Efficient Mergesort

Christian Sternagel

February 6, 2026

Abstract

We provide a formalization of the mergesort algorithm as used in GHC's Data.List module, proving correctness and stability. Furthermore, experimental data suggests that generated (Haskell-)code for this algorithm is much faster than for previous algorithms available in the Isabelle distribution.

theory *Efficient-Sort*
  imports *HOL−Library.Multiset*
begin

1 GHC Version of Mergesort

In the following we show that the mergesort implementation used in GHC (see http://hackage.haskell.org/package/base-4.11.1.0/docs/src/Data.OldList.html#sort) is a correct and stable sorting algorithm. Furthermore, experimental data suggests that generated code for this implementation is much more efficient than for the implementation provided by *HOL−Library.Multiset*.

A high-level overview of an older version of this formalization as well as some experimental data is to be found in [1].

1.1 Definition of Natural Mergesort

context
  fixes *key* :: *'a* ⇒ *'k*::*linorder*
begin

Split a list into chunks of ascending and descending parts, where descending parts are reversed on the fly. Thus, the result is a list of sorted lists.

fun *sequences* :: *'a list* ⇒ *'a list list*
  and *asc* :: *'a* ⇒ (*'a list* ⇒ *'a list*) ⇒ *'a list* ⇒ *'a list list*
  and *desc* :: *'a* ⇒ *'a list* ⇒ *'a list* ⇒ *'a list list*
  where
    sequences (*a* # *b* # *xs*) =

```
    (if key a > key b then desc b [a] xs else asc b ((#) a) xs)
| sequences [x] = [[x]]
| sequences [] = []
| asc a as (b # bs) =
    (if key a ≤ key b then asc b (λys. as (a # ys)) bs
    else as [a] # sequences (b # bs))
| asc a as [] = [as [a]]
| desc a as (b # bs) =
    (if key a > key b then desc b (a # as) bs
    else (a # as) # sequences (b # bs))
| desc a as [] = [a # as]

fun merge :: 'a list ⇒ 'a list ⇒ 'a list
  where
    merge (a # as) (b # bs) =
      (if key a > key b then b # merge (a # as) bs else a # merge as (b # bs))
  | merge [] bs = bs
  | merge as [] = as

fun merge-pairs :: 'a list list ⇒ 'a list list
  where
    merge-pairs (a # b # xs) = merge a b # merge-pairs xs
  | merge-pairs xs = xs

lemma length-merge [simp]:
  length (merge xs ys) = length xs + length ys
  ⟨proof⟩

lemma length-merge-pairs [simp]:
  length (merge-pairs xs) = (1 + length xs) div 2
  ⟨proof⟩

fun merge-all :: 'a list list ⇒ 'a list
  where
    merge-all [] = []
  | merge-all [x] = x
  | merge-all xs = merge-all (merge-pairs xs)

fun msort-key :: 'a list ⇒ 'a list
  where
    msort-key xs = merge-all (sequences xs)
```

1.2 The Functional Argument of *local.asc*

f is a function that only adds some prefix to a given list.

```
definition ascP f = (∀ xs. f xs = f [] @ xs)

lemma ascP-Cons [simp]: ascP ((#) x) ⟨proof⟩
```

lemma *ascP-comp-append-Cons* [*simp*]:
ascP (λ*xs*. *f* [] @ *x* # *xs*)
⟨*proof*⟩

lemma *ascP-f-Cons*:
assumes *ascP f*
shows *f* (*x* # *xs*) = *f* [] @ *x* # *xs*
⟨*proof*⟩

lemma *ascP-comp-Cons* [*simp*]:
assumes *ascP f*
shows *ascP* (λ*ys*. *f* (*x* # *ys*))
⟨*proof*⟩

lemma *ascP-f-singleton*:
assumes *ascP f*
shows *f* [*x*] = *f* [] @ [*x*]
⟨*proof*⟩

1.3 Facts about Lengths

lemma
shows *length-sequences*: *length* (*sequences xs*) ≤ *length xs*
and *length-asc*: *ascP f* ⟹ *length* (*asc a f ys*) ≤ *1* + *length ys*
and *length-desc*: *length* (*desc a xs ys*) ≤ *1* + *length ys*
⟨*proof*⟩

lemma *length-concat-merge-pairs* [*simp*]:
length (*concat* (*merge-pairs xss*)) = *length* (*concat xss*)
⟨*proof*⟩

