Efficient Mergesort

Christian Sternagel
August 28, 2014

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 $P$ ($y$#zs) ⇒ linked $P$ ($x$#y#zs)

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
take-chain a P [] = []
| take-chain a P (x#xs) = (if P a x then x # take-chain x P xs else [])

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 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 (x#ys) ⇒ linked P (rev (take-chain x (λx y. P y x) xs) @ 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) xs) @ [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 (\(\text{Cons} \ y \ ys\)) with IH [rule-format, of \(ys \ y\)]
show \(?\text{thesis}\) by simp
qed simp

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

1.1 Sorted is a Special Case of Linked

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

abbreviation \(\text{lt} :: \text{input} \ \text{lt} : (\text{key} x \ y \equiv \text{key} x < \text{key} y)\)
abbreviation \(\text{le} :: \text{input} \ \text{le} : (\text{key} x \ y \equiv \text{key} x \leq \text{key} y)\)
abbreviation \(\text{gt} :: \text{input} \ \text{gt} : (\text{key} x \ y \equiv \text{key} x > \text{key} y)\)
abbreviation \(\text{ge} :: \text{input} \ \text{ge} : (\text{key} x \ y \equiv \text{key} x \geq \text{key} y)\)

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

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

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

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

## 2 GHC Version of Mergesort

In the following we show that the mergesort implementation used in GHC (see [http://haskell.org/ghc/docs/7.0-latest/html/libraries/base-4.3.1.0/src/Data-List.html#sort](http://haskell.org/ghc/docs/7.0-latest/html/libraries/base-4.3.1.0/src/Data-List.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 **Multiset**.

**Context** **linorder**

**Begin**

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

**Fun** **sequences** :: (′b ⇒ ′a) ⇒ ′b list ⇒ ′b list list

**And** **asc** :: (′b ⇒ ′a) ⇒ ′b ⇒ (′b list ⇒ ′b list) ⇒ ′b list ⇒ ′b list list

**And** **desc** :: (′b ⇒ ′a) ⇒ ′b ⇒ ′b list ⇒ ′b list ⇒ ′b list list

**Where**

\[
\text{sequences key}\ (a\#b\#xs) = \\
| \text{if key a > key b then desc key b [a] xs else asc key b (op \# a) xs} \\
| \text{sequences key xs} = [xs] \\
| \text{asc key a f (b#bs)} = (\text{if} \neg\ \text{key a > key b}) \text{then asc key b (f \circ op \# a) bs} \text{else f [a] \# sequences key (b#bs)} \\
| \text{desc key a as (b#bs)} = (\text{if key a > key b}) \text{then desc key b (a#as) bs} \text{else (a#as) \# sequences key (b#bs)} \\
| \text{desc key a as bs} = (a#as) \# \text{sequences key bs}
\]

**Fun** **merge** :: (′b ⇒ ′a) ⇒ ′b list ⇒ ′b list ⇒ ′b list

**Where**

\[
\text{merge key\ (a#as) (b#bs) = (if key a > key b}} \text{then b \# merge key (a#as) bs} \\
| \text{else a \# merge key as (b#bs)} \\
| \text{merge key [] bs = bs} \\
| \text{merge key as [] = as}
\]

**Fun** **merge-pairs** :: (′b ⇒ ′a) ⇒ ′b list list ⇒ ′b list list

**Where**

\[
\text{merge-pairs key (a#b#xs) = merge key a b \# merge-pairs key xs} \\
| \text{merge-pairs key xs = xs}
\]
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 :: ('b ⇒ 'a) ⇒ '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 ∀x∈set xs. sorted (map key x)
  shows sorted (map key (merge-all key xs))
  using assms by (induct xs rule: merge-all.induct) simp-all

lemma desc-take-chain-drop-chain-conv[simp]:
  desc key a bs xs
  = (rev (take-chain a (gt key) xs) @ a # bs) # sequences key (drop-chain a (gt key) xs)
proof (induct xs arbitrary: a bs)
  case (Cons x xs) thus ?case by (cases key a < key x) simp-all
qed simp

lemma asc-take-chain-drop-chain-conv-append:
  assumes ⋀xs ys. f (xs@ys) = f xs @ ys
  shows asc key a (f ◦ op @ as) xs
  = (f as @ a # take-chain a (le key) xs) # sequences key (drop-chain a (le key) xs)
using assms
proof (induct xs arbitrary: as a)
  case (Cons x xs)
  show ?case
    proof (cases le key a x)
    case False with Cons show ?thesis by auto
    next
    case True
    with Cons(1)[of x as@[]] and Cons(2)
    show ?thesis by (simp add: o-def)
    qed
  qed simp

lemma asc-take-chain-drop-chain-conv[simp]:
  asc key b (op # a) xs
  = (a # b # take-chain b (le key) xs) # sequences key (drop-chain b (le key) xs)
proof
  let ?f = op # a
  have ⋀xs ys. (op # a) (xs@ys) = (op # a) xs @ ys by simp
  from asc-take-chain-drop-chain-conv-append[of ?f b [] xs, OF this]
  show ?thesis by (simp add: o-def)
qed

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

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

lemma merge-simp[simp]:
assumes sorted (map key xs)
shows merge key xs (y#ys)
  = takeWhile (ge key y) xs @ y # 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)

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

lemma filter-by-key-dropWhile[simp]:
  assumes sorted (map \( \text{key} \) \( \text{xs} \))
  shows \( \text{y←dropWhile (λx. \text{key} x ≤ \text{key} z) \text{xs} z = \text{key} y} = \text{[]} \)
  (is \( \text{y←dropWhile ?P \text{xs} z = \text{key} y} = \text{[]} \))
  using assms
  proof (induct \text{xs} rule: rev-induct)
    case Nil thus ?case by simp
  next
    case (snoc \text{x} \text{xs})
    hence IH: \( \text{y←dropWhile ?P \text{xs} z = \text{key} y} = \text{[]} \)
    by (auto simp: sorted-append)
    show ?case
    proof (cases \( \forall \text{z} ∈ \text{set} \text{xs}. ?P \text{z} \))
      \( \text{True} \)
      show ?thesis
      using dropWhile-append2[of \text{xs} ?P \text{x}] and \( \text{True} \) by simp
    next
      \( \text{False} \)
      then obtain \( \text{a} \) where \( \text{a} ∈ \text{set} \text{xs} \) \( \neg ?P \text{a} \) by auto
      show ?thesis
      unfolding dropWhile-append1[of \text{a} \text{xs} ?P, OF \text{a}]
      using snoc and \( \text{False} \) by (auto simp: IH sorted-append)
    qed
  qed

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

done
end

References