J Autom Reasoning (2013) 51:357–370
DOI 10.1007/s10817-012-9260-7

# Proof Pearl—A Mechanized Proof of GHC's Mergesort

**Christian Sternagel**

Received: 15 February 2012 / Accepted: 30 August 2012 / Published online: 26 September 2012


**Abstract** We present our Isabelle/HOL formalization of GHC's sorting algorithm for lists, proving its correctness and stability. This constitutes another example of applying a state-of-the-art proof assistant to real-world code. Furthermore, it allows users to take advantage of the formalized algorithm in generated code.

**Keywords** Mergesort · Theorem proving · Code generation

## 1 Introduction

In proof assistants, like Isabelle/HOL [4], it is common to use definitions of algorithms that look more like specifications than actual implementations, in the following sense: Specifications are typically easy to understand (but possibly inefficient) and prefer abstract datatypes (like sets) over concrete datatypes (like lists). In contrast, implementations are often tuned for performance and incomprehensible for the uninitiated.

Specifications facilitate high-level proofs that are mostly concerned with abstract properties and avoid "implementation details" that tend to be tedious. From the logical viewpoint this is mostly the end of the story: we define an algorithm and prove its desired properties. For actual use in real-world code, however, such specifications are often not efficient enough. This is where *algorithm refinement* comes into play. That is, we implement an alternative, more efficient, variant of our algorithm and formally prove that both versions are equivalent, i.e., their extensional behaviors coincide. Or put differently: given equal arguments, both variants yield the same results.

This research is supported by the Austrian Science Fund (FWF): J3202.

C. Sternagel (✉)
School of Information Science, Japan Advanced Institute of Science and Technology,
Nomi, Ishikawa, Japan
e-mail: c-sterna@jaist.ac.jp

Additionally, Isabelle/HOL, allows for code generation [2], i.e., to automatically generate actual source code in various target languages (currently, Haskell, OCaml, Scala, and StandardML) from a given formalization of an algorithm. The resulting code is correct by construction.

Together with algorithm refinement, code generation allows for the following workflow for obtaining efficient verified programs in three steps: First formalize *easy* variants of the constituting algorithms and prove all desired properties. Then, formalize *efficient* variants of the same algorithms and prove them equivalent. Finally, use code generation and obtain an efficient program that is guaranteed to satisfy all properties that have been proven in the initial formalization.

In the following we present our Isabelle/HOL[1] formalization of GHC's sorting algorithm for lists[2] (for brevity, referred to as *sort* in the remainder). Along the way, we prove its correctness and stability. In this work, we just give an overview of the most important ideas and refer to the *Archive of Formal Proofs* [7] for details.

The remainder is structured as follows. In Section 2, we describe the implementation of *sort* in GHC's standard library. Before we enter the main section, we give some preliminaries (Section 3) that are related to sorting and already provided by Isabelle/HOL. Our formalization is given in Section 4, where we prove correctness and stability of *sort*. We finally conclude in Section 5.

*Motivation* Our original motivation was to tune CeTA,[3] a fully verified program whose code is generated from an underlying Isabelle/HOL formalization [8]. CeTA is a certifier for termination proofs of first-order term rewrite systems (TRSs). Such proofs are highly modular, i.e., a given TRS is split into several TRSs for which termination is proven separately, and often use transformation techniques (like semantic labeling) that can blow up the number of rewrite rules exponentially. Moreover, for reduction pairs, which are employed to delete rewrite rules from TRSs that cannot be the cause of nontermination, a common task for a certifier is to check that the remaining TRS is a subset of the original one. Since in CeTA, TRSs are represented as lists of rewrite rules, this check incorporates sorting those lists and was identified as one of the bottle-necks. Our first step was to replace Isabelle/HOL's default sorting algorithm (an insertsort variant provided in the `List` theory) by a supposedly more efficient version from the library (a quicksort variant provided in `Multiset`). Since this did not give the desired speedup (unfortunately, CeTA does not work properly together with `Efficient_Nat`; see also the remark in Section 5) and our target programming language is Haskell, we decided to formalize the sorting algorithm of GHC's standard library.

