MMT}\ \mathit{Monoid}_U\ M_{\mathit{ne\_tree}}$.

Definition $\mathit{lookup\_cmp}\ (x\ y : \mathbb{N}) : \mathit{comparison} :=$
$\mathit{cmp}\ (\mathit{nth}\ (\mathit{sort}\ l)\ x)\ (\mathit{nth}\ (\mathit{sort}\ l)\ y)$.

Definition $\mathit{cmp}_U\ (x\ y : \mathbb{N}) : U\ \mathit{comparison} :=$
$\mathit{ret}\ ((\textbf{if}\ x \leqslant y\ \textbf{then}\ (x, y)\ \textbf{else}\ (y, x)) :: \mathit{nil}, \mathit{lookup\_cmp}\ x\ y)$.

Definition $\mathit{qs}_U : \mathit{list}\ \mathbb{N} \to \mathit{list}\ \mathbb{N} := \mathit{qs}\ \mathit{cmp}_U\ \mathit{pick}_U$.

qs_U operates directly on a list of natural numbers representing indices into $\mathit{sort}\ l$. Comparison of indices is defined by comparison of the T values they denote in $\mathit{sort}\ l$. Furthermore, rather than producing a grand total comparison count the

way *NDP* does, U records every pair of compared indices, by using *MMT* with the free monoid over $\mathbb{N} * \mathbb{N}$ pairs, instead of the additive monoid on $\mathbb{N}$ we used up until now.

The goal can now be rewritten using

$$\begin{aligned}&\mathit{expec\ cost}\ (\mathit{qs}_{\mathit{NDP}}\ (\mathit{map}\ (\mathit{nth}\ (\mathit{sort}\ l))\ \mathit{li}))\\&\quad = \mathit{monoid\_expec\ length}\ (\mathit{qs}_U\ \mathit{li}) = \mathit{expec}\ (\mathit{length}\ \circ\ \mathit{fst})\ (\mathit{qs}_U\ \mathit{li}).\end{aligned}$$

The first equality is justified by a separate induction matching either side's qs's recursion, proving that counting comparisons as they appear is really the same as recording them in a list and then computing the length of the resulting list.

After rewriting once more, this time using

$$\mathit{expec\_fcmp} : \forall\ f\ g\ t, \mathit{expec}\ (f\ \circ\ g)\ t = \mathit{expec}\ f\ (\mathit{ne\_tree.map}\ g\ t),$$

the goal becomes

$$\mathit{expec\ length}\ (\mathit{ne\_tree.map\ fst}\ (\mathit{qs}_U\ \mathit{li})) \leqslant \sum_{(i,j)\in IJ} \frac{2}{S(j-i)}.$$

We now invoke another lemma which bounds a nondeterministically computed list's expected length by the expected number of occurrences of specific values in that list. More specifically, it states that

$$\begin{aligned}&\forall\ (X : \mathit{Set})\ (\mathit{fr} : X \to \mathbb{R})\ (q : \mathit{list}\ X)\ (t : \mathit{ne\_tree}\ (\mathit{list}\ X)) :\\&\quad (\forall\ x \in q, \mathit{expec}\ (\mathit{count}\ x)\ t \leqslant \mathit{fr}\ i) \to\\&\quad (\forall\ x \notin q, \mathit{expec}\ (\mathit{count}\ x)\ t = 0) \to\\&\quad \mathit{NoDuplicates}\ q \to \mathit{expec\ length}\ t \leqslant \sum_{i\in q} \mathit{fr}\ i.\end{aligned}$$

The last of the three subgoals thus generated states that there are no duplicate values in IJ, which trivially holds. The second subgoal generated is

$$\forall\ (i,j) \notin IJ, \mathit{expec}\ (\mathit{count}\ (i,j))\ (\mathit{ne\_tree.map\ fst}\ (\mathit{qs}_U\ \mathit{li})) = 0.$$

Rewriting this using *expec_fcmp* backwards, then rewriting the *expec* as a *monoid_expec*, and then generalizing the premise, results in the following property which is useful in its own right, as we will see later.

$$\begin{aligned}&\mathit{sound\_cmp\_expec\_0} : \forall\ i\ j\ \mathit{li}, (i \notin \mathit{li} \vee j \notin \mathit{li}) \to\\&\quad \mathit{monoid\_expec}\ (\mathit{count}\ (i,j))\ (\mathit{qs}_U\ \mathit{li}) = 0,\end{aligned}$$

The proof of *sound_cmp_expec_0*, which we won't show here, is by a separate induction matching qs's recursion.

