Efficient Mergesort

Christian Sternagel

May 28, 2015

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 ~/src/HOL/Library/Multiset
begin

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

1 Chaining Lists by Predicates

Make sure that some binary predicate \( P \) is satisfied between every two consecutive elements of a list. We call such a list a chain in the following.

inductive
linked :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ bool
for P::'a ⇒ 'a ⇒ bool
where
Nil[iff]: linked P []
| singleton[iff]: linked P [x]
| many: P x y ⇒ linked P (y#ys) ⇒ linked P (x#y#ys)

declare eqTrueI[OF Nil, code] eqTrueI[OF singleton, code]

lemma linked-many-eq[simp, code]:
linked P (x#y#zs) ⇔ P x y ∧ linked P (y#zs)
by (blast intro: linked.many elim: linked.cases)

Take the longest prefix of a list that forms a chain.

fun take-chain :: 'a ⇒ ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a list where
Drop the longest prefix of a list that forms a chain.

fun drop-chain :: 'a ⇒ ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a list
where
drop-chain a P [] = []
drop-chain a P (x#xs) = (if P a x then x # drop-chain x P xs
else x#xs)

lemma take-chain-drop-chain-id[simp]:
take-chain a P xs @ drop-chain a P xs = xs
by (induct xs arbitrary: a) simp-all

lemma linked-take-chain:
linked P (x # take-chain x P xs)
by (induct xs arbitrary: x) simp-all

lemma linked-rev-take-chain-append:
linked P (rev (take-chain x (λx y. P y x) x) @ x#ys)
by (induct xs arbitrary: x ys) simp-all

lemma linked-rev-take-chain:
linked P (rev (take-chain x (λx y. P y x) x) @ [x])
using linked-rev-take-chain-append[of P x [] xs] by simp

lemma linked-append:
linked P (xs@ys) ←→ linked P xs ∧ linked P ys
∧ (if xs ≠ [] ∧ ys ≠ [] then P (last xs) (hd ys) else True)
(is ?lhs = ?rhs)
proof
  assume ?lhs thus ?rhs
  proof (induct xs)
  case (Cons x xs) thus ?case by (cases xs, simp-all) (cases ys, auto)
  qed simp
next
  assume ?rhs thus ?lhs
  proof (induct xs)
  case (Cons x xs) thus ?case by (cases ys, auto) (cases xs, auto)
  qed simp
  qed

lemma length-drop-chain[termination-simp]:
length (drop-chain b P xs) ≤ length xs (is ?P b xs)
proof (induct xs arbitrary: b rule: length-induct)
  fix xs::'a list and b
  assume IH: ∀ ys. length ys < length xs → (∀ x. ?P x ys)
show \( P \ b \ xs \)
proof (cases \( xs \))
  case (Cons \( y \) \( ys \)) with IH[rule-format, of \( ys \) \( y \)] show \( \text{thesis} \) by simp
qed simp

qed

lemma take-chain-map[simp]:
take-chain \((f \ x)\) \( P \) (map \( f \) \( xs \)) = map \( f \) (take-chain \( x \) (\( \lambda x \) \( y \). \( P \) (\( f \) \( x \)) (\( f \) \( y \))) \( xs \))
by (induct \( xs \) arbitrary: \( x \)) simp-all

1.1 Sorted is a Special Case of Linked

lemma (in linorder) linked-le-sorted-conv[simp]:
linked \((op \leq)\) \( xs \) = sorted \( xs \)
proof
  assume sorted \( xs \) thus linked \((op \leq)\) \( xs \)
proof (induct \( xs \) rule: sorted.induct)
  case (Cons \( xs \) \( x \)) thus \( ?\text{case} \) by (cases \( xs \)) simp-all
qed simp
qed (induct \( xs \) rule: linked.induct, simp-all)

abbreviation (in linorder) (input) \( \text{lt} :: (\ 'b \Rightarrow \ 'a) \Rightarrow \ 'b \Rightarrow \ 'b \Rightarrow \ 'b \Rightarrow \ 'b \Rightarrow \ 'b \Rightarrow \ bool \) where
\( \text{lt} \) \( \text{key} \) \( x \) \( y \) \( \equiv \text{key} \ x \) \( < \) \( \text{key} \ y \)

