Verification of Functional Binomial Queues

René Neumann
Technische Universität München, Institut für Informatik
http://www.in.tum.de/neumannr/

Abstract. Priority queues are an important data structure and efficient implementations of them are crucial. We implement a functional variant of binomial queues in Isabelle/HOL and show its functional correctness. A verification against an abstract reference specification of priority queues has also been attempted, but could not be achieved to the full extent.

1 Abstract priority queues

1.1 Generic Lemmas

lemma tl-set:
\[ \text{distinct } q \implies \text{set} (\text{tl } q) = \text{set } q - \{\text{hd } q\} \]
by (cases q) simp-all

1.2 Type of abstract priority queues

typedef ('a, 'b::linorder) pq = 
\{xs :: ('a × 'b) list. \text{distinct} (\text{map} \text{fst} \text{xs}) \land \text{sorted} (\text{map} \text{snd} \text{xs})\}

morphisms alist-of Abs-pq
proof –
  have [] \in \?pq by simp
  then show \?thesis by blast
qed

lemma alist-of-Abs-pq:
  assumes \text{distinct} (\text{map} \text{fst} \text{xs})
  and \text{sorted} (\text{map} \text{snd} \text{xs})
  shows \text{alist-of} (Abs-pq \text{xs}) = \text{xs}
  by (rule Abs-pq-inverse) (simp add: assms)

lemma [code abstype]:
  Abs-pq (alist-of \text{q}) = \text{q}
  by (fact alist-of-inverse)
lemma distinct-fst-alist-of [simp]:
distinct (map fst (alist-of q))
  using alist-of [of q] by simp

lemma distinct-alist-of [simp]:
distinct (alist-of q)
  using distinct-fst-alist-of [of q] by (simp add: distinct-map)

lemma sorted-snd-alist-of [simp]:
sorted (map snd (alist-of q))
  using alist-of [of q] by simp

lemma alist-of-eqI:
alist-of p = alist-of q =⇒ p = q
proof –
  assume alist-of p = alist-of q
  then have Abs-pq (alist-of p) = Abs-pq (alist-of q) by simp
  thus p = q by (simp add: alist-of-inverse)
qed

definition values :: ('a, 'b::linorder) pq ⇒ 'a list (|{-} |)
  where
values q = map fst (alist-of q)

definition priorities :: ('a, 'b::linorder) pq ⇒ 'b list (\|{-} \|)
  where
priorities q = map snd (alist-of q)

lemma values-set:
  set |q| = fst ' set (alist-of q)
  by (simp add: values-def)

lemma priorities-set:
  set \|q\| = snd ' set (alist-of q)
  by (simp add: priorities-def)

definition is-empty :: ('a, 'b::linorder) pq ⇒ bool where
  is-empty q =⇒ alist-of q = []

definition priority :: ('a, 'b::linorder) pq ⇒ 'a ⇒ 'b option where
  priority q = map-of (alist-of q)

definition min :: ('a, 'b::linorder) pq ⇒ 'a where
  min q = fst (hd (alist-of q))

definition empty :: ('a, 'b::linorder) pq where
  empty = Abs-pq []
lemma is-empty-alist-of [dest]:
  is-empty q \implies\ \text{alist-of q} = []
  by (simp add: is-empty-def)

lemma not-is-empty-alist-of [dest]:
  \neg is-empty q \implies\ \text{alist-of q} \neq []
  by (simp add: is-empty-def)

lemma alist-of-empty [simp, code abstract]:
  \text{alist-of empty} = []
  by (simp add: empty-def Abs-pq-inverse)

lemma values-empty [simp]:
  \text{|empty|} = []
  by (simp add: values-def)

lemma priorities-empty [simp]:
  \|empty\| = []
  by (simp add: priorities-def)

lemma values-empty-nothing [simp]:
  \forall k. k \not\in \text{set |empty|}
  by (simp add: values-def)

lemma is-empty-empty:
  is-empty q \iff q = empty
proof (rule iffI)
  assume is-empty q
  then have \text{alist-of q} = [] by (simp add: is-empty-alist-of)
  then have Abs-pq (alist-of q) = Abs-pq [] by simp
  then show q = empty by (simp add: empty-def alist-of-inverse)
qed (simp add: is-empty-def)

lemma is-empty-empty-simp [simp]:
  \text{is-empty empty}
by (simp add: is-empty-empty)

lemma map-snd-alist-of:
  map (\circ \text{priority q}) (\text{values q}) = map \text{snd} (\text{alist-of q})
by (auto simp add: values-def priority-def)

lemma image-snd-alist-of:
  \text{the '} priority q ' set (\text{values q}) = \text{snd'} set (\text{alist-of q})
proof =
from map-snd-alist-of [of q]
have set (map (the ◦ priority q) (values q)) = set (map snd (alist-of q))
  by (simp only:)
then show ?thesis by (simp add: image-comp)
qed

lemma Min-snd-alist-of:
assumes ¬ is-empty q
shows Min (snd ' set (alist-of q)) = snd (hd (alist-of q))
proof -
from assms obtain ps p where q: map snd (alist-of q) = p # ps
  by (cases map snd (alist-of q)) auto
then have hd (map snd (alist-of q)) = p by simp
with assms have p: snd (hd (alist-of q)) = p by (auto simp add: hd-map)
have sorted (map snd (alist-of q)) by simp
with q have sorted (p # ps) by simp
then have ∀ p′ ∈ set ps. p′ ≥ p by (simp add: sorted-Cons)
then have Min (set (p # ps)) = p by (auto intro: Min-eqI)
with p q have Min (set (map snd (alist-of q))) = snd (hd (alist-of q))
  by simp
then show ?thesis by simp
qed

lemma priority-fst:
assumes xp ∈ set (alist-of q)
shows priority q (fst xp) = Some (snd xp)
using assms by (simp add: priority-def)

lemma priority-Min:
assumes ¬ is-empty q
shows priority q (min q) = Some (Min (the ' priority q ' set (values q)))
using assms
  by (auto simp add: min-def image-snd-alist-of Min-snd-alist-of priority-fst)

lemma priority-Min-priorities:
assumes ¬ is-empty q
shows priority q (min q) = Some (Min (set ||q||))
using assms
  by (simp add: priority-Min image-snd-alist-of priorities-def)

definition push :: 'a ⇒ 'b::linorder ⇒ ('a, 'b) pq ⇒ ('a, 'b) pq where
push k p q = Abs-pq (if k ∉ set (values q) then insort-key snd {k, p} (alist-of q)
else alist-of q)
lemma Min-snd-hd:
\[ q \neq [] \implies \text{sorted } (\text{map} \ \text{snd} \ q) \implies \text{Min} (\text{snd} \ \text{set} \ q) = \text{snd} \ (\text{hd} \ q) \]

proof (induct q)
case (Cons x xs) then show ?case by (cases xs) (auto simp add: ord-class.min-def)
qed simp

lemma hd-construct:
assumes \( \neg \text{is-empty } q \)
shows \( \text{hd} \ (\text{alist-of } q) = \text{(min } q, \text{the } (\text{priority } q \ (\text{min } q))) \)

proof
from assms have \( \text{the } (\text{priority } q \ (\text{min } q)) = \text{snd} \ (\text{hd} \ (\text{alist-of } q)) \)
using Min-snd-hd [of \text{alist-of } q]
by (auto simp add: priority-Min-priorities priorities-def)
then show ?thesis by (simp add: min-def)
qed

lemma not-in-first-image:
\( x \notin \text{fst } s \implies (x, p) \notin s \)
by (auto simp add: image-def)

lemma alist-of-push [simp, code abstract]:
alist-of \( (\text{push } k \ p \ q) = \)
(if \( k \notin \text{set } (\text{values } q) \) then \( \text{insort-key } \text{snd} \ (k, p) \ (\text{alist-of } q) \) else \( \text{alist-of } q \))
using distinct-fst-alist-of [of \( q \)]
by (auto simp add: distinct-map insert-map insert-key intro: alist-of-Abs-pq)