This leaves but one subgoal which, expressed using a *monoid_expec*, reads

$$\forall\ (i,j) \in IJ, \mathit{monoid\_expec}\ (\mathit{count}\ (i,j))\ (\mathit{qs}_U\ \mathit{li}) \leqslant 2\ /\ S\ (j-i).$$

This is exactly to the property described at the beginning of this section. We prove it in the next section.

6 Finishing the Proof

To get a stronger induction hypothesis, we slightly generalize the goal to

$$\forall\ i\ j, i < j \to \forall\ (li : \mathit{list}\ \mathbb{N})\ (b : \mathbb{N}), \mathit{IndexSeq}\ b\ li \to \\ \mathit{monoid\_expec}\ (\mathit{count}\ (i,j))\ (\mathit{qs}_U\ li) \leqslant 2\ /\ S\ (j-i).$$

The $(i,j) \in IJ$ hypothesis is dropped, because the statement is also true if $(i,j) \notin IJ$, per *sound_cmp_expec_0*. The *IndexSeq* premise expresses that li is a permutation of $[b\ ...\ b + \mathit{length}\ l]$ (for some b).

Again, the proof is by induction matching qs's recursion. In the base case, li is nil, and the left side of the inequality reduces to 0. In the recursive case, qs unfolds:

$$\begin{array}{l}\mathit{monoid\_expec}\ (\mathit{count}\ (i,j))\ (\\ \quad \mathit{pi} \leftarrow \mathit{pick}\ [0\ ...\ n]; \\ \quad \textbf{let}\ \mathit{pivot} := \mathit{nth}\ \mathit{li}\ \mathit{pi}\ \textbf{in} \\ \quad \mathit{part} \leftarrow \mathit{partition}_U\ \mathit{pivot}\ (\mathit{remove}\ \mathit{li}\ \mathit{pi}); \\ \quad \mathit{low} \leftarrow \mathit{qs}_U\ (\mathit{part}\ \mathit{Lt}); \\ \quad \mathit{upp} \leftarrow \mathit{qs}_U\ (\mathit{part}\ \mathit{Gt}); \\ \quad \mathit{ret}\ (\mathit{low} +\!\!+ \mathit{pivot} :: \mathit{part}\ \mathit{Eq} +\!\!+ \mathit{upp}) \\ )\ \leqslant 2\ /\ S\ (j-i)\end{array}$$

Since cmp_U is deterministic, $\mathit{partition}_U$ is deterministic here as well. Furthermore, since we know exactly what monadic effects *partition* has in this case, we can split those effects off and then revert to simple uneffectful *filter* passes. Finally, the outer *monoid_expec* can be partially evaluated because of the immediately visible *pick*. Using these observations, the goal can be rewritten into a form that uses less monadic indirection:

$$\begin{array}{l}\mathit{avg}\ (\mathit{map}\ (\mathit{monoid\_expec}\ (\mathit{count}\ (i,j)) \circ (\lambda \mathit{pi} \Rightarrow \\ \quad \textbf{let}\ \mathit{pivot} := \mathit{nth}\ \mathit{li}\ \mathit{pi}\ \textbf{in} \\ \quad \textbf{let}\ \mathit{rest} := \mathit{remove}\ \mathit{li}\ \mathit{pi}\ \textbf{in} \\ \quad \mathit{ne\_tree.map}\ (\mathit{map\_fst}\ (+\!\!+ \mathit{map}\ (U.\mathit{unordered\_nat\_pair}\ \mathit{pivot})\ \mathit{rest}))\ (\\ \quad\quad \mathit{lower} \leftarrow \mathit{qs}_U\ (\mathit{filter}\ ((=\mathit{Lt}) \circ \mathit{lookup\_cmp}\ \mathit{pivot})\ \mathit{rest}); \\ \quad\quad \mathit{upper} \leftarrow \mathit{qs}_U\ (\mathit{filter}\ ((=\mathit{Gt}) \circ \mathit{lookup\_cmp}\ \mathit{pivot})\ \mathit{rest}); \\ \quad\quad \mathit{ret}\ (\mathit{lower} +\!\!+ (\mathit{pivot} :: \mathit{filter}\ ((=\mathit{Eq}) \circ \mathit{lookup\_cmp}\ \mathit{pivot})\ \mathit{rest}) +\!\!+ \mathit{upper}) \\ \quad)))\ [0\ ...\ n])\ \leqslant 2\ /\ S\ (j-i).\end{array}$$

Here, *map_fst* applies a function to a pair's first component.