abbreviation (in linorder) (input) \( \text{le} :: (\ 'b \Rightarrow \ 'a) \Rightarrow \ 'b \Rightarrow \ 'b \Rightarrow \ 'b \Rightarrow \ 'b \Rightarrow \ 'b \Rightarrow \ bool \) where
\( \text{le} \) \( \text{key} \) \( x \) \( y \) \( \equiv \text{key} \ x \) \( \leq \) \( \text{key} \ y \)

abbreviation (in linorder) (input) \( \text{gt} :: (\ 'b \Rightarrow \ 'a) \Rightarrow \ 'b \Rightarrow \ 'b \Rightarrow \ 'b \Rightarrow \ 'b \Rightarrow \ 'b \Rightarrow \ bool \) where
\( \text{gt} \) \( \text{key} \) \( x \) \( y \) \( \equiv \text{key} \ x \) \( > \) \( \text{key} \ y \)

abbreviation (in linorder) (input) \( \text{ge} :: (\ 'b \Rightarrow \ 'a) \Rightarrow \ 'b \Rightarrow \ 'b \Rightarrow \ 'b \Rightarrow \ 'b \Rightarrow \ 'b \Rightarrow \ bool \) where
\( \text{ge} \) \( \text{key} \) \( x \) \( y \) \( \equiv \text{key} \ x \) \( \geq \) \( \text{key} \ y \)

lemma (in linorder) sorted-take-chain-le[simp]:
sorted \((\text{key} \ # \ # \text{map} \ \text{key} \ (\text{take-chain} \ x \ (\text{le} \ \text{key} \) \( xs \)))\)
using linked-take-chain[of \( \text{op} \leq \), of \( \text{key} \) \( \text{x} \) \( \text{map} \) \( \text{key} \) \( \text{xs} \)] by simp

lemma (in linorder) sorted-rev-take-chain-gt-append:
assumes \( \text{linked} \ (op <) \) \( \text{(key} \ # \ # \text{map} \ \text{key} \ y) \)
shows \( \text{sorted} \ (\text{map} \ \text{key} \ (\text{rev} \ (\text{take-chain} \ x \ (gt \ \text{key} \) \( xs \)))) \# \text{key} \ # \ # \text{map} \ \text{key} \ y \)
using linked-less-imp-sorted[OF \( \text{linked-rev-take-chain-append} \) \( \text{OF} \) \( \text{assms} \), \( \text{of} \) \( \text{map} \) \( \text{key} \) \( \text{xs} \)]]
by (simp add: rev-map)

lemma multiset-of-take-chain-drop-chain[simp]:
\[
\text{multiset-of } \text{(take-chain } x \ P \ \text{xs)} + \text{multiset-of } \text{(drop-chain } x \ P \ \text{xs)} = \text{multiset-of } \text{xs}
\]
\by (\text{induct } \text{xs arbitrary: } x) (\text{simp-all add: ac-simps})

\[\text{lemma multiset-of-drop-chain-take-chain[simp]:} \]
\[
\text{multiset-of } \text{(drop-chain } x \ P \ \text{xs)} + \text{multiset-of } \text{(take-chain } x \ P \ \text{xs)} = \text{multiset-of } \text{xs}
\]
\by (\text{induct } \text{xs arbitrary: } x) (\text{simp-all add: ac-simps})

2 GHC Version of Mergesort

In the following we show that the mergesort implementation used in GHC (see [link to GHC documentation]) 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 \textit{Multiset}.

