An Assertional Proof of the Stability and Correctness of Natural Mergesort

K. RUSTAN M. LEINO, Microsoft Research
PAQUI LUCIO, The University of the Basque Country (UPV/EHU)

We present a mechanically verified implementation of the sorting algorithm *Natural Mergesort* that consists of a few methods specified by their contracts of pre/post conditions. Methods are annotated with assertions that allow 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 property, but also that the algorithm performs a stable sort. Throughout the article, we provide and explain the complete text of the program-proof.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Assertions, Invariants, Mechanical Verification

Additional Key Words and Phrases: Verification, theorem proving, formal methods, dafny, natural mergesort, software engineering, sorting, stability

ACM Reference Format:
K. Rustan M. Leino and Paqui Lucio. 2015. An assertional proof of the stability and correctness of natural mergesort. ACM Trans. Comput. Logic 17, 1, Article 6 (November 2015), 22 pages.
DOI: http://dx.doi.org/10.1145/2814571

6

1. INTRODUCTION

Natural Mergesort [Knuth 1973] is a sorting algorithm for linear data structures (arrays and lists) that has been widely studied mainly due to its good properties. It has $N \log N$ worst-case complexity and, even in the case of arrays, is slightly easier to code than heapsort. Furthermore, 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 different (primary, secondary, etc.) keys. Stability is, as we show in lemma EqMultisets (see Section 4.3), stronger than the property of preserving the multiset of elements (from the input list to the sorted output list). Hence, stability, along with sortedness, implies the correctness of sorting algorithms (including the permutation property).

Recently, Sternagel [2013] has published an Isabelle/HOL proof of the correctness and stability of natural mergesort as a proof pearl. Sternagel [2013] first specifies the algorithm as a functional program and then formalizes and proves the desired properties using the proof-assistant Isabelle/HOL. The proof is nonassertional and

---

This work has been partially supported by the Spanish Project TIN2013-46181-C2-2-R and the Basque Project GIU12/26 and grant UFI11/45.
Authors' addresses: K. R. M. Leino, Microsoft Research, One Microsoft Way, Redmond, WA 98052 USA; email: leino@microsoft.com; P. Lucio, Dpto de Lenguajes y Sistemas Informáticos, Facultad de Informática, Paseo Manuel de Lardizabal, 1, 20018-San Sebastián, Spain; email: paqui.lucio@ehu.eus.

DOI: http://dx.doi.org/10.1145/2814571

uses higher order constructions. Indeed, it 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, and the second induction scheme related to the merging of the ascending lists. Correctness and stability are deduced from auxiliary lemmas that 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 particular, the stability property is formalized in higher order logic.

More recently, de Gouw et al. [2014] discussed a semiautomated formal proof of the correctness and stability of two sorting algorithms on arrays: Counting sort and Radix sort. This proof is formalized using the theorem-prover KeY [Beckert et al. 2007]. The implementation code is written in Java. The specification is written (using the Java Modeling Language, JML) in an extension of first-order logic with *permutation predicates*, which have recently been added [Beckert et al. 2013] to the KeY system.

There are many other formalizations of the natural mergesort algorithm and also of different sorting algorithms (e.g., insertion sort, quicksort, heapsort, radix sort, etc.) in various systems, such as Coq [Bertot and Castéran 2004], Isabelle/HOL [Nipkow et al. 2002], Why3 [Filliâtre and Paskevich 2013], ACL2 [Kaufmann et al. 2000], KeY [Beckert et al. 2007], and others. However, to the best of our knowledge, stability is only considered in Sternagel [2013] and de Gouw et al. [2014], and in our assertional proof.

In this article, we present an implementation of natural mergesort over an algebraic datatype of lists. The code is enriched with its contract-based specification and a proof of its correctness and its stability. Our proof is assertional: That is, 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. The proof is supported by a few definitions that are easy to understand and a few lemmas that isolate useful properties. Moreover, only nontrivial lemmas have detailed proofs, and these are short and easy to read and understand. Hence, in our opinion, the presented proof is quite clear and elegant.

The program proof is implemented in the state-of-the-art verifier Dafny [Leino 2010]. The Dafny programming language supports a mixture of imperative, object-oriented programming and functional programming. In this article, we use mostly functions, methods, and algebraic datatypes. The Dafny specification language includes the usual assertional language for contracts of pre/post conditions, invariants, decreasing expressions for termination proofs, and the like. Since Dafny is designed with the main purpose of facilitating 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 constructs for structuring proofs such as lemmas and calculational proofs [Leino and Polikarpova 2014]. Dafny automatically generates executable .NET code for verified programs. The presented proof is made on the basis of some lemmas that ensure natural properties. Most of the proofs are inductive and use calculations 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 example is useful for the introduction of formal software development methods and tools in software engineering courses. In this article, we give and explain in detail the complete text of the program-proof.

Outline of the Paper. In Section 2, we introduce the algorithm of natural mergesort and give an example to demonstrate how it works. In Section 3, we present some preliminaries on Dafny. Section 4 is devoted to the basic definitions and lemmas on which the verification of the algorithm relies. This section is split into three subsections for lists, sortedness, and stability. In Section 5, we provide all the methods that make up the implementation of the natural mergesort algorithm. We explain the assertional proof of each method. Section 5 contains two subsections, each one focused on the code of one of the two main phases of the sorting algorithm. In Section 6, we report on the experience of using Dafny to verify the natural mergesort algorithm. Finally, we give some concluding remarks.