lemma push-values [simp]:
\( \text{set } (\text{push } k \ p \ q) = \text{set } q \cup \{k\} \)
by (auto simp add: values-def set-insort)

lemma push-priorities [simp]:
\( k \notin \text{set } q \implies \text{set } (\text{push } k \ p \ q) = \text{set } q \cup \{p\} \)
\( k \in \text{set } q \implies \text{set } (\text{push } k \ p \ q) = \text{set } q \)
by (auto simp add: priorities-def set-insort)

lemma not-is-empty-push [simp]:
\( \neg \text{is-empty } (\text{push } k \ p \ q) \)
by (auto simp add: values-def is-empty-def)

lemma push-commute:
assumes \( a \neq b \) and \( v \neq w \)
shows \( \text{push } w \ b \ (\text{push } v \ a \ q) = \text{push } v \ a \ (\text{push } w \ b \ q) \)
using assms by (auto intro: alist-of-eqI insert-key-left-comm)
definition remove-min :: ('a::linorder) pq ⇒ ('a::linorder) pq where
remove-min q = (if is-empty q then empty else Abs-pq (tl (alist-of q)))

lemma alift-of-remove-min-if [code abstract]:
alist-of (remove-min q) = (if is-empty q then [] else tl (alist-of q))
by (auto simp add: remove-min-def map-tl sorted-tl distinct-tl alist-of-Abs-pq)

lemma remove-min-empty [simp]:
is-empty q ⇒ remove-min q = empty
by (simp add: remove-min-def)

lemma alist-of-remove-min [simp]:
¬ is-empty q ⇒ alist-of (remove-min q) = tl (alist-of q)
by (simp add: alift-of-remove-min-if)

lemma values-remove-min [simp]:
¬ is-empty q ⇒ values (remove-min q) = tl (values q)
by (simp add: values-def map-tl)

lemma set-alist-of-remove-min:
¬ is-empty q ⇒ set (alist-of (remove-min q)) =
set (alist-of q) − { (min q, the (priority q (min q))) }
by (simp add: tl-set hd-construct)

definition pop :: ('a::linorder) pq ⇒ ('a×('a::linorder)) pq option where
pop q = (if is-empty q then None else Some (min q, remove-min q))

lemma pop-simps [simp]:
is-empty q ⇒ pop q = None
¬ is-empty q ⇒ pop q = Some (min q, remove-min q)
by (simp-all add: pop-def)

hide-const (open) Abs-pq alist-of values priority empty is-empty push min pop

no-notation
PQ.values ([|(-)|])
and PQ.priorities ([||(-)||])

2 Functional Binomial Queues

2.1 Type definition and projections
datatype ('a,'b::linorder) bintree = Node 'a 'b ('a,'b) bintree list
primrec priority :: ('a,'b) bintree ⇒ 'a where
priority \( (\text{Node } a - -) = a \)

primrec \( \text{val} :: (\acute{a}, \acute{b}) \text{ bintree} \Rightarrow \acute{b} \) where 
\[ \text{val} (\text{Node} - v -) = v \]

primrec \( \text{children} :: (\acute{a}, \acute{b}) \text{ bintree} \Rightarrow (\acute{a}, \acute{b}) \text{ bintree list} \) where 
\[ \text{children} (\text{Node} - - ts) = ts \]

type-synonym \( (\acute{a}, \acute{b}) \text{ binqueue} = (\acute{a}, \acute{b}) \text{ bintree option list} \)

lemma \( \text{binqueue-induct} \) [case-names Empty None Some, induct type: binqueue]: 
assumes \( P [] \) and \( \bigwedge xs. P xs \implies P (\text{None} \# xs) \) and \( \bigwedge x xs. P xs \implies P (\text{Some } x \# xs) \) shows \( P xs \)
using \( \text{assms} \) proof (induct \( xs \)) 
case (\text{Cons } x xs) thus \?case by (cases \( x \)) simp-all
qed simp

Terminology:
- values \( v, w \) or \( v_1, v_2 \)
- priorities \( a, b \) or \( a_1, a_2 \)
- bintrees \( t, r \) or \( t_1, t_2 \)
- bintree lists \( ts, rs \) or \( ts_1, ts_2 \)
- bintree element \( x, y \) or \( x_1, x_2 \)
- binquenes = bintree element lists \( xs, ys \) or \( xs_1, xs_2 \)
- abstract priority queues \( q, p \) or \( q_1, q_2 \)

2.2 Binomial queue properties

Binomial tree property

inductive \( \text{is-bintree-list} :: \text{nat} \Rightarrow (\acute{a}, \acute{b}) \text{ bintree list} \Rightarrow \text{bool} \) where 
\( \text{is-bintree-list-Nil} \) [simp]: \( \text{is-bintree-list} 0 [] \)
| \( \text{is-bintree-list-Cons} \) : \( \text{is-bintree-list} l ts \implies \text{is-bintree-list} l (\text{children } t) \) 
\implies \( \text{is-bintree-list} (\text{Suc } l) (t \# ts) \)

abbreviation (input) \( \text{is-bintree } k t \equiv \text{is-bintree-list } k \) (\text{children } t)

lemma \( \text{is-bintree-list-triv} \) [simp]:
\( \text{is-bintree-list} 0 ts \iff ts = [] \)
\( \text{is-bintree-list} l [] \iff l = 0 \)
by (auto intro: \( \text{is-bintree-list.intros} \) elim: \( \text{is-bintree-list.cases} \))
lemma \textit{is-bintree-list-simp} [simp]:
\begin{align*}
\text{is-bintree-list } (\text{Suc } l) (t \neq ts) & \iff \\
\text{is-bintree-list } l \{ \text{children } t \} \land \text{is-bintree-list } l \ ts & \\
\end{align*}
by (auto intro: \textit{is-bintree-list.intros elim: is-bintree-list.cases})

lemma \textit{is-bintree-list-length} [simp]:
\begin{align*}
\text{is-bintree-list } l \ ts & \implies \text{length } ts = l \\
\end{align*}
by (erule \textit{is-bintree-list.induct}) simp-all

lemma \textit{is-bintree-list-children-last}:
\begin{align*}
\text{assumes } \text{is-bintree-list } l \ ts & \land ts \neq [] \\
\text{shows } \text{children } \{ \text{last } ts \} & = [] \\
\text{using } \text{assms } & \text{by induct auto} \\
\end{align*}

lemma \textit{is-bintree-children-length-desc}:
\begin{align*}
\text{assumes } \text{is-bintree-list } l \ ts & \\
\text{shows } \text{map } (\text{length } \circ \text{children}) \ ts & = \text{rev } [0..<l] \\
\text{using } \text{assms } & \text{by (induct } ts \text{) simp-all} \\
\end{align*}

Heap property

inductive \textit{is-heap-list} :: \(\forall a::\text{lindorder } \Rightarrow \{a, \ 'b\} \ \text{bintree list } \Rightarrow \text{bool}\) where
\begin{align*}
\text{is-heap-list-Nil: } & \text{is-heap-list } h \ [] \\
\text{is-heap-list-Cons: } & \text{is-heap-list } h \ ts \implies \text{is-heap-list } (\text{priority } t) \ (\text{children } t) \\
& \implies (\text{priority } t) \geq h \implies \text{is-heap-list } h \ (t \neq ts) \\
\end{align*}

abbreviation (input) \textit{is-heap } t \equiv \text{is-heap-list } (\text{priority } t) \ (\text{children } t)

lemma \textit{is-heap-list-simps} [simp]:
\begin{align*}
\text{is-heap-list } h \ [] & \iff \text{True} \\
\text{is-heap-list } h \ (t \neq ts) & \iff \text{is-heap-list } h \ ts \land \text{is-heap-list } (\text{priority } t) \ (\text{children } t) \land \text{priority } t \geq h \\
\end{align*}
by (auto intro: \textit{is-heap-list.intros elim: is-heap-list.cases})

lemma \textit{is-heap-list-append-dest} [dest]:
\begin{align*}
\text{is-heap-list } l \ (ts\circ rs) & \implies \text{is-heap-list } l \ ts \\
\text{is-heap-list } l \ (ts\circ rs) & \implies \text{is-heap-list } l \ rs \\
\end{align*}
by (induct \(ts\)) (auto intro: \textit{is-heap-list.intros elim: is-heap-list.cases})

lemma \textit{is-heap-list-rev}:
\begin{align*}
\text{is-heap-list } l \ ts & \implies \text{is-heap-list } l \ (\text{rev } ts) \\
\end{align*}
by (induct \(ts\) rule: \textit{rev-induct}) auto

lemma \textit{is-heap-children-larger}:
\begin{align*}
\text{is-heap } t & \implies \forall x \in \text{set } (\text{children } t). \ \text{priority } x \geq \text{priority } t \\
\end{align*}
by (erule is-heap-list.induct) simp-all