\[\text{context linorder} \begin{align*} \quad & \text{begin} \quad \text{begin} \\
\quad & \text{Split a list into chunks of ascending and descending parts, where descending parts are reversed. Hence, the result is a list of sorted lists.} \\
\quad & \text{fun} \quad \text{sequences :: } (\forall b \Rightarrow \forall a) \Rightarrow \forall b \text{ list} \Rightarrow \forall b \text{ list list} \\
\quad & \quad \text{and} \quad \text{asc :: } (\forall b \Rightarrow \forall a) \Rightarrow \forall b \Rightarrow (\forall b \Rightarrow \forall b \text{ list}) \Rightarrow \forall b \text{ list} \Rightarrow \forall b \text{ list list} \\
\quad & \quad \text{and} \quad \text{desc :: } (\forall b \Rightarrow \forall a) \Rightarrow \forall b \Rightarrow \forall b \text{ list} \Rightarrow \forall b \text{ list} \Rightarrow \forall b \text{ list list} \\
\quad & \text{where} \quad \text{sequences key } (a \# b \# x s) = \\
\quad & \quad (\text{if key } a > \text{ key } b \text{ then desc key } b \ [a] \ x s \text{ else asc key } b \ (\text{op } \# \ a) \ x s) \\
\quad & \quad \text{sequences key } x s = [x s] \\
\quad & \quad \text{asc key a f } (b \# b s) = (\text{if } \neg \text{ key } a > \text{ key } b \text{ then asc key b } (f \circ \text{op } \# \ a) \ b s \\
\quad & \quad \text{else } f \ [a] \ # \ \text{sequences key } (b \# b s)) \\
\quad & \quad \text{asc key a f b s = f } [a] \ # \ \text{sequences key } b s \\
\quad & \quad \text{desc key a as } (b \# b s) = (\text{if key } a > \text{ key } b \\
\quad & \quad \text{then desc key b } (a \# a s) \ b s \\
\quad & \quad \text{else } (a \# a s) \ # \ \text{sequences key } (b \# b s)) \\
\quad & \quad \text{desc key a as } b s = (a \# a s) \ # \ \text{sequences key } b s \\
\quad & \text{fun} \quad \text{merge :: } (\forall b \Rightarrow \forall a) \Rightarrow \forall b \text{ list} \Rightarrow \forall b \text{ list} \Rightarrow \forall b \text{ list} \ \text{where} \\
\quad & \quad \text{merge key } (a \# a s) \ (b \# b s) = (\text{if key } a > \text{ key } b \\
\quad & \quad \text{then } b \# \ \text{merge key } (a \# a s) \ b s \\
\quad & \quad \text{else } a \# \ \text{merge key as } (b \# b s)) \\
\quad & \quad \text{merge key } [] \ b s = b s \\
\quad & \quad \text{merge key } a s = [] = a s \\
\quad & \text{fun} \quad \text{merge-pairs :: } (\forall b \Rightarrow \forall a) \Rightarrow \forall b \text{ list list} \Rightarrow \forall b \text{ list list} \ \text{where} \\
\quad & \quad \text{merge-pairs key } (a \# b \# x s) = \text{merge key } a b \# \ \text{merge-pairs key } x s \\
\quad & \quad \text{merge-pairs key } x s = x s \end{align*} \]

4
lemma merge-nil2[simp]: merge key as [] = as by (cases as) simp-all

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

lemma merge-pairs-length[termination-simp]:
  length (merge-pairs key xs) ≤ length xs
by (induct xs rule: merge-pairs.induct) simp-all

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

lemma multiset-of-merge[simp]:
  multiset-of (merge key xs ys) = multiset-of xs + multiset-of ys
by (induct xs ys rule: merge.induct) (simp-all add: ac-simps)

lemma set-merge[simp]:
  set (merge key xs ys) = set xs ∪ set ys
unfolding set-of-multiset-of[symmetric] by simp

lemma multiset-of-concat-merge-pairs[simp]:
  multiset-of (concat (merge-pairs key xs)) = multiset-of (concat xs)
by (induct xs rule: merge-pairs.induct) (auto simp: ac-simps)

lemma set-concat-merge-pairs[simp]:
  set (concat (merge-pairs key xs)) = set (concat xs)
unfolding set-of-multiset-of[symmetric] by simp

lemma multiset-of-merge-all[simp]:
  multiset-of (merge-all key xs) = multiset-of (concat xs)
by (induct xs rule: merge-all.induct) (simp-all add: ac-simps)

lemma set-merge-all[simp]:
  set (merge-all key xs) = set (concat xs)
unfolding set-of-multiset-of[symmetric] by simp

