Efficient Mergesort

Christian Sternagel

September 19, 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 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 zs 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 \(IH[\text{rule-format},\ \text{of}\ ys\ y]\)
  show \(?\text{thesis}\) by simp
qed simp

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

lemma (in linorder) linked-less-imp-sorted:
  linked \((op\ <)\) \(xs\) = \(\Rightarrow\) sorted \(xs\)
by (induct \(xs\) rule: linked.induct) simp-all

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

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

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

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

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

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

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

**Lemma** \( \text{mset-dro...n-\text{take-chain}[\text{simp}]:} \)
mset \( \text{drop-chain} \ x \ P \ xs \) + mset \( \text{take-chain} \ x \ P \ xs \) = mset \( xs \)
by \( \text{induct} \ xs \) 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 [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

\[
\text{sequences key} \ a \# b \# xs = \\
\begin{cases} 
\text{if key a > key b then desc key b [a] xs else asc key b (op \# a) xs} & \\
\text{sequences key xs = [xs]} & \\
\text{asc key a f (b#bs) = (if \neg \text{key a > key b}} & \\
\text{then asc key b (f ∘ op \# a) bs} & \\
\text{else f [a] # sequences key (b#bs))} & \\
\text{desc key a as (b#bs) = (if key a > key b} & \\
\text{then desc key b (a#as) bs} & \\
\text{else (a#as) # sequences key (b#bs))} & \\
\text{desc key a as bs = (a#as) # sequences key bs} & \\
\end{cases}
\]

**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]:` \text{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 mset-merge[simp]:
mset (merge key xs ys) = mset xs + mset 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-mset-mset[symmetric] by simp

lemma mset-concat-merge-pairs[simp]:
mset (concat (merge-pairs key xs)) = mset (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-mset-mset[symmetric] by simp

lemma mset-merge-all[simp]:
mset (merge-all key xs) = mset (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-mset-mset[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:
\[ \forall x \in \text{set } xs . \text{sorted } (\text{map key } x) \]
\[ \text{shows } \text{sorted } (\text{map key } (\text{merge-all key } xs)) \]
\[ \text{using } \text{assms by (induct } xs \text{ rule: merge-all.induct) simp-all} \]

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

**Lemma**: asc-take-chain-drop-chain-conv-append:
\[ \text{assumes } \forall xs ys . f (xs@ys) = f xs @ ys \]
\[ \text{shows } \text{asc key a } (f \circ op \oplus a) xs \]
\[ = (f as @ a \# \text{take-chain a } (\text{le key }) xs) \# \text{sequences key } (\text{drop-chain a } (\text{le key }) xs) \]
\[ \text{using } \text{assms} \]
\[ \text{proof (induct } xs \text{ arbitrary: as a)} \]
\[ \text{case } (\text{Cons } x \text{ xs)} \]
\[ \text{show ?case by auto} \]
\[ \text{next } \]
\[ \text{case True } \]
\[ \text{with } \text{Cons}(1)[\text{of } x @ [a]] \text{ and } \text{Cons}(2) \]
\[ \text{show ?thesis by (simp add: o-def)} \]
\[ \text{qed simp} \]

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

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

6
lemma sorted-sequences:
\( \forall x \in \text{set} (\text{sequences key xs}). \text{sorted} (\text{map} \ x) \)
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 mset-sequences[simp]:
mset (concat (sequences key xs)) = mset 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 \([y\leftarrow\text{drop-chain b (gt key)} \text{xs}. \text{key a} = \text{key y}] = [y\leftarrow\text{xs}. \text{key a} = \text{key y}]\)
using assms by (induct xs arbitrary: b) auto

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

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

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

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

lemma merge-simp[simp]:
assumes sorted (map key xs)
shows merge key xs (y\#ys)
  = takeWhile (ge key y) \text{xs} @ y \# merge key (dropWhile (ge key y) \text{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) \text{xs}))
using sorted-dropWhile[OF assms] by (simp add: dropWhile-map o-def)

lemma sorted-merge-induct[consumes 1, case-names Nil IH]:
assumes \( \text{sorted} (\text{map key } xs) \)
and \( \bigwedge x . \ P \ x s \ [\] \)
and \( \bigwedge x s y s . \ \text{sorted} (\text{map key } xs) \implies P (\text{dropWhile} \ (g e \ \text{key} \ y) \ x s) \ y s \)
\implies P x s (y \# y s)
shows \( P x s y s \)
using \( \text{assms}(2-) \) \( \text{assms}(1) \)
by (induction-schema) (case-tac \( y s \), simp-all, lexicographic-order)

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

lemma \( \text{filter-by-key-takeWhile}[^{\text{simp}}] \):
assumes \( \text{sorted} (\text{map key } xs) \)
shows \( \lbrack y \leftarrow \text{takeWhile} \ (\lambda x . \ \text{key} \ x \leq \text{key} \ z) \ x s . \ \text{key} \ z = \text{key} \ y \rbrack = \lbrack y \leftarrow x s . \ \text{key} \ z = \text{key} \ y \rbrack \)
(is \( \lbrack y \leftarrow \text{takeWhile} \ ?P \ x s . \ \text{key} \ z = \text{key} \ y \rbrack = - \) 
using \( \text{assms} \)
proof (induct \( x s \) rule: rev-induct)
case \( \text{Nil} \) thus \( \text{?case} \) by simp
next
case \( \text{snoc} \ x x s \)
hence \( \text{IH} \): \( \lbrack y \leftarrow \text{takeWhile} \ ?P \ x s . \ \text{key} \ z = \text{key} \ y \rbrack = \lbrack y \leftarrow x s . \ \text{key} \ z = \text{key} \ y \rbrack \)
by (auto simp: \( \text{sorted-append} \))
show \( \text{?case} \)
proof (cases \( \forall z \in \text{set} \ x s . \ ?P \ z \))
case \( \text{True} \)
show \( \text{?thesis} \)
qed
using (takeWhile-append2[of xs ?P [x]]) and True by simp
next
  case False
  then obtain a where a: a \in set xs \sim ?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 \forall xs\in 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 \forall xs\in 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 \circ 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 \circ sequences key.
declare sort-key-merge-all-sequences[code]

end
end
References

s10817-012-9260-7.