**lemma** is-heap-Min-children-larger:

\[ \text{is-heap } t \implies \text{children } t \neq [] \implies \text{priority } t \leq \text{Min} (\text{priority ' set} \text{(children } t)) \]

by (simp add: is-heap-children-larger)

Combination of both: binqueue property

**inductive** is-binqueue :: nat ⇒ ('a::linorder, 'b) binqueue ⇒ bool where

*Empty*:

\[ \text{is-binqueue } l [] \]

| *None*:

\[ \text{is-binqueue } (\text{Suc } l) \text{ } xs \implies \text{is-binqueue } l \text{ (None # xs)} \]

| *Some*:

\[ \text{is-binqueue } (\text{Suc } l) \text{ } xs \implies \text{is-bintree } l \text{ } t \]

\[ \implies \text{is-heap } t \implies \text{is-binqueue } l \text{ (Some } t \neq xs) \]

**lemma** is-binqueue-simp [simp]:

\[ \text{is-binqueue } l [] \iff \text{True} \]

\[ \text{is-binqueue } l \text{ (Some } t \neq xs) \iff \]

\[ \text{is-bintree } l \text{ } t \land \text{is-heap } t \land \text{is-binqueue } (\text{Suc } l) \text{ } xs \]

\[ \text{is-binqueue } l \text{ (None # xs)} \iff \text{is-binqueue } \text{ (Suc } l) \text{ } xs \]

by (auto intro: is-binqueue.intros elim: is-binqueue.cases)

**lemma** is-binqueue-trans:

\[ \text{is-binqueue } l \text{ } x \# xs \implies \text{is-binqueue } \text{ (Suc } l) \text{ } xs \]

by (cases x) simp-all

**lemma** is-binqueue-head:

\[ \text{is-binqueue } l \text{ } x \# xs \implies \text{is-binqueue } l \text{ } [x] \]

by (cases x) simp-all

**lemma** is-binqueue-append:

\[ \text{is-binqueue } l \text{ } xs \implies \text{is-binqueue } \text{ (length } xs + l) \text{ } ys \implies \text{is-binqueue } l \text{ } (xs \circ ys) \]

by (induct xs arbitrary: l) (auto intro: is-binqueue.intros elim: is-binqueue.cases)

**lemma** is-binqueue-append-dest [dest]:

\[ \text{is-binqueue } l \text{ } (xs \circ ys) \implies \text{is-binqueue } l \text{ } xs \]

by (induct xs arbitrary: l) (auto intro: is-binqueue.intros elim: is-binqueue.cases)

**lemma** is-binqueue-children:

assumes is-bintree-list l ts

and is-heap-list t ts

shows is-binqueue 0 (map Some (rev ts))

using assms by (induct ts) (auto simp add: is-binqueue-append)

**lemma** is-binqueue-select:
is-binqueue l xs \Rightarrow Some t \in set xs \Rightarrow \exists k. is-bintree k t \land is-heap t

by (induct xs arbitrary: l) (auto intro: is-binqueue.intros elim: is-binqueue.cases)

Normalized representation

inductive normalized :: ('a, 'b) binqueue \Rightarrow bool where
  normalized-Nil: normalized []
| normalized-single: normalized [Some t]
| normalized-append: xs \neq [] \Rightarrow normalized xs \Rightarrow normalized (ys @ xs)

lemma normalized-last-not-None:
  — sometimes the inductive definition might work better
normalized xs \leftrightarrow xs = [] \lor last xs \neq None

proof
  assume normalized xs
  then show xs = [] \lor last xs \neq None
    by (rule normalized.induct) simp-all
next
  assume \ast: xs = [] \lor last xs \neq None
  show normalized xs proof (cases xs rule: rev-cases)
    case Nil then show \asthesis by (simp add: normalized.intros)
next
  case (snoc ys x) with \ast obtain t where last xs = Some t by auto
  with snoc have xs = ys @ [Some t] by simp
  then show \asthesis by (simp add: normalized.intros)
qed

lemma normalized-simps [simp]:
  normalized [] \leftrightarrow True
normalized (Some t \# xs) \leftrightarrow normalized xs
normalized (None \# xs) \leftrightarrow xs \neq [] \land normalized xs
by (simp-all add: normalized-last-not-None)

lemma normalized-map-Some [simp]:
normalized (map Some xs)
by (induct xs) simp-all

lemma normalized-Cons:
normalized (x\#xs) \Rightarrow normalized xs
by (auto simp add: normalized-last-not-None)

lemma normalized-append:
normalized xs \Rightarrow normalized ys \Rightarrow normalized (xs@ys)
by (cases ys) (simp-all add: normalized-last-not-None)
lemma normalized-not-None:
  normalized xs \Rightarrow set xs \neq \{None\}
by (induct xs) (auto simp add: normalized-Cons [of - ts] dest: subset_singletonD)

primrec normalize' :: ('a, 'b) binqueue \Rightarrow ('a, 'b) binqueue where
  normalize' [] = []
| normalize' (x # xs) = (case x of None \Rightarrow normalize' xs | Some t \Rightarrow (x # xs))

definition normalize :: ('a, 'b) binqueue \Rightarrow ('a, 'b) binqueue where
  normalize xs = rev (normalize' (rev xs))

lemma normalized-normalize:
  normalized (normalize xs)
proof (induct xs rule: rev-induct)
  case (snoc y ys) then show ?case
    by (cases y) (simp-all add: normalized-last-not-None normalize-def)
qed (simp add: normalize-def)

lemma is-binqueue-normalize:
  is-binqueue l xs \Rightarrow is-binqueue l (normalize xs)
unfolding normalize-def
  by (induct xs arbitrary: l rule: rev-induct) (auto split: option.split)

2.3 Operations
Adding data

definition merge :: ('a::linorder, 'b) bintree \Rightarrow ('a, 'b) bintree \Rightarrow ('a, 'b) bintree
where
  merge t1 t2 = (if priority t1 < priority t2
    then Node (priority t1) (val t1) (t2 # children t1)
    else Node (priority t2) (val t2) (t1 # children t2))

lemma is-bintree-list-merge:
  assumes is-bintree l t1 is-bintree l t2
  shows is-bintree (Suc l) (merge t1 t2)
  using assms by (simp add: merge-def)

lemma is-heap-merge:
  assumes is-heap t1 is-heap t2
  shows is-heap (merge t1 t2)
  using assms by (auto simp add: merge-def)
fun
\texttt{add} :: (\texttt{\textquoteleft\texttt{a}:linorder, \textquoteleft\texttt{b}}) \texttt{bintree option \Rightarrow (\textquoteleft\texttt{a}, \textquoteleft\texttt{b}}) \texttt{binqueue \Rightarrow (\textquoteleft\texttt{a}, \textquoteleft\texttt{b}}) \texttt{binqueue}

where
\texttt{add None xs} = \texttt{xs}
| \texttt{add (Some t) []} = [\texttt{Some t}]
| \texttt{add (Some t) (\texttt{None \# xs})} = \texttt{Some t \# xs}
| \texttt{add (Some t) (\texttt{Some r \# xs})} = \texttt{None \# add (Some (merge t r)) xs}

lemma \texttt{add-Some-not-Nil [simp]:}
\texttt{add (Some t) xs} \neq [\texttt{[]}]
by (\texttt{induct Some t xs rule: add.induct}) simp-all