2. NATURAL MERGESORT

Mergesort is a classic algorithm for sorting lists and arrays. As invented by John von Neumann in 1945, it divides the input into two halves, recursively sorts each half, and finally merges the two sorted halves. There are several variants of this algorithm that share the idea of splitting the unordered input into two or more ordered slices and merging them. One outstanding variant is the natural mergesort algorithm [Knuth 1973], which, taking advantage of the ascending and descending chains appearing in the input list, splits the data in as many ascending sublists as required. These sub-lists are then merged to produce the sorted output list. For example, given the input list of numbers:

$$1, 2, 8, 6, 5, 1, 7, 6, 5, 4, 1, 0, 1, 3$$

it is first partitioned into the following five lists:

$$[1, 2, 8], [1, 5, 6], [0, 1, 4, 5, 6, 7], [1, 3], [\,]$$

Then, these lists are merged pairwise into:

$$[1, 1, 2, 5, 6, 8], [0, 1, 1, 3, 4, 5, 6, 7], [\,]$$

Another round of pairwise merging gives:

$$[0, 1, 1, 1, 1, 2, 3, 4, 5, 5, 6, 6, 7, 8], [\,]$$

and one final round of pairwise merging obtains the final list:

$$[0, 1, 1, 1, 1, 2, 3, 4, 5, 5, 6, 6, 7, 8]$$

In our implementation, the first step of splitting the input into ascending sequences is performed by three mutually recursive methods. These take one pass over all input elements, thus requiring time $O(N)$. The second step of merging the ascending sequences is also performed by three methods. Starting with, say, K ascending sequences (where K is no larger than N), method mergeAll performs rounds of merging operations. Each round applies the traditional merge to consecutive pairs of ascending sequences, thus reducing the number of ascending sequences by a factor of 2. After $\log K$ such rounds, only one ascending sequence remains. Since each round touches every element once, the algorithm has $O(N \log N)$ worst-case complexity. The stability of this sorting procedure is a subtle property that is stronger than the permutation property.

3. DAFNY

Dafny [Leino 2010] is an automatic program verifier for functional correctness. The Dafny programming language supports a mixture of imperative, object-oriented programming and functional programming. Dafny programs are statically verified for *total correctness*; that is, that every terminating execution satisfies its specification (*partial correctness*) and that every execution does indeed terminate. Dafny's program verifier works by translating a given Dafny program into the *intermediate verification language* Boogie [Barnett et al. 2006] in such a way that the correctness of the Boogie program implies the correctness of the Dafny program. Thus, the semantics of Dafny are defined in terms of Boogie. Boogie is a layer on which to build program verifiers for other languages. For example, the program verifiers VCC [Cohen et al. 2009] for C, AutoProof for Eiffel [Tschannen et al. 2015], and Spec# [Barnett et al. 2011] are built on the top of Boogie. The Boogie tool is used to generate first-order verification conditions that are passed to a logic reasoning engine. In particular, for Dafny, they are passed to the Satisfiability Modulo Theories (SMT) solver Z3 [de Moura and Bjørner 2008].

The Dafny Integrated Development Environment (IDE) is an extension of Microsoft Visual Studio (VS). The IDE is designed to reduce the effort required by the user to make use of the proof system. For example, the IDE runs the program verifier in the background, thus providing design time feedback. Also, verification error messages can have a lot of associated information, and the user can get information about the possible values of variables for a reported error using the Boogie Verification Debugger (BVD) [Le Goues et al. 2011] that is deeply integrated into the Dafny IDE. The interested reader is referred to Leino and Wüstholz [2014] for further information on the several ways that Dafny IDE helps to build verified software.

In the remainder of this section, we provide the preliminary notions of Dafny to facilitate the understanding of the paper.

The basic unit of a Dafny program is the **method**. A method is a piece of executable code with a head where multiple named parameters and multiple named results are declared. Dafny has built-in specification constructs for assertions, such as **requires** for preconditions, **ensures** for postconditions, **invariant** for loop invariants, and **assert** for inline assertions. Multiple **requires** have the same meaning as their conjunction in a single **requires**, and the same applies to **ensures**, **invariant**, and **assert**. Dafny does not generate invariants; they must be specified by the user, as do preconditions and postconditions. The most common use of inline assertions is to provide hints to the verifier whenever it cannot complete a correctness proof by itself. A hint is an assertion that the verifier is required to prove. Once the assertion is proved, it turns into a usable property for completing the correctness proof.

Dafny offers user-defined **function**s, built-in immutable types, and algebraic/ inductive **datatype**s. Dafny also provide mutable types, like arrays and objects, along with a notion of **class** for object-oriented programming (which is not used in this article). By default, in Dafny, functions can be used only in specifications, hence they do not generate code. To override this default so that the compiler will generate code for a function, the function is declared with **function method**. A **predicate** is a boolean function, and a **predicate method** is a predicate for which code is generated.

Dafny sets out to prove termination of all loops and of all recursion among methods and functions by employing **decreases** annotations for termination metrics. A decreases annotation specifies an expression whose value is compared for successive loop iterations and for caller and callee. If the successive values become strictly smaller according to a built-in well-founded order, then termination follows. Dafny has rules for guessing terminations metrics. If the guessed metric is not fine enough for proving termination, Dafny asks the user to provide one. Although the most common metrics are of type integer, other types of expressions also work, including, for example, (finite) sequences (whose well-founded order Dafny defines to be proper-prefix ordering). In particular, tuples are very useful as termination metrics. Dafny compare tuples lexicographically. The three methods in Figure 3 have a pair of integer expressions as metric. In the three cases, the first expression is a variable and the second is a constant (either 0 or 1), so a strict decrease happens if the value of the variable strictly goes down or if the variable remains unchanged and the caller has the 1 and the callee has the 0.

Dafny supports polymorphic types. That is, any class, inductive datatype, method, and function can have type parameters. An inductive (algebraic) **datatype** is a type whose values are created using a fixed set of constructors. In Section 4.1, we define the usual type of polymorphic lists with constructors Nil and Cons. There, we also declare the two usual destructors: head and tail. The **match** statement (respectively, expression) is provided to do pattern-matching in methods (respectively, functions) on values whose type is an inductive datatype. It binds the constructor parameters to the given names and executes (respectively, applies) the corresponding case. Dafny built-in immutable types include: **set**⟨T⟩, **multiset**⟨T⟩, and **seq**⟨T⟩, which respectively denote the types of finite sets, multisets, and sequences of elements of type T. Operations on sets and multisets include the usual operations of + (union), $*$ (intersection), and $-$ (set difference); comparison operators <= (subset), !! (disjointness), **in** (membership), |_| (cardinality); and the multiplicity of an element x in a multiset S, which is denoted by S[x]. Similarly, for sequences, Dafny provides + (concatenation), <= (prefix), **in** (membership), |_| (length), and many other operations. The expression S[j] denotes the element at index j of sequence S.[1]

