An Assertional Proof of the Stability and Correctness of Natural Mergesort §

K. Rustan M. Leino[1] and Paqui Lucio[2]

[1] Microsoft Research, Redmond, WA, USA
leino@microsoft.com
[2] The University of the Basque Country, 20080-San Sebastián, Spain
paqui.lucio@ehu.es

Abstract. We present a mechanically verified implementation of the sorting algorithm commonly known as *Natural Mergesort*. The implementation consists in a few methods specified in the contract style of pre- and post-conditions. In addition, methods are annotated with assertions that, both, explain how it works, and allows the automatic verification of the contract satisfaction. This program-proof is made using the state-of-the-art verifier *Dafny*. We verify not only the standard sortedness condition of the algorithm, but also that it performs a stable sort. Along the paper we provide —and explain— the complete text of the program-proof.

Keywords: Formal Methods, Verification, Software Engineering, Dafny, Natural Mergesort, Theorem Proving, Sorting Algorithms, Stability.

0 Introduction

Natural Mergesort ([4]) is a sorting algorithm for linear data structures (mainly, arrays and lists) that has been widely studied mainly due to its good properties. It has $Nlog(N)$ worst-case complexity and, even in the case of arrays, is slightly easier to code than heapsort. Further, it performs very well on input data that are already mostly sorted. Another good property is stability. A sorting algorithm is stable if it maintains the relative order of records with equal keys. The most obvious application of a stable algorithm is sorting using a different (primary, secondary, etc.) keys. The natural mergesort algorithm —taking advantage of the ascending and descending chains appearing in the input list— splits the data in as many ascending sublists as required. These sublists are then merged to produce the sorted output list. The first step of splitting the input into ascending sequences is performed by three mutually recursive operations. The second step uses the traditional merge of two lists, for merging all the lists (two by two) until a single list remains. The stability of each step is a subtle and

§ This work has been partially supported by the Spanish Project TIN2007-66523 and the Basque Projects S-PE12UN050 and GIU12/26.

strong property. Stability is, as we show in Sec. 3, stronger than the property of preserving the multiset of elements (from the input list to the sorted output list). Hence, stability, along with sortedness, strictly implies the correctness of sorting algorithms.

Recently, an Isabelle/HOL proof of the correctness and stability of natural mergesort has been published as a proof pearl in [9]. The author of [9], firstly, specifies the algorithm as a functional program and, then, formalizes and proves the desired properties using the proof-assistant Isabelle/HOL. The proof is extrinsic to the program and uses high-order constructions. This means that the proof is strongly based on two skillful ad-hoc induction schemes. The first one for handling the mutually recursive functions involved in the splitting of the input into ascending sequences. The second induction scheme is related to the merging of the ascending lists. Correctness and stability are deduced from auxiliary lemmas which are proved by means of these induction schemes and with the help of a subtle generalization of the predicate sorted. The definition of that generalization and the induction schemes require the power of higher-order logic.

In this paper we present an implementation of Natural Mergesort over an algebraic data type of lists. The code is enriched with its contract-based specification and a proof of its correctness and its stability. Our proof is assertional, i.e. it uses assert statements —inserted in the code— to enable the (fully) automatic verification. The assertions are first-order formulas that explain how and why the program works. In our opinion the proof is clear and elegant. The proof uses the state-of-the-art verifier Dafny [5], and it is made on the basis of some lemmas that ensure natural properties. Most of the proofs are inductive and use calculations [7] when appropriate. We believe that our program-proof is a simple and intuitive example of how a practical verification tool can be used by software developers with a minimum of familiarity with contract-based specifications and first-order assertions. We aim to contribute to the spread of the educational use of automatic tools in the development of formally verified software. We are convinced that this kind of examples are very useful for the introduction of formal software development methods and tools in software engineering courses.

Along the rest of the paper we give and explain in detail the complete text of the program-proof. Section 2 is devoted to the basic definitions and lemmas on which the verification of the algorithm relies. In Section 3 we provide all the methods that make up the implementation of the natural mergesort algorithm. We explain the assertional proof of each method. In Section 4 we provide the proof of the lemma ensuring that stability is a stronger property than the invariability of the elements in the input and output lists. Finally, we give some concluding remarks.

1 Preliminary definitions and lemmas

In this section, we give the basic definitions and lemmas. Of course, they were arising during the design of the program-proof using the language and verifier Dafny [5]. The Dafny programming language is imperative, sequential and sup-

ports user-defined algebraic datatypes, generic classes and many other features for object-oriented programming. The Dafny specification language includes the usual assertional language for pre-post contract, invariant, decreasing expressions for termination proofs, etc. Since Dafny is designed with the main purpose of facilitate the construction of correct code, Dafny notation is compact and easy to understand. For the sake of readability and conciseness, the Dafny proof language includes constructors for structuring proofs such as lemmas and the more recently calculation proofs [7]. Dafny automatically generates executable .NET code for verified programs.