lemma \texttt{normalized-add:}
\texttt{assumes normalized xs}
\texttt{shows normalized \ (add x xs)}
using \texttt{assms by (induct xs rule: add.induct)} simp-all

lemma \texttt{is-binqueue-add-None:}
\texttt{assumes is-binqueue l xs}
\texttt{shows is-binqueue l \ (add None xs)}
using \texttt{assms by simp}

lemma \texttt{is-binqueue-add-Some:}
\texttt{assumes is-binqueue l xs}
\texttt{and is-bintree l t}
\texttt{and is-heap t}
\texttt{shows is-binqueue l \ (add (Some t) xs)}
using \texttt{assms by (induct xs arbitrary: t) (simp-all add: is-bintree-list-merge is-heap-merge)}

function
\texttt{meld} :: (\texttt{\textquoteleft\texttt{a}:linorder, \textquoteleft\texttt{b}}) \texttt{binqueue \Rightarrow (\textquoteleft\texttt{a}, \textquoteleft\texttt{b}}) \texttt{binqueue \Rightarrow (\textquoteleft\texttt{a}, \textquoteleft\texttt{b}}) \texttt{binqueue}

where
\texttt{meld [] ys} = \texttt{ys}
| \texttt{meld xs []} = \texttt{ys}
| \texttt{meld (\texttt{None \# xs}) \ (y \# ys)} = \texttt{y \# meld xs ys}
| \texttt{meld (x \# xs) \ (\texttt{None \# ys})} = \texttt{x \# meld xs ys}
| \texttt{meld (\texttt{Some t \# xs}) \ (\texttt{Some r \# ys})} =
  \texttt{None \# add (Some (merge t r)) (meld xs ys)}
by pat-completeness auto termination by lexicographic-order

lemma \texttt{meld-singleton-add [simp]:}
\texttt{meld [Some t] xs} = \texttt{add (Some t) xs}
by (\texttt{induct Some t xs rule: add.induct}) simp-all
lemma nonempty-meld [simp]:
  \( xs \neq [] \implies \text{meld} \ xs \ ys \neq [] \)
  \( ys \neq [] \implies \text{meld} \ xs \ ys \neq [] \)
  by (induct \( xs \ ys \) rule: meld.induct) auto

lemma nonempty-meld-commute:
  \( \text{meld} \ xs \ ys \neq [] \implies \text{meld} \ xs \ ys \neq [] \)
  by (induct \( xs \ ys \) rule: meld.induct) auto

lemma is-binqueue-meld:
  assumes is-binqueue l xs
  and is-binqueue l ys
  shows is-binqueue l (meld xs ys)
  using assms
  proof (induct xs ys arbitrary: l rule: meld.induct)
    fix \( xs \ ys :: (\'a, \'b) \text{ binqueue} \)
    fix y :: (\'a, \'b) bintree option
    fix l :: nat
    assume \( \forall \ l. \text{is-binqueue} \ l \ xs \implies \text{is-binqueue} \ l \ ys \)
    \( \implies \text{is-binqueue} \ l \ (\text{meld} \ xs \ ys) \)
    and is-binqueue l (None # xs)
    and is-binqueue l (y # ys)
    then show is-binqueue l (meld (None # xs) (y # ys)) by (cases y) simp-all
  next
    fix \( xs \ ys :: (\'a, \'b) \text{ binqueue} \)
    fix x :: (\'a, \'b) bintree option
    fix l :: nat
    assume \( \forall \ l. \text{is-binqueue} \ l \ xs \implies \text{is-binqueue} \ l \ ys \)
    \( \implies \text{is-binqueue} \ l \ (\text{meld} \ xs \ ys) \)
    and is-binqueue l (x # xs)
    and is-binqueue l (None # ys)
    then show is-binqueue l (meld (x # xs) (None # ys)) by (cases x) simp-all
  qed (simp-all add: is-bintree-list-merge is-heap-merge is-binqueue-add-Some)

lemma normalized-meld:
  assumes normalized xs
  and normalized ys
  shows normalized (meld xs ys)
  using assms
  proof (induct xs ys rule: meld.induct)
    fix \( xs \ ys :: (\'a, \'b) \text{ binqueue} \)
    fix y :: (\'a, \'b) bintree option
    assume normalized xs \( \implies \) normalized ys \( \implies \) normalized (meld xs ys)
    and normalized (None # xs)
    and normalized (y # ys)
then show \( \text{normalized} \ (\text{meld} \ (\text{None} \ # \ xs) \ (y \ # \ ys)) \) by (cases \( y \)) simp-all
next
fix \( xs \ y :: ('a, 'b) \text{ bintree} \)
fix \( x :: ('a, 'b) \text{ bintree} \)
assume \( \text{normalized} \ xs \implies \text{normalized} \ ys \implies \text{normalized} \ (\text{meld} \ xs \ ys) \)
and \( \text{normalized} \ (x \ # \ xs) \)
and \( \text{normalized} \ (\text{None} \ # \ ys) \)
then show \( \text{normalized} \ (\text{meld} \ (x \ # \ xs) \ (\text{None} \ # \ ys)) \) by (cases \( x \)) simp-all
qed (simp-all add: \text{normalized-add})

lemma \text{normalized-meld-weak}: 
assumes \( \text{normalized} \ xs \)
and \( \text{length} \ ys \leq \text{length} \ xs \)
shows \( \text{normalized} \ (\text{meld} \ xs \ ys) \)
using assms
proof (induct \( xs \ ys \) rule: meld.induct)
fix \( xs \ y :: ('a, 'b) \text{ bintree} \)
fix \( x :: ('a, 'b) \text{ bintree} \)
assume \( \text{normalized} \ xs \implies \text{length} \ ys \leq \text{length} \ xs \implies \text{normalized} \ (\text{meld} \ xs \ ys) \)
and \( \text{normalized} \ (\text{None} \ # \ xs) \)
and \( \text{length} \ (y \ # \ ys) \leq \text{length} \ (\text{None} \ # \ xs) \)
then show \( \text{normalized} \ (\text{meld} \ (\text{None} \ # \ xs) \ (y \ # \ ys)) \) by (cases \( y \)) simp-all
next
fix \( xs \ y :: ('a, 'b) \text{ bintree} \)
fix \( x :: ('a, 'b) \text{ bintree} \)
assume \( \text{normalized} \ xs \implies \text{length} \ ys \leq \text{length} \ xs \implies \text{normalized} \ (\text{meld} \ xs \ ys) \)
and \( \text{normalized} \ (x \ # \ xs) \)
and \( \text{length} \ (\text{None} \ # \ ys) \leq \text{length} \ (x \ # \ xs) \)
then show \( \text{normalized} \ (\text{meld} \ (x \ # \ xs) \ (\text{None} \ # \ ys)) \) by (cases \( x \)) simp-all
qed (simp-all add: \text{normalized-add})

definition \text{least} :: 'a::\text{linorder} \text{ option} \Rightarrow 'a \text{ option} \Rightarrow 'a \text{ option} \ where 
\text{least} \ x \ y = (\text{case} \ x \ of 
None ⇒ y 
| \text{Some} \ x' ⇒ (\text{case} \ y \ of 
None ⇒ x 
| \text{Some} \ y' ⇒ (x' \leq y' \text{ then} x \text{ else} y))

lemma \text{least-simps} [simp, code]:
\text{least} \ \text{None} \ \text{=} \ \text{x}
\text{least} \ \text{x} \ \text{None} \ \text{=} \ \text{x}
\text{least} \ (\text{Some} \ x') \ (\text{Some} \ y') = (\text{if} \ x' \leq y' \text{ then} \text{Some} \ x' \text{ else} \text{Some} \ y')

unfolding \text{least-def} by (simp-all) (cases \text{x}, simp-all)

lemma \text{least-split}:
assumes least x y = Some z
shows x = Some z \lor y = Some z
using assms proof (cases x)
case (Some x') with assms show ?thesis by (cases y) (simp-all add: eq-commute)
qed simp

