Efficient Mergesort

Christian Sternagel

March 17, 2025

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 $key :: {'a} \Rightarrow {'k}::linorder$
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 $sequences :: {'a}\ list \Rightarrow {'a}\ list\ list$
  and $asc :: {'a} \Rightarrow ({'a}\ list \Rightarrow {'a}\ list) \Rightarrow {'a}\ list \Rightarrow {'a}\ list\ list$
  and $desc :: {'a} \Rightarrow {'a}\ list \Rightarrow {'a}\ list \Rightarrow {'a}\ list\ list$
  where
    $sequences\ (a\ \#\ b\ \#\ xs) =$

$(if\ key\ a > key\ b\ then\ desc\ b\ [a]\ xs\ else\ asc\ b\ ((\#)\ a)\ xs)$
$|\ sequences\ [x] = [[x]]$
$|\ sequences\ [] = []$
$|\ asc\ a\ as\ (b\ \#\ bs) =$
$(if\ key\ a \leq key\ b\ then\ asc\ b\ (\lambda ys.\ as\ (a\ \#\ ys))\ bs$
$else\ as\ [a]\ \#\ sequences\ (b\ \#\ bs))$
$|\ asc\ a\ as\ [] = [as\ [a]]$
$|\ desc\ a\ as\ (b\ \#\ bs) =$
$(if\ key\ a > key\ b\ then\ desc\ b\ (a\ \#\ as)\ bs$
$else\ (a\ \#\ as)\ \#\ sequences\ (b\ \#\ bs))$
$|\ desc\ a\ as\ [] = [a\ \#\ as]$

fun *merge* :: $'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$
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* :: $'a\ list\ list \Rightarrow 'a\ list\ list$
where
$merge\text{-}pairs\ (a\ \#\ b\ \#\ xs) = merge\ a\ b\ \#\ merge\text{-}pairs\ xs$
$|\ merge\text{-}pairs\ xs = xs$

lemma *length-merge* [*simp*]:
$length\ (merge\ xs\ ys) = length\ xs + length\ ys$
by (*induct xs ys rule*: *merge.induct*) *simp-all*

lemma *length-merge-pairs* [*simp*]:
$length\ (merge\text{-}pairs\ xs) = (1 + length\ xs)\ div\ 2$
by (*induct xs rule*: *merge-pairs.induct*) *simp-all*

fun *merge-all* :: $'a\ list\ list \Rightarrow 'a\ list$
where
$merge\text{-}all\ [] = []$
$|\ merge\text{-}all\ [x] = x$
$|\ merge\text{-}all\ xs = merge\text{-}all\ (merge\text{-}pairs\ xs)$

fun *msort-key* :: $'a\ list \Rightarrow 'a\ list$
where
$msort\text{-}key\ xs = merge\text{-}all\ (sequences\ xs)$

1.2 The Functional Argument of *local.asc*

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

definition $ascP\ f = (\forall xs.\ f\ xs = f\ []\ @\ xs)$

lemma *ascP-Cons* [*simp*]: $ascP\ ((\#)\ x)$ **by** (*simp add*: *ascP-def*)

lemma *ascP-comp-append-Cons* [*simp*]:
  ascP (*λxs. f* [] @ *x* # *xs*)
  by (*auto simp*: *ascP-def*)

lemma *ascP-f-Cons*:
  assumes *ascP f*
  shows *f* (*x* # *xs*) = *f* [] @ *x* # *xs*
  using ‹*ascP f*› [*unfolded ascP-def*, *THEN spec*, *of x* # *xs*] **.**

lemma *ascP-comp-Cons* [*simp*]:
  assumes *ascP f*
  shows *ascP* (*λys. f* (*x* # *ys*))
proof (*unfold ascP-def*, *intro allI*)
  fix *xs* **show** *f* (*x* # *xs*) = *f* [*x*] @ *xs*
    using *assms* **by** (*simp add*: *ascP-f-Cons*)
qed

lemma *ascP-f-singleton*:
  assumes *ascP f*
  shows *f* [*x*] = *f* [] @ [*x*]
  by (*rule ascP-f-Cons* [*OF assms*])

