

# A Machine-checked Proof of the Average-case Complexity of Quicksort in Coq

Eelis van der Weegen\* and James McKinna  
eelis@eelis.net, james.mckinna@cs.ru.nl

Institute for Computing and Information Sciences  
Radboud University Nijmegen  
Heijendaalseweg 135, 6525 AJ Nijmegen, The Netherlands

**Abstract.** As a case-study in machine-checked reasoning about the complexity of algorithms in type theory, we describe a proof of the average-case complexity of Quicksort in Coq. The proof attempts to follow a textbook development, at the heart of which lies a technical lemma about the behaviour of the algorithm for which the original proof only gives an intuitive justification.

We introduce a general framework for algorithmic complexity in type theory, combining some existing and novel techniques: algorithms are given a shallow embedding as monadically expressed functional programs; we introduce a variety of operation-counting monads to capture worst- and average-case complexity of deterministic and nondeterministic programs, including the generalization to count in an arbitrary monoid; and we give a small theory of expectation for such non-deterministic computations, featuring both general map-fusion like results, and specific counting arguments for computing bounds.

Our formalization of the average-case complexity of Quicksort includes a fully formal treatment of the ‘tricky’ textbook lemma, exploiting the generality of our monadic framework to support a key step in the proof, where the expected comparison count is translated into the expected length of a recorded list of all comparisons.

## 1 Introduction

Proofs of the  $O(n \log n)$  average-case complexity of Quicksort [1] are included in many textbooks on computational complexity [9, for example]. 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 [2].

The formalisation is based on the “paper proof” in [9], which consists of three parts. The first part shows that the total number of comparisons performed by the algorithm (the usual complexity metric for sorting algorithms) can be written as a sum of expected comparison counts for individual pairs of input list elements. The second part derives from the algorithm a specific formula for

---

\* Research carried out as part of the Radboud Master’s programme in “Foundations”

this expectation. The third and last part employs some analysis involving the harmonic series to derive the  $O(n \log n)$  bound from the sum-of-expectations.

Of these three parts, only the first two involve the actual algorithm itself—the third part is strictly numerical. While the original proof provides a thorough treatment of the third part, its treatment of the first two parts is informal in two major ways.

First, it never actually justifies anything in terms of the algorithm’s formal semantics. Indeed, it does not even formally define the algorithm in the first place, relying instead on assertions which are taken to be intuitively true. While this practice is common and perfectly reasonable for paper proofs intended for human consumption, it is a luxury we can not afford ourselves.

Second, the original proof (implicitly) assumes that the input list does not contain any duplicate elements, which significantly simplifies its derivation of the formula for the expected comparison count for pairs of individual input list elements. We take care to avoid appeals to such an assumption.

The key to giving a proper formal treatment of both these aspects lies in using an appropriate representation of the algorithm, capable of capturing its computational behaviour—specifically, its use of comparisons—in a way suitable for subsequent formal reasoning. The approach we take is to consider such operation-counting as a *side effect*, and to use the general framework of *monads* for representing side-effecting computation in pure functional languages. Accordingly we use a shallow embedding, in which the algorithm, here Quicksort, is written as a monadically expressed functional program in Coq. This definition is then instantiated with refinements of operation-counting monads to make the comparison count observable.

The embedding is introduced in section 2, where we demonstrate its use by first giving a simple deterministic monadic Quicksort definition, and then instantiating it with a simple operation counting monad that lets us prove its quadratic worst-case complexity.

For the purposes of the more complex average-case theorem, we then give (in section 3) a potentially-nondeterministic monadic Quicksort definition, and compose a monad that combines operation counting with nondeterminism, supporting a formal definition of the notion of the *expected* comparison count, with which we state the main theorem in section 4.

The next two sections detail the actual formalised proof. Section 5 corresponds to the first part in the original proof described above, showing how the main theorem can be split into a lemma (stated in terms of another specialized monad) giving a formula for the expected comparison count for individual pairs of input elements, and a strictly numerical part. Since we were able to fairly directly transcribe the latter from the paper proof, using the existing real number theory in the Coq standard library with few complications and additions, we omit discussion of it here and refer the interested reader to the paper proof.

Section 6 finishes the proof by proving the lemma about the expected comparison count for individual input list elements. Since this is the part where the original proof omits the most detail, and makes the assumption regarding dupli-

cate elements, and where we really have to reason in detail about the behaviour of the algorithm, it is by far the most involved part of the formalisation.

Section 7 ends with conclusions and final remarks.

The Coq source files containing the entire formalisation can be downloaded from <http://www.eelis.net/research/quicksort/>. We used Coq version 8.2.

*Related work* In his Ph.D thesis [12], Hurd presents an approach to formal analysis of probabilistic programs based on a comprehensive formalisation of measure-theoretic constructions of probability spaces, representing probabilistic programs using a state-transforming monad in which bits from an infinite supply of random bits may be consumed. He even mentions the problem of proving the average-case complexity of Quicksort, but leaves it for future work.