1.0 Lists

We start defining a polymorphic data type of lists with the usual destructors functions of head and tail.

```
datatype List⟨T⟩ = Nil | Cons(head:T, tail:List⟨T⟩)
```

Over this data type, we define some common functions that enable us to specify the contracts of methods of our implementation in a natural way. By default in Dafny, functions do not generate code and can then be used only in specifications. To override this default, so that the compiler will generate code for a function, the function is declared with "**function method**" and similarly for predicates, which are boolean functions. The **lemma** declarations we will see later are like methods, but no code is generated for them.

```
function length⟨T⟩ (xs:List⟨T⟩): nat
{
    match xs
    case Nil ⇒ 0
    case Cons(_,t) ⇒ 1+length(t)
}

function append⟨T⟩ (xs: List⟨T⟩, ys:List⟨T⟩): List⟨T⟩
{
    match xs
    case Nil ⇒ ys
    case Cons(h,t) ⇒ Cons(h,append(t,ys))
}

function method reverse⟨T⟩ (xs: List⟨T⟩, acc: List⟨T⟩): List⟨T⟩
{
  match xs
  case Nil ⇒ acc
  case Cons(h,t) ⇒ reverse(t,Cons(h,acc))
}

function flatten⟨T⟩ (xxs: List⟨List⟨T⟩⟩): List⟨T⟩
{
    match xxs
    case Nil ⇒ Nil
    case Cons(h,t) ⇒ append(h,flatten(t))
}

function multiset_of⟨T⟩ (xs: List⟨T⟩): multiset⟨T⟩
{
    match xs
    case Nil ⇒ multiset{}
    case Cons(h,t) ⇒ multiset{h} ∪ multiset_of(t)
}
```

The function length is used only in decreasing expressions required for termination proofs. The remaining functions are mainly used in assertions. The function reverse is also called from real code, hence it have been declared as **function method**s. We will see later that the function append is also call from real code, but the call is in the actual parameter of a ghost variable. Ghost variables are not represented at run time, they are only used by the verifier. Hence, code for append is not required.

The following three natural lemmas on append and flatten have an easy proof by induction on their first argument xs. Indeed, they are automatically proved by Dafny. Hence, the three proofs (bodies) are empty, represented by {}. Dafny automatically sets up the induction hypothesis ans also heuristically identifies user-supplied properties whose proof may benefit from induction, see [6].

```
lemma AppendNil⟨T⟩ (xs: List⟨T⟩)
 ensures append(xs,Nil) = xs;
{}

lemma AssocAppend⟨T⟩(xs: List⟨T⟩,ys: List⟨T⟩,zs: List⟨T⟩)
 ensures append(xs,append(ys,zs)) = append(append(xs,ys),zs);
{}

lemma FlattenConsAppend⟨T⟩ (xs: List⟨T⟩, ys: List⟨T⟩, zzs: List⟨List⟨T⟩⟩)
 ensures flatten(Cons(append(xs,ys),zzs)) = append(xs,append(ys,flatten(zzs)));
{}
```

Next lemma easily follows from the asserted commutativity property of append and reverse, which is automatically proved (by induction on a).

```
lemma ReverseCons⟨T⟩ (xs: List⟨T⟩,rev: List⟨T⟩,x: T)
  requires xs = reverse(rev,Nil);
  ensures append(xs, Cons(x,Nil)) = reverse(Cons(x,rev),Nil);
{
  assert ∀ a,b,c • append(reverse(a,b),c) = reverse(a,append(b,c));
}
```

1.1 Sortedness

The rest of the program-proof is parametric in the type E of the elements of the list to be sorted, and also in an abstract (non-body) function that associates a key with each element of type E. Since function key is abstract, we implicitly assume that it is total. Rather than axiomatizing some order relation on E, we simply let the key be an integer number.

```
type E
function method key (e:E): int
```

Lists are ordered on the basis of that key. Hence, we define the predicates greater-than (GT), equal (EQ) and sorted as follows.

```
predicate method GT (x:E, y:E)  { key(x) > key(y) }
predicate method EQ (x:E, y:E)  { key(x) = key(y) }
predicate sorted (xs: List⟨E⟩)
{
  xs ≠ Nil ⟹ (∀ x • x in multiset_of(xs.tail) ⟹ ¬GT(xs.head,x))
              ∧ sorted(xs.tail)
}
```

Now, we prove two lemmas on (respectively) sorted lists of elements of type E and lists of sorted lists. The first of these requires induction, the second just needs a consideration of cases. Dafny proves both of them automaticaly. The sortedness-part of our correctness proof is based on these two lemmas.

```
lemma SortedAppend (xs: List⟨E⟩, u:E)
    requires sorted(xs);
    requires ∀ z • z in multiset_of(xs) ⟹ ¬GT(z,u);
    ensures sorted(append(xs,Cons(u,Nil)));
{}

lemma SortedConsList (ys: List⟨E⟩, xxs: List⟨List⟨E⟩⟩)
    requires sorted(ys);
    requires ∀ xs • xs in multiset_of(xxs) ⟹ sorted(xs);
    ensures ∀ xs • xs in multiset_of(Cons(ys,xxs)) ⟹ sorted(xs);
{}
```