interpretation least!: semilattice least proof
qed (auto simp add: least-def split: option.split)

declaration min :: ('a::linorder, 'b) binqueue \Rightarrow 'a option where
min xs = fold least (map (map-option priority) xs) None

lemma min-simps [simp]:
min [] = None
min (None # xs) = min xs
min (Some t # xs) = least (Some (priority t)) (min xs)
by (simp-all add: min-def fold-commute-apply [symmetric]
fun-eq-iff least-simps)

lemma [code]:
min xs = fold (\lambda x. least (map-option priority x)) xs None
by (simp add: min-def fold-map o-def)

lemma min-single:
min [x] = Some a \Longrightarrow priority (the x) = a
min [x] = None \Longrightarrow x = None
by (auto simp add: min-def)

lemma min-Some-not-None:
min (Some t # xs) \neq None
by (cases min xs) simp-all

lemma min-None-trans:
assumes min (x#xs) = None
shows min xs = None
using assms proof (cases x)
case None with assms show ?thesis by simp
next
case (Some t) with assms show ?thesis by (simp only: min-Some-not-None)
qed

lemma min-None-None:
min xs = None \Longleftrightarrow xs = [] \lor set xs = {None}
proof (rule iffI)
  have splitQ: \forall xs. xs \subseteq {None} \Longrightarrow xs = {} \lor xs = {None} by auto

15
assume $\text{min } xs = \text{None}$
then have $\text{set } xs \subseteq \{\text{None}\}$
proof (induct $xs$)
  case (None $ys$) thus ?case using min-None-trans[of - $ys$] by simp-all
next
  case (Some $t$ $ys$) thus ?case using min-Some-not-None[of $t$ $ys$] by simp
qed simp with $\text{splitQ}$ show $xs = [] \lor \text{set } xs = \{\text{None}\}$ by auto
next
  show $xs = [] \lor \text{set } xs = \{\text{None}\} \Longrightarrow \text{min } xs = \text{None}$
  by (induct $xs$) (auto dest: subset-singletonD)
qed

lemma normalized-min-not-None:
normalized $xs \Longrightarrow xs \neq [] \Longrightarrow \text{min } xs \neq \text{None}$
by (simp add: min-None-None normalized-not-None)

lemma min-is-min:
assumes normalized $xs$
and $xs \neq []$
and $\text{min } xs = \text{Some } a$
shows $\forall x \in \text{set } xs. x = \text{None} \lor a \leq \text{priority } (\text{the } x)$
using assms proof (induct $xs$ arbitrary: $a$ rule: bqueue-induct)
  case (Some $t$ $ys$) thus ?case
proof (cases $ys = []$)
  case False
  with Some have $N$: normalized $ys$ using normalized-Cons[of - $ys$] by simp
  with $ys \neq []$ have $\text{min } ys \neq \text{None}$
  by (simp add: normalized-min-not-None)
  then obtain $a'$ where $oa': \text{min } ys = \text{Some } a'$ by auto
  with Some $N$ False
  have $\forall y \in \text{set } ys. y = \text{None} \lor a' \leq \text{priority } (\text{the } y)$ by simp
  with Some $oa'$ show ?thesis
  by (cases $a' \leq \text{priority } t$) (auto simp add: least.commute)
qed simp
qed simp-all

lemma min-exists:
assumes $\text{min } xs = \text{Some } a$
shows $\text{Some } a \in \text{map-option priority } \cdot \text{set } xs$
proof (rule ccontr)
  assume $\text{Some } a \notin \text{map-option priority } \cdot \text{set } xs$

then have $\forall x \in \text{set } xs, x = \text{None} \lor \text{priority (the } x) \neq a$ by (induct $xs$) auto
then have $\min xs \neq \text{Some } a$
proof (induct $xs$ arbitrary: $a$)
case (Some $t$ $ys$)
  hence priority $t \neq a$ and $\min ys \neq \text{Some } a$ by simp-all
  show $?case
proof (rule contr, simp)
  assume least (Some (priority $t$)) ($\min ys$) = Some $a$
  hence Some (priority $t$) = Some $a$ \lor $\min ys$ = Some $a$ by (rule least-split)
  with $\min ys \neq \text{Some } a$ have priority $t$ = $a$ by simp
  with ($\text{priority } t \neq a$) show False by simp
qed
qed simp-all
with assms show False by simp
qed

primrec find :: 'a::linorder \Rightarrow ('a, 'b) binqueue \Rightarrow ('a, 'b) bintree option
where
  find $a$ [] = None
| find $a$ (x#xs) = (case x of None \Rightarrow find $a$ $xs$
  | Some $t$ \Rightarrow if priority $t$ = $a$ then Some $t$ else find $a$ $xs$)

declare find-simps [simp del]

lemma find-simps [simp, code]:
  find $a$ [] = None
  find $a$ (None # $xs$) = find $a$ $xs$
  find $a$ (Some $t$ # $xs$) = (if priority $t$ = $a$ then Some $t$ else find $a$ $xs$)
  by (simp-all add: find-def)

lemma find-works:
  assumes Some $a$ \in \text{set } (\map \text{(map-option priority) } xs)$
  shows $\exists t$. find $a$ $xs$ = Some $t$ \AND priority $t$ = $a$
  using assms by (induct $xs$) auto

lemma find-works-not-None:
  Some $a$ \in \text{set } (\map \text{(map-option priority) } xs) \Longrightarrow find a $xs$ \neq None
  by (drule find-works) auto

lemma find-None:
  find $a$ $xs$ = None \Longrightarrow Some $a$ \notin \text{set } (\map \text{(map-option priority) } xs)$
  by (auto simp add: find-works-not-None)

lemma find-exist:
  find $a$ $xs$ = Some $t$ \Longrightarrow Some $t$ \in \text{set } $xs$
  by (induct $xs$) (simp-all add: eq-commute)
definition \texttt{find-min} :: \texttt{('a::linorder, 'b) binqueue \Rightarrow ('a, 'b) bintree option} \texttt{ where}
\[
\texttt{find-min xs} = (\texttt{case min xs of None \Rightarrow None | Some a \Rightarrow find a xs})
\]

\textbf{lemma} \texttt{find-min-simps} [\texttt{simp}]:
\[
\begin{align*}
\texttt{find-min} \ [\] & = \texttt{None} \\
\texttt{find-min} \ (\texttt{None} \# \texttt{xs}) & = \texttt{find-min} \ \texttt{xs}
\end{align*}
\]
by (\texttt{auto simp add: find-min-def split: option.split})

\textbf{lemma} \texttt{find-min-single}:
\[
\texttt{find-min} \ [\texttt{x}] = \texttt{x}
\]
by (\texttt{cases x}) (\texttt{auto simp add: find-min-def})

\textbf{lemma} \texttt{min-eq-find-min-None}:
\[
\texttt{min} \ \texttt{xs} = \texttt{None} \iff \texttt{find-min} \ \texttt{xs} = \texttt{None}
\]
\textbf{proof} (\texttt{rule iffI})
\[
\begin{align*}
\texttt{show} & \quad \texttt{min} \ \texttt{xs} = \texttt{None} \Longrightarrow \texttt{find-min} \ \texttt{xs} = \texttt{None} \\
\texttt{by} & \quad \texttt{(simp add: find-min-def)}
\end{align*}
\]
next
\[
\begin{align*}
\texttt{assume} & \quad *: \quad \texttt{find-min} \ \texttt{xs} = \texttt{None} \\
\texttt{show} & \quad \texttt{min} \ \texttt{xs} = \texttt{None}
\end{align*}
\]
\textbf{proof} (\texttt{rule ccontr})
\[
\begin{align*}
\texttt{assume} & \quad \texttt{min} \ \texttt{xs} \neq \texttt{None} \\
\texttt{then} & \quad \texttt{obtain} \quad \texttt{a} \quad \texttt{where} \quad \texttt{min} \ \texttt{xs} = \texttt{Some} \ \texttt{a} \quad \texttt{by} \quad \texttt{auto} \\
\texttt{hence} & \quad \texttt{find-min} \ \texttt{xs} \neq \texttt{None} \\
\texttt{by} & \quad \texttt{(simp add: find-min-def min-exists find-works-not-None)} \\
\texttt{with} & \quad * \quad \texttt{show} \quad \texttt{False} \quad \texttt{by} \quad \texttt{simp}
\]
\texttt{qed}
\texttt{qed}