*Spoiler* Note that we do not prove anything about the runtime or space complexity of *sort* (in the sources of GHC's library it is claimed that the current version performs better than earlier ones on several benchmarks; however, we are not aware of any formal proof). For us it suffices that on the examples we tested, *sort* actually outperforms Isabelle/HOL's quicksort variant. Furthermore, *sort* is part of GHC's

[1] Our development is based on version `Isabelle2012` (May 2012).

[2] www.haskell.org/ghc/docs/7.0-latest/html/libraries/base-4.3.1.0/src/Data-List.html#sort

[3] More precisely, to make its runtime scale better on huge inputs.

standard library and thus our formalization constitutes a verification of real-world code that is (at least implicitly) used in many Haskell programs.

## 2 GHC's Sorting Algorithm

Consider GHC's sorting algorithm for lists depicted in Listing 1. It is a mergesort variant that takes advantage of (reverse) sorted subsequences occurring in the input. The three mutually recursive functions `sequences`, `descending`, and `ascending` take care of transforming an input list into a list of sorted lists. To this end, `ascending` detects sorted subsequences and returns them unchanged, while `descending` detects reverse sorted subsequences and flips them along the way. The resulting sequence of sorted lists is merged into a single list by `mergeAll`. Note that this implementation behaves especially well on typically problematic cases like sorted lists or reverse sorted lists as input. In both cases `sequences` just needs a single traversal and no merging is required.

Before we treat our Isabelle/HOL formalization of *sort*, some words on it's origin. According to the GHC sources, the algorithm is rumored to be based on code

```
sort = sortBy compare
sortBy cmp = mergeAll . sequences
  where
    sequences (a:b:xs)
      | a `cmp` b == GT = descending b [a] xs
      | otherwise = ascending b (a:) xs
    sequences xs = [xs]

    descending a as (b:bs)
      | a `cmp` b == GT = descending b (a:as) bs
    descending a as bs = (a:as) : sequences bs

    ascending a as (b:bs)
      | a `cmp` b /= GT = ascending b (\ys -> as (a:ys)) bs
    ascending a as bs = as [a] : sequences bs

    mergeAll [x] = x
    mergeAll xs = mergeAll (mergePairs xs)

    mergePairs (a:b:xs) = merge a b : mergePairs xs
    mergePairs xs = xs

    merge as@(a:as') bs@(b:bs')
      | a `cmp` b == GT = b : merge as bs'
      | otherwise = a : merge as' bs
    merge [] bs = bs
    merge as [] = as
```

**Listing 1** GHC's sort

```
sequences key (a # b # xs) =
  (if gt key a b then desc key b [a] xs else asc key b (op # a) xs)
sequences key [] = [[]]
sequences key [v] = [[v]]

asc key a as (b # bs) =
  (if ¬ gt key a b then asc key b (λx. as (a # x)) bs
    else as [a] # sequences key (b # bs))
asc key a as [] = as [a] # sequences key []

desc key a as (b # bs) =
  (if gt key a b then desc key b (a # as) bs
    else (a # as) # sequences key (b # bs))
desc key a as [] = (a # as) # sequences key []
```

**Fig. 1** Formalization of `sequences`

by Lennart Augustsson[4] and possibly to bear similarities to an algorithm of [5] (which does not seem to be available any longer) by Richard O'Keefe. This rumor is supported by the chapter about sorting of [6]. However, we could not find any definite answer.

In our Isabelle/HOL formalization we define `sequences` and `merge_all` as shown in Figs. 1 and 2, respectively. When comparing this definitions to the one from Listing 1, there are some differences that may need explanation. First, partly for brevity and partly to conform to Isabelle/HOL's naming conventions, we changed the names of some functions. Furthermore, Isabelle/HOL's syntax is slightly different from Haskell's. More specifically, '`#`' denotes list-cons ('`:`' in Haskell) and the notation `op f` is used to turn an infix operator into a function (i.e., `op #` corresponds to `(:)` in Haskell). Another difference is that instead of Haskell's `Ord` typeclass, we are using Isabelle/HOL's built-in typeclass `linorder`, whose instances are all linearly ordered types. As a consequence we do not parametrize our functions over a compare-function, but rather over a key-function that turns list-elements into elements of some linearly ordered type. For brevity, we use the abbreviation `gt key` $\equiv \lambda$`y x. key x < key y` (later, we will also use `lt key` $\equiv \lambda$`x y. key x < key y` and `ge key` $\equiv \lambda$`y x. key x` $\leq$ `key y`).