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*
  by (*induct xs* **and** *a f ys* **and** *a xs ys rule*: *sequences-asc-desc.induct*) *auto*

lemma *length-concat-merge-pairs* [*simp*]:
  length (*concat* (*merge-pairs xss*)) = *length* (*concat xss*)
  by (*induct xss rule*: *merge-pairs.induct*) *simp-all*

1.4 Functional Correctness

lemma *mset-merge* [*simp*]:
  mset (*merge xs ys*) = *mset xs* + *mset ys*
  by (*induct xs ys rule*: *merge.induct*) *simp-all*

lemma *set-merge* [*simp*]:
  set (*merge xs ys*) = *set xs* ∪ *set ys*
  by (*simp flip*: *set-mset-mset*)

lemma *mset-concat-merge-pairs* [*simp*]:
  mset (*concat* (*merge-pairs xs*)) = *mset* (*concat xs*)
  by (*induct xs rule*: *merge-pairs.induct*) *auto*

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

by (*simp flip*: *set-mset-mset*)

lemma *mset-merge-all* [*simp*]:
mset (*merge-all xs*) = *mset* (*concat xs*)
by (*induct xs rule*: *merge-all.induct*) *simp-all*

lemma *set-merge-all* [*simp*]:
set (*merge-all xs*) = *set* (*concat xs*)
by (*simp flip*: *set-mset-mset*)

lemma
shows *mset-seqeuences* [*simp*]: *mset* (*concat* (*sequences xs*)) = *mset xs*
and *mset-asc*: *ascP f* $\Longrightarrow$ *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*
by (*induct xs* **and** *x f ys* **and** *x xs ys rule*: *sequences-asc-desc.induct*)
(*auto simp*: *ascP-f-singleton*)

lemma *mset-msort-key*:
mset (*msort-key xs*) = *mset xs*
by (*auto*)

lemma *sorted-merge* [*simp*]:
assumes *sorted* (*map key xs*) **and** *sorted* (*map key ys*)
shows *sorted* (*map key* (*merge xs ys*))
using *assms* **by** (*induct xs ys rule*: *merge.induct*) (*auto*)

lemma *sorted-merge-pairs* [*simp*]:
assumes $\forall$ *x*$\in$*set xs*. *sorted* (*map key x*)
shows $\forall$ *x*$\in$*set* (*merge-pairs xs*). *sorted* (*map key x*)
using *assms* **by** (*induct xs rule*: *merge-pairs.induct*) *simp-all*

lemma *sorted-merge-all*:
assumes $\forall$ *x*$\in$*set xs*. *sorted* (*map key x*)
shows *sorted* (*map key* (*merge-all xs*))
using *assms* **by** (*induct xs rule*: *merge-all.induct*) *simp-all*

lemma
shows *sorted-sequences*: $\forall$ *x* $\in$ *set* (*sequences xs*). *sorted* (*map key x*)
and *sorted-asc*: *ascP f* $\Longrightarrow$ *sorted* (*map key* (*f* [])) $\Longrightarrow$ $\forall$ *x*$\in$*set* (*f* []). *key x* $\leq$ *key a* $\Longrightarrow$ $\forall$ *x*$\in$*set* (*asc a f ys*). *sorted* (*map key x*)
and *sorted-desc*: *sorted* (*map key xs*) $\Longrightarrow$ $\forall$ *x*$\in$*set xs*. *key a* $\leq$ *key x* $\Longrightarrow$ $\forall$ *x*$\in$*set* (*desc a xs ys*). *sorted* (*map key x*)
by (*induct xs* **and** *a f ys* **and** *a xs ys rule*: *sequences-asc-desc.induct*)
(*auto simp*: *ascP-f-singleton sorted-append not-less dest*: *order-trans*, *fastforce*)

lemma *sorted-msort-key*:
sorted (*map key* (*msort-key xs*))
by (*unfold msort-key.simps*) (*intro sorted-merge-all sorted-sequences*)