\textbf{lemma} \texttt{min-eq-find-min-Some}:
\[
\texttt{min} \ \texttt{xs} = \texttt{Some} \ \texttt{a} \iff \exists \ t. \ \texttt{find-min} \ \texttt{xs} = \texttt{Some} \ t \land \texttt{priority} \ t = \texttt{a}
\]
\textbf{proof} (\texttt{rule iffI})
\[
\begin{align*}
\texttt{show} & \quad \exists \ t. \ \texttt{find-min} \ \texttt{xs} = \texttt{Some} \ t \land \texttt{priority} \ t = \texttt{a} \\
\texttt{by} & \quad \texttt{(simp add: find-min-def find-works min-exists)}
\end{align*}
\]
\textbf{assume} \*: \exists \ t. \ \texttt{find-min} \ \texttt{xs} = \texttt{Some} \ t \land \texttt{priority} \ t = \texttt{a}
\[
\texttt{show} \quad \texttt{min} \ \texttt{xs} = \texttt{Some} \ \texttt{a}
\]
\textbf{proof} (\texttt{rule ccontr})
\[
\begin{align*}
\texttt{assume} & \quad \texttt{min} \ \texttt{xs} \neq \texttt{Some} \ \texttt{a} \quad \texttt{thus} \quad \texttt{False} \\
\texttt{proof} & \quad \texttt{(cases min} \ \texttt{xs}) \\
\texttt{case} & \quad \texttt{None} \\
\texttt{hence} & \quad \texttt{find-min} \ \texttt{xs} = \texttt{None} \quad \texttt{by} \quad \texttt{(simp only: min-eq-find-min-None)}
\end{align*}
\]

18
with show False by simp
next
  case (Some b)
    with min xs ≠ Some a have a ≠ b by simp
    with show False using D1 by auto
  qed
qed

lemma find-min-exist:
assumes find-min xs = Some t
shows Some t ∈ set xs
proof –
  from assms have min xs ≠ None by (simp add: min-eq-find-min-None)
  with assms show ?thesis by (auto simp add: find-min-def find-exist)
qed

lemma find-min-is-min:
assumes normalized xs
and xs ≠ []
and find-min xs = Some t
shows ∀ x ∈ set xs. x = None ∨ (priority t) ≤ priority (the x)
using assms by (simp add: min-eq-find-min-Some min-is-min)

lemma normalized-find-min-exists:
normalized xs ⇒ xs ≠ [] ⇒ ∃t. find-min xs = Some t
by (drule normalized-min-not-None) (simp-all add: min-eq-find-min-None)

primrec
match :: 'a::linorder ⇒ ('a, 'b) bintree option ⇒ ('a, 'b) bintree option
where
  match a None = None
| match a (Some t) = (if priority t = a then None else Some t)

definition delete-min :: ('a::linorder, 'b) bintree ⇒ ('a, 'b) bintree
where
  delete-min xs = (case find-min xs
  of Some (Node a v ts) ⇒
    normalize (meld (map Some (rev ts)) (map (match a) xs))
    | None ⇒ [])

lemma delete-min-empty [simp]:
delete-min [] = []
by (simp add: delete-min-def)

lemma delete-min-nonempty [simp]:

\[ \text{normalized } xs \implies xq \neq [] \implies \text{find-min } xs = \text{Some } t \]
\[ \implies \text{delete-min } xs = \text{normaliz}e \]
\[ (\text{meld (map Some (rev (children t)))) (\text{map (match (priority t)) } xs)) \]
\[ \text{unfolding delete-min-def by (cases } t \text{) simp} \]

\textbf{lemma is-binqueue-delete-min:}

\textbf{assumes} is-binqueue 0 xs
\textbf{shows} is-binqueue 0 (delete-min xs)
\textbf{proof} (cases find-min xs)
\hspace{1em} \textbf{case (Some } t \text{)}
\hspace{2em} \textbf{from assms have} is-binqueue 0 (map (match (priority t)) xs)
\hspace{3em} \textbf{by (induct } xs \text{) simp-all}
\hspace{1em} \textbf{moreover}
\hspace{2em} \textbf{from Some have} Some } t \in \text{ set } xs \text{ by (rule find-min-exist)}
\hspace{3em} \textbf{with assms have } \exists l. \text{ is-bintree } l \ t \text{ and is-heap } t
\hspace{4em} \textbf{using is-binqueue-select[of } 0 \text{ } xs \text{ } t \text{] by auto}
\hspace{3em} \textbf{with assms have} is-binqueue 0 (\text{map Some (rev (children } t \text{))})
\hspace{4em} \textbf{by (auto simp add: is-binqueue-children)}
\hspace{1em} \textbf{ultimately show } \text{thesis using Some}
\hspace{2em} \textbf{by (auto simp add: is-binqueue-meld delete-min-def is-binqueue-normalize}
\hspace{3em} \text{split: bintree.split)}
\textbf{qed (simp add: delete-min-def)}

\textbf{lemma normalized-delete-min:}

\textbf{normalized } (\text{delete-min } xs)
\hspace{1em} \textbf{by (cases } find-min xs)
\hspace{2em} \textbf{(auto simp add: delete-min-def normalized-normalize split: bintree.split)}

\textbf{Dedicated grand unified operation for generated program}

\textbf{definition}
\textbf{meld'} :: ("a", "b") bintree option \Rightarrow ("a":linorder, "b") binqueue
\hspace{1em} \Rightarrow ("a", "b") binqueue \Rightarrow ("a", "b") binqueue
\textbf{where}
\textbf{meld'} } z \hspace{0.5em} xs \hspace{0.5em} ys = \text{add } z (\text{meld } xs \hspace{0.5em} ys)

\textbf{lemma [code]:}
\textbf{add z } xs = \text{meld'} z \hspace{0.5em} [] \hspace{0.5em} xs
\textbf{meld } xs \hspace{0.5em} ys = \text{meld'} } \text{None } \hspace{0.5em} xs \hspace{0.5em} ys
\hspace{1em} \textbf{by (simp-all add: meld'-def)}

\textbf{lemma [code]:}
\textbf{meld'} } z (\text{Some } t \neq xs) (\text{Some } r \neq ys) =
z ≠ (meld′ (Some (merge t r)) xs ys)
meld′ (Some t) (Some r ≠ xs) (None ≠ ys) =
  None ≠ (meld′ (Some (merge t r)) xs ys)
meld′ (Some t) (None ≠ xs) (Some r ≠ ys) =
  None ≠ (meld′ (Some (merge t r)) xs ys)
meld′ None (x ≠ xs) (None ≠ ys) = x ≠ (meld′ None xs ys)
meld′ None (None ≠ xs) (y ≠ ys) = y ≠ (meld′ None xs ys)
meld′ z (None ≠ xs) (None ≠ ys) = z ≠ (meld′ None xs ys)
meld′ z xs [] = meld′ z [] xs
meld′ z [] (y ≠ ys) = meld′ None [z] (y ≠ ys)
meld′ (Some t) [] ys = meld′ None [Some t] ys
meld′ None [] ys = ys
by (simp add: meld′-def | cases z)+