1.4 Functional Correctness

lemma *mset-merge* [*simp*]:
mset (*merge xs ys*) = *mset xs* + *mset ys*
⟨*proof*⟩

lemma *set-merge* [*simp*]:
set (*merge xs ys*) = *set xs* ∪ *set ys*
⟨*proof*⟩

lemma *mset-concat-merge-pairs* [*simp*]:
mset (*concat* (*merge-pairs xs*)) = *mset* (*concat xs*)
⟨*proof*⟩

lemma *set-concat-merge-pairs* [*simp*]:
set (*concat* (*merge-pairs xs*)) = *set* (*concat xs*)
⟨*proof*⟩

lemma *mset-merge-all* [*simp*]:

mset (*merge-all xs*) = *mset* (*concat xs*)
⟨*proof*⟩

lemma *set-merge-all* [*simp*]:
set (*merge-all xs*) = *set* (*concat xs*)
⟨*proof*⟩

lemma
shows *mset-seqeuences* [*simp*]: *mset* (*concat* (*sequences xs*)) = *mset xs*
and *mset-asc*: *ascP f* ⟹ *mset* (*concat* (*asc x f ys*)) = {#*x*#} + *mset* (*f* []) + *mset ys*
and *mset-desc*: *mset* (*concat* (*desc x xs ys*)) = {#*x*#} + *mset xs* + *mset ys*
⟨*proof*⟩

lemma *mset-msort-key*:
mset (*msort-key xs*) = *mset xs*
⟨*proof*⟩

lemma *sorted-merge* [*simp*]:
assumes *sorted* (*map key xs*) **and** *sorted* (*map key ys*)
shows *sorted* (*map key* (*merge xs ys*))
⟨*proof*⟩

lemma *sorted-merge-pairs* [*simp*]:
assumes ∀ *x*∈*set xs*. *sorted* (*map key x*)
shows ∀ *x*∈*set* (*merge-pairs xs*). *sorted* (*map key x*)
⟨*proof*⟩

lemma *sorted-merge-all*:
assumes ∀ *x*∈*set xs*. *sorted* (*map key x*)
shows *sorted* (*map key* (*merge-all xs*))
⟨*proof*⟩

lemma
shows *sorted-sequences*: ∀ *x* ∈ *set* (*sequences xs*). *sorted* (*map key x*)
and *sorted-asc*: *ascP f* ⟹ *sorted* (*map key* (*f* [])) ⟹ ∀ *x*∈*set* (*f* []). *key x* ≤ *key a* ⟹ ∀ *x*∈*set* (*asc a f ys*). *sorted* (*map key x*)
and *sorted-desc*: *sorted* (*map key xs*) ⟹ ∀ *x*∈*set xs*. *key a* ≤ *key x* ⟹ ∀ *x*∈*set* (*desc a xs ys*). *sorted* (*map key x*)
⟨*proof*⟩

lemma *sorted-msort-key*:
sorted (*map key* (*msort-key xs*))
⟨*proof*⟩

1.5 Stability

lemma
shows *filter-by-key-sequences* [*simp*]: [*y*←*concat* (*sequences xs*). *key y* = *k*] =

$[y \leftarrow xs.\ key\ y = k]$
and *filter-by-key-asc*: $ascP\ f \Longrightarrow [y \leftarrow concat\ (asc\ a\ f\ ys).\ key\ y = k] = [y \leftarrow f\ [a]\ @\ ys.\ key\ y = k]$
and *filter-by-key-desc*: $sorted\ (map\ key\ xs) \Longrightarrow \forall x \in set\ xs.\ key\ a \leq key\ x \Longrightarrow [y \leftarrow concat\ (desc\ a\ xs\ ys).\ key\ y = k] = [y \leftarrow a\ \#\ xs\ @\ ys.\ key\ y = k]$
⟨*proof*⟩

lemma *filter-by-key-merge-is-append* [*simp*]:
assumes $sorted\ (map\ key\ xs)$
shows $[y \leftarrow merge\ xs\ ys.\ key\ y = k] = [y \leftarrow xs.\ key\ y = k]\ @\ [y \leftarrow ys.\ key\ y = k]$
⟨*proof*⟩