We now distinguish between five different cases that can occur for the nondeterministically picked *pivot*; it can either be less than i, equal to i, between i and j, equal to j, or greater than j. Each case occurs a certain number of times, and has an associated expected number of (i,j) comparisons (coming either from the *map_fst* term representing the *partition* pass, or from the two recursive qs_U calls). To represent this split, we first rewrite the right side of the inequality to

$$(2\ /\ S\ (j-i) * (i-b) + 1 + 0 + 1 + 2\ /\ S\ (j-i) * (b+n-j))\ /\ S\ n.$$

This form reflects the facts that

- the case where *pivot* is less than i occurs $i - b$ times, and in each instance, the expected number of (i, j) comparisons is no more than $2 \,/\, S\ (j - i)$;
- the case where the *pivot* is equal to i occurs once, and in this case no more than a single (i, j) comparison is expected;
- in the case where *pivot* lies between i and j, the number of expected (i, j) comparisons is 0, and consequently it does not matter how many times this case occurs;
- the case where the *pivot* is equal to j occurs once, and in this case no more than a single (i, j) comparison is expected;
- the case where the *pivot* is greater than j occurs $b + n - j$ times, and in each instance, the expected number of (i, j) comparisons is no more than $2 \,/\, S\ (j - i)$.

With the right side of the inequality in this form, we unfold the *avg* application on the left into $sum\ (...) \,/\, S\ n$, and then cancel the division by $S\ n$ on both sides. Next, to actually realize the split, we apply a specialized lemma stating that

$$
\begin{array}{l}
\forall\ b\ i\ j\ X\ f\ n\ (li : list\ \mathbb{N}) \\
\quad (g : [0\,..\,n] \to U\ X), IndexSeq\ b\ li \to \\
\quad b \leqslant i < j < b + S\ n \to \forall\ ca\ cb, 0 \leqslant ca \to 0 \leqslant cb \to \\
\quad (\forall\ pi, nth\ li\ pi < i \to expec\ f\ (g\ pi) \leqslant ca) \to \\
\quad (\forall\ pi, nth\ li\ pi = i \to expec\ f\ (g\ pi) \leqslant cb) \to \\
\quad (\forall\ pi, i < nth\ li\ pi < j \to expec\ f\ (g\ pi) = 0) \to \\
\quad (\forall\ pi, nth\ li\ pi = j \to expec\ f\ (g\ pi) \leqslant cb) \to \\
\quad (\forall\ pi, j < nth\ li\ pi \to expec\ f\ (g\ pi) \leqslant ca) \to \\
\quad\quad sum\ (map\ (expec\ f\ \circ\ g)\ [0\,..\,n]) \leqslant \\
\quad\quad\quad ca * (i - b) + cb + 0 + cb + ca * (b + n - j).
\end{array}
$$

The proof of this lemma is somewhat tedious [TODO: because..].

Five subgoals remain after applying the above lemma—one for each of the listed cases. The first one reads

$$
\begin{array}{l}
\forall\ pi, \\
\quad \textbf{let}\ pivot := nth\ li\ pi\ \textbf{in} \\
\quad \textbf{let}\ rest := remove\ li\ pi\ \textbf{in} \\
\quad\quad pivot < i \to \\
\quad\quad monoid\_expec\ (count\ (i, j)) \\
\quad\quad\quad (ne\_tree.map\ (map\_fst\ (+\!\!+ map\ (U.unordered\_nat\_pair\ pivot)\ rest))\ (\\
\quad\quad\quad\quad foo \leftarrow qs_U\ (filter\ ((= Lt)\ \circ\ lookup\_cmp\ pivot)\ rest); \\
\quad\quad\quad\quad bar \leftarrow qs_U\ (filter\ ((= Gt)\ \circ\ lookup\_cmp\ pivot)\ rest); \\
\quad\quad\quad\quad ret\ (foo +\!\!+ (pivot :: filter\ ((= Gt)\ \circ\ lookup\_cmp\ pivot)\ rest) +\!\!+ bar))) \\
\quad\quad\quad \leqslant 2 \,/\, S\ (j - i).
\end{array}
$$