In [11], Audebaud and Paulin-Mohring describe a different monadic approach in which programs are interpreted directly as measures representing probability distributions. A set of axiomatic rules is defined for estimating the probability that programs interpreted this way satisfy certain properties.

Compared to these approaches, our infrastructure for reasoning about non-deterministic programs is rather less ambitious, in that we only consider finite expectation based on naïve counting probability, using a monad for nondeterminism which correctly supports weighted expectation. In particular, we do not need to reason explicitly with probability distributions.

A completely different approach to type-theoretic analysis of computational complexity is to devise a special-purpose type theory in which the types of terms include some form of complexity guarantees. Such an approach is taken in [4], for example.

## 2 A Shallow Monadic Embedding

As stated before, the key to giving a proper formal treatment of those parts of the proof for which the original contents itself with appeals to intuition, lies in the use of an appropriate representation of the algorithm. Indeed, we cannot even formally state the main theorem until we have both an algorithm definition and the means to denote its use of comparisons.

Since we are working in Coq, we already have at our disposal a full functional programming language, in the form of Coq’s CIC [3]. However, just writing the algorithm as an ordinary Coq function would not let us observe its use of comparisons. We can however see comparison counting as a *side effect*. As is well known and standard practice in functional languages such as Haskell, side effects can be represented using *monads*: a side-effecting function  $f$  from  $A$  to  $B$  is represented as a function  $A \rightarrow M B$  where  $M$  is a type constructor encapsulating the side effects. “Identity” and “composition” for such functions are given by *ret* (for “return”) of type  $A \rightarrow M A$  and *bind* (infix:  $\gg$ ) of type  $M A \rightarrow (A \rightarrow M B) \rightarrow M B$  satisfying certain identities (the *monad laws*). For a general introduction to monadic programming and monad laws, see [5]. Furthermore, we use Haskell’s “do-notation”, declared in Coq as follows

**Notation** "x <- y ; z" := (*bind* y ( $\lambda x : \_ \Rightarrow z$ ))

and freely use standard monadic functions such as:

$$\begin{aligned} \textit{liftM} &: \forall (M : \textit{Monad}) (A B : \textit{Set}), (A \rightarrow B) \rightarrow (M A \rightarrow M B) \\ \textit{filterM} &: \forall (M : \textit{Monad}) (A : \textit{Set}), (A \rightarrow M \textit{bool}) \rightarrow \textit{list} A \rightarrow M (\textit{list} A) \end{aligned}$$

Here, the Coq type *Monad* is a dependent record containing the (coercible) carrier of type *Set*  $\rightarrow$  *Set*, along with the *bind* and *ret* operations, and proofs of the three monad laws.

We now express Quicksort in this style, parameterizing it on both the monad itself and on the comparison operation. 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*  $\rightarrow$  *T*  $\rightarrow$  *M bool*).

**Definition** *gt* (*x y* : *T*) : *M bool* := *liftM* *negb* (*le* *x y*).

**Program Fixpoint** *qs* (*l* : *list T*) {**measure** *length l*} : *M (list T)* :=

```

match l with
| nil  $\Rightarrow$  ret nil
| pivot :: t  $\Rightarrow$ 
  lower  $\leftarrow$  filterM (gt pivot) t  $\gg=$  qs;
  upper  $\leftarrow$  filterM (le pivot) t  $\gg=$  qs;
  ret (lower ++ pivot :: upper)
end.

```

We use Coq’s **Program Fixpoint** facility [7] to cope with Quicksort’s non-structural recursion, specifying list length as an input measure function that is separately shown to strongly decrease for each recursive call. For this definition of *qs*, these proof obligations are trivial enough for Coq to prove mostly by itself.

For recursive functions defined this way, Coq does not automatically define corresponding induction principles matching the recursive call structure. Hence, for this *qs* definition as well as the one we will introduce in section 3, we had to define these induction principles manually. To make their use as convenient as possible, we further customized and specialized them to take advantage of specific monad properties. We will omit further discussion of these issues in this paper, and will henceforth simply say: “by induction on *qs*, ...”.

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 (which are included in the formalisation for completeness). 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 construct the monad with which we will instantiate the above definitions using a monad transformer [8] *MMT* (for “monoid monad transformer”), which piggybacks a monoid onto an existing monad by pairing.

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

**Let**  $C_{MMT} (T : Set) : Set := monad (monoid \times T)$ .

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

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

**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 now use *MMT* to piggyback the additive monoid structure 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 trivially have identities such as  $cost (ret_{SP} x) = 0$ ,  $cost (le_{SP} x\ y) = 1$ , and  $cost (x \gg_{SP} f) = cost\ x + cost (f (result\ x))$ . This very modest amount of machinery is sufficient for a straightforward proof of Quicksort’s quadratic worst-case complexity.