1.5 Stability

lemma
shows *filter-by-key-sequences* [*simp*]: [y←*concat* (*sequences xs*). *key* y = k] = [y←*xs*. *key* y = k]
and *filter-by-key-asc*: *ascP f* ⟹ [y←*concat* (*asc a f ys*). *key* y = k] = [y←*f* [a] @ *ys*. *key* y = k]
and *filter-by-key-desc*: *sorted* (*map key xs*) ⟹ ∀ *x*∈*set xs*. *key a* ≤ *key x* ⟹ [y←*concat* (*desc a xs ys*). *key* y = k] = [y←*a* # *xs* @ *ys*. *key* y = k]
proof (*induct xs* **and** *a f ys* **and** *a xs ys rule*: *sequences-asc-desc.induct*)
case (*4 a f b bs*)
then show *?case*
by (*auto simp*: *o-def ascP-f-Cons* [**where** *f* = *f*])
next
case (*6 a as b bs*)
then show *?case*
proof (*cases key b* < *key a*)
case *True*
with *6* **have** [y←*concat* (*desc b* (*a* # *as*) *bs*). *key* y = k] = [y←*b* # (*a* # *as*) @ *bs*. *key* y = k]
by (*auto simp*: *less-le order-trans*)
then show *?thesis*
using *True* **and** *6*
by (*cases key a* = k, *cases key b* = k)
(*auto simp*: *Cons-eq-append-conv intro*!: *filter-False*)
qed *auto*
qed *auto*

lemma *filter-by-key-merge-is-append* [*simp*]:
assumes *sorted* (*map key xs*)
shows [y←*merge xs ys*. *key* y = k] = [y←*xs*. *key* y = k] @ [y←*ys*. *key* y = k]
using *assms*
by (*induct xs ys rule*: *merge.induct*) (*auto simp*: *Cons-eq-append-conv leD intro*!: *filter-False*)

lemma *filter-by-key-merge-pairs* [*simp*]:
assumes ∀ *xs*∈*set xss*. *sorted* (*map key xs*)
shows [y←*concat* (*merge-pairs xss*). *key* y = k] = [y←*concat xss*. *key* y = k]
using *assms* **by** (*induct xss rule*: *merge-pairs.induct*) *simp-all*

lemma *filter-by-key-merge-all* [*simp*]:
assumes ∀ *xs*∈*set xss*. *sorted* (*map key xs*)
shows [y←*merge-all xss*. *key* y = k] = [y←*concat xss*. *key* y = k]
using *assms* **by** (*induct xss rule*: *merge-all.induct*) *simp-all*

lemma *filter-by-key-merge-all-sequences* [*simp*]:
[x←*merge-all* (*sequences xs*) . *key* x = k] = [x←*xs* . *key* x = k]
using *sorted-sequences* [*of xs*] **by** *simp*

lemma *msort-key-stable*:

$[x \leftarrow msort\text{-}key\ xs.\ key\ x = k] = [x \leftarrow xs.\ key\ x = k]$
by *auto*

lemma *sort-key-msort-key-conv*:
sort-key key = *msort-key*
using *msort-key-stable* [*of key x* **for** *x*]
by (*intro ext properties-for-sort-key mset-msort-key sorted-msort-key*)
(*metis* (*mono-tags*, *lifting*) *filter-cong*)

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*:
$x > 0 \Longrightarrow x \leq y \Longrightarrow log\ 2\ x \leq log\ 2\ y$
by *auto*

2 Counting the Number of Comparisons

context
fixes *key* :: $'a \Rightarrow 'k$::*linorder*
begin

fun *c-merge* :: $'a\ list \Rightarrow 'a\ list \Rightarrow nat$
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* :: $'a\ list\ list \Rightarrow nat$
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* :: $'a\ list\ list \Rightarrow nat$
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
 by (induct xs ys rule: c-merge.induct) simp-all

lemma c-merge-pairs:
 c-merge-pairs xss ≤ length (concat xss)
proof (induct xss rule: c-merge-pairs.induct)
 case (1 xs ys zss)
 then show ?case using c-merge [of xs ys] by simp
qed simp-all