Further note that Isabelle/HOL disambiguates the patterns on the left-hand sides of equations such that at most one defining equation is applicable on any term. In Haskell, on the other hand, this is guaranteed by trying patterns from top to bottom.

Apart from this rather cosmetic changes, we hope that it is still sufficiently obvious that our formalization is indeed handling the function of Listing 1. (By the way, if you want to see the Haskell code that can be generated from the formalization, just use

**export_code** `sequences merge_all` **in** `Haskell` **file** -

inside Isabelle/HOL.)

*Note* The Haskell implementation of `mergeAll` is possibly nonterminating (when called on the empty list), however, by construction the result of `sequences` contains at least one element. Hence there is no problem. In Isabelle/HOL all functions must

[4] www.mail-archive.com/haskell@haskell.org/msg01822.html

```
merge key (a # as) (b # bs) =
  (if gt key a b then b # merge key (a # as) bs
     else a # merge key as (b # bs))
merge key [] bs = bs
merge key (v # va) [] = v # va

merge_pairs key (a # b # xs) = merge key a b # merge_pairs key xs
merge_pairs key [] = []
merge_pairs key [v] = [v]

merge_all key [] = []
merge_all key [x] = x
merge_all key (v # vb # vc) = merge_all key (merge_pairs key (v # vb # vc))
```

**Fig. 2** Formalization of *merge_all*

be terminating and hence the Haskell version is not accepted. That is, why our formalization of *merge_all* contains an extra case for the empty list (which is never used for sorting).

## 3 Preliminaries

Before we describe the default sorting algorithm of Isabelle/HOL, let us have a closer look at the properties that we are interested in. The two properties of main interest are *correctness* and *stability*. In the following, we investigate each of them in turn and show how they are formalized in Isabelle/HOL's library.

*Correctness* Probably the first thing that comes to our mind, when we think about the correctness of a sorting algorithm, is that its result should be, well, sorted.

**Definition 1** (Sortedness) A list is *sorted* when every two consecutive elements are in order. In Isabelle/HOL this is expressed as an inductive predicate given by the rules:

```
sorted []
⟦∀y∈set xs. x ≤ y; sorted xs⟧ ⟹ sorted (x # xs)
```

Note, however, that sortedness on its own is not sufficient to describe the correctness of a sorting algorithm. Consider, e.g., the function *wrongsort xs = []*. Besides its result being sorted, it is clearly not a correct sorting algorithm. It turns out that we have to make sure that a potential sorting algorithm does not add or remove elements. This property is formulated using multisets in Isabelle/HOL. Where a multiset is like a set in that the order of elements is not important, but may contain multiple copies of equal elements.

**Definition 2** (Element Invariance) A function *f::α list ⇒ α list* is *element invariant* if it does neither add nor remove elements. More formally, *f* has to satisfy

```
multiset_of (f xs) = multiset_of xs
```

where *multiset_of* (defined in theory *Multiset*) turns a list into a multiset.

Together, the above two properties allow us to define the *correctness* of a sorting algorithm.

**Definition 3** (Correctness) A function `f:: α list ⇒ α list` is a *correct* sorting algorithm whenever it is element invariant and produces only sorted results.

In the standard Isabelle/HOL distribution an archetypical sorting algorithm is provided by

```
sort_key f xs = foldr (insort_key f) xs []        (1)
```

(in theory `List`) where `insort_key` is defined by the equations

```
insort_key f x [] = [x]
insort_key f x (y # ys) =
  (if ge f y x then x # y # ys else y # insort_key f x ys)
```

with corresponding sortedness and element invariance proofs in the theories `List` and `Multiset`, respectively:

```
sorted_sort_key:   sorted (map f (sort_key f xs))
multiset_of_sort:  multiset_of (sort_key f xs) = multiset_of xs
```

*Stability* Informally, a sorting algorithm is stable when it does not change the relative order of equal elements. Since in Isabelle/HOL equality is built-in (and hence there is no way to distinguish between two equal elements), stability of a sorting algorithm can only be expressed in presence of a key-function, i.e., a function that, given an element, produces a key according to which this element should be sorted.