**Proposition.**  $qs\_worst : \forall l, cost (qs_{SP} l) \leq (length\ l)^2$ .<sup>1</sup>

*Proof.* The proof is by induction on *qs*. For  $l = nil$ , we have  $cost (qs_{SP} nil) = cost (ret\ nil) = 0 \leq (length\ l)^2$ . For  $l = h :: t$ , the cost decomposes into

$$\begin{aligned} & cost (filter (le\ h) t) + cost (qs_{SP} (result (filter (le\ h) t))) + \\ & cost (filter (gt\ h) t) + cost (qs_{SP} (result (filter (gt\ h) t))) + \\ & cost (ret (result (qs_{SP} (result (filter (le\ h) t)))) \text{ ++ } \\ & h :: result (qs_{SP} (result (filter (gt\ h) t))))). \end{aligned}$$

<sup>1</sup> We do not use big-O notation for this simple statement, as it would only obfuscate. Big-O complexity is discussed in section 4.

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

$$\text{length } (\text{result } (\text{filter } (le \ h) \ t)) + \text{length } (\text{result } (\text{filter } (gt \ h) \ t)) \leq \text{length } t,$$

because the two predicates filtered on are mutually exclusive. Abstracting the filter terms as  $ft$  and  $ft'$ , this leaves

$$\begin{aligned} \text{length } ft + \text{length } ft' &\leq \text{length } t \rightarrow \\ \text{length } t + (\text{length } ft)^2 + \text{length } t + (\text{length } ft')^2 + 0 &\leq (S (\text{length } t))^2, \end{aligned}$$

which is true by elementary arithmetic. □

We now extend the technique to prepare for the average-case proof.

### 3 Nondeterminism and Expected Values

The version of Quicksort used in the average-case complexity proof in [9] differs from the one presented in the last section in two ways. This is also reflected in our formalisation.

First, the definition of  $qs$  is modified to use a single three-way partition pass, instead of two calls to *filter*, thus avoiding the pathological quadratic behaviour which can arise when the input list does not consist of distinct elements.

Second, and more significantly, we use *nondeterministic* pivot selection, thus avoiding the pathological quadratic behaviour from which any deterministic pivot selection strategy inevitably suffers. While this means that we have proved our result for a subtly different presentation of Quicksort, this nevertheless follows the textbook treatment, in line with common practice.

These two modifications together greatly simplify the formalisation, because they remove the need to carefully track input distributions in order to show that ‘good’ inputs (for which the original deterministic version of the algorithm performs well) sufficiently outnumber ‘bad’ inputs (for which the original version performs poorly). They further ensure that the  $O(n \log n)$  average-case bound holds not just averaged over all possible input lists, but for each individual input list as well. In particular, it means that once we prove that the bound holds for an arbitrary input, the global bound immediately follows.

This also means that for a key lemma near the end of our proof, we can use straightforward induction over the algorithm’s recursive call structure, without having to show that given appropriately distributed inputs, the partition step yields lists that are again appropriately distributed. Such issues are a major technical concern in more ambitious approaches to average-case complexity analysis [10, for example] and to the analysis of probabilistic algorithms.

The second modification is based on a new monad (again defined using *MMT*, but this time transforming a nondeterminism monad) with which the new definition can be instantiated, capturing the *expected* comparison count.

The first modification is relatively straightforward. Instead of calling *filterM*, which uses a two-way comparison operation producing a monadic *bool*, we define a function *partition*. It takes a three-way comparison operation producing a monadic *comparison*, which is an enumeration with values *Lt*, *Eq*, and *Gt*. We represent the resulting partitioning by a function of type *comparison*  $\rightarrow$  *list T* rather than a record or tuple type containing three lists, because in the actual formalisation, this saves us from having to constantly map *comparison* values to corresponding record field accessors or tuple projections. This is only a matter of minor convenience; a record or tuple could have been used instead without problems.

**Variables** ( $T : \text{Set}$ ) ( $M : \text{Monad}$ ) ( $\text{cmp} : T \rightarrow T \rightarrow M \text{ comparison}$ ).

**Fixpoint** *partition* ( $t : T$ ) ( $l : \text{list } T$ ) :  $M (\text{comparison} \rightarrow \text{list } T) :=$   
**match**  $l$  **with**  
|  $nil \Rightarrow \text{ret } (const \text{ nil})$   
|  $h :: l' \Rightarrow$   
   $c \leftarrow \text{cmp } h \ t; f \leftarrow \text{partition } t \ l';$   
   $\text{ret } (\lambda c' \Rightarrow \text{if } c = c' \text{ then } h :: f \ c' \text{ else } f \ c')$   
**end.**

Next, we redefine *qs* to use *partition*, and have it take as an additional parameter a *pick* operation, representing nondeterministic selection of an element of a non-empty list of choices. An *ne\_list T* is a non-empty list of *T*'s, inductively defined in the obvious way.