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

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

**Lemma** desc-take-chain-drop-chain-conv[simp]:
desc key a bs x = (rev (take-chain a (gt key) x) $\@ a \# bs)$ $\# \text{sequences key } (\text{drop-chain a } (gt key) x)$
proof (induct x s arbitrary: a bs)
case (Cons x x s) thus $?\text{case}$ by (cases key a $< key x$) simp-all
qed simp

**Lemma** asc-take-chain-drop-chain-conv-append:
assumes $\forall x s y s. f (x s @ y s) = f x s @ y s$
s-shows asc key a (f $\circ$ op $\@ a s) x s$
  = (f as $\@ a \# \text{take-chain a } (le key) x s) \# \text{sequences key } (\text{drop-chain a } (le key) x s)$
using assms
proof (induct x s arbitrary: as a)
case (Cons x x s)
  show $?\text{case}$
    proof (cases le key a x)
    case False with Cons show $?\text{thesis}$ by auto
    next
    case True with Cons (2) $[a] x s$ and Cons(2)
    show $?\text{thesis}$ by (simp add: o-def)
  qed
qed simp

**Lemma** asc-take-chain-drop-chain-conv[simp]:
asc key b (op $\# a) x s$
  = (a $\# b \# \text{take-chain b } (le key) x s) \# \text{sequences key } (\text{drop-chain b } (le key) x s)$
proof
let $?f = op \# a$
have $\forall x s y s. (op \# a) (x s @ y s) = (op \# a) x s @ y s$ by simp
from asc-take-chain-drop-chain-conv-append[of $?f b [] x s$, OF this]
show $?\text{thesis}$ by (simp add: o-def)
qed

**Lemma** sequences-induct[case-names Nil singleton many]:
assumes $\forall key. P \text{ key }[]$ and $\forall key x. P \text{ key }[x]$
and $\forall key a b x s. (le key a b \implies P \text{ key } (\text{drop-chain b } (le key) x s))$
$\implies (\neg \text{ le key a b } \implies P \text{ key } (\text{drop-chain b } (gt key) x s))$
$\implies P \text{ key } (a \# b \# x s)$
shows $P\ key\ xs$
using assms by (induction-schema) (pat-completeness, lexicographic-order)

lemma sorted-sequences:
$\forall x \in \text{set} (\text{sequences key xs}). \text{sorted (map key x)}$
proof (induct key xs rule: sequences-induct)
case (many key a b xs)
thus ?case using sorted-rev-take-chain-gt-append[of key b [a] xs]
  by (cases le key a b) auto
qed simp-all

lemma multiset-of-sequences[simp]:
multiset-of (concat (sequences key xs)) = multiset-of xs
by (induct key xs rule: sequences-induct) (simp-all add: ac-simps)

lemma filter-by-key-drop-chain-gt[simp]:
assumes key b $\leq$ key a
shows $[y\leftarrow\text{drop-chain b (gt key)}\ xs.\ key a = key y] = [y\leftarrow xs.\ key a = key y]$
using assms by (induct xs arbitrary: b) auto

lemma filter-by-key-take-chain-gt[simp]:
assumes key b $\leq$ key a
shows $[y\leftarrow\text{take-chain b (gt key)}\ xs.\ key a = key y] = []$
using assms by (induct xs arbitrary: b) auto

lemma filter-take-chain-drop-chain[simp]:
filter $P$ (take-chain x Q xs) @ filter $P$ (drop-chain x Q xs) = filter $P$ xs
by (simp add: filter-append[symmetric])

lemma filter-by-key-rev-take-chain-gt-conv[simp]:
$[y\leftarrow\text{rev (take-chain b (gt key)) xs.\ key x = key y}] = [y\leftarrow\text{take-chain b (gt key)}\ xs.\ key x = key y]$
by (induct xs arbitrary: b) auto