lemma c-merge-all:
 c-merge-all xss ≤ length (concat xss) ∗ ⌈log 2 (length xss)⌉
proof (induction xss rule: c-merge-all.induct)
 case (3 xs ys zss)
 let ?clen = λxs. length (concat xs)
 let ?xss = xs # ys # zss
 let ?xss2 = merge-pairs key ?xss

 have ∗: ⌈log 2 (real n + 2)⌉ = ⌈log 2 (Suc n div 2 + 1)⌉ + 1 for n :: nat
  using ceiling-log2-div2 [of n + 2] by (simp add: algebra-simps)

 have c-merge-all ?xss = c-merge-pairs ?xss + c-merge-all ?xss2 by simp
 also have . . . ≤ ?clen ?xss + c-merge-all ?xss2
  using c-merge [of xs ys] and c-merge-pairs [of ?xss] by auto
 also have . . . ≤ ?clen ?xss + ?clen ?xss2 ∗ ⌈log 2 (length ?xss2)⌉
```

using *3.IH* **by** *simp*
also have $\ldots \leq$ *?clen ?xss* $*$ $\lceil$*log 2* (*length ?xss*)$\rceil$
by (*auto simp*: $*$ *algebra-simps*)
finally show *?case* **by** *simp*
qed *simp-all*

lemma
shows *c-sequences*: *c-sequences xs* $\leq$ *length xs* $-$ *1*
and *c-asc*: *c-asc x ys* $\leq$ *length ys*
and *c-desc*: *c-desc x ys* $\leq$ *length ys*
by (*induct xs* **and** *x ys* **and** *x ys rule*: *c-sequences-c-asc-c-desc.induct*) *simp-all*

lemma
shows *length-concat-sequences* [*simp*]: *length* (*concat* (*sequences key xs*)) = *length xs*
and *length-concat-asc*: *ascP f* $\Longrightarrow$ *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*
by (*induct xs* **and** *a f ys* **and** *a xs ys rule*: *sequences-asc-desc.induct*)
(*auto simp*: *ascP-f-singleton*)

lemma
shows *sequences-ne*: *xs* $\neq$ [] $\Longrightarrow$ *sequences key xs* $\neq$ []
and *asc-ne*: *ascP f* $\Longrightarrow$ *asc key a f ys* $\neq$ []
and *desc-ne*: *desc key a xs ys* $\neq$ []
by (*induct xs* **and** *a f ys* **and** *a xs ys taking*: *key rule*: *sequences-asc-desc.induct*) *simp-all*

lemma *c-msort*:
assumes [*simp*]: *length xs* = *n*
shows *c-msort xs* $\leq$ *n* + *n* $*$ $\lceil$*log 2 n*$\rceil$
proof $-$
have [*simp*]: *xs* = [] $\longleftrightarrow$ *length xs* = *0* **by** *blast*
have *int* (*c-merge-all* (*sequences key xs*)) $\leq$ *int n* $*$ $\lceil$*log 2* (*length* (*sequences key xs*))$\rceil$
using *c-merge-all* [*of sequences key xs*] **by** *simp*
also have $\ldots \leq$ *int n* $*$ $\lceil$*log 2 n*$\rceil$
using *length-sequences* [*of key xs*]
by (*cases n*) (*auto intro*!: *sequences-ne mult-mono ceiling-mono log2-mono*)
finally have *int* (*c-merge-all* (*sequences key xs*)) $\leq$ *int n* $*$ $\lceil$*log 2 n*$\rceil$ **.**
moreover have *c-sequences xs* $\leq$ *n* **using** *c-sequences* [*of xs*] **by** *auto*
ultimately show *?thesis* **by** (*auto intro*: *add-mono*)
qed

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 (*induct xss rule*: *C-merge-adj.induct*)
case (*3 xs ys zss*)
then show *?case* **using** *C-merge-ub* [*of xs ys*] **by** *simp*
qed *simp-all*

lemma *length-concat-merge-adj*:
length (*concat* (*merge-adj xss*)) = *length* (*concat xss*)