**Variable** *pick* :  $\forall A : \text{Set}, \text{ne\_list } A \rightarrow M A.$

**Program Fixpoint** *qs* ( $l : \text{list } T$ ) {**measure** *length l*} :  $M (\text{list } T) :=$   
**match**  $l$  **with**  
|  $nil \Rightarrow \text{ret } nil$   
|  $- \Rightarrow$   
   $i \leftarrow \text{pick } [0 \dots \text{length } l - 1];$   
  **let**  $\text{pivot} := \text{nth } l \ i$  **in**  
   $\text{part} \leftarrow \text{partition } \text{pivot } (\text{remove } l \ i);$   
   $\text{low} \leftarrow \text{qs } (\text{part } Lt);$   
   $\text{upp} \leftarrow \text{qs } (\text{part } Gt);$   
   $\text{ret } (\text{low} \ ++ \ \text{pivot} :: \text{part } Eq \ ++ \ \text{upp})$   
**end.**

The functions *nth* and *remove* select and remove the *n*th element of a list, respectively.

Note that the deterministic Quicksort definition in section 2 could also have been implemented with a *partition* pass instead, which might well have made the worst-case proof even simpler. We chose not to do this, in order to emphasise that the properties the average-case proof demands of the algorithm rule out the naïve but familiar implementation using *filter* passes.

Nondeterminism can now be emulated by instantiating these definitions with a suitable monad and *pick* operation. A deterministic, non-instrumented version

can still be obtained, simply by using the identity monad and any deterministic *pick* operation, such as *head* or ‘median-of-three’ (not considered here).

Let us now consider what kind of nondeterminism monad would be suitable for reasoning about the expected value of a nondeterministic program like

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

When executed in the list monad (commonly used to emulate nondeterministic computation), this program 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  $\text{avg } [0, \text{avg } [1, 2]] = \frac{3}{4} \neq 1 = \text{avg } [0, 1, 2]$ . This makes list nondeterminism unsuitable for our purposes.

Using tree nondeterminism instead solves the problem: we introduce the type *ne\_tree* of non-empty trees, building on *ne\_list*:

**Inductive** *ne\_tree* ( $T : \text{Set}$ ) : *Set* :=

| *Leaf* :  $T \rightarrow \text{ne\_tree } T$   
| *Node* :  $\text{ne\_list } (\text{ne\_tree } T) \rightarrow \text{ne\_tree } T$ .

**Definition**  $\text{ret}_{\text{ne\_tree}} \{A : \text{Set}\} : A \rightarrow \text{ne\_tree } A := \text{Leaf}$ .

**Fixpoint**  $\text{bind}_{\text{ne\_tree}} (A B : \text{Set})$

$(m : \text{ne\_tree } A) (k : A \rightarrow \text{ne\_tree } B) : \text{ne\_tree } B :=$

**match**  $m$  **with**

| *Leaf*  $a \Rightarrow k a$

| *Node*  $ts \Rightarrow \text{Node } (\text{ne\_list.map } (\lambda x \Rightarrow \text{bind}_{\text{ne\_tree}} x k) ts)$

**end**.

**Definition**  $M_{\text{ne\_tree}} : \text{Monad} := \text{Build\_Monad } \text{ne\_tree } \text{bind}_{\text{ne\_tree}} \text{ret}_{\text{ne\_tree}}$ .

**Definition**  $\text{pick}_{\text{ne\_tree}} (T : \text{Set}) : \text{ne\_list } T \rightarrow M_{\text{ne\_tree}} T$

$:= \text{Node } \circ \text{ne\_list.map } \text{Leaf}$ .

We use non-empty trees because we do not consider partial functions, and using potentially empty trees would complicate the definition of a tree’s average value below. This is also why we used *ne\_list* for *pick*.

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

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

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.

To denote the expected value of a discrete measure  $f$  of the output of a program, we define

**Definition**  $\text{expec} (T : \text{Set}) (f : T \rightarrow \mathbb{N}) : \text{ne\_tree } T \rightarrow \mathbb{R}$

$:= \text{ne\_tree.avg} \circ \text{ne\_tree.map } f$ .

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  $T$ .

The function  $expec$  gives rise to a host of identities, such as

$$\begin{aligned}
& 0 \leq expec\ f\ t \\
& expec\ (\lambda x \Rightarrow f\ x + g\ x)\ t = expec\ f\ t + expec\ g\ t \\
& expec\ ((*c) \circ f) = (*c) \circ expec\ f \\
& (\forall x \in t \rightarrow f\ x \leq g\ x) \rightarrow expec\ f\ t \leq expec\ g\ t \\
& (\forall x \in t \rightarrow f\ x = c) \rightarrow expec\ f\ t = c \\
& (\forall x \in t \rightarrow f\ x = 0) \leftrightarrow expec\ f\ t = 0 \\
& expec\ f\ (t \ggg (ret \circ g)) = expec\ (f \circ g)\ t \\
& expec\ (f \circ g)\ t = expec\ f\ (ne\_tree.map\ g\ t) \tag{1}
\end{aligned}$$

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 “nondeterministically profiled”):

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

**Definition**  $cmp_{NDP}\ (x\ y : T) : M_{NDP}\ bool := ret_{ne\_tree}\ (1, cmp\ x\ y)$ .

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

We can now denote the expected comparison count for a  $qs_{NDP}$  application by  $expec\ cost\ (qs_{NDP}\ l)$ , and will use this in our statement of the main theorem in the next section.