*Example 1* Consider the list *[2, 3, 2]*. After sorting we obtain *[2, 2, 3]*. It is impossible to say inside Isabelle/HOL whether the first *2* in the result is the same as the first one in the input. Having a key-function, we can apply a simple trick. First we add the indices of elements to the input *[(2, 0), (3, 1), (2, 2)]*. Then we sort the list using the key-function *fst* (i.e., projecting to the first components of the pairs). Finally, we can see for each *2*, from which index in the input list it originates. If the result is *[(2, 0), (2, 2), (3, 1)]* sorting was indeed stable.

**Definition 4** (Stability) A sort function $f::(\alpha \Rightarrow \beta) \Rightarrow \alpha\ list \Rightarrow \alpha\ list$ is *stable* (w.r.t. the key-function $key::\alpha \Rightarrow \beta$) whenever the relative order of elements having the same key does not change between *xs* and *f key xs*. In Isabelle/HOL this is expressed as follows:

```
x ∈ set xs ⟹ [y←f key xs. key x = key y] = [y←xs. key x = key y]
```

We sometimes (as above) use the convenience syntax `[x←xs. P x]` instead of `filter P xs` (where *filter* keeps just those elements of a list that satisfy the given predicate).

Why are we actually interested in the above properties? Correctness should be clear, we want to make sure that *sort* really is a sorting algorithm. But why do we need stability? In principle there are several reasons why stability is interesting: only stable sorting algorithms allow for incremental sorting (e.g., sort according to key A

and for those with equal A, sort according to key B), swapping elements may cause memory updates on physical media, etc. However, our interest in stability has more ad hoc reasons. Those will become clear after showing the following lemma (which is to be found in theory `Multiset`)

```
⟦multiset_of (f key xs) = multiset_of xs;
 ⋀k. k ∈ set (f key xs) ⟹
      [x←f key xs. key k = key x] =
       [x←xs. key k = key x];
 sorted (map key (f key xs))⟧
⟹ sort_key key xs = f key xs
```
(2)

which states that it is sufficient for a function *f* to be a correct (w.r.t. the key-function `key`) and stable sorting algorithm, in order to be logically equivalent to `sort_key`. Hence, if we succeed in proving the above three assumptions for some function `f`, we may use it interchangeably with `sort_key`. This, in turn, allows us to install a more efficient sorting algorithm than (1) for code generation. Thus, every formalization using `sort_key` can take advantage of the more efficient algorithm in generated code for free.

## 4 Efficient Mergesort

The definition of sorting as given in (1) is a reasonable implementation and a good compromise between efficiency and ease of specification. In the end, efficiency is irrelevant for the logic and hence definitions should be as natural and easy as possible. For code generation on the other hand, efficiency is a concern. The typical way of handling this situation is starting with a natural, (maybe) inefficient, but easy to use definition and use it throughout the formalization. Then, before generating code, prove so called *code equations* that show the equivalence of this natural definition to some more efficient variant. The remainder of this article does exactly this, i.e., provide a code equation for `sort_key` that tunes its performance. As we have seen at the end of Section 3, we need to show element invariance, stability, and sortedness in order to prove a function equivalent to `sort_key`.

In the following, we describe our corresponding formalization and mention how we managed to turn an initial formalization with tedious manual proofs and having more than a thousand lines into a mere 400 lines (at least 100 lines less, if we disregard auxiliary definitions that might be of general interest) where most of the proofs are automatic (i.e., solved by automatic methods like `auto`, `blast`, `simp`, etc., after indicating the used induction schema).

Obviously most non-trivial proofs about *sequences* require induction. Since we have a mutual dependency on *asc* and *desc* this requires to prove simultaneously according facts about those two functions. The induction schema that is provided by Isabelle/HOL can be seen in Fig. 3 in the Appendix. Applying this schema turned out to be quite tedious and required to strengthen the induction hypothesis and introduce additional assumptions (both modifications were however only necessary for *asc*, because of its function argument). We achieved a drastic simplification by introducing an alternative induction schema for `sequences`. Before giving this schema, we need two auxiliary functions which are generalizations of the well known functions `takeWhile` and `dropWhile` (whose definitions are, e.g., available

in theory `List` or Haskell's standard prelude), where `takeWhile p xs` returns the longest prefix of `xs` in which every element satisfies the predicate `p` and its counterpart `dropWhile p xs` returns the remaining elements after removing such a prefix. In the generalization, the predicate that decides whether we still take/drop does not only depend on the current element, but additionally on the previous one. Since in this way, starting from some default element, we can take/drop a sequence in which every two consecutive elements are linked by the predicate, we call such a sequence a *chain*. Here are the corresponding definitions:

```
take_chain a P [] = []
take_chain a P (x # xs) = (if P a x
 then x # take_chain x P xs else [])

drop_chain a P [] = []
drop_chain a P (x # xs) = (if P a x
 then drop_chain x P xs else x # xs)
```