Rewriting using a lemma saying that

$$
\begin{array}{l}
\forall\ (A : Set)\ (g : m)\ (h : m \to \mathbb{N})\ (t : MMT\ m\ M_{ne\_tree}\ A) : \\
\quad MonoidHomomorphism\ h \to
\end{array}
$$

$$\begin{aligned}&\mathit{monoid\_expec}\ h\ (\mathit{ne\_tree.map}\ (\mathit{map\_fst}\ (\mathit{monoid\_mult}\ m\ g))\ t) =\\ &h\ g + \mathit{monoid\_expec}\ t,\end{aligned}$$

the head reduces to

$$\begin{aligned}&\mathit{count}\ (i,j)\ (\mathit{map}\ (U.\mathit{unordered\_nat\_pair}\ \mathit{pivot})\ \mathit{rest}) +\\ &\mathit{monoid\_expec}\ (\mathit{count}\ (i,j))\\ &\quad (\mathit{foo} \leftarrow \mathit{qs}_U\ (\mathit{filter}\ ((=\mathit{Lt})\ \circ\ \mathit{lookup\_cmp}\ \mathit{pivot})\ \mathit{rest});\\ &\quad \mathit{bar} \leftarrow \mathit{qs}_U\ (\mathit{filter}\ ((=\mathit{Gt})\ \circ\ \mathit{lookup\_cmp}\ \mathit{pivot})\ \mathit{rest});\\ &\quad \mathit{ret}\ (\mathit{foo} +\!\!+ (\mathit{nth}\ v\ \mathit{pi} :: \mathit{filter}\ ((=\mathit{Eq})\ \circ\ \mathit{lookup\_cmp}\ \mathit{pivot})\ \mathit{rest}) +\!\!+ \mathit{bar}))\\ &\leqslant 2\ /\ S\ (j-i).\end{aligned}$$

From $\mathit{pivot} < i$ and $i < j$, we have $\mathit{pivot} < j$. Since each of the comparisons in $\mathit{map}\ (U.\mathit{unordered\_nat\_pair}\ \mathit{pivot})\ \mathit{rest}$ involves the pivot element, it follows that none of them can represent comparisons between i and j. Hence, the first term vanishes. Furthermore, since $\mathit{count}\ (i,j)$ is a monoid homomorphism, *monoid_expec_plus* lets us distribute *monoid_expec*. Since the *ret* term does not produce any comparisons either (by definition), its *monoid_expec* term vanishes, too. What remains are the two recursive calls:

$$\begin{aligned}&\mathit{monoid\_expec}\ (\mathit{count}\ (i,j))\\ &\quad (\mathit{qs}_U\ (\mathit{filter}\ ((=\mathit{Lt})\ \circ\ \mathit{lookup\_cmp}\ \mathit{pivot})\ \mathit{rest})) +\\ &\mathit{monoid\_expec}\ (\mathit{count}\ (i,j))\\ &\quad (\mathit{qs}_U\ (\mathit{filter}\ ((=\mathit{Gt})\ \circ\ \mathit{lookup\_cmp}\ \mathit{pivot})\ \mathit{rest})) \leqslant 2\ /\ S\ (j-i)\end{aligned}$$

All indices in the lower filtered part denote elements that compare less than the element denoted by the pivot. Since the former precede the latter in $\mathit{sort}\ l$, it must be the case that the indices areless than *pivot*. By *sound_cmp_expec_0*, it follows that the first qs_U term will produce no (i,j) comparisons, so the first *monoid_expec* term vanishes as well, leaving

$$\begin{aligned}&\mathit{monoid\_expec}\ (\mathit{count}\ (i,j))\\ &\quad (\mathit{qs}_U\ (\mathit{filter}\ ((=\mathit{Gt})\ \circ\ \mathit{lookup\_cmp}\ \mathit{pivot})\ \mathit{rest})) \leqslant 2\ /\ S\ (j-i).\end{aligned}$$

We now compare $\mathit{nth}\ (\mathit{sort}\ l)\ i$ with $\mathit{nth}\ (\mathit{sort}\ l)\ \mathit{pivot}$.