But before we do so, we define a slight refinement of  $expec$  that specifically observes the monoid component of computations in monads formed by transforming  $M_{ne\_tree}$  using  $MMT$  (like  $NDP$ ).

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

Since  $cost = fst$ , we have  $expec\ cost\ t = monoid\_expec\ id\ t$ .

In addition to all the identities  $monoid\_expec$  inherits from  $expec$ , it has some of its own. One identity 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:

$$\begin{aligned}
& monoid\_expec\_plus : \forall (m : Monoid)\ (h : m \rightarrow (\mathbb{N}, 0, +)), \\
& monoid\_homo\ h \rightarrow \forall (A\ B : Set) \\
& (f : MMT\ m\ M_{ne\_tree}\ A)\ (g : A \rightarrow MMT\ m\ M_{ne\_tree}\ B) : \\
& (\forall x\ y \in f \rightarrow monoid\_expec\ h\ (g\ (snd\ x)) = monoid\_expec\ h\ (g\ (snd\ y))), \\
& monoid\_expec\ h\ (f \ggg g) = \\
& monoid\_expec\ h\ f + monoid\_expec\ h\ (g\ (snd\ (ne\_tree.head\ f))).
\end{aligned}$$

Since  $id$  is a monoid homomorphism,  $monoid\_expec\_plus$  applies to  $NDP$  and  $expec\_cost$ . In section 5, we will use  $monoid\_expec\_plus$  with another monoid and homomorphism.

## 4 The Statement

The last thing needed before the main theorem can be stated, is the notion of big-O complexity. We use the standard textbook definition, except that we make explicit how we measure inputs to  $f$ , namely with respect to a measure function  $m$ :

**Definition**  $bigO (X : Set) (m : X \rightarrow \mathbb{N}) (f : X \rightarrow \mathbb{R}) (g : \mathbb{N} \rightarrow \mathbb{R}) : Prop$   
 $:= \exists c n, \forall x, n \leq m x \rightarrow f x \leq c * g (m x).$

**Notation** “ $wrt m, f = O (g)$ ”  $:= bigO m f g.$

We now state the main theorem.

**Theorem**  $qs\_avg : wrt length, expec\_cost \circ qs_{NDP} = O (\lambda n \Rightarrow n * \log_2 n).$

Thanks to the property discussed at the start of the previous section,  $qs\_avg$  follows as a corollary from the stronger statement

$$qs\_expec\_cost : \forall l, expec\_cost (qs_{NDP} l) \leq 2 * length l * (1 + \log_2 (length l)),$$

the proof of which is described in the next two sections.

## 5 Reduction to Pairwise Comparison Counts

As described in the introduction, the key ingredient in the proof is a lemma giving a formula for the expected comparison count for individual pairs of input list elements, indexed a certain way. More specifically, if  $X \equiv X_{I_0} \dots X_{I_{n-1}}$  is the input list, with  $I$  a permutation of  $[0 \dots n - 1]$  such that  $X_0 \dots X_n$  is sorted, then the expected comparison count for any  $X_i$  and  $X_j$  with  $i < j$  is at most  $2/(1 + j - i)$ . In other words, the expected comparison count for two input 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 comparison count ought to equal the sum of the expected comparison count for each individual pair of input elements, the property described above suggests breaking up the inequality into

$$expec\_cost (qs_{NDP} l) \leq \sum_{(i,j) \in IJ} ecc\ i\ j \leq 2 * length l * (1 + \log_2 (length l)),$$

where  $IJ := \{(i, j) \in [0, length l] \mid i < j\}$ , and  $ecc\ i\ j := 2 / (1 + j - i)$ .

The right-hand inequality is a strictly numerical affair, requiring a bit of analysis involving the harmonic series. As stated before, this part of the proof was fairly directly transcribed from the paper proof, with few complications and additions, and so we will not discuss it.

The left inequality is the challenging one. To bring it closer to the index summation, we first write  $l$  on the left-hand side as  $map\ (nth\ (sort\ l))\ li$ , where  $sort$  may be any sorting function (including  $qs$  itself), and where  $li$  is a permutation of  $[0 \dots n - 1]$  such that  $map\ (nth\ (sort\ l))\ li = l$  (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**  $Monoid_U : Monoid := (list\ (\mathbb{N} \times \mathbb{N}), nil, ++)$ .

**Definition**  $U : Monad := MMT\ Monoid_U\ M_{ne\_tree}$ .

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

**Definition**  $unordered\_nat\_pair\ (x\ y : \mathbb{N}) : \mathbb{N} \times \mathbb{N} := if\ x \leq y\ then\ (x, y)\ else\ (y, x)$ .

**Definition**  $cmp_U\ (x\ y : \mathbb{N}) : U\ comparison := ret\ (unordered\_nat\_pair\ x\ y :: nil, lookup\_cmp\ x\ y)$ .