It is easily shown by induction that `takeWhile` and `dropWhile` are just special cases of `take_chain` and `drop_chain`, i.e.,

```
take_chain a (λx. P) xs = takeWhile P xs

drop_chain a (λx. P) xs = dropWhile P xs
```

A characteristic property of `take_chain` and `drop_chain` that we will need later (and which is easily proven by induction) is the following:

```
take_chain a P xs @ drop_chain a P xs = xs
```
(3)

Having `take_chain` and `drop_chain`, we can get rid of all occurrences of `asc` and `desc` inside the definition of `sequences` as is shown by the lemmas

```
asc key b (op # a) xs =
  (a # b # take_chain b (λx y. ge key y x) xs) #
  sequences key (drop_chain b (λx y. ge key y x) xs)

desc key a bs xs =
  (rev (take_chain a (gt key) xs) @ a # bs) #
  sequences key (drop_chain a (gt key) xs)
```

and (relatively) easily proven by induction over *xs* (we need to generalize the first equation to an arbitrary `f` satisfying `f (xs @ ys) = f xs @ ys` and an arbitrary list `as`, instead of `op #` and *a*, for the induction to run through). This gives rise to an alternative induction schema for `sequences`

```
⟦⋀key. P key []; ⋀key x. P key [x];
 ⋀key a b xs.
    ⟦ge key b a ⟹
      P key (drop_chain b (λx y. ge key y x) xs);
     ¬ ge key b a ⟹ P key (drop_chain b (gt key) xs)⟧
    ⟹ P key (a # b # xs)⟧
⟹ P key xs
```

Now, this looks much better! We will no longer bother with `asc` and `desc`.

Our next step on the way to sortedness is a generalization of the `sorted` predicate that works well together with `take_chain` and `drop_chain`. We call the inductive predicate `linked` and define it by the rules:

```
linked P []
linked P [x]
⟦P x y; linked P (y # ys)⟧ ⟹ linked P (x # y # ys)
```

In contrast to `sorted`, it makes tests on consecutive elements explicit (whereas for `sorted`, *all* remaining elements are checked in a single rule) and allows us to use an arbitrary predicate (which is a perfect fit for our take and drop generalizations). Having `linked`, it is easy to show

```
linked P (x # take_chain x P xs)
linked op ≤ xs = sorted xs
```

thereby showing that `sorted` is just a special case of `linked` and that the result of `take_chain` is always a chain (w.r.t. the given predicate), which is needed in the proof that `sequences` generates a list of sorted lists.

Now we have the main ingredients to prove two important facts, `sequences` does not remove or add elements and generates a list of sorted lists. Both proofs are by induction using our newly introduced induction schema for `sequences` and run through automatically in Isabelle/HOL. Hence, we just give the lemmas

```
∀x∈set (sequences key xs). sorted (map key x)                    (4)
multiset_of (concat (sequences key xs)) = multiset_of xs         (5)
```

where `concat` concatenates all elements of a list of lists into a single list.

The corresponding facts for `merge_all` are automatically proven by induction:

```
∀x∈set xs. sorted (map key x) ⟹
sorted (map key (merge_all key xs))                              (6)
multiset_of (merge_all key xs) = multiset_of (concat xs)         (7)
```

Together, (5) and (7) yield element invariance of *merge_all key* ∘ *sequences key* (where '∘' denotes function composition, i.e., *(f ∘ g) x = f (g x)*), whereas (4) and (6) yield sortedness, i.e.,

$$\mathit{multiset\_of}\ (\mathit{merge\_all}\ \mathit{key}\ (\mathit{sequences}\ \mathit{key}\ \mathit{xs})) = \mathit{multiset\_of}\ \mathit{xs} \tag{8}$$

$$\mathit{sorted}\ (\mathit{map}\ \mathit{key}\ (\mathit{merge\_all}\ \mathit{key}\ (\mathit{sequences}\ \mathit{key}\ \mathit{xs}))) \tag{9}$$

showing that `merge_all key ∘ sequences key` is a correct sorting algorithm.