lemma filter-by-key-sequences[simp]:
$[y\leftarrow\text{concat (sequences key xs).\ key x = key y}]$
$= \text{is}\ ?P$
by (induct key xs rule: sequences-induct) auto

lemma merge-simp[simp]:
assumes sorted (map key xs)
shows merge key xs (y#ys)
  $= \text{takeWhile (ge key y) xs @ y} \neq \text{merge key (dropWhile (ge key y) xs) ys}$
using assms by (induct xs arbitrary: y ys) (auto simp: sorted-Cons)

lemma sorted-map-dropWhile[simp]:
assumes sorted (map key xs)
shows sorted (map key (dropWhile (ge key y) xs))
using sorted-dropWhile[OF assms] by (simp add: dropWhile-map o-def)
lemma sorted-merge-induct[consumes 1, case-names Nil IH]:
assumes sorted (map key xs)
and \( \forall x s. P x s [] \)
and \( \forall x s y s. \) sorted (map key xs) \( \implies \) \( P (\text{dropWhile} (\ge key y) xs) ys \)
shows \( P x s y s \)
using assms(2−) assms(1)
by (induction-schema) (case-tac ys, simp-all, lexicographic-order)

lemma filter-by-key-dropWhile[simp]:
assumes sorted (map key xs)
shows \( [y ← \text{dropWhile} (λ. \text{key} z ≤ \text{key} y) \text{xs}. \text{key} z = \text{key} y] = [] \)
using assms
proof (induct xs rule: rev-induct)
case Nil thus ?case by simp
next
case (snoc x xs)
hence IH: \( [y ← \text{dropWhile} ?P \text{xs}. \text{key} z = \text{key} y] = [] \)
by (auto simp: sorted-append)
show ?case
proof (cases \( \forall z ∈ \text{set} \text{xs}. ?P z \))
8
next
case False
then obtain a where a: a ∈ set xs → ?P a by auto
show ?thesis
unfolding dropWhile-append1[of a xs ?P, OF a]
using snoc and False by (auto simp: IH sorted-append)
qed

lemma filter-by-key-takeWhile[simp]:
assumes sorted (map key xs)
shows \( [y ← \text{takeWhile} (λ. \text{key} x ≤ \text{key} z) \text{xs}. \text{key} z = \text{key} y] = [y ← \text{xs}. \text{key} z = \text{key} y] = - \)
using assms
proof (induct xs rule: rev-induct)
case Nil thus ?case by simp
next
case (snoc x xs)
hence IH: \( [y ← \text{takeWhile} ?P \text{xs}. \text{key} z = \text{key} y] = [y ← \text{xs}. \text{key} z = \text{key} y] \)
by (auto simp: sorted-append)
show ?case
proof (cases \( \forall z ∈ \text{set} \text{xs}. ?P z \))

8
case True
  show ?thesis
  using takeWhile-append2[of xs ?P [x]] and True by simp
next
  case False
  then obtain a where a : a ∈ set xs ¬ ?P a by auto
  show ?thesis
  unfolding takeWhile-append1[of a xs ?P, OF a]
  using snoc and False by (auto simp: IH sorted-append)
qed
qed

lemma filter-takeWhile-dropWhile-id[simp]:
  filter P (takeWhile Q xs) @ filter P (dropWhile Q xs) = filter P xs
by (simp add: filter-append[symmetric])

lemma filter-by-key-merge-is-append[simp]:
  assumes sorted (map key xs)
  shows [y← merge key xs ys. key x = key y]
  = [y← xs. key x = key y] @ [y← ys. key x = key y]
  using assms by (induct xs ys rule: sorted-merge-induct) auto

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

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

lemma sort-key-merge-all-sequences:
  sort-key key = merge-all key o sequences key
  by (intro ext properties-for-sort-key)
     (simp-all add: sorted-merge-all[OF sorted-sequences])

Replace existing code equations for sort-key by merge-all key o sequences key.

declare sort-key-merge-all-sequences[code]

end
References