lemma *filter-by-key-merge-pairs* [*simp*]:
assumes $\forall xs \in set\ xss.\ sorted\ (map\ key\ xs)$
shows $[y \leftarrow concat\ (merge\text{-}pairs\ xss).\ key\ y = k] = [y \leftarrow concat\ xss.\ key\ y = k]$
⟨*proof*⟩

lemma *filter-by-key-merge-all* [*simp*]:
assumes $\forall xs \in set\ xss.\ sorted\ (map\ key\ xs)$
shows $[y \leftarrow merge\text{-}all\ xss.\ key\ y = k] = [y \leftarrow concat\ xss.\ key\ y = k]$
⟨*proof*⟩

lemma *filter-by-key-merge-all-sequences* [*simp*]:
$[x \leftarrow merge\text{-}all\ (sequences\ xs)\ .\ key\ x = k] = [x \leftarrow xs\ .\ key\ x = k]$
⟨*proof*⟩

lemma *msort-key-stable*:
$[x \leftarrow msort\text{-}key\ xs.\ key\ x = k] = [x \leftarrow xs.\ key\ x = k]$
⟨*proof*⟩

lemma *sort-key-msort-key-conv*:
$sort\text{-}key\ key = msort\text{-}key$
⟨*proof*⟩

end

Replace existing code equations for *sort-key* by *msort-key*.

declare *sort-key-by-quicksort-code* [*code del*]
declare *sort-key-msort-key-conv* [*code*]

end

theory *Mergesort-Complexity*
imports
Efficient-Sort
Complex-Main
begin

lemma *log2-mono*:
x > 0 ⟹ x ≤ y ⟹ log 2 x ≤ log 2 y
⟨*proof*⟩

2 Counting the Number of Comparisons

context
fixes *key* :: *'a ⇒ 'k*::*linorder*
begin

fun *c-merge* :: *'a list ⇒ 'a list ⇒ nat*
where
c-merge (*x # xs*) (*y # ys*) =
1 + (if key y < key x then c-merge (*x # xs*) *ys else c-merge xs* (*y # ys*))
| *c-merge* [] *ys = 0*
| *c-merge xs* [] *= 0*

fun *c-merge-pairs* :: *'a list list ⇒ nat*
where
c-merge-pairs (*xs # ys # zss*) = *c-merge xs ys + c-merge-pairs zss*
| *c-merge-pairs* [] *= 0*
| *c-merge-pairs* [*x*] *= 0*

fun *c-merge-all* :: *'a list list ⇒ nat*
where
c-merge-all [] *= 0*
| *c-merge-all* [*x*] *= 0*
| *c-merge-all xss = c-merge-pairs xss + c-merge-all* (*merge-pairs key xss*)

fun *c-sequences* :: *'a list ⇒ nat*
and *c-asc* :: *'a ⇒ 'a list ⇒ nat*
and *c-desc* :: *'a ⇒ 'a list ⇒ nat*
where
c-sequences (*x # y # zs*) = *1 + (if key y < key x then c-desc y zs else c-asc y zs*)
| *c-sequences* [] *= 0*
| *c-sequences* [*x*] *= 0*
| *c-asc x* (*y # ys*) = *1 + (if ¬ key y < key x then c-asc y ys else c-sequences* (*y # ys*))
| *c-asc x* [] *= 0*
| *c-desc x* (*y # ys*) = *1 + (if key y < key x then c-desc y ys else c-sequences* (*y # ys*))
| *c-desc x* [] *= 0*

fun *c-msort* :: *'a list ⇒ nat*
where
c-msort xs = c-sequences xs + c-merge-all (*sequences key xs*)

lemma *c-merge*:

c-merge xs ys ≤ *length xs* + *length ys*
⟨*proof*⟩

lemma *c-merge-pairs*:
c-merge-pairs xss ≤ *length* (*concat xss*)
⟨*proof*⟩

lemma *c-merge-all*:
c-merge-all xss ≤ *length* (*concat xss*) ∗ ⌈*log 2* (*length xss*)⌉
⟨*proof*⟩

lemma
shows *c-sequences*: *c-sequences xs* ≤ *length xs* − *1*
and *c-asc*: *c-asc x ys* ≤ *length ys*
and *c-desc*: *c-desc x ys* ≤ *length ys*
⟨*proof*⟩