At this point (corresponding roughly to the first half of our formalization), we turn our attention to stability. Stability (or at least a very similar property) of `sequences` is proven by the lemma

```
[y←concat (sequences key xs). key x = key y] =
  [y←xs. key x = key y]                                          (10)
```

again, using our custom induction schema together with (3) and the auxiliary lemmas

```
ge key a b ⟹
  [y←take_chain b (gt key) xs. key a = key y] = []
```

```
filter P (take_chain x Q xs) @ filter P (drop_chain x Q xs)
  = filter P xs
```

```
[y←rev (take_chain b (gt key) xs). key x = key y] =
  [y←take_chain b (gt key) xs. key x = key y]
```

all of which are automatically proven by induction.

The first step towards stability of `merge_all`, is proving the lemma

```
sorted (map key xs) ⟹
[y←merge key xs ys. key x = key y] =                    (11)
  [y←xs. key x = key y] @ [y←ys. key x = key y]
```

which states that for sorted lists `xs`, `merge` behaves like list-append on lists that are filtered corresponding to a specific key. To this end, we apply the same ideas that were already used for `sequences`: First, we introduce an alternative induction schema that combines several recursive calls into a single one by means of an auxiliary function. (In the case of `merge` the auxiliary function is `dropWhile` rather than the slightly more complicated *drop_chain*.) Then, we prove some easy lemmas about the auxiliary function and its comrade (`takeWhile` instead of `take_chain`, in the case of `merge`). Finally, we put everything together by applying the new induction schema. Since you have seen all this for `sequences`, we just give the alternative induction schema

```
⟦sorted (map key xs); ⋀xs. P xs [];
 ⋀xs y ys.
    ⟦sorted (map key xs); P (dropWhile (ge key y) xs) ys⟧
    ⟹ P xs (y # ys)⟧
⟹ P xs ys
```

This induction schema allows us to prove facts about `merge` in a context where its first argument is sorted (which is the case for all lists in the result of `sequences`, as we showed earlier) and combines multiple recursive calls (as if `merge` took elements from the first argument as long as all of them were greater than or equal to the head of the second argument).

We have to show the corresponding properties for `merge_pairs` and `merge_all`, which are

```
∀xs∈set xss. sorted (map key xs) ⟹
[y←concat (merge_pairs key xss). key x = key y] =       (12)
  [y←concat xss. key x = key y]
```

```
∀xs∈set xss. sorted (map key xs) ⟹
[y←merge_all key xss. key x = key y] =                  (13)
  [y←concat xss. key x = key y]
```

and proven by induction using (11). An easy consequence of (13) and (4) is

```
[x←merge_all key (sequences key xs). key y = key x] =
  [x←xs. key y = key x]                                    (14)
```

showing stability of `merge_all key ∘ sequences key`.

Finally, using (2), whose assumptions are discharged by (8), (14), and (9), we can establish the equation:

```
sort_key key = merge_all key ∘ sequences key
```

## 5 Conclusions and Related Work

We have given an Isabelle/HOL formalization of GHC's mergesort algorithm, showing correctness and stability. On the one hand, this showcases once more that state-of-the-art proof assistants, like Isabelle/HOL, can be used to verify real-world code. On the other hand, our formalization allows existing theories that rely on Isabelle/HOL's default sorting algorithm to take advantage of the more efficient *sort* during code generation. Doing this is as easy as importing `Efficient_Sort` (from the Archive of Formal Proofs) in the header of your theory.

The key points to achieve such a compact formalization are *custom induction schemes* and *generalizations*. The former is greatly alleviated by a bunch of Isabelle/HOL commands that were originally developed as part of the function package [3] and deserve broader attention: the `induction_schema` command together with `pat_completeness` and `lexicographic_order` (or any other way of proving well-foundedness of the induction relation automatically) makes writing customized induction schemes a breeze. The latter is of course well-known, nevertheless, we think that the generalizations `linked`, `take_chain`, and `drop_chain` constitute another nice example of this concept.