**Definition**  $qs_U : list\ \mathbb{N} \rightarrow list\ \mathbb{N} := qs\ U\ cmp_U\ pick_U$ .

The function  $qs_U$  operates directly on lists of indices into  $sort\ l$ . Comparison of indices is defined by comparison of the values they denote in  $sort\ l$ . Furthermore, rather than producing a grand total comparison count the way  $NDP$  does,  $U$  records every pair of indices compared, by using  $MMT$  with  $Monoid_U$ , the free monoid over  $\mathbb{N} \times \mathbb{N}$  pairs, instead of the additive monoid on  $\mathbb{N}$  we used until now.

We now rewrite

$$\begin{aligned} & expec\ cost\ (qs_{NDP}\ (map\ (nth\ (sort\ l))\ li)) \\ & = monoid\_expec\ length\ (qs_U\ li) = expec\ (length \circ fst)\ (qs_U\ li). \end{aligned}$$

The first equality expresses that the expected number of comparisons counted by  $NDP$  is equal to the expected length of the list of comparisons recorded by  $U$ . In the formalisation, this is a separate lemma proved by induction on  $qs$ . The second equality merely unfolds the definition of  $monoid\_expec$ .

After rewriting with identity **1** in section **3** on page **9**, the goal becomes

$$expec\ length\ (ne\_tree.map\ fst\ (qs_U\ li)) \leq \sum_{(i,j) \in IJ} ecc\ i\ j.$$

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

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

We end up with two subgoals, the first of which is

$$\forall (i, j) \notin IJ, \text{expec} (\text{count} (i, j)) (\text{ne\_tree.map fst} (q_{s_U} li)) = 0.$$

Rewriting this using identity 1 from section 3 in reverse, then rewriting the *expec* as a *monoid\_expec*, and then generalizing the premise, results in

$$\forall i j li, (i \notin li \vee j \notin li) \rightarrow \text{monoid\_expec} (\text{count} (i, j)) (q_{s_U} li) = 0 \quad (2)$$

which can be shown by induction on *qs*, although we will not do so in this paper. We *will* use this property again in the next section.

The second subgoal, expressed with *monoid\_expec*, becomes

$$\forall (i, j) \in IJ, \text{monoid\_expec} (\text{count} (i, j)) (q_{s_U} li) \leq \text{ecc } i j \quad (3)$$

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

## 6 Finishing the Proof

Again, the proof of (3) is by induction on *qs*. But to get a better induction hypothesis, we drop the  $(i, j) \in IJ$  premise (because as was shown in the last section, the statement is also true if  $(i, j) \notin IJ$ ), and add a premise saying *li* is a permutation of a contiguous sequence of indices.

$$\forall i j, i < j \rightarrow \forall (li : \text{list } \mathbb{N}) (b : \mathbb{N}), \text{Permutation } [b \dots b + \text{length } li - 1] li \rightarrow \text{monoid\_expec} (\text{count} (i, j)) (q_{s_U} li) \leq \text{ecc } i j.$$

In the base case, *li* is *nil*, and the left-hand side of the inequality reduces to 0. In the recursive case, *qs* unfolds:

$$\begin{aligned} & \text{monoid\_expec} (\text{count} (i, j)) ( \\ & \quad pi \leftarrow \text{pick } [0 \dots n - 1]; \\ & \quad \text{let } pivot := \text{nth } li \text{ } pi \text{ in} \\ & \quad part \leftarrow \text{partition}_U \text{ } pivot (\text{remove } li \text{ } pi); \\ & \quad lower \leftarrow q_{s_U} (part Lt); \\ & \quad upper \leftarrow q_{s_U} (part Gt); \\ & \quad \text{ret } (lower \text{ ++ } pivot \text{ :: part } Eq \text{ ++ upper}) \\ & ) \leq \text{ecc } i j. \end{aligned}$$

Since *cmp<sub>U</sub>* is deterministic, *partition<sub>U</sub>* is as well. Furthermore, since we know exactly what monadic effects *partition<sub>U</sub>* has, we can split those effects off and revert to simple effect-free *filter* passes. Finally, we rewrite using the following *monoid\_expec* identity:

$$\text{monoid\_expec } f (\text{pick } l \gg m) = \text{avg} (\text{map} (\text{monoid\_expec } f \circ m) l).$$

This way, the goal ends up in a form using less monadic indirection:

```

avg (map (monoid_expec (count (i, j)) o (λpi ⇒
  let pivot := nth li pi in
  let rest := remove li pi in
  let flt := λc ⇒ filter ((= c) o lookup_cmp pivot) rest in
  ne_tree.map (map_fst (λmap (unordered_nat_pair pivot) rest)) (
    lower ← qs_U (flt Lt);
    upper ← qs_U (flt Gt);
    ret (lower ++ pivot :: flt Eq ++ upper)
  ))) [0 ... n - 1]) ≤ ecc i j.

```

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

We now distinguish between five different cases that can occur for the non-deterministically picked `pivot` (which, because we are in the  $U$  monad, is an index). 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-hand side of the inequality to

$$\frac{ecc\ i\ j * (i - b) + 1 + 0 + 1 + ecc\ i\ j * (b + n - j)}{n}.$$

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  $ecc\ i\ j$ ;
- 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 hence it does not matter how often 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  $ecc\ i\ j$ .