Dafny distinguishes between *ghost* entities and *executable* entities. Ghost entities are used only during verification; the compiler omits them from the executable code. A variable x of some type T can be declared a ghost variable as:

ghost var x : T;

Also, parameters and results of methods can be declared to be ghost by preceding the declaration with the keyword **ghost**. As we alluded to earlier, a function is ghost by default and thus cannot be called from non-ghost code. The **lemma** declarations are like methods, but no code is generated for them (i.e., **lemma** is equivalent to ghost method). Ghost variables are useful whenever a computed value x is helpful for specification purposes, but the value x is not really needed in the executable code. For example, ghost variables can help to simplify specifications, to prove termination, and also to specify class invariants in OO programming. In the method ascending of Figure 4, we use a ghost variable (named grow) to facilitate the specification of the method and its correctness proof.

Although Dafny uses the powerful SMT-solver Z3, sometimes Dafny cannot complete a proof (i.e., Z3 cannot prove all verification conditions) by itself. Then, the user can provide assertions as hints: properties that, once verified, can be used for completing the proof. Indeed, "**assert** φ" tells Dafny to check that φ holds (whenever control reaches that part of the code) and to use the condition φ (as a lemma) to prove the verification conditions beyond this program point. To help the user construct proofs, in particular to guess hints, Dafny offers two features: the construct **assume** and the use of a declared (but yet not proved) **lemma**. Of course, a proof is not complete until all verification conditions have been discharged (i.e., all assume statements have been removed [or replaced by asserts], and all the lemmas have been proved). However, throughout the construction of a proof, we can introduce an assumed condition φ to check whether φ is the condition that Dafny needs to complete the proof. In other words, Dafny tries to complete the proof, assuming that φ is true, without having tried to prove φ. If Dafny succeeds, then "**assume** φ" should be changed to "**assert** φ" to force Dafny to prove φ. Now, if the assertion is violated, either φ is too strong a property (hence, the user should weaken it) or φ is a heavier weight property that must be separately proved. In

[1]The mentioned symbols are Dafny notation. For easy reading of the code snippets, we show them as the usual mathematical symbol (e.g., $\cup$ for union, $\in$ for membership, etc.).

the latter case, the user could declare a lemma (here, parameterized by one parameter x: T) like

```
lemma L(x: T)
  ensures ψ // desirable property
```

and use it to validate the assert φ. That is, if a concrete lemma call (namely L(a) for some a: T) placed just before "**assert** φ" works (in the sense that φ is satisfied), then the user could comment (or drop) the assert. After that, it remains to prove the lemm (i.e., to write its body). Reusability by instantiation (of the parameters) is an advantage of lemmas. In other words, the **assert** mechanism provides a non-instantiable lemma, which Dafny is able to prove without extra help.

As in other proof assistants (e.g., Isabelle/HOL and Coq) and verifiers (e.g., Why3 and KeY), Dafny allows proofs to be written in different styles and with different levels of description of the outcome of every logical transformation. Therefore, proof readability and easy checking by humans is part of the work of the Dafny user. For writing lemma proofs, Dafny also provides a notation that is easy to read and understand: *calculations* [Leino and Polikarpova 2014]. This notation was extracted from the *calculational method* [Backhouse 1995], whereby a theorem is established by a chain of formulas, each transformed in some way into the next. The relationship between successive formulas (e.e., equality, implication, double implication, etc.) is notated, or it can be omitted if it is the default relationship (equality). In addition, the hints (usually asserts or lemma calls) that justify a step can also be notated (in curly brackets after the relationship). Calculations are written inside the environment **calc**{ }.

To finish this section, we show in Figure 1 two different proofs for the same property: For all non-negative integers n, f(n) is divisible by 3, where f(n) = n∗n∗n + 2∗n. The first proof (**lemma** fnIsDivBy3) is divided into two cases, as indicated by an if statement (remember that a lemma in Dafny is nothing but a ghost method). The trivial case, for n=0, is automatically proved on Lines 8–9. The other case uses a calculational proof, one of whose hints (Line 21) calls the lemma recursively. This call is treated in accordance with programming rules: The precondition of the callee is checked, termination—that is, a strict decrease of the termination metric, which Dafny in this case supplies automatically as the parameter n—is checked, and then the postcondition can be assumed. In effect, this sets up a proof by induction, where the recursive call to the lemma obtains the inductive hypothesis for a smaller n. The second proof (**lemma** fnIsDivBy3') is equally convincing to Dafny but may take more head-scratching for a human to understand. Provided enough hints are supplied for Dafny to complete the proof, the tradeoff between clarity and clutter is up to the user and depends on how many details the user wants to show for human readers.

4. BASIC DEFINITIONS AND LEMMAS

In this section, we give the basic definitions and lemmas to be used in assertions and proofs.

4.1. Lists

We start defining a polymorphic datatype of lists with the usual destructor functions: head and tail.

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

```
function f (n: int): int
{ n*n*n + 2*n }

lemma fnIsDivBy3 (n: int)
  requires 0 ≤ n
  ensures f(n) % 3 = 0
{
  if n = 0 {
    // base case is proved automatically
  } else {
    calc {
      f(n) % 3;
    = // def. of f(n)
      (n*n*n + 2*n) % 3;
    = { assert f(n−1) = n*n*n − 3*n*n + 5*n − 3; }
      (f(n−1) + 3*n*n − 3*n + 3) % 3;
    = // distribute 3*
      (f(n−1) + 3*(n*n − n + 1)) % 3;
    = // modular arithmetic
      f(n−1) % 3;
    = { fnIsDivBy3'(n−1); } // invoke ind. hypothesis by calling the lemma recursively
      0;
    }
  }
}

lemma fnIsDivBy3' (n: int)
  requires 0 ≤ n
  ensures f(n) % 3 = 0
{
  if n ≠ 0 {
    fnIsDivBy3(n−1);
    assert f(n) = f(n−1) + 3*n*n − 3*n + 3;
  }
}
```

Fig. 1. Two different proofs for the same property. The first proof shows a pedantic number of details and also uses a calculation in the inductive step. The second proof has been written to be short and includes only the essential hints that Dafny needs to verify the lemma. A user can decide to keep as many details as are deemed helpful for human understanding and to elide those that seem more like clutter.