1.2 Stability

The binary predicate stable characterizes the stability property as a binary relation on lists. For defining stable, we first introduce a function that filters all the elements of a given list that have the same key as a given element.

```
function filterEQ (e: E,xs: List⟨E⟩): List⟨E⟩
{
    match xs
    case Nil ⇒ Nil
    case Cons(h, t) ⇒ if EQ(e,h)
                      then Cons(h, filterEQ(e,t))
                      else filterEQ(e,t)
}

predicate stable(xs: List⟨E⟩, ys: List⟨E⟩)
{       ∀ x • filterEQ(x,xs) = filterEQ(x,ys)  }
```

The following two lemmas prove two basic properties of the function filterEQ that are useful for proving the stability property of our implementation. Lemma DistrFilterApp ensures that filtering is distributive with respect to append. Lemma NullFilter warrants that filtering w.r.t. an element, whose key does not appear in the given list, produces a null list. Both are automatically proved.

```
lemma DistrFilterApp(x:E,xs: List⟨E⟩,ys: List⟨E⟩)
    ensures filterEQ(x,append(xs,ys)) = append(filterEQ(x,xs),filterEQ(x,ys));
{}

lemma NullFilter (x:E,ys: List⟨E⟩)
    requires ∀ y • y in multiset_of(ys) ⟹ ¬EQ(x,y);
    ensures filterEQ(x,ys) = Nil;
{
}
```

On the basis of these two properties we give an easy calculation proof of the following lemma StableLifting , which states that whenever GT(zs.head,ws.head) and zs is sorted (non-decreasing), the list append(zs,ws) is stable-related to the list that results from lifting in append(zs,ws) the head of ws to the first position, i.e. the list append(Cons(ws.head,zs),ws. tail).

```
lemma StableLifting (zs: List⟨E⟩,ws: List⟨E⟩)
    requires zs ≠ Nil ∧ ws ≠ Nil ;
    requires GT(zs.head,ws.head);
```

```
    requires sorted(zs);
    ensures stable(append(zs,ws),append(Cons(ws.head,zs),ws.tail));
{
  forall x:E {
       calc {
            filterEQ(x,append(zs,ws));
            ={DistrFilterApp(x,zs,ws);}
            append(filterEQ(x,zs),filterEQ(x,ws));
            = // ws = Cons(ws.head,ws.tail)
            append(filterEQ(x,zs),filterEQ(x,Cons(ws.head,ws.tail)));
            =// definitions of filterEQ and append
            append(filterEQ(x,zs),
               append(filterEQ(x,Cons(ws.head,Nil)),filterEQ(x,ws.tail)));
            ={AssocAppend(filterEQ(x,zs),
                          filterEQ(x,Cons(ws.head,Nil)),
                          filterEQ(x,ws.tail));
            append(append(filterEQ(x,zs),filterEQ(x,Cons(ws.head,Nil))),
                   filterEQ(x,ws.tail));
            ={if EQ(x,ws.head) {NullFilter(x,zs);}
              else {AppendNil(filterEQ(x,zs));}}
            append(filterEQ(x,Cons(ws.head,zs)),filterEQ(x,ws.tail));
            ={DistrFilterApp(x,Cons(ws.head,zs),ws.tail);}
            filterEQ(x,append(Cons(ws.head,zs),ws.tail));
            =
            filterEQ(x, Cons(ws.head,append(zs,ws.tail)));
            }
            }
}
```

The proof is a calculation that has been parametrized in the universal variable x. We prove that the result of filtering (any) x through append(zs,ws) is equal to filtering x through append(Cons(ws.head,zs),ws.tail)). First, we use the previous lemma DistrFilterApp that ensures the distributive property of filtering w.r.t. append. Note that the used lemma is enclosed in curly-brackets after the symbol = which it helps to prove. Then, we unfold ws into "the cons of its head and its tail" and apply the definitions of append and filterEQ. After that, we apply the associativity of append (also previously established as a lemma). Then, depending on the case of EQ(x,ws.head), a different lemma allows us to reduce the first argument subexpression append(filterEQ(x,zs), filterEQ(x,Cons(ws.head, Nil))) to filterEQ(x,Cons(ws.head,zs)). Note that the preconditions GT(zs.head,ws.head) and sorted(zs) are crucial in this calculation step. In the last two steps, we use again DistrFilterApp and then the definition of the function append.