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

$$\begin{aligned}
& \forall b\ i\ j\ X\ f\ n\ (li : list\ \mathbb{N}) \\
& (g : [0 \dots n - 1] \rightarrow U\ X),\ Permutation\ [b \dots b + length\ li - 1]\ li \rightarrow \\
& b \leq i < j < b + S\ n \rightarrow \forall ca\ cb,\ 0 \leq ca \rightarrow 0 \leq cb \rightarrow \\
& (\forall pi,\ nth\ li\ pi < i \rightarrow expec\ f\ (g\ pi) \leq ca) \rightarrow \\
& (\forall pi,\ nth\ li\ pi = i \rightarrow expec\ f\ (g\ pi) \leq cb) \rightarrow \\
& (\forall pi,\ i < nth\ li\ pi < j \rightarrow expec\ f\ (g\ pi) = 0) \rightarrow \\
& (\forall pi,\ nth\ li\ pi = j \rightarrow expec\ f\ (g\ pi) \leq cb) \rightarrow \\
& (\forall pi,\ j < nth\ li\ pi \rightarrow expec\ f\ (g\ pi) \leq ca) \rightarrow \\
& sum\ (map\ (expec\ f\ o\ g)\ [0 .. n]) \leq \\
& ca * (i - b) + cb + 0 + cb + ca * (b + n - j).
\end{aligned}$$

Five subgoals remain after applying this lemma—one for each listed case. The first one reads

$$\begin{aligned}
& \forall pi, \\
& \text{let } pivot := nth\ li\ pi \text{ in} \\
& \text{let } rest := remove\ li\ pi \text{ in} \\
& \quad pivot < i \rightarrow \\
& \quad monoid\_expec\ (count\ (i, j)) \\
& \quad \quad (ne\_tree.map\ (map\_fst\ (+map\ (unordered\_nat\_pair\ pivot)\ rest))\ ( \\
& \quad \quad \quad foo \leftarrow qs_U\ (filter\ ((= Lt) \circ lookup\_cmp\ pivot)\ rest); \\
& \quad \quad \quad bar \leftarrow qs_U\ (filter\ ((= Gt) \circ lookup\_cmp\ pivot)\ rest); \\
& \quad \quad \quad ret\ (foo ++ (pivot :: filter\ ((= Gt) \circ lookup\_cmp\ pivot)\ rest) ++ bar))) \\
& \leq ecc\ i\ j.
\end{aligned}$$

Since  $count\ (i, j)$  is a monoid homomorphism, we may rewrite using another lemma saying that

$$\begin{aligned}
& \forall (m : Monoid)\ (h : m \rightarrow (\mathbb{N}, 0, +)),\ monoid\_homo\ h \rightarrow \\
& \forall (g : m)\ (A : Set)\ (t : MMT\ m\ M_{ne\_tree}\ A), \\
& \quad monoid\_expec\ h\ (ne\_tree.map\ (map\_fst\ (monoid\_mult\ m\ g))\ t) = \\
& \quad h\ g + monoid\_expec\ h\ t.
\end{aligned}$$

This leaves

$$\begin{aligned}
& count\ (i, j)\ (map\ (unordered\_nat\_pair\ pivot)\ rest) + \\
& monoid\_expec\ (count\ (i, j)) \\
& \quad (foo \leftarrow qs_U\ (filter\ ((= Lt) \circ lookup\_cmp\ pivot)\ rest); \\
& \quad \quad bar \leftarrow qs_U\ (filter\ ((= Gt) \circ lookup\_cmp\ pivot)\ rest); \\
& \quad \quad ret\ (foo ++ (nth\ v\ pi :: filter\ ((= Eq) \circ lookup\_cmp\ pivot)\ rest) ++ bar)) \\
& \leq ecc\ i\ j.
\end{aligned}$$

From  $pivot < i$  and  $i < j$ , we have  $pivot < j$ . Since each of the comparisons in  $map\ (unordered\_nat\_pair\ pivot)\ rest$  involves the pivot element, it follows that none of them can represent comparisons between  $i$  and  $j$ . Hence, the first term vanishes. Furthermore,  $monoid\_expec\_plus$  lets us distribute  $monoid\_expec$  over the  $bind$  applications. 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:

$$\begin{aligned}
& monoid\_expec\ (count\ (i, j))\ (qs_U\ (filter\ ((= Lt) \circ lookup\_cmp\ pivot)\ rest)) + \\
& monoid\_expec\ (count\ (i, j))\ (qs_U\ (filter\ ((= Gt) \circ lookup\_cmp\ pivot)\ rest)) \\
& \leq ecc\ i\ j.
\end{aligned}$$