Over this datatype, we define some common functions that enable us to specify the contracts of methods of our implementation in a natural way.

```
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)
}
```

Remember that function methods generate code, whereas functions are used only in specifications and do not generate code. 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 executable code; hence, it is declared as **function method**. For efficiency reasons, we define a tail-recursive reverse that uses an accumulator. We will see later that the function append is also called from executable code, but the call is in the parameter of a ghost variable. Ghost variables are not represented at run time; they are only used by the verifier. Hence, compiled code for append is not required.

The function multiset_of enables expressing that a list is a permutation of another list, in particular, for the input and the output list of a sorting algorithm. We also use multiset_of to write universal assertions about all the elements in a list. In Dafny, there is also a built-in notion of set that could be used to write these assertions. We could write another function set_of⟨T⟩ (xs: List⟨T⟩): **set**⟨T⟩ and substitute set_of for multiset_of in all the assertions where the multiplicity of elements is irrelevant. However, we think that it is clearer and simpler to use only multisets instead of mixing sets and multisets.

The following three 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 and also heuristically identifies user-supplied properties whose proof may benefit from induction, see [Leino 2012].

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

The next lemma follows easily from the asserted commutativity property of append and reverse, which is automatically proved (by induction on xs) by Dafny.

```
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: List⟨T⟩ • append(reverse(a, b), c) = reverse(a, append(b, c));
}
```

4.2. 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 a function that associates a key with each element of type E. The function key is abstract (i.e., it does not have a defining body) and since it does not have any other declared precondition, the function is assumed to be total. The opaque type E and abstract function key can easily be instantiated in a module using Dafny's refinement features, but we do not concern ourselves with that in this article. Rather than axiomatizing some order relation on E, we simply let the key be an integer (alternatively, we could have declared it to be a real).

```
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 lemma just needs a consideration of cases; that is, for any list in the multiset of Cons(ys,xxs), either xs = ys or xs **in** multiset_of (xxs). Dafny proves both of them automatically. 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)
{}
```

4.3. Stability

The binary predicate stable characterizes the stability property as a binary relation on lists. For defining stable, we first introduce a function filterEQ that filters all the elements of a given list that have the same key as a given element. The predicate stable relates two lists whenever filtering both lists with respect to any element yields the same result. Hence, we use the predicate stable to relate the input and output of algorithmic operations on lists, in particular the sorting algorithm.

```
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 filterEQ is distributive with respect to append, and it is 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 says that filtering an element whose key does not appear in the given list produces a null list and has a trivial inductive proof:

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

We prove the following lemma, StableLifting, on the basis of the previous two properties. The contract of StableLifting states that, provided that zs.head is greater than ws.head and zs is sorted, then append(zs,ws) is stable-related to append(Cons (ws.head,zs),ws.tail). The first precondition zs ≠ Nil ∧ ws ≠ Nil is required for the existence of the two mentioned heads.

```
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) { // assert ∀ z • z in multiset_of(zs) ⟹ ¬EQ(x, z);
                                      NullFilter(x, zs);
                                      // assert filterEQ(x, zs) = Nil;
                                     }
                  // else {assert filterEQ(x, Cons(ws.head, Nil)) = Nil;}
                  }
            if EQ(x, ws.head)
            then append(append(Nil, filterEQ(x, Cons(ws.head, zs))), filterEQ(x, ws.tail))
            else append(append(filterEQ(x, zs), Nil), filterEQ(x, ws.tail));
            =     {
                  if ¬EQ(x, ws.head) {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));
            }
    }
}
```

The proof is a very detailed 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 the result of filtering x through append(Cons(ws.head,zs),ws.tail)). In the first step, the hint is a call to the previous lemma DistrFilterApp (enclosed in curly-brackets after the symbol =) that ensures the distributivity of filterEQ with regard to append. According to the precondition ws $\neq$ nil, hence, in the second step, we unfold ws into "the cons of its head and its tail." Third, we apply the definitions of append and filterEQ, as we have noted in comments. After that, we apply the associativity of append to the three lists passed as parameters of the lemma AssocAppend in the hint for this step. For the next calculation step, the preconditions GT(zs.head,ws.head) and sorted(zs) are crucial. There, depending on whether x and ws.head are equal or not, a different subexpression is reduced to Nil. When EQ(x,ws.head), according to the preconditions GT(zs.head,ws.head) and sorted(zs), we have that x is less than (hence, different from) any element in the list zs. So, the lemma call NullFilter (x,zs) proves that filterEQ (x,zs) is Nil. On the contrary case, it is trivial that filterEQ (x,Cons(ws.head, Nil)) is Nil. In the next step, the subexpression append(Nil,filterEQ(x,Cons(ws.head, Nil))) (in the **then** branch) trivially reduces to filterEQ(x,Cons(ws.head,zs)). To prove the equivalence between append(filterEQ(x,zs),Nil) (in the **else** branch) and filterEQ(x,Cons(ws.head,zs)), we apply the lemma AppendNil. Note that also ¬EQ(x,ws.head) is needed. The proof ends with another application of lemma DistrFilterApp.

The following lemma, StableAppend, states that append preserves stability. More precisely, given two pairs of lists, each pair related by stability, the result of appending the lists of each pair is also stable. The calculational proof of the lemma is easy to follow.

```
lemma StableAppend (xs: List⟨E⟩, xs': List⟨E⟩, ws: List⟨E⟩, ws': List⟨E⟩)
      requires stable(xs, xs') ∧ stable(ws, ws')
      ensures stable(append(xs, ws), append(xs', ws'))
{
   forall     z: E {
         calc  {
               filterEQ(z, append(xs, ws));
               =      { DistrFilterApp(z, xs, ws);  }
               append(filterEQ(z, xs), filterEQ(z, ws));
               =      // by precondition
               append(filterEQ(z, xs'), filterEQ(z, ws'));
               =      { DistrFilterApp(z, xs', ws');  }
               filterEQ(z, append(xs', ws'));
               }
   }
}
```

Lemma StableAppendL states that appending a given list xs to the left preserves stability. This is proved as a corollary of lemma StableAppend since the list xs is stable with itself.