The last two preliminary lemmas state that the operation of appending a given list xs preserves stability whatever would be the hand-side where xs is appended. Each lemma StableAppendL and StableAppendR respectively considers the left and right hand-side for appending xs. It is worthy to note their different (though equivalent) kind of contracts.

```
lemma StableAppendL (xs:List⟨E⟩,ws:List⟨E⟩,ws':List⟨E⟩)
    ensures stable(ws,ws') ⟹ stable(append(xs,ws),append(xs,ws'));
{
    match xs
    {
    case Nil ⇒
    case Cons(x,t) ⇒  StableAppendL(t,ws,ws');
    }
}
lemma StableAppendR (ws:List⟨E⟩,ws':List⟨E⟩,xs:List⟨E⟩)
    requires stable(ws,ws');
    ensures stable(append(ws,xs),append(ws',xs));
```

```
{
    forall x:E { calc {
                        filterEQ(x,append(ws,xs));
                       ={DistrFilterApp(x,ws,xs);}
                        append(filterEQ(x,ws),filterEQ(x,xs));
                        = //by precondition
                        append(filterEQ(x,ws'),filterEQ(x,xs));
                       ={DistrFilterApp(x,ws',xs);}
                        filterEQ(x,append(ws',xs));
                        }
               }
}
```

The first lemma has an easy inductive proof. The calculation proof in the second lemma is easy to follow.

2 The code

In this section we explain the annotated methods that make up the implementation, and which are compiled into executable .NET code. Each body method contains the assertions that ensures the Dafny-verification of its contract.

The following method natural_mergesort is the basis of the our program-proof. The contract and proof of natural_mergesort is based on the fact that the conjunction of stability and sortedness is a stronger enough property for warranting the correctness of a sorting algorithm. Indeed, any two lists in a stable pair have exactly the same (multisef of) elements, as we prove in Section 3. [0]

```
method natural_mergesort (xs:List⟨E⟩) returns (ys:List⟨E⟩)
    ensures sorted(ys);
    ensures stable(xs,ys);
    ensures multiset_of(xs) = multiset_of(ys);
{
  var aux := sequences(xs);
  ys := mergeAll(aux);

    assert stable(flatten(aux),xs);
    EQMultisetsOfStables(xs,ys);          //Lemma
}
```

To check that natural_mergesort satisfies its contract we only need to inspect the specifications (contracts) of the two methods and the lemma involved in its body:

```
method sequences (xs:List⟨E⟩) returns (xxs:List⟨List⟨E⟩⟩)
    ensures ∀ zs • zs in multiset_of(xxs) ⟹ sorted(zs);
    ensures xxs ≠ Nil;
    ensures stable(flatten(xxs),xs);
⊞{...}
method mergeAll (xxs:List⟨List⟨E⟩⟩) returns (ys:List⟨E⟩)
    requires xxs ≠ Nil;
    requires ∀ zs • zs in multiset_of(xxs) ⟹ sorted(zs);
    ensures sorted(ys);
    ensures stable(ys,flatten(xxs));
⊞{...}
lemma EQMultisetsOfStables (xs:List⟨E⟩,ys:List⟨E⟩)
    requires stable(xs,ys);
    ensures multiset_of(xs) = multiset_of(ys);
⊞{...}
```

[0] In order to facilitate the view of the real code, we have indented the assertions and the lemma calls. We also use comments which are prefixed by //. Sometimes we also use comments to give illustrative, although unnecessary, assertions.

Let us check that natural_mergesort satisfies its contract whenever the three pieces above also satisfy their contracts. First, sequences has a trivial precondition and the preconditions of mergeAll follow directly from the postconditions of sequences. The sortedness postcondition of natural_mergesort follows from the postcondition of mergeAll and the same-elements postcondition follows from the lemma EQMultisetsOfStables. The stability postcondition of natural_mergesort, which is also a precondition of the lemma, follows from stable(flatten(aux),xs) which is a postcondition of sequences, and stable(ys, flatten(aux)), which is a postcondition of mergeAll. However, because of the way quantifiers and functions are involved, the Dafny verifier needs the hint that stable(flatten(aux),xs) also holds after the call to mergeAll, so we assert that condition explicitly.[1]

In the rest of this section, we separately concentrate in the verification of the methods sequences and mergeAll. The lemma EQMultisetsOfStables will be presented in the next section.

2.1 The method sequences

The method sequences is implemented by simultaneous recursion with the two methods ascending and descending. Bellow we depict the annotated body of sequences and the contract specification of ascending and descending. For now, we omit the three decreases clauses, which are related to their simultaneous recursion. Termination is explained at the end of this subsection.

```
method sequences (xs: List⟨E⟩) returns (xxs: List⟨List⟨E⟩⟩)
    ensures ∀ zs • zs in multiset_of(xxs) ⟹ sorted(zs);
    ensures xxs ≠ Nil;
    ensures stable(flatten(xxs),xs);
{
    match xs {
    case Nil ⇒ xxs := Cons(Nil,Nil);
    case Cons(h,t) ⇒
        match t {
        case Nil ⇒ xxs := Cons(Cons(h,Nil),Nil);
        case Cons(ht,tt) ⇒
          if GT(h,ht)
               { xxs := descending(ht,Cons(h,Nil),tt);

                  //by simultaneous induction hypothesis:
                  //assert stable(flatten(xxs),Cons(ht,append(Cons(h,Nil),tt)));
                  assert stable(Cons(ht,append(Cons(h,Nil),tt)),xs);
               }
          else { xxs := ascending(ht,Cons(h,Nil),Cons(h,Nil),tt);

                  //by simultaneous induction hypothesis
                  //assert stable(flatten(xxs),append(Cons(h,Nil),Cons(ht,tt)));
                  assert stable(append(Cons(h,Nil),Cons(ht,tt)), xs);
               }
        }
     }
}

method descending (min:E, grow: List⟨E⟩, xs: List⟨E⟩) returns (xxs: List⟨List⟨E⟩⟩)
    requires grow ≠ Nil ∧ sorted(grow);
    requires ¬GT(min,grow.head);
```