All indices in the first filtered list denote elements less than the element denoted by the pivot. Since the former precede the latter in  $sort\ l$ , it must be the case that these indices are all less than  $pivot$ . And since  $pivot < i$ , it follows that the first  $qs_U$  term will produce no  $(i, j)$  comparisons (using property (2) at the end of section 5 on page 12). Hence, the first  $monoid\_expec$  term vanishes, leaving

$$\begin{aligned} & \text{monoid\_expec } (\text{count } (i, j)) \\ & (\text{qs}_U (\text{filter } ((= \text{Gt}) \circ \text{lookup\_cmp } \text{pivot}) \text{ rest})) \leq \text{ecc } i \ j. \end{aligned}$$

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

- If the two are equal, then  $i$  will not occur in the *filter* term, and so (again) no  $(i, j)$  comparisons are performed.
- If  $\text{nth } (\text{sort } l) \ i < \text{nth } (\text{sort } l) \ \text{pivot}$ , then we must have  $i < \text{pivot}$ , contradicting the assumption that  $\text{pivot} < i$ .
- If  $\text{nth } (\text{sort } l) \ i > \text{nth } (\text{sort } l) \ \text{pivot}$ , then we apply the induction hypothesis. For this, it must be shown that filtering the list of indices preserves contiguity, which follows from the fact that the indices share the order of the elements they denote in *sort l*.

This concludes the case where  $\text{pivot} < i$ . The case where  $j < \text{pivot}$  is symmetric. The other three cases use similar arguments. The proof is now complete.

## 7 Final Remarks

In the interest of brevity, we have omitted lots of detail and various lemmas in the description of the proof. Still, the parts shown are reasonably faithful to the actual formalisation, with two notable exceptions.

First, we have pretended to have used ordinary natural numbers as indices into ordinary lists, completely ignoring issues of index validity that could not be ignored in the actual formalisation. 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  $i < \text{length } l$  proofs that need to be produced, converted, and passed around, but this solution is still far from painless.

Second, using the **Program** facility to deal with Quicksort’s non-structural recursion is not completely as trivial as we made it out to be. Since the recursive calls are nested in lambda abstractions passed to the *bind* operation of an unspecified monad, the relation between their arguments and the function’s parameters is not locally known, resulting in unprovable proof obligations. To make these provable, we  $\Sigma$ -decorated the types of *filter* and *partition* in the actual formalisation with modest length guarantees.

The formalised development successfully adopted from the original proof the idea of using a nondeterministic version of the algorithm to make the  $O(n \log n)$  bound hold for any input list, the idea of taking an order-indexed perspective to reduce the problem to a sum-of-expected-comparison-counts, and the use of the standard bound for harmonic series for the strictly numerical part. However, for the actual reduction and the derivation of the formula for the expected comparison count, the intuitive arguments essentially had to be reworked from scratch, building on the monadic representation of the algorithm and the various comparison counting/nondeterminism monads.

The shallow monadic embedding provides a simple but effective representation of the algorithm. Being parameterized on the monad used, it allows a single

definition to be instantiated either with basic monads (like the identity monad or bare nondeterminism monads) to get a non-instrumented version suitable for extraction and correctness proofs, or with *MMT*-transformed monads to support complexity proofs. Furthermore, since this approach lets us re-use all standard Coq data types and facilities, including the powerful **Program Fixpoint** command, the actual algorithm definition itself is reasonably clean.

We have shown that it is straightforward to give a fully formal treatment in type theory of a classical result in complexity theory. This clearly shows the utility and applicability of the general monadic approach we have developed.

## References

1. Hoare, C.: Quicksort. *The Computer Journal* **5** (1962) 10–15
2. The Coq Development Team: The Coq Proof Assistant Reference Manual – Version V8.2. (February 2009) <http://coq.inria.fr>.
3. Bertot, Y., Castéran, P.: *Coq’Art: Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science. Springer (2004)
4. Constable, R.L.: Expressing computational complexity in constructive type theory. In Leivant, D., ed.: *LCC ’94*. Volume 960 of LNCS., Springer (1995) 131–144
5. Wadler, P.: Monads for functional programming. In Jeuring, J., Meijer, E., eds.: *Advanced Functional Programming*. Volume 925 of LNCS., Springer (1995) 24–52
6. Sedgewick, R.: The analysis of quicksort programs. *Acta Inf.* **7** (1977) 327–355
7. Sozeau, M.: Subset coercions in Coq. In Altenkirch, T., McBride, C., eds.: *Types for Proofs and Programs*. Volume 4502 of LNCS. Springer (2007) 237–252
8. Liang, S., Hudak, P., Jones, M.P.: Monad transformers and modular interpreters. In: *POPL ’95*, ACM (1995) 333–343
9. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: *Introduction to Algorithms*, Second Edition. MIT Press (September 2001)
10. Schellekens, M.: *A Modular Calculus for the Average Cost of Data Structuring*. Springer (2008)
11. Audebaud, P., Paulin-Mohring, C.: Proofs of Randomized Algorithms in Coq. In Uustalu, T., ed.: *MPC’06*. Volume 4014 of LNCS., Springer (2006) 49–68
12. Hurd, J.: *Formal Verification of Probabilistic Algorithms*. PhD thesis, University of Cambridge (2002)