```
lemma StableAppendL (xs: List⟨E⟩, ws: List⟨E⟩, ws': List⟨E⟩)
      requires stable(ws, ws')
      ensures stable(append(xs, ws), append(xs, ws'))
{
   StableAppend(xs, xs, ws, ws');
}
```

The last lemma, EqMultisets, ensures that stability is stronger than equivalence of multisets. In other words, any pair of stable lists has identical multisets.

```
lemma EqMultisets (xs: List⟨E⟩, ys: List⟨E⟩)
      requires stable(xs, ys)
      ensures multiset_of(xs) = multiset_of(ys)
{
   assert ∀ z: E, zs : List⟨E⟩ • multiset_of(filterEQ(z, zs))[z] = multiset_of(zs)[z];
}
```

The proof is based on the hint that the multiplicity (see Section 3) of any element z in the multiset of any list zs is preserved by filtering zs with regard to z. After proving this hint, Dafny uses it to prove the lemma since the stability precondition ensures identical filterings for xs and ys with regard to any element, and two multisets are equal if and only if every element has identical multiplicity on both multisets.

5. THE CODE

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

To facilitate the view of the executable code, we have indented the assertions and the lemma calls. We sometimes also use comments to give illustrative assertions, although they are unnecessary for automatic verification. Most of the commented assertions come from the **assume** annotations used to guess hints, as explained in Section 3, during construction of the proof.

The contract of our main method, natural_mergesort, is complete in the sense that it ensures both properties: correctness (split into sortedness and permutation) and

```
lemma EqMultisets (xs: List⟨E⟩,ys: List⟨E⟩)
    requires stable(xs,ys);
    ensures multiset_of(xs) = multiset_of(ys)
⊞{...}                                        // already proved

method natural_mergesort (xs: List⟨E⟩) returns (ys: List⟨E⟩)
    ensures sorted(ys)
    ensures multiset_of(xs) = multiset_of(ys)
    ensures stable(xs,ys)
{
  var aux := sequences(xs);
  ys := mergeAll(aux);
    assert stable(flatten(aux),xs);
    EqMultisets(xs,ys);                       // Lemma
}

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))
⊞{...}
```

Fig. 2. The method natural_mergesort and the contracts of the methods and lemma it invokes.

stability. So that, after any call to natural_mergesort, Dafny can assume the three postconditions for its parameters.

The proof of natural_mergesort is based on the fact that the conjunction of stability and sortedness is a strong enough property for warranting the correctness of a sorting algorithm, as ensured by lemma EqMultisets in Section 4.3.

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. All of them are depicted in Figure 2.

Let us check that natural_mergesort satisfies its contract whenever sequences, mergeAll, and EqMultisets also satisfy their contracts. First, sequences has a trivial precondition (no **requires** clause), 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 EqMultisets. The stability postcondition stable(xs,ys), which is the precondition of the lemma, is automatically inferred from stable(flatten(aux),xs) and stable(ys,flatten(aux)), according to the respective postconditions of sequences and 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. The reason is that Dafny encodes functions, like filterEQ, as if they could depend on the heap even if they do not.[2] Since mergeAll could change the heap, Dafny must check stable(flatten(aux),xs) again after the execution of mergeAll(aux).

This section is devoted to the annotated Dafny code that generates executable code. In the remainder of this section, we focus on the verification of the methods sequences and mergeAll.

[2]We hope that this may change in a future version of Dafny.

```
   method sequences (xs: List⟨E⟩) returns (xxs: List⟨List⟨E⟩⟩)
       ensures ∀ zs • zs in multiset_of(xxs) ⟹ sorted(zs)
 3     ensures xxs ≠ Nil
       ensures stable(flatten(xxs),xs)
       decreases xs,0
 6 {
     match xs
     case Nil ⇒ xxs := Cons(Nil, Nil);
 9   case Cons(h,t) ⇒
       match t
       case Nil ⇒ xxs := Cons(Cons(h, Nil), Nil);
12     case Cons(ht, tt) ⇒
         if GT(h,ht)
                 {
15               xxs := descending(ht,Cons(h, Nil), tt);
                   //by simultaneous induction hypothesis:
                   //assert stable(flatten(xxs),Cons(ht,append(Cons(h, Nil), tt)));
18                 assert stable(Cons(ht,append(Cons(h, Nil), tt)),xs);
                 }
          else {
21               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)));
24                 assert stable(append(Cons(h, Nil),Cons(ht, tt)),xs);
                 }
   }
27
   method descending (min: E,grow: List⟨E⟩,xs: List⟨E⟩) returns (xxs: List⟨List⟨E⟩⟩)
       requires grow ≠ Nil ∧ sorted(grow)
30     requires ¬GT(min,grow.head)
       ensures ∀ zs • zs in multiset_of(xxs) ⟹ sorted(zs)
       ensures stable(flatten(xxs),append(Cons(min,grow),xs))
33     decreases xs,1
   ⊞{...}

36 method ascending (max: E, ghost grow: List⟨E⟩, shrink: List⟨E⟩,xs: List⟨E⟩)
                                                    returns (xxs: List⟨List⟨E⟩⟩)
       requires grow ≠ Nil ∧ sorted(grow)
39     requires reverse(shrink, Nil) = grow
       requires ∀ z • z in multiset_of(grow) ⟹ ¬GT(z,max)
       ensures ∀ zs • zs in multiset_of(xxs) ⟹ sorted(zs)
42     ensures stable(flatten(xxs),append(grow,Cons(max,xs)))
       decreases xs,1
   ⊞{...}
```

Fig. 3. The method sequences and the contracts of descending and ascending.

5.1. The method sequences

The method sequences is implemented by mutual recursion with respect to the two methods ascending and descending. In Figure 3, we depict the annotated body of sequences and the contract specifications of ascending and descending. The annotated bodies of ascending and descending are shown in Figure 4. Remember that ghost variable grow in descending is only used by the verifier. We discuss this ghost variable later. 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 calls in sequences, but it does not work for the mutually recursive calls where sequences is called with the same parameter. Remember that Dafny allows—in **decreases** clauses—tuples of expressions and interprets them in lexicographic order. Hence, we add **decreases** xs,0 (line 5) to the contract of sequences and **decreases** xs,1 to the contract of descending (Line 33) and ascending (Line 43). This works because for calls where the first components coincide, the second component decreases.

The first two postconditions of sequences (Lines 2 and 3 in Figure 3) are automatically inferred from the contracts of the invoked methods. Only the stability property (Line 4) needs an assert statement in each branch of the if-then-else (Lines 18 and 24).