[1] The reason is that Dafny encodes functions, like filterEQ, as if they could depend on the heap even if they do not. This may change in a future version of Dafny.

```
    ensures ∀ zs • zs in multiset_of(xxs) ⟹ sorted(zs);
    ensures stable(flatten(xxs), append(Cons(min,grow),xs));
⊞{...}

method ascending (max:E, ghost grow:List⟨E⟩, shrink: List⟨E⟩, xs:List⟨E⟩)
                                                    returns (xxs:List⟨List⟨E⟩⟩)
    requires grow ≠ Nil ∧ sorted(grow);
    requires reverse(shrink,Nil) = grow;
    requires ∀ z • z in multiset_of(grow) ⟹ ¬GT(z, max);
    ensures ∀ zs • zs in multiset_of(xxs) ⟹ sorted(zs);
    ensures stable(flatten(xxs), append(grow,Cons(max,xs)));
⊞{...}
```

The first two postconditions of sequences are automatically inferred from the contracts of the invoked methods. Only the stability property needs an assert statement in each branch of the if-then-else. This assert, along with the respective induction hypothesis (which follows from the contracts), allows Dafny to prove the third postcondition stable(flatten(xxs),xs), by transitivity of the relation stable. It should be noted that this property is also automatically deduced.

Now, let us provide the code of the methods descending and ascending (for now, without decreases clauses).

```
method descending (min:E, grow:List⟨E⟩, xs:List⟨E⟩) returns (xxs:List⟨List⟨E⟩⟩)
    requires grow ≠ Nil ∧ sorted(grow);
    requires ¬GT(min,grow.head);
    ensures ∀ zs • zs in multiset_of(xxs) ⟹ sorted(zs);
    ensures stable(flatten(xxs), append(Cons(min,grow),xs));
{
  if xs ≠ Nil ∧ GT(min,xs.head)
       {
       xxs := descending(xs.head,Cons(min,grow),xs.tail);

         //by induction hypothesis
         //assert stable(flatten(xxs),
                         append(Cons(xs.head,Cons(min,grow)),xs.tail));
         StableLifting(Cons(min,grow),xs);
         //assert stable(append(Cons(xs.head,Cons(min,grow)),xs.tail),
                        append(Cons(min,grow),xs));
       }
  else {
        var aux := sequences(xs);
        xxs := Cons(Cons(min,grow),aux);

          SortedConsList(Cons(min,grow),aux);
          //by simultaneous induction hypothesis
          //assert stable(flatten(aux), xs);
          StableAppendL(Cons(min,grow),flatten(aux),xs);
          //assert stable(append(Cons(min,grow),flatten(aux)),
                          append(Cons(min,grow),xs));
          assert append(Cons(min,grow),flatten(aux))
                 = flatten(Cons(Cons(min,grow),aux));
          //assert flatten(xxs) = flatten(Cons(Cons(min,grow),aux));
       }
}
```

Almost all the assertions and lemma calls annotating the body of descending are designed for proving stability. The only exception is the call to the lemma SortedConsList (in the else-branch) which forces Dafny to check that Cons(min,grow) and every list in aux is sorted, which easily follows from the two preconditions of descending. Then, it infers that every member of the value taken by xxs —i.e. Cons(Cons(min,grow),aux))— is sorted. Hence, the first postcondition of descending is proved. Regarding the second (stability) postcondition, in the then-

branch, the induction hypothesis stable-relates the list flatten(xxs) to the list append(Cons(xs.head, Cons(min, grow)), xs.tail). The latter, by lemma StableLifting, is stable-related to append(Cons(min,grow),xs). Hence, the postcondition is achieved by the transitivity of the relation stable. A very similar reasoning is used in the else-branch. In this case, by induction hypothesis, the list flatten(xxs) is stable-related to xs. Then, by lemma StableAppendL, we can relate the two lists that result from respectively append flatten(xxs) and xs to the left hand-side of Cons(min,grow). Finally, we assert that the first component of such stable pair coincides with flatten(xxs) (for the current value of xxs). Hence, this list is also stable-related to the second component in the pair, as ensured by the second postcondition.