lemma
shows *length-concat-sequences* [*simp*]: *length* (*concat* (*sequences key xs*)) = *length xs*
and *length-concat-asc*: *ascP f* ⟹ *length* (*concat* (*asc key a f ys*)) = *1* + *length* (*f* []) + *length ys*
and *length-concat-desc*: *length* (*concat* (*desc key a xs ys*)) = *1* + *length xs* + *length ys*
⟨*proof*⟩

lemma
shows *sequences-ne*: *xs* ≠ [] ⟹ *sequences key xs* ≠ []
and *asc-ne*: *ascP f* ⟹ *asc key a f ys* ≠ []
and *desc-ne*: *desc key a xs ys* ≠ []
⟨*proof*⟩

lemma *c-msort*:
assumes [*simp*]: *length xs* = *n*
shows *c-msort xs* ≤ *n* + *n* ∗ ⌈*log 2 n*⌉
⟨*proof*⟩

end

end

theory *Natural-Mergesort*
imports *HOL−Data-Structures.Sorting*
begin

2.1 Auxiliary Results

lemma *C-merge-adj′*:

C-merge-adj xss $\leq$ *length* (*concat xss*)
⟨*proof*⟩

lemma *length-concat-merge-adj*:
length (*concat* (*merge-adj xss*)) = *length* (*concat xss*)
⟨*proof*⟩

lemma *C-merge-all'*:
C-merge-all xss $\leq$ *length* (*concat xss*) $*$ $\lceil$*log 2* (*length xss*)$\rceil$
⟨*proof*⟩

2.2 Definition of Natural Mergesort

Partition input into ascending and descending subsequences. (The latter are reverted on the fly.)

fun *runs* :: ('*a*::*linorder*) *list* ⇒ '*a list list* **and**
asc :: '*a* ⇒ ('*a list* ⇒ '*a list*) ⇒ '*a list* ⇒ '*a list list* **and**
desc :: '*a* ⇒ '*a list* ⇒ '*a list* ⇒ '*a list list*
where
runs (*a* # *b* # *xs*) = (*if a* > *b then desc b* [*a*] *xs else asc b* ((#) *a*) *xs*)
| *runs* [*x*] = [[*x*]]
| *runs* [] = []
| *asc a as* (*b* # *bs*) = (*if* ¬ *a* > *b then asc b* (*as* ∘ (#) *a*) *bs else as* [*a*] # *runs* (*b* # *bs*))
| *asc a as* [] = [*as* [*a*]]
| *desc a as* (*b* # *bs*) = (*if a* > *b then desc b* (*a* # *as*) *bs else* (*a* # *as*) # *runs* (*b* # *bs*))
| *desc a as* [] = [*a* # *as*]

definition *nmsort* :: ('*a*::*linorder*) *list* ⇒ '*a list*
where
nmsort xs = *merge-all* (*runs xs*)

2.3 Functional Correctness

definition *ascP f* = (∀ *xs ys*. *f* (*xs* @ *ys*) = *f xs* @ *ys*)

lemma *ascP-Cons* [*simp*]: *ascP* ((#) *x*) ⟨*proof*⟩

lemma *ascP-comp-Cons* [*simp*]: *ascP f* ⟹ *ascP* (*f* ∘ (#) *x*)
⟨*proof*⟩

lemma *ascP-simp* [*simp*]:
assumes *ascP f*
shows *f* [*x*] = *f* [] @ [*x*]
⟨*proof*⟩

lemma
shows *mset-runs*: $\sum_{\#}$ (*image-mset mset* (*mset* (*runs xs*))) = *mset xs*

and *mset-asc*: $ascP\ f \Longrightarrow \sum_{\#}\ (image\text{-}mset\ mset\ (mset\ (asc\ x\ f\ ys))) = \{\#x\#\} + mset\ (f\ []) + mset\ ys$
and *mset-desc*: $\sum_{\#}\ (image\text{-}mset\ mset\ (mset\ (desc\ x\ xs\ ys))) = \{\#x\#\} + mset\ xs + mset\ ys$
⟨*proof*⟩