*Assessment* In order to compare the generated code for *sort* to Isabelle/HOL's default insertsort (is) and the alternative quicksort (qs) from theory `Multiset`, we conducted some experiments whose results can be seen in Table 1. We tested the code generated for different target languages (Haskell, OCaml, Scala, and

**Table 1** Relative speedup of *sort*

| | #-elements | Haskell | | OCaml | | Scala | | StandardML | |
|---|---|---|---|---|---|---|---|---|---|
| | | is | qs | is | qs | is | qs | is | qs |
| | 100,000 | 6.9 | 1.2 | 0.8 | 4.7 | 1.1 | 17.2 | 1.4 | 3.6 |
| inc | 500,000 | ∞ | 1.5 | 2.4 | 9.9 | 2.4 | 61.2 | 1.5 | 4.2 |
| | 1,000,000 | ∞ | 1.8 | 4.3 | 16.0 | 2.1 | 19.2 | 10.2 | 17.9 |
| | 100,000 | ∞ | 1.2 | ∞ | 16.8 | ∞ | 15.0 | ∞ | 4.3 |
| dec | 500,000 | ∞ | 1.6 | ∞ | 41.4 | ∞ | 91.2 | ∞ | 5.0 |
| | 1,000,000 | ∞ | 2.0 | ∞ | 74.3 | ∞ | 37.8 | ∞ | 18.9 |
| | 100,000 | ∞ | 1.1 | ∞ | 1.4 | ∞ | 3.0 | ∞ | 1.2 |
| rnd | 500,000 | ∞ | 1.2 | ∞ | 1.7 | ∞ | 3.9 | ∞ | 1.2 |
| | 1,000,000 | ∞ | 1.4 | ∞ | 1.8 | ∞ | 4.8 | ∞ | 1.4 |

StandardML) on ascending (inc), descending (dec), and random (rnd) lists of integers of various sizes (100,000 elements, 500,000 elements, and 1,000,000 elements, respectively). In each column of the table, the speedup of *sort* with respect to the given algorithm is listed (i.e., a number greater than 1 indicates that *sort* was faster), where we aborted tests after a timeout of 60 s (indicated by a speedup of $\infty$). Each value corresponds to the average results on 100 samples. For every target language, a small wrapper program reads a list of integers and applies the sorting algorithm under consideration. Note that qs performs orders of magnitude worse, if it is not used together with the theory `Efficient_Nat`, since the pivot of a list is computed using Isabelle/HOL's *nat* type which by default uses Peano numbers (also in generated code).

*A Note on Efficient_Nat* The default representation of natural numbers in Isabelle/HOL is by the datatype

**datatype** `nat = 0 | Suc nat`

that is, a unary encoding by so called Peano numbers. Compared to the *integer* types which are typically part of any programming language, arithmetic operations on Peano numbers are quite slow. To solve this problem, the theory *Efficient_Nat* (which in turn is based on *Num*) may be loaded to set up the code generator such that it uses the following more efficient binary encoding of natural numbers:

**datatype** `num = One | Bit0 num | Bit1 num`

In the quicksort variant of Isabelle/HOL, the pivot is computed by division on natural numbers. An advantage of *sort* is that it does not involve any arithmetic operations on natural numbers and thus performs well even without loading `Efficient_Nat`.

It turns out that *sort* is the algorithm of choice, independent of the used target language, performing slightly better than qs, even when `Efficient_Nat` is loaded.

*Related Work* We are aware of two other formalizations of mergesort. The first is a Coq formalization[5] which does, however, not consider stability (which we personally found to be the most challenging part). The second is an ACL2 formalization[6] which, again, does not consider stability and is based on a theory of so called *powerlists*.

There are also formalizations of other sorting algorithms in various systems, e.g., insertsort, quicksort, and heapsort in Coq [1]; insertsort (theory *List*) and quicksort (theory *Multiset*) in Isabelle/HOL.