The method ascending is almost dual to descending, though there is a difference that is immediately apparent: the variable grow now is ghost and a new variable shrink is introduced. We next explain the reason for that change.

```
method ascending (max:E, ghost grow:List⟨E⟩, shrink: List⟨E⟩, xs:List⟨E⟩)
                                                    returns (xxs:List⟨List⟨E⟩⟩)
    requires grow ≠ Nil ∧ sorted(grow);
    requires reverse(shrink,Nil) = grow;
    requires ∀ z • z in multiset_of(grow) ⟹ ¬GT(z, max);
    ensures ∀ zs • zs in multiset_of(xxs) ⟹ sorted(zs);
    ensures stable(flatten(xxs), append(grow,Cons(max,xs)));
{
  if xs ≠ Nil ∧ ¬GT(max,xs.head)
      {
        assert ∀ xs,ys,z:E • z in multiset_of(append(xs,ys)) ⟺
                             z in multiset_of(xs) ∨ z in multiset_of(ys);
        //assert ∀ x • x in multiset_of(append(grow,Cons(max,Nil)))
                                        ⟹ ¬GT(x,max);
        SortedAppend(grow,max);
        ReverseCons(grow,shrink,max);

       xxs := ascending(xs.head,append(grow,Cons(max,Nil)),Cons(max,shrink),
                                                                xs.tail);
        //by induction hypothesis
        //assert stable(flatten(xxs),append(append(grow,Cons(max,Nil)),xs));
        AssocAppend(grow,Cons(max,Nil),xs);
        //assert append(grow,append(Cons(max,Nil),xs))
        //       = append(grow,Cons(max,xs));
      }
  else {
      var aux := sequences(xs);
      xxs := Cons(reverse(Cons(max,shrink),Nil),aux);

         ReverseCons(grow,shrink,max);
         SortedAppend(grow,max);
         //assert flatten(xxs) = flatten(Cons(append(grow,Cons(max,Nil)),aux));
         FlattenConsAppend(grow,Cons(max,Nil),aux);
         //assert flatten(Cons(append(grow,Cons(max,Nil)),aux))
         //         = append(grow,append(Cons(max,Nil),flatten(aux)))
         //         = append(grow,Cons(max,flatten(aux)));
         //by induction hypothesis: assert stable(flatten(aux), xs);
         assert stable(Cons(max,flatten(aux)),Cons(max,xs));
         StableAppendL(grow,Cons(max,flatten(aux)),Cons(max,xs));
         //assert stable(append(grow,Cons(max,flatten(aux))),
                       append(grow,Cons(max,xs)));
       }
}
```

The use of grow allows us to write a contract for ascending that reflects the natural duality to descending and enables a similar assertional proof. However,

shrink is used to bound the (else-branch) computation of xxs to linear complexity. That is, leaving aside shrink (and keeping grow to be non-ghost) the else-branch[2] assignment to xs should be

```
xxs := Cons(append(grow,Cons(max,Nil)),aux);
```

Doing so, the computation of xxs is quadratic on length(grow). We use the variable shrink to overcome this problem. The precondition states that grow is the reverse of shrink. The starting assert in the then-branch and the lemma calls to SortedAppend and ReverseCons are all designed to ensure that the actual parameter of the recursive call satisfies the preconditions that the method ascending imposes to the formal parameters grow and shrink. The lemma ReverseCons is also used in the else-branch for showing that reverse(Cons(max, shrink), Nil)) —which is the first element of xxs— is equal to append(grow,Cons(max,Nil)). The remaining details of the assertional proof for this method are very similar to the previously explained proof of descending. We have also provided commented asserts to further aid the interested reader.

Since sequences, descending and ascending are mutually recursive methods, their termination proofs must be jointly explained. A clause **decreases** xs would be perfect for the then-branch of sequences, but it does not work for the else-branch where sequences is called with the same parameter. Dafny allows —in decreases clauses— tuples of expressions and interprets them in lexicographic order. Hence, we add **decreases** xs,0 to the contract of sequences and **decreases** xs,1 to the contract of descending. This works since when the first component coincides, the second component decreases ($0 < 1$). Likewise **decreases** xs,1 ensures termination for ascending.

2.2 The method mergeAll

The method mergeAll is implemented as a repeated application of the following function method merge. For easy reading of the code, the result of the function appears as the last expression of every branch and is non-indented.

```
function method merge (xs: List⟨E⟩, ys: List⟨E⟩): List⟨E⟩
    requires sorted(xs) ∧ sorted(ys);
    ensures sorted(merge(xs,ys));
    ensures stable(merge(xs,ys), append(xs,ys));
{
    match xs
    case Nil ⇒  ys
    case Cons(hxs,txs) ⇒
        match ys
            case Nil ⇒     AppendNil(xs);
                           xs
            case Cons(hys,tys) ⇒
                if GT(hxs,hys)
                then
                     //by induction hypothesis:
                     //assert stable(merge(xs,tys),append(xs,tys));
                     //assert stable(Cons(hys,merge(xs,tys)),
                                     Cons(hys,append(xs,tys)));
```