lemma *mset-nmsort*:
$mset\ (nmsort\ xs) = mset\ xs$
⟨*proof*⟩

lemma
shows *sorted-runs*: $\forall x{\in}set\ (runs\ xs).\ sorted\ x$
and *sorted-asc*: $ascP\ f \Longrightarrow sorted\ (f\ []) \Longrightarrow \forall x{\in}set\ (f\ []).\ x \leq a \Longrightarrow \forall x{\in}set\ (asc\ a\ f\ ys).\ sorted\ x$
and *sorted-desc*: $sorted\ xs \Longrightarrow \forall x{\in}set\ xs.\ a \leq x \Longrightarrow \forall x{\in}set\ (desc\ a\ xs\ ys).\ sorted\ x$
⟨*proof*⟩

lemma *sorted-nmsort*:
$sorted\ (nmsort\ xs)$
⟨*proof*⟩

2.4 Running Time Analysis

fun *C-runs* :: $('a{::}linorder)\ list \Rightarrow nat$ **and**
C-asc :: $('a{::}linorder) \Rightarrow 'a\ list \Rightarrow nat$ **and**
C-desc :: $('a{::}linorder) \Rightarrow 'a\ list \Rightarrow nat$
where
$C\text{-}runs\ (a\ \#\ b\ \#\ xs) = 1 + (if\ a > b\ then\ C\text{-}desc\ b\ xs\ else\ C\text{-}asc\ b\ xs)$
| $C\text{-}runs\ xs = 0$
| $C\text{-}asc\ a\ (b\ \#\ bs) = 1 + (if\ \neg\ a > b\ then\ C\text{-}asc\ b\ bs\ else\ C\text{-}runs\ (b\ \#\ bs))$
| $C\text{-}asc\ a\ [] = 0$
| $C\text{-}desc\ a\ (b\ \#\ bs) = 1 + (if\ a > b\ then\ C\text{-}desc\ b\ bs\ else\ C\text{-}runs\ (b\ \#\ bs))$
| $C\text{-}desc\ a\ [] = 0$

fun *C-nmsort* :: $('a{::}linorder)\ list \Rightarrow nat$
where
$C\text{-}nmsort\ xs = C\text{-}runs\ xs + C\text{-}merge\text{-}all\ (runs\ xs)$

lemma
fixes $a :: 'a{::}linorder$ **and** $xs\ ys :: 'a\ list$
shows *C-runs*: $C\text{-}runs\ xs \leq length\ xs - 1$
and *C-asc*: $C\text{-}asc\ a\ ys \leq length\ ys$
and *C-desc*: $C\text{-}desc\ a\ ys \leq length\ ys$
⟨*proof*⟩

lemma
shows *length-runs*: $length\ (runs\ xs) \leq length\ xs$
and *length-asc*: $ascP\ f \Longrightarrow length\ (asc\ a\ f\ ys) \leq 1 + length\ ys$

and *length-desc*: *length* (*desc a xs ys*) $\leq$ *1* + *length ys*
⟨*proof*⟩

lemma
shows *length-concat-runs* [*simp*]: *length* (*concat* (*runs xs*)) = *length xs*
and *length-concat-asc*: *ascP f* $\Longrightarrow$ *length* (*concat* (*asc a f ys*)) = *1* + *length* (*f* []) + *length ys*
and *length-concat-desc*: *length* (*concat* (*desc a xs ys*)) = *1* + *length xs* + *length ys*
⟨*proof*⟩

lemma *log2-mono*:
x > *0* $\Longrightarrow$ *x* $\leq$ *y* $\Longrightarrow$ *log 2 x* $\leq$ *log 2 y*
⟨*proof*⟩

lemma
shows *runs-ne*: *xs* $\neq$ [] $\Longrightarrow$ *runs xs* $\neq$ []
and *ascP f* $\Longrightarrow$ *asc a f ys* $\neq$ []
and *desc a xs ys* $\neq$ []
⟨*proof*⟩

lemma *C-nmsort*:
assumes [*simp*]: *length xs* = *n*
shows *C-nmsort xs* $\leq$ *n* + *n* $*$ $\lceil$*log 2 n*$\rceil$
⟨*proof*⟩

end

References

[1] Christian Sternagel. Proof pearl - a mechanized proof of GHC's mergesort. *Journal of Automated Reasoning*, 2012. doi:10.1007/s10817-012-9260-7.