```
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))
    decreases xs, 1
{
  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));
      }
}

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)))
    decreases xs, 1
{
  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));
          FlattenConsApp(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)));
      }
}
```

Fig. 4. The methods descending and ascending.

This assert, along with the respective induction hypothesis (which follows from the contracts), allows Dafny to prove the third postcondition (i.e., stable(flatten(xxs),xs)) by transitivity of the relation stable. It should be noted that the transitivity property of stable is also automatically deduced.

Almost all the assertions and lemma calls annotating the body of descending (see Figure 4) are designed for proving stability. The only exception is the call to the lemma SortedConsList (Line 21 in Figure 4), 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 (Lines 2 and 3) and the postcondition of sequences (Line 2). Then, it infers that every member of Cons(Cons(min,grow),aux)) (i.e., the value of xxs) is sorted. Hence, the first postcondition (Line 4) of descending is proved. For the second postcondition (Line 5), in the then-branch, the induction hypothesis (Lines 11–13) stable-relates the two lists: flatten(xxs) and append(Cons(xs.head,Cons(min,grow)),xs.tail). The latter, by lemma StableLifting (Line 14), is stable-related to the list append(Cons(min,grow),xs). Hence, the postcondition is established by the transitivity of the relation stable. A very similar reasoning is used in the else-branch (Lines 19–29): 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 a 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, although there is a difference that is immediately apparent: The variable grow now is ghost, and a new variable shrink is introduced (Line 33). The use of grow allows us to write a contract for ascending (Lines 35–40) 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 assignment to xs would be xxs := Cons(append(grow,Cons(max,Nil)),aux);. Doing so, the computation of xxs would be quadratic in length(grow). We use the variable shrink to overcome this problem. The precondition states that grow is the reverse of shrink (Line 36). In the then-branch, append(grow,Cons(max,Nil)) (Line 49) is the parameter of a ghost variable, whereas calculation is performed through Cons(max,shrink). The starting assert in the then-branch and the lemma calls to SortedAppend and Reverse Cons (Lines 44 and 45) are all designed to ensure that the parameter of the recursive call satisfies the preconditions that the method ascending imposes on the formal parameters grow and shrink (Lines 35–37). The lemma ReverseCons is also used in the else-branch (Line 58) for showing that reverse(Cons(max,shrink),Nil)) (the first element of xxs) is equal to append(grow,Cons(max,Nil)). The remaining details of the proof of ascending are very similar to the previously explained for descending; see Lines 56–68 of Figure 4, where commented asserts are provided for further aid.

5.2. The method mergeAll

The method mergeAll merges a list of sorted lists into a single sorted list. It is implemented as a repeated application of the function method mergePairs. In Figure 5, we provide the annotated code of mergeAll and mergePairs, along with the contract of the function method merge, which is called by mergePairs.

The first two cases in the code of mergeAll are almost trivial: Only the lemma AppendNil is needed to prove that flattening of the input list is identical to the output list. Dafny requires that lemma to check the postcondition stable(ys, flatten(xxs)). However, the postcondition sorted(ys) is automatically verified by Dafny in all branches of the code. Regarding the assertions of the inductive case, we first establish that the length of xxs is at least 2, from which the postcondition of mergePairs tells us that