[2] In the then-branch append(grow,Cons(max,Nil)) is the actual parameter of a ghost variable, whereas calculation is performed through Cons(max,shrink).

```
                    StableLifting(xs,ys);
                    //assert stable(Cons(hys,append(xs,tys)),
                    //                append(xs,Cons(hys,tys)));
                    //assert stable(Cons(hys,merge(xs,tys)),append(xs,ys));
                Cons(hys,merge(xs,tys))
                else
                    //by induction hypothesis:
                    //assert stable(merge(txs,ys),append(txs,ys));
                    //assert stable(Cons(hxs,merge(txs,ys)),
                    //                Cons(hxs,append(txs,ys)));
                    assert stable(Cons(hxs,merge(txs,ys)),append(xs,ys));
                 Cons(hxs,merge(txs,ys))
}
```

In the first case, the result of merge is xs that is sorted by precondition. Since the other list is null, AppendNil is used to ensure that the append of both list also yields xs. The first postcondition of merge (sortedness) is automatically proved by Dafny, also in the remaining two cases. The second case (then-branch) use the lemma StableLifting to prove that the lifting of the head (hys) of ys to the first position in Cons(hys,merge(xs,tys)) preserves stability. The third case (else-branch) is much easier. It is based in the following fact: any pair of lists constructed from a fixed head and respective tails taken from a pair of stable lists is also stable.

Now, the method for merging all the sorted lists into a unique sorted list is:

```
method mergeAll (xxs: List⟨List⟨E⟩⟩) returns (ys: List⟨E⟩)
    requires xxs ≠ Nil;
    requires ∀ zs • zs in multiset_of(xxs) ⟹ sorted(zs);
    ensures sorted(ys);
    ensures stable(ys,flatten(xxs));
    decreases length(xxs);
{
  match xxs  {
  case Cons(hxs,txs) ⇒
       match txs {
       case Nil ⇒
            ys := hxs;
                // assert flatten(xxs) = append(hxs,Nil);
                AppendNil(hxs);
       case Cons(htxs,ttxs) ⇒
                assert htxs in multiset_of(txs);
                assert length(xxs) = 1 + length(Cons(merge(hxs,htxs),ttxs));
            ys := mergeAll(Cons(merge(hxs,htxs),ttxs));
                calc {
                    stable(merge(hxs,htxs),append(hxs,htxs));
                    ⟹{StableAppendR(merge(hxs,htxs),append(hxs,htxs),
                                    flatten(ttxs));}
                    stable(append(merge(hxs,htxs),flatten(ttxs)),
                           append(append(hxs,htxs),flatten(ttxs)));
                    ⟹ {assert append(merge(hxs,htxs),flatten(ttxs))
                              = flatten(Cons(merge(hxs,htxs),ttxs));}
                    stable(flatten(Cons(merge(hxs,htxs),ttxs)),
                           append(append(hxs,htxs),flatten(ttxs)));
                    ⟹ {AssocAppend(hxs,htxs,flatten(ttxs));}
                    stable(flatten(Cons(merge(hxs,htxs),ttxs)),
                           append(hxs,append(htxs,flatten(ttxs))));
                    ⟹ //by induction hypothesis:
                       {assert stable(ys,
                                      flatten(Cons(merge(hxs,htxs),ttxs)));}
                    stable(ys,flatten(xxs));
                    }
         }
    }
}
```

The base case is trivial, but the lemma AppendNil is needed to prove that the flatten of the input list is identical to the output list. In the inductive case, we first prove an assertion that allows Dafny to infer that the second list is sorted,[3] so that it satisfies the precondition of merge. After that, we prove the assertion that ensures the termination of the method or, in other words, the validity of the clause decreases. The first postcondition is automatically inferred from the induction hypothesis. Finally, the second postcondition is proved through a succession of implications (with their explanations enclosed in curly-brackets) starting on the stability property ensured by the merge contract.

3 The lemma EQMultisetsOfStables

In this section we prove that stability is stronger that equivalence of multisets. That is, the lemma EQMultisetsOfStables ensures that any pair of stable lists has identical multisets. We first give a definition and two natural properties (lemmas) on the relationships between filtering and multisets. Using them, a short and elegant proof —for lemma EQMultisetsOfStables— is constructed.
The following function filtertNotEQ is dual to the previously defined function filtertEQ . Consequently, lemma ComplFilters is automatically proved.

```
function filterNotEQ (e: E, xs: List⟨E⟩): List⟨E⟩
{
    match xs
    case Nil ⇒ Nil
    case Cons(h, t) ⇒ if ¬EQ(e,h) then Cons(h,filterNotEQ(e,t))
                                  else filterNotEQ(e,t)
}

lemma ComplFilters (xs: List⟨E⟩)
    ensures ∀ z • multiset_of(xs) = multiset_of(filterEQ(z,xs))
                                  ∪ multiset_of(filterNotEQ(z,xs));
{}
```

The second auxiliary lemma FilterNotHeadPresStab ensures that the operation of filtering all the elements whose key is different to xs.head, from each of a pair of (non-empty) lists xs and ys, preserves stability.

```
lemma FilterNotHeadPresStab (xs: List⟨E⟩,ys: List⟨E⟩)
    requires stable(xs,ys);
    ensures xs ≠ Nil
            ⟹ stable(filterNotEQ(xs.head,xs),filterNotEQ(xs.head,ys));
{
    assert ∀ z,zs • filterEQ(z,filterNotEQ(z,zs)) = Nil;
    assert ∀ z, z', zs • ¬EQ(z,z') ⟹ filterEQ(z,filterNotEQ(z',zs))
                                   = filterEQ(z,zs);
    assert ∀ z, z', zs • EQ(z,z') ⟹ filterEQ(z,zs)
                                   = filterEQ(z',zs);
}
```

The proof reflects that the result is a logical consequence of three natural and general properties on the duality of filterNotEQ and filterEQ .