by (*induct xss rule*: *merge-adj.induct*) (*simp-all add*: *length-merge*)

lemma *C-merge-all′*:
C-merge-all xss $\leq$ *length* (*concat xss*) $*$ $\lceil$*log 2* (*length xss*)$\rceil$
proof (*induction xss rule*: *C-merge-all.induct*)
case (*3 xs ys zss*)
let *?xss* = *xs* # *ys* # *zss*
let *?m* = *length* (*concat ?xss*)

have $*$: $\lceil$*log 2* (*real n* + *2*)$\rceil$ = $\lceil$*log 2* (*Suc n div 2* + *1*)$\rceil$ + *1* **for** *n* :: *nat*
using *ceiling-log2-div2* [*of n* + *2*] **by** (*simp add*: *algebra-simps*)

have *C-merge-all ?xss* = *C-merge-adj ?xss* + *C-merge-all* (*merge-adj ?xss*) **by** *simp*
also have . . . $\leq$ *?m* + *C-merge-all* (*merge-adj ?xss*)
using *C-merge-adj′* [*of ?xss*] **by** *auto*
also have . . . $\leq$ *?m* + *length* (*concat* (*merge-adj ?xss*)) $*$ $\lceil$*log 2* (*length* (*merge-adj ?xss*))$\rceil$
using *3.IH* **by** *simp*
also have . . . = *?m* + *?m* $*$ $\lceil$*log 2* (*length* (*merge-adj ?xss*))$\rceil$
by (*simp only*: *length-concat-merge-adj*)
also have . . . $\leq$ *?m* $*$ $\lceil$*log 2* (*length ?xss*)$\rceil$
by (*auto simp*: $*$ *algebra-simps*)
finally show *?case* **by** *simp*
qed *simp-all*

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* $a\ as\ (b \# bs) = (if\ \neg\ a > b\ then\ asc\ b\ (as \circ (\#)\ a)\ bs\ else\ as\ [a] \#\ runs\ (b \# bs))$
| *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* :: $('a::linorder)\ list \Rightarrow 'a\ list$
where
  nmsort xs = *merge-all* (*runs xs*)

2.3 Functional Correctness

definition *ascP f* = $(\forall\, xs\ ys.\ f\ (xs\ @\ ys) = f\ xs\ @\ ys)$

lemma *ascP-Cons* [*simp*]: *ascP* ((#) *x*) **by** (*simp add*: *ascP-def*)

lemma *ascP-comp-Cons* [*simp*]: *ascP f* $\Longrightarrow$ *ascP* $(f \circ (\#)\ x)$
by (*auto simp*: *ascP-def simp flip*: *append-Cons*)

lemma *ascP-simp* [*simp*]:
assumes *ascP f*
shows $f\ [x] = f\ []\ @\ [x]$
using *assms* [*unfolded ascP-def*, *THEN spec*, *THEN spec*, *of* [] [*x*]] **by** *simp*

lemma
shows *mset-runs*: $\sum_{\#}$ (*image-mset mset* (*mset* (*runs xs*))) = *mset xs*
and *mset-asc*: *ascP f* $\Longrightarrow \sum_{\#}$ (*image-mset mset* (*mset* (*asc x f ys*))) = {#*x*#} + *mset* (*f* []) + *mset ys*
and *mset-desc*: $\sum_{\#}$ (*image-mset mset* (*mset* (*desc x xs ys*))) = {#*x*#} + *mset xs* + *mset ys*
by (*induct xs* **and** *x f ys* **and** *x xs ys rule*: *runs-asc-desc.induct*) *auto*

lemma *mset-nmsort*:
mset (*nmsort xs*) = *mset xs*
by (*auto simp*: *mset-merge-all nmsort-def mset-runs*)

lemma
shows *sorted-runs*: $\forall\, x \in set\ (runs\ xs).\ sorted\ x$
and *sorted-asc*: $ascP\ f \Longrightarrow sorted\ (f\ []) \Longrightarrow \forall\, x \in set\ (f\ []).\ x \le a \Longrightarrow \forall\, x \in set\ (asc\ a\ f\ ys).\ sorted\ x$