- If the two are equal, then i will not occur in the *filter term*, allowing us to invoke *sound_cmp_expec_0* yet again.
- If $\mathit{nth}\ (\mathit{sort}\ l)\ i < \mathit{nth}\ (\mathit{sort}\ l)\ \mathit{pivot}$, then we must have $i < \mathit{pivot}$, contradicting the assumption that $\mathit{pivot} < i$.
- If $\mathit{nth}\ (\mathit{sort}\ l)\ i > \mathit{nth}\ (\mathit{sort}\ l)\ \mathit{pivot}$, then we apply the induction hypothesis. For this, it must be shown that the filter term is a proper contiguous index sequence, which we relegated to a lemma we won't show here.

This concludes the case where $i < \mathit{pivot}$. The case where $j < \mathit{pivot}$ is symmetric. The other cases use similar arguments.

The proof is now complete.

7 Final Remarks

In the interest of brevity, we have omitted innumerable details and lemmas in the description of the proof. Still, the parts shown are reasonably faithful to the actual formalization. For completeness, we briefly list some complicating factors that we have ignored. In some cases, resolving these took significant effort.

- We have pretended to have used ordinary natural numbers as indices into ordinary lists, completely ignoring the inescapable issues of index validity that could not be ignored in the actual formalization. There, we use vectors (lists whose size is part of their type) and bounded natural numbers in many places instead. Using these substantially reduces the amount of $i < \mathit{length}\ l$ proofs that need to be produced, converted, and passed around, but the issue is still far from painless.
- Using the **Program** facility to deal with quicksort's non-structural recursion was not completely as trivial as we made it out to be. To make proving the generated proof obligations possible, the types of *filter* and *partition* had to be sigma-decorated with modest length guarantees, because the opacity of the unspecified monad's *bind* operation prevented **Program** from being able to "see" the relation between the function's parameters and the arguments it passed in its recursive calls.
- The definitions produced by **Program** use clever recursion operators that translate a well-founded relation into a form that can be structurally recursed on. Consequently, corresponding induction principles are similarly clever. Using these induction principles for a function already defined with two layers of monadicity quickly became unwieldy. To deal with this, we defined a series of induction principles specialized for specific instances of qs, which put the recursive proof obligation in as simple terms as possible, with as little monadic indirection as possible, using among other things the observations regarding determinism of the comparison operation, as described in section 6.

For all the gruesome details, we refer the interested reader to the Coq source files.

In conclusion, we have found the shallow monadic embedding to be a straightforward, clean, and solid basis for reasoning about an algorithm's computational complexity. Calling it an embedding almost seems misplaced, since the algorithm definitions are essentially just monadically expressed functional programs in Coq's calculus. Furthermore, since monads and monad transformers have been used in functional programming for decades, the basic idea should be immediately clear to any functional programmer. Still, it is worth noting that this pleasing convergence of subject and object languages is due solely to the fact that Coq is based on type theory.

Quicksort turned out to be a good stress test for the technique, showing that it copes well with nondeterminism and nontrivial average-case bounds. The U monad shows that custom profiling monads can be defined to analyse very

algorithm-specific traits for a specific part of a larger complexity proof, which was crucial for this particular proof.

For future work, a logical next challenge is to try and formalize a (likely more complex) proof that the $O(n \log n)$ bound holds also for deterministic quicksorts (with sensible pivot selection strategies).

References

1. H. P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.
2. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.
3. R. L. Constable. Expressing computational complexity in constructive type theory. In *LCC '94: Selected Papers from the International Workshop on Logical and Computational Complexity*, pages 131–144, London, UK, 1995. Springer.
4. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.
5. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
6. C.A.R. Hoare. Quicksort. *The Computer Journal*, 5:10–15, 1962.
7. W.A. Howard. *The formulae-as-types notion of constructions*, pages 479–490. to H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism. 1980.
8. Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming*, pages 97–136, 1995.
9. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343. ACM, 1995.
10. P. Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Bibliopolis, 1984.
11. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
12. H.R. Nielson and F. Nielson. *Semantics with applications: a formal introduction*. John Wiley & Sons, Inc., 1992.
13. R. Sedgewick. The analysis of quicksort programs. *Acta Inf.*, 7:327–355, 1977.
14. Natarajan Shankar and Sam Owre. Principles and pragmatics of subtyping in PVS. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, WADT '99*, volume 1827 of *LNCS*, pages 37–52. Springer, sep 1999.
15. Matthieu Sozeau. Subset coercions in Coq. In Thorsten Altenkirch and Conor Mcbride, editors, *Types for Proofs and Programs*, LNCS, pages 237–252. Springer, 2007.
16. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.1 gamma*, November 2006. http://coq.inria.fr.
17. P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer, 1993.