Finally, we can prove the main lemma of this section.

[3] This fact is automatically inferred for the first list, but not for the second.

```
lemma EQMultisetsOfStables (xs: List⟨E⟩, ys: List⟨E⟩)
    requires stable(xs, ys);
    ensures multiset_of(xs) = multiset_of(ys);
    decreases length(xs);
{
    if xs = Nil { assert ys = Nil; }
    else {
          assert ∀ z, zs • length(filterNotEQ(z, zs)) ≤ length(zs);
            // assert length(filterNotEQ(xs.head, xs)) < length(xs);
          calc {
               multiset_of(xs);
               ={ComplFilters(xs);}
               multiset_of(filterEQ(xs.head, xs))
               ∪ multiset_of(filterNotEQ(xs.head, xs));
               ={FilterNotHeadPresStab(xs, ys);
                    //by induction hypothesis
                  EQMultisetsOfStables(filterNotEQ(xs.head, xs),
                                       filterNotEQ(xs.head, ys));}
               multiset_of(filterEQ(xs.head, ys))
               ∪ multiset_of(filterNotEQ(xs.head, ys));
               ={ComplFilters(ys);}
               multiset_of(ys);
               }
     }
}
```

The proof is by induction on xs. The base case relies on the fact that any list stable-related to the empty list is also empty. The else-branch is the inductive case. First, in order to ensure the well-foundedness of the induction, we force Dafny to prove a more general property relating the lengths of any list and the result of filtering any arbitrary element. From that, and the definition of filterNotEQ, the well-foundedness condition (i.e. the commented assertion) is deduced by Dafny. Then, we prove the inductive case by a calculation that uses the lemma ComplFilters to split xs (and symmetrically ys at the end of the proof) into the union on two multisets. The first one contains all the elements in xs whose key coincides with the xs.head key and the second contains the remaining elements in xs. By stability, the first set coincides with all the elements in ys whose key coincides with the xs.head key. In order to prove that the second set also coincides with the remaining elements in ys, we use the lemma FilterNotHeadPresStab — which ensures the stability of the pair (filterNotEQ(xs.head,xs), filterNotEQ(xs.head,ys))— then the induction hypothesis does the remaining work. The last step, similar to the first one, is due to ComplFilters.

4 Conclusion

State-of-the-art tools for software verification are becoming more and more valuable from the practical point of view. Nowadays, tools can be applied to real world software with a reasonable effort and skill. Hoare logic and first-order assertions are in the foundations of programming. Indeed, fundamental courses in programming technology and programming languages include these topics. This kind of formal software development is, in general, well understood in the computer science community. Program-proofs —even when subtle properties are

involved— can be written in a clear and structured style. Software developers can isolate the required properties and concentrate once in its proof and separately in its application.

There are many formalizations of the natural mergesort algorithm —and also of different sorting algorithms (such as insertsort, quicksort, heapsort, etc)— in various systems, such as Coq [1], Isabelle/HOL [8], Why3 [2], ACL2 [3], etc. However, to the best of our knowledge, stability is only considered in [9] and in our assertional proof. Our program-proof —as it is written in this paper— consists of 352 (non-blank) lines. Many of them are dedicated to very common function definitions and obvious lemmas that can be automatically proved ot have an easy proof. The program-proof is composed by 9 functions (including 3 function methods), 4 predicates (including 2 predicate methods), 14 lemmas and 5 (pure) methods. The verified algorithm has $Nlog(N)$ complexity in the worst-case and linear complexity in the case of already sorted (or reverse-sorted) lists. The interested reader can access the file (and verify it online) at the following permalink: `http://rise4fun.com/Dafny/U5uw`.

Acknoledgments We are very grateful to Jean-Christophe Filliâtre for many valuable comments on a previous draft of this paper.

References

1. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.
2. Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.
3. Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.
4. Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.
5. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.
6. K. Rustan M. Leino. Automating induction with an SMT solver. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI12)*, pages 315–331. Springer-Verlag, 2012.
7. K. Rustan M. Leino and Nadia Polikarpova. Verified calculations. In Ernie Cohen and Andrey Rybalchenko, editors, *VSTTE*, volume 8164 of *Lecture Notes in Computer Science*, pages 170–190. Springer, 2013.
8. Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.
9. Christian Sternagel. Proof pearl - a mechanized proof of ghc's mergesort. *J. Autom. Reasoning*, 51(4):357–370, 2013.