Interface operations

abbreviation (input) empty :: ('a,'b) binqueue where
  empty ≡ []

definition
  insert :: 'a::linorder ⇒ 'b ⇒ ('a,'b) binqueue ⇒ ('a,'b) binqueue
  where
  insert a v xs = add (Some (Node a v [])) xs

lemma insert-simps [simp]:
  insert a v [] = [Some (Node a v [])]
  insert a v (None ≠ xs) = Some (Node a v []) ≠ xs
  insert a v (Some t ≠ xs) = None ≠ add (Some (merge (Node a v [])) t)) xs
  by (simp-all add: insert-def)

lemma is-binqueue-insert:
  is-binqueue 0 xs ⇒ is-binqueue 0 (insert a v xs)
  by (simp add: is-binqueue-add-Some insert-def)

lemma normalized-insert:
  normalized xs ⇒ normalized (insert a v xs)
  by (simp add: normalized-add-Some insert-def)

definition
  pop :: ('a::linorder, 'b) binqueue ⇒ ('b × 'a) option × ('a,'b) binqueue
  where
  pop xs = (case find-min xs of
    None ⇒ (None, xs)
  | Some t ⇒ (Some (val t, priority t), delete-min xs))
lemma pop-empty [simp]:
  pop empty = (None, empty)
  by (simp add: pop-def empty-def)

lemma pop-nonempty [simp]:
  normalized xs \Rightarrow xs \neq [] \Rightarrow find-min xs = Some t
  \Rightarrow pop xs = (Some (val t, priority t), normalize
    (meld (map Some (rev (children t))) (map (match (priority t)) xs))))
  by (simp add: pop-def)

lemma pop-code [code]:
  pop xs = (case find-min xs of
    None \Rightarrow (None, xs)
  | Some t \Rightarrow (Some (val t, priority t)
    (meld (map Some (rev (children t))) (map (match (priority t)) xs))))
  by (cases find-min xs) (simp-all add: pop-def delete-min-def split: bintree.split)

3 Relating Functional Binomial Queues To The Abstract Priority Queues

notation
  PQ.values (|(-)|)
  and PQ.priorities (||(-)||)

Naming convention: prefix bt- for bintrees, bts- for bintree lists, no prefix for binqueues.

primrec bt-dfs :: ('a::linorder, 'b) bintree \Rightarrow 'c list
  and bts-dfs :: ('a::linorder, 'b) bintree \Rightarrow 'c list

where
  bt-dfs f (Node a v ts) = f (Node a v ts) # bt-dfs f ts
  | bts-dfs f [] = []
  | bts-dfs f (t # ts) = bt-dfs f t @ bts-dfs f ts

lemma bt-dfs-simp:
  bt-dfs f t = f t # bt-dfs f (children t)
  by (cases t) simp-all

lemma bts-dfs-append [simp]:
  bts-dfs f (ts @ rs) = bts-dfs f ts @ bts-dfs f rs
  by (induct ts) simp-all

lemma set-bts-dfs-rev:
set (bts-dfs f (rev ts)) = set (bts-dfs f ts)
by (induct ts) auto

lemma bts-dfs-rev-distinct:
distinct (bts-dfs f ts) \implies distinct (bts-dfs f (rev ts))
by (induct ts) (auto simp add: set-bts-dfs-rev)

lemma bt-dfs-comp:
bts-dfs (f \circ g) t = map f (bt-dfs g t)
bts-dfs (f \circ g) ts = map f (bts-dfs g ts)
by (induct t and ts rule: bt-dfs.induct bts-dfs.induct) simp-all

lemma bt-dfs-comp-distinct:
distinct (bts-dfs (f \circ g) t) \implies distinct (bts-dfs g t)
distinct (bts-dfs (f \circ g) ts) \implies distinct (bts-dfs g ts)
by (simp-all add: bt-dfs-comp distinct-map [of f])

lemma bt-dfs-distinct-children:
distinct (bts-dfs f x) \implies distinct (bts-dfs f (children x))
by (cases x) simp

fun dfs :: ([‘a::linorder, ‘b] bintree \Rightarrow ‘c) \Rightarrow ([‘a, ‘b] binqueue \Rightarrow ‘c) list
where
dfs f [] = []
| dfs f (None # xs) = dfs f xs
| dfs f (Some t # xs) = bt-dfs f t @ dfs f xs

lemma dfs-append:
dfs f (xs @ ys) = (dfs f xs) @ (dfs f ys)
by (induct xs) simp-all

lemma set-dfs-rev:
set (dfs f (rev xs)) = set (dfs f xs)
by (induct xs) (auto simp add: dfs-append)

lemma set-dfs-Cons:
set (dfs f (x # xs)) = set (dfs f xs) \cup set (dfs f [x])
proof –
have set (dfs f (x # xs)) = set (dfs f (rev xs @ [x]))
using set-dfs-rev[of f rev xs @ [x]] by simp
thus ?thesis by (simp add: dfs-append set-dfs-rev)
qed

lemma dfs-comp:
dfs (f \circ g) xs = map f (dfs g xs)
by (induct xs) (simp-all add: bt-dfs-comp del: o-apply)
lemma dfs-comp-distinct:
  distinct (dfs (f ∘ g) xs) \implies distinct (dfs g xs)
  by (simp add: dfs-comp distinct-map[of f])

lemma dfs-distinct-member:
  distinct (dfs f xs) \implies
  Some x \in set xs \implies
  distinct (bt-dfs f x)
proof (induct xs arbitrary: x)
  case (Some r xs t) then show ?case
  by (cases t = r) simp-all
qed simp-all

lemma dfs-map-Some-idem:
  dfs f (map Some xs) = bts-dfs f xs
by (induct xs) simp-all

primrec alist :: ('a, 'b) bintree ⇒ ('b × 'a)
where
  alist (Node a v -) = (v, a)

lemma alist-split-pre:
  val t = (fst o alist) t
  priority t = (snd o alist) t
by (cases t, simp)+

lemma alist-split:
  val = fst o alist
  priority =snd o alist
by (auto intro!: ext simp add: alist-split-pre)

lemma alist-split-set:
  set (dfs val xs) = fst ' set (dfs alist xs)
  set (dfs priority xs) = snd ' set (dfs alist xs)
by (auto simp add: dfs-comp alist-split)

lemma in-set-in-alist:
  assumes Some t ∈ set xs
  shows (val t, priority t) ∈ set (dfs alist xs)
using assms
proof (induct xs)
  case (Some x xs) then show ?case
  proof (cases Some t ∈ set xs)
    case False with Some show ?thesis
    by (cases t) (auto simp add: bl-dfs-simp)
  qed simp
  qed simp-all

24
abbreviation vals where vals ≡ dfs val
abbreviation prios where prios ≡ dfs priority
abbreviation elements where elements ≡ dfs alist

primrec
  bt-augment :: ('a::linorder, 'b) bintree ⇒ ('b, 'a) PQ.pq
and
  bts-augment :: ('a::linorder, 'b) bintree list ⇒ ('b, 'a) PQ.pq ⇒ ('b, 'a) PQ.pq
where
  bt-augment (Node a v ts) q = PQ.push v a (bts-augment ts q)
| bts-augment [] q = q
| bts-augment (t # ts) q = bts-augment ts (bt-augment t q)

lemma bts-augment [simp]:
  bts-augment = fold bt-augment
proof (rule ext)
  fix ts :: ('a, 'b) bintree list
  show bts-augment ts = fold bt-augment ts
  by (induct ts) simp-all
qed

lemma bt-augment-Node [simp]:
  bt-augment (Node a v ts) q = PQ.push v a (fold bt-augment ts q)
by (simp add: bts-augment)

lemma bt-augment-simp:
  bt-augment t q = PQ.push (val t) (priority t) (fold bt-augment (children t) q)
by (cases t) (simp-all add: bts-augment)

declare bt-augment.simps [simp del] bts-augment.simps [simp del]

fun pqueue :: ('a::linorder, 'b) binqueue ⇒ ('b, 'a) PQ.pq where
  Empty: pqueue [] = PQ.empty
| None: pqueue (None # xs) = pqueue xs
| Some: pqueue (Some t # xs) = bt-augment t (pqueue xs)

lemma bt-augment-v-subset:
  set |q| ⊆ set |bt-augment t q|
  set |q| ⊆ set |bts-augment ts q|
by (induct t and ts arbitrary: q and q rule: bt-augment.induct bts-augment.induct)
auto

lemma bt-augment-v-in:
  v ∈ set |q| ⇒ v ∈ set |bt-augment t q|
\[ v \in \text{set}\ |\ q \implies v \in \text{set}\ |\ \text{bts-augment}\ ts\ q \]

using \( \text{bt-augment-v-subset[of q]} \) by auto

**Lemma bt-augment-v-union:**

\[ \text{set}\ |\ \text{bt-augment}\ t\ (\text{bt-augment}\ r\ q)\] = 
\[ \text{set}\ |\ \text{bt-augment}\ t\ q\ \cup\ \text{set}\ |\ \text{bt-augment}\ r\ q\]

\[ \text{set}\ |\ \text{bts-augment}\ ts\ (\text{bt-augment}\ r\ q)\] = 
\[ \text{set}\ |\ \text{bts-augment}\ ts\ q\ \cup\ \text{set}\ |\ \text{bt-augment}\ r\ q\]