and *sorted-desc*: $sorted\ xs \Longrightarrow \forall\, x \in set\ xs.\ a \le x \Longrightarrow \forall\, x \in set\ (desc\ a\ xs\ ys).\ sorted\ x$
by (*induct xs* **and** *a f ys* **and** *a xs ys rule*: *runs-asc-desc.induct*)
  (*auto simp*: *sorted-append not-less dest*: *order-trans*, *fastforce*)

lemma *sorted-nmsort*:
sorted (*nmsort xs*)
by (*auto intro*: *sorted-merge-all simp*: *nmsort-def sorted-runs*)

2.4 Running Time Analysis

fun *C-runs* :: (*'a*::*linorder*) *list* ⇒ *nat* **and**
C-asc :: (*'a*::*linorder*) ⇒ *'a list* ⇒ *nat* **and**
C-desc :: (*'a*::*linorder*) ⇒ *'a list* ⇒ *nat*
where
C-runs (*a* # *b* # *xs*) = *1* + (*if a* > *b then C-desc b xs else C-asc b xs*)
| *C-runs xs* = *0*
| *C-asc a* (*b* # *bs*) = *1* + (*if* ¬ *a* > *b then C-asc b bs else C-runs* (*b* # *bs*))
| *C-asc a* [] = *0*
| *C-desc a* (*b* # *bs*) = *1* + (*if a* > *b then C-desc b bs else C-runs* (*b* # *bs*))
| *C-desc a* [] = *0*

fun *C-nmsort* :: (*'a*::*linorder*) *list* ⇒ *nat*
where
C-nmsort xs = *C-runs xs* + *C-merge-all* (*runs xs*)

lemma
fixes *a* :: *'a*::*linorder* **and** *xs ys* :: *'a list*
shows *C-runs*: *C-runs xs* ≤ *length xs* − *1*
and *C-asc*: *C-asc a ys* ≤ *length ys*
and *C-desc*: *C-desc a ys* ≤ *length ys*
by (*induct xs* **and** *a ys* **and** *a ys rule*: *C-runs-C-asc-C-desc.induct*) *auto*

lemma
shows *length-runs*: *length* (*runs 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*
by (*induct xs* **and** *a f ys* **and** *a xs ys rule*: *runs-asc-desc.induct*) *auto*

lemma
shows *length-concat-runs* [*simp*]: *length* (*concat* (*runs xs*)) = *length xs*
and *length-concat-asc*: *ascP f* ⟹ *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*
by (*induct xs* **and** *a f ys* **and** *a xs ys rule*: *runs-asc-desc.induct*) *auto*

lemma *log2-mono*:
x > *0* ⟹ *x* ≤ *y* ⟹ *log 2 x* ≤ *log 2 y*
by *auto*

lemma
shows *runs-ne*: *xs* ≠ [] ⟹ *runs xs* ≠ []

and *ascP f* $\Longrightarrow$ *asc a f ys* $\neq$ []
and *desc a xs ys* $\neq$ []
by (*induct xs* **and** *a f ys* **and** *a xs ys rule*: *runs-asc-desc.induct*) *simp-all*

lemma *C-nmsort*:
assumes [*simp*]: *length xs* = *n*
shows *C-nmsort xs* $\leq$ *n* + *n* $*$ $\lceil$*log 2 n*$\rceil$
proof −
have [*simp*]: *xs* = [] $\longleftrightarrow$ *length xs* = *0* **by** *blast*
have *int* (*C-merge-all* (*runs xs*)) $\leq$ *int n* $*$ $\lceil$*log 2* (*length* (*runs xs*))$\rceil$
using *C-merge-all'* [*of runs xs*] **by** *simp*
also have ... $\leq$ *int n* $*$ $\lceil$*log 2 n*$\rceil$
using *length-runs* [*of xs*]
by (*cases n*) (*auto intro*!: *runs-ne mult-mono ceiling-mono log2-mono*)
finally have *int* (*C-merge-all* (*runs xs*)) $\leq$ *int n* $*$ $\lceil$*log 2 n*$\rceil$ **.**
moreover have *C-runs xs* $\leq$ *n* **using** *C-runs* [*of xs*] **by** *auto*
ultimately show *?thesis* **by** (*auto intro*: *add-mono*)
qed

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.