**Acknowledgements** We thank the anonymous referees for helpful suggestions.



[5] http://coq.inria.fr/stdlib/Coq.Sorting.Mergesort.html

[6] www.cs.utexas.edu/users/moore/acl2/books/books/powerlists/merge-sort.lisp

## Appendix

```
⟦⋀key a b xs.
    ⟦gt key a b ⟹ R key b [a] xs; ¬ gt key a b ⟹ Q key b (op # a) xs⟧
    ⟹ P key (a # b # xs);
 ⋀key. P key []; ⋀key v. P key [v];
 ⋀key a f b bs.
    ⟦¬ gt key a b ⟹ Q key b (f ∘ op # a) bs;
     ¬ ¬ gt key a b ⟹ P key (b # bs)⟧
    ⟹ Q key a f (b # bs);
 ⋀key a f. P key [] ⟹ Q key a f [];
 ⋀key a as b bs.
    ⟦gt key a b ⟹ R key b (a # as) bs; ¬ gt key a b ⟹ P key (b # bs)⟧
    ⟹ R key a as (b # bs);
 ⋀key a as. P key [] ⟹ R key a as []⟧
⟹ P a0.0 a1.0
⟦⋀key a b xs.
    ⟦gt key a b ⟹ R key b [a] xs; ¬ gt key a b ⟹ Q key b (op # a) xs⟧
    ⟹ P key (a # b # xs);
 ⋀key. P key []; ⋀key v. P key [v];
 ⋀key a f b bs.
    ⟦¬ gt key a b ⟹ Q key b (f ∘ op # a) bs;
     ¬ ¬ gt key a b ⟹ P key (b # bs)⟧
    ⟹ Q key a f (b # bs);
 ⋀key a f. P key [] ⟹ Q key a f [];
 ⋀key a as b bs.
    ⟦gt key a b ⟹ R key b (a # as) bs; ¬ gt key a b ⟹ P key (b # bs)⟧
    ⟹ R key a as (b # bs);
 ⋀key a as. P key [] ⟹ R key a as []⟧
⟹ Q a2.0 a3.0 a4.0 a5.0
⟦⋀key a b xs.
    ⟦gt key a b ⟹ R key b [a] xs; ¬ gt key a b ⟹ Q key b (op # a) xs⟧
    ⟹ P key (a # b # xs);
 ⋀key. P key []; ⋀key v. P key [v];
 ⋀key a f b bs.
    ⟦¬ gt key a b ⟹ Q key b (f ∘ op # a) bs;
     ¬ ¬ gt key a b ⟹ P key (b # bs)⟧
    ⟹ Q key a f (b # bs);
 ⋀key a f. P key [] ⟹ Q key a f [];
 ⋀key a as b bs.
    ⟦gt key a b ⟹ R key b (a # as) bs; ¬ gt key a b ⟹ P key (b # bs)⟧
    ⟹ R key a as (b # bs);
 ⋀key a as. P key [] ⟹ R key a as []⟧
⟹ R a6.0 a7.0 a8.0 a9.0
```

**Fig. 3** Default induction schema for `sequences`

## References


1. Filliâtre, J.C., Magaud, N.: Certification of sorting algorithms in the Coq system. In: Theorem Proving in Higher Order Logics: Emerging Trends (1999). http://www-sop.inria.fr/croap/TPHOLs99/proceeding.html
2. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) Functional and Logic Programming, FLOPS 2010, Lecture Notes in Computer Science, vol. 6009, pp. 103–117. Springer (2010). doi:10.1007/978-3-642-12251-4_9
3. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. J. Autom. Reasoning **44**(4), 303–336 (2010). doi:10.1007/s10817-009-9157-2
4. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—a proof assistant for higher-order logic. In: Lecture Notes in Computer Science, vol. 2283. Springer (2002). doi:10.1007/3-540-45949-9
5. O'Keefe, R.: A smooth applicative merge sort. Tech. rep., Department of Artificial Intelligence, University of Edinburgh (1982)

6. Paulson, L.C.: ML for the Working Programmer, 2nd edn. Cambridge University Press, New York (1996)
7. Sternagel, C.: Efficient Mergesort. In: Klein, G., Nipkow, T., Paulson, L.C. (eds.) The Archive of Formal Proofs. http://afp.sf.net/entries/Efficient-Mergesort.shtml (2011, formal proof development)
8. Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, TPHOLs 2009, Lecture Notes in Computer Science, vol. 5674, pp. 452–468. Springer (2009). doi:10.1007/978-3-642-03359-9_31