```
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(Cons(hxs, Nil)) = append(hxs, Nil);
            AppendNil(hxs);
    case Cons(htxs, ttxs) ⇒
            assert length(xxs) = 1 + length(txs) = 2 + length(ttxs);
            // assert length(mergePairs(xxs)) < length(xxs);
        ys := mergeAll(mergePairs(xxs));
            assert stable(ys, flatten(mergePairs(xxs)));
            // assert stable(ys, flatten(xxs));
}
function method mergePairs (xxs: List⟨List⟨E⟩⟩) : List⟨List⟨E⟩⟩
    requires ∀ zs • zs in multiset_of(xxs) ⟹ sorted(zs)
    ensures ∀ zs • zs in multiset_of(mergePairs(xxs)) ⟹ sorted(zs)
    ensures stable(flatten(mergePairs(xxs)), flatten(xxs))
    ensures mergePairs(xxs) = Nil ⟹ xxs = Nil
    ensures length(mergePairs(xxs)) ≤ length(xxs)
    ensures length(xxs) ≥ 2 ⟹ length(mergePairs(xxs)) < length(xxs)
    decreases length(xxs)
{ match xxs
  case Nil ⇒ Nil
  case Cons(hxs, txs) ⇒
     match txs
     case Nil ⇒ xxs
     case Cons(htxs, ttxs) ⇒
            assert htxs in multiset_of(txs);
            assert length(xxs) = 1 + length(txs) = 2 + length(ttxs);
            calc {
                 stable(merge(hxs, htxs), append(hxs, htxs));
                 ⟹ {
                     StableAppend(merge(hxs, htxs), append(hxs, htxs),
                                  flatten(mergePairs(ttxs)), flatten(ttxs));
                     }
                 stable(append(merge(hxs, htxs), flatten(mergePairs(ttxs))),
                        append(append(hxs, htxs), flatten(ttxs)));
                 ⟹ {
                     AssocAppend(hxs, htxs, flatten(ttxs));
                     }
                 stable(append(merge(hxs, htxs), flatten(mergePairs(ttxs))),
                        append(hxs, append(htxs, flatten(ttxs))));
                 ⟹ // definition of flatten
                 stable(flatten(Cons(merge(hxs, htxs), mergePairs(ttxs))),
                        append(hxs, append(htxs, flatten(ttxs))));
                 ⟹ // definition of flatten
                 stable(flatten(Cons(merge(hxs, htxs), mergePairs(ttxs))), flatten(xxs));
                 }
        Cons(merge(hxs, htxs), mergePairs(ttxs))
}
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))
⊞{...}
```

Fig. 5. The method mergeAll, the function method mergePairs, and the contract of merge.

length(mergePairs(xxs)) is strictly less than length(xxs), which is needed to prove termination of the recursive call to mergeAll. The postcondition stable(ys,flatten(xxs)) is proved through the postcondition of the recursive call, which we have repeated in an assert statement, the mergePairs postcondition about stability, and the transitivity of the stability relation. Remember that we write commented assertions as explanations;

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

Fig. 6. The function method merge.

that is, they are not required by Dafny, but Dafny can prove them (the user can simply drop the // to check that issue).

The proof of function method mergePairs starts by establishing two properties about tails of xxs. The subsequent proof calculation then uses the StableAppend and AssocAppend lemmas together with the definition of flatten to establish the postconditions.

In Figure 6, we show the code of merge. For easy reading, the result of the function appears as the last expression of every branch and is nonindented. In the first case, the result of merge is the list xs, which is sorted according to the precondition. Since the other list is empty, AppendNil is used to ensure that the append of both lists 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) uses the lemma StableLifting to prove that the lifting of hys (i.e., the head 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 on the following fact: Any pair of lists constructed from a fixed head and respective tails taken from a pair of stable lists is stable.

6. EXPERIENCE

Our program proof has been developed in the Dafny IDE [Leino and Wüstholz 2014], which lends itself to increase user productivity. Both the type checker and the verifier are run in the background. Type-checking and verification errors are displayed as colored underlining marks. When an attempted verification fails, a red dot (and a red squiggly line) indicate the return path along which the error is reported. The error message appears as hover text for the squiggly line. The locations related to the error are also marked by squiggly lines, and hover text is provided. In general, the Dafny IDE uses hover text for any additional information about the code (such as inferred types, termination metrics, co-induction desugaring, code inherited through refinement, etc.).

By clicking on a red dot, the Dafny IDE will display information from a counterexample that is relevant for analyzing the cause of the focused verification failure. The blue dots that then appear in the program text trace the control path from the start of the enclosing routine and leading to the error. There is state information associated with each blue dot, and the user can click on a blue dot to select a particular state.

When automated verification fails, telling the user that an assertion (i.e., a postcondition) cannot be proved, the user has the **assume** construct (along with the counterexample) to look for the required hints. Of course, the code itself is essential information for guessing induction hypotheses, invariants, termination metrics, and inline assertions. We exemplify this use of **assume** with the method merge in Figure 6. If one writes the contract and the body as follows:

```
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 ⇒ xs
    case Cons(hys, tys) ⇒
        if GT(hxs, hys)
        then Cons(hys, merge(xs, tys))
        else Cons(hxs, merge(txs, ys))
}
```

then a red dot in the second **case** Nil appears, the hover text on it says "Error: A postcondition might not hold on this return path," and the last postcondition is marked. Indeed, if the last postcondition was deleted (commented), the method would be automatically verified. So, the problem is that stable(xs,append(xs,Nil)) cannot be automatically inferred. We know this because if we assume that stable(xs,append(xs, Nil)), then the path is verified, whereas if we assert it, the assertion is not proved. Assuming xs = append(xs,Nil) also works, but to be automatically proved (by induction on zs), we should assert ∀ zs: List⟨E⟩ • zs = append(zs,Nil). Since this property is reused many times in the code for different lists, we write the parametrized lemma AppendNil in Section 4.1 and call AppendNil(xs) where the assume worked (see Figure 6). After that, the red dot jumps to the statement **if** GT(hxs,hys). First, in the then-branch, we know (looking at the value to be returned) what should be satisfied: stable(Cons(hys,merge(xs,tys)), append(xs,ys)); and we also know the induction hypothesis: stable(merge(xs,tys), append(xs,tys)). So, we use asserts and assumes to isolate the property that should be proved:

```
if GT(hxs, hys)  then  // by induction hypothesis:  stable(merge(xs, tys), append(xs, tys));
                       assert stable(Cons(hys, merge(xs, tys)), Cons(hys, append(xs, tys)));
                       assert append(Cons(hys, xs), tys) = Cons(hys, append(xs, tys));
                       // property to be proved
                       assume stable(append(xs, ys), append(Cons(hys, xs), tys));
                       assert stable(Cons(hys, merge(xs, tys)), append(xs, ys));
                 Cons(hys, merge(xs, tys))
```

Before proving lemma StableLifting, we construct its contract and write the call StableLifting(xs,ys) that allows us to convert the assume to an assert, hence completing the verification of the method merge. Indeed, the else-branch is automatically verified. To facilitate understanding, we have also provided commented assertions in the else-branch.

Our program-proof—as written in this article—is about 380 lines of relevant text (including comments). Many of them are dedicated to common function definitions and obvious lemmas that can be automatically proved or have an easy proof. The program proof is composed of 6 functions, 4 predicates, 12 lemmas, 5 methods, and 2 function methods. In the following two tables, we summarize the main proof statistics:

(fn.) Method	Contract Lines	Lem. Calls	Asserts	Comment Lines	Calc Steps	Code Line
nat_merge	3	1	1	0	0	2
sequences	3	0	2	4	0	13
descending	4	3	0	8	0	8
ascending	5	7	1	11	0	8
mergeAll	4	1	2	3	0	7
mergePairs	6	2	2	2	4	7
merge	3	2	0	10	0	12

Lemma	Contract Lin.	Lem. Call	Asserts	Comment Lin.	Calc St.	Automat.?
AppendNil	1	0	0	0	0	yes
AssocAppend	1	0	0	0	0	yes
FlattenConsApp	1	0	0	0	0	yes
ReverseCons	2	0	1	0	0	no
SortedAppend	3	0	0	0	0	yes
SortedConsList	3	0	0	0	0	yes
DistrFilterApp	1	0	0	0	0	yes
NullFilter	2	1	1	0	0	no
StableLifting	4	5	0	5	7	no
StableAppend	2	2	0	1	3	no
StableAppendL	2	1	0	0	0	no
EqMultisets	2	0	1	0	0	no

Once the code and lemmas have been verified, which altogether takes about 25 seconds, the Dafny compiler generates executable code for the .NET platform. The natural_mergesort routine is then callable from other .NET programs. However, since .NET does not have a standard format for inductive datatypes, the data format used by the Dafny compiler may not agree with the data formats used by other .NET languages like C#, Visual Basic, and F#. Therefore, to use our verified sorting algorithm from other languages may require some data conversions.

7. CONCLUSION

There is no doubt that sorting algorithms are useful and important in software. Good algorithms based on ingenious ideas can be subtle and warrant formal proofs. Indeed, if the comparison GT(min,xs.head) in function descending were replaced by ¬GT(xs.head,min), then the algorithm would no longer be stable. An excellent example is the recent revelation [de Gouw et al. 2015] of the incorrectness of a very popular sorting algorithm that has been running since 2002 in billions of computers, cloud services, and mobile phones. Indeed, it is the default sorting algorithm for Android SDK, Sun's JDK, and OpenJDK. The bug was discovered and fixed using the formal verification tool KeY [Beckert et al. 2007]. The bug appeared already in the original implementation in Python.

In this article, we have demonstrated that assertional proofs of correctness written as part of the program text are within reach of today's state-of-the-art verifiers. We are encouraged that the proof explains the reasons of correctness: The proof ingredients are provided by the user, but the proof steps themselves are carried out automatically by the verifier. Good programmers today are already used to putting assertions in their code. This gives us hope that verification of important algorithms will be carried out routinely by software engineers in the future.

Two aspects of the proof are worth extra attention. First, the Dafny program text given in this article is all that is fed as input to the verifier. No additional guidance is needed. Second, whereas a previous proof in Isabelle/HOL [Sternagel 2013] required innovation in defining appropriate induction schemes, induction in Dafny is as simple as a recursive call (see, e.g., the mutually recursive calls in the methods in Figures 3 and 4, or the use of the induction hypothesis via a recursive call in lemma NullFilter).

The interested reader can access the file (and verify it online) at the permalink: `http://rise4fun.com/Dafny/TFCr`.

ACKNOWLEDGMENTS

We are very grateful to Jean-Christophe Filliâtre for many valuable comments on a previous draft of this article. We also thank the anonymous referees for their thorough reviews and constructive comments, in particular for suggesting the one-line proof of the lemma EqMultisets.

REFERENCES

Roland Backhouse. 1995. The calculational method. *Information Processing Letter* 53, 3, 121.

Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005 (LNCS)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.), Vol. 4111. Springer, 364–387.

Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. 2011. Specification and verification: The spec# experience. *Communications of the ACM* 54, 6 (June 2011), 81–91.

Bernhard Beckert, Daniel Bruns, Vladimir Klebanov, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. 2013. *Secure Information Flow for Java. A Dynamic Logic Approach. Extended Version*. Karlsruhe reports in informatics. Fakultät für Informatik.

Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. 2007. *Verification of Object-oriented Software: The KeY Approach*. Springer.

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer.

Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009 (LNCS)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.), Vol. 5674. Springer, 23–42.

Stijn de Gouw, Frank S. de Boer, and Jurriaan Rot. 2014. Proof pearl: The key to correct and stable sorting. *Journal of Automated Reasoning* 53, 2, 129–139.

Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hähnle. 2015. OpenJDK's `java.utils.Collection.sort()` is broken: The good, the bad and the worst case. In *Computer Aided Verification. 27th International Conference, CAV 2015 (LNCS)*. Springer.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008 (LNCS)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, 337–340.

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where programs meet provers. In *Programming Languages and Systems. 22nd European Symposium on Programming, ESOP 2013 (LNCS)*, Matthias Felleisen and Philippa Gardner (Eds.), Vol. 7792. Springer, 125–128.

Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. 2000. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers.

Donald E. Knuth. 1973. *The Art of Computer Programming, Vol. III: Sorting and Searching*. Addison-Wesley.

Claire Le Goues, K. Rustan M. Leino, and Michał Moskal. 2011. The boogie verification debugger (tool paper). In *Software Engineering and Formal Methods. 9th International Conference, SEFM 2011 (LNCS)*, Gilles Barthe, Alberto Pardo, and Gerardo Schneider (Eds.), Vol. 7041. Springer, 407–414.

K. Rustan and M. Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *LPAR-16 (LNCS)*, Edmund M. Clarke and Andrei Voronkov (Eds.), Vol. 6355. Springer, 348–370.

K. Rustan and M. Leino. 2012. Automating induction with an SMT solver. In *Verification, Model Checking, and Abstract Interpretation. 13th International Conference, VMCAI 2012 (LNCS)*, Viktor Kuncak and Andrey Rybalchenko (Eds.), Vol. 7148. Springer, 315–331.

K. Rustan, M. Leino and Nadia Polikarpova. 2014. Verified calculations. In *Verified Software: Theories, Tools, Experiments. 5th International Conference, VSTTE 2013, Revised Selected Papers (LNCS)*, Ernie Cohen and Andrey Rybalchenko (Eds.), Vol. 8164. Springer, 170–190.

K. Rustan M. Leino, and Valentin Wüstholz. 2014. The Dafny integrated development environment. In *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014 (EPTCS)*, Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry (Eds.), Vol. 149. 3–15.

Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. 2002. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer.

Christian Sternagel. 2013. Proof pearl: A mechanized proof of GHC's mergesort. *Journal of Automated Reasoning* 51, 4, 357–370.

Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. 2015. Autoproof: Auto-active functional verification of object-oriented programs. In *Tools and Algorithms for the Construction and Analysis of Systems. 21st International Conference, TACAS 2015 (LNCS)*, Christel Baier and Cesare Tinelli (Eds.), Vol. 9035. Springer, 566–580.

Received February 2015; revised June 2015; accepted August 2015