**Proof** (induct \( t \) and \( ts \) arbitrary; \( q \) \& \( r \) rule: bt-augment.induct bts-augment.induct)

**Case** Nil-bintree

from \( \text{bt-augment-v-subset[of q]} \) show \( \text{?case by auto} \)

**Qed** auto

**Lemma bt-val-augment:**

shows \( \text{set}\ (\text{bt-dfs val}\ t)\ \cup\ \text{set}\ |\ q\] = \( \text{set}\ |\ \text{bt-augment}\ t\ q\]

and \( \text{set}\ (\text{bts-dfs val}\ ts)\ \cup\ \text{set}\ |\ q\] = \( \text{set}\ |\ \text{bts-augment}\ ts\ q\]

**Proof** (induct \( t \) and \( ts \) rule: bt-augment.induct bts-augment.induct)

**Case** (Cons-bintree \( r\ rs \))

have \( \text{set}\ |\ \text{bts-augment}\ rs\ (\text{bt-augment}\ r\ q)\] = 
\[ \text{set}\ |\ \text{bts-augment}\ rs\ q\ \cup\ \text{set}\ |\ \text{bt-augment}\ r\ q\]

by (simp only: bt-augment-v-union)

with \( \text{bt-augment-v-subset[of q]} \)

have \( \text{set}\ |\ \text{bts-augment}\ rs\ (\text{bt-augment}\ r\ q)\] = 
\[ \text{set}\ |\ \text{bts-augment}\ rs\ q\ \cup\ \text{set}\ |\ \text{bt-augment}\ r\ q\ \cup\ \text{set}\ |\ q\]

by auto

with Cons-bintree show \( \text{?case by auto} \)

**Qed** auto

**Lemma vals-pqueue:**

\( \text{set}\ (\text{vals}\ xs)\) = \( \text{set}\ |\ \text{pqueue}\ xs\]

by (induct \( xs \)) (simp-all add: bt-val-augment)

**Lemma bt-augment-v-push:**

\[ \text{set}\ |\ \text{bt-augment}\ t\ (\text{PQ.}\ \text{push}\ v\ a\ q)\] = \( \text{set}\ |\ \text{bt-augment}\ t\ q\ \cup\ \{v\}\]

\[ \text{set}\ |\ \text{bts-augment}\ ts\ (\text{PQ.}\ \text{push}\ v\ a\ q)\] = \( \text{set}\ |\ \text{bts-augment}\ ts\ q\ \cup\ \{v\}\]

using \( \text{bt-val-augment[where q = PQ.\ push}\ v\ a\ q} \) by (simp-all add: bt-val-augment)

**Lemma bt-augment-v-push-commute:**

\[ \text{set}\ |\ \text{bt-augment}\ t\ (\text{PQ.}\ \text{push}\ v\ a\ q)\] = \( \text{set}\ |\ \text{PQ.}\ \text{push}\ v\ a\ (\text{bt-augment}\ t\ q)\]

\[ \text{set}\ |\ \text{bts-augment}\ ts\ (\text{PQ.}\ \text{push}\ v\ a\ q)\] = \( \text{set}\ |\ \text{PQ.}\ \text{push}\ v\ a\ (\text{bts-augment}\ ts\ q)\]

by (simp-all add: bt-augment-v-push del: bts-augment)

**Lemma bts-augment-v-union:**

\[ \text{set}\ |\ \text{bt-augment}\ t\ (\text{bts-augment}\ rs\ q)\] =
proof (induct t and ts arbitrary; q rs and q rs rule; bt-augment.induct bts-augment.induct)

next

next case (Cons-bintree x xs)
let ?L = set | bts-augment xs (bt-augment x (bts-augment rs q)) |

from btaugment-v-union
have *: \( q \cdot \) set | bts-augment xs (bt-augment x q) | =
set | bts-augment xs q | \( \cup \) set | bt-augment x q | by simp

with Cons-bintree
have ?L =
set | bts-augment xs q | \( \cup \) set | bts-augment rs q | \( \cup \) set | bt-augment x q | by auto

with * show ?case by auto
qed simp

lemma btaugment-v-commute:

set | bt-augment t (bt-augment r q) | = set | bt-augment r (bt-augment t q) |
set | bt-augment t (bts-augment rs q) | = set | bts-augment rs (bt-augment t q) |
set | bts-augment ts (bts-augment rs q) |

unfolding bts-augment-v-union bt-augment-v-union by auto

lemma btaugment-v-merge:

set | bt-augment (merge t r) q | = set | bt-augment t (bt-augment r q) |
by (simp add: bt-augment-simp [symmetric] btaugment-v-push
btaugment-v-commute merge-def)

lemma vals-merge [simp]:

set (bt-dfs val (merge t r)) = set (bt-dfs val t) \( \cup \) set (bt-dfs val r)
by (auto simp add: bt-dfs-simp merge-def)

lemma vals-merge-distinct:

distinct (bt-dfs val t) \( \Rightarrow \) distinct (bt-dfs val r) \( \Rightarrow \)
set (bt-dfs val t) \( \cap \) set (bt-dfs val r) = \( \{ \} \) \( \Rightarrow \)
distinct (bt-dfs val (merge t r))
by (auto simp add: bt-dfs-simp merge-def)

lemma vals-add-Cons:
\[
\text{set } (\text{vals } (\text{add } x \text{ xs})) = \text{set } (\text{vals } (x \# \text{xs}))
\]

\textbf{proof} (cases \textit{x})

\textbf{case} (Some \textit{t}) \textbf{ then show} ?thesis 
\textbf{by} (induct \textit{xs} arbitrary: \textit{x \ t}) \textbf{auto}
\textbf{qed simp}

\textbf{lemma vals-add-distinct:}
\textbf{assumes} \text{distinct } (\text{vals } \text{xs})
\textbf{and} \text{distinct } (\text{dfs val } [x])
\textbf{and} \text{set } (\text{vals } \text{xs}) \cap \text{set } (\text{dfs val } [x]) = \{\}
\textbf{shows} \text{distinct } (\text{vals } (\text{add } x \text{ xs}))
\textbf{using} \text{assms}
\textbf{proof} (cases \textit{x})
\textbf{case} (Some \textit{t}) \textbf{with} \text{assms} \textbf{show} ?thesis
\textbf{proof} (induct \textit{xs} arbitrary: \textit{x \ t})
\textbf{case} (Some \textit{r \ xs})
\textbf{then have} \text{set } (\text{bt-dfs val } t) \cap \text{set } (\text{bt-dfs val } r) = \{\} \textbf{by} \textbf{auto}
\textbf{with} \text{Some have} \text{distinct } (\text{bt-dfs val } (\text{merge } t \ r)) \textbf{by} (\text{simp add: vals-merge-distinct})
\textbf{moreover}
\textbf{with} \text{Some have} \text{set } (\text{vals } \text{xs}) \cap \text{set } (\text{bt-dfs val } (\text{merge } t \ r)) = \{\} \textbf{by} \textbf{auto}

\textbf{moreover note} \textit{Some} 
\textbf{ultimately show} ?\textit{case} \textbf{by} \textbf{simp}
\textbf{qed} \textbf{auto}
\textbf{qed simp}

\textbf{lemma vals-insert} [simp]:
\text{set } (\text{vals } (\text{insert } a \ v \text{ xs})) = \text{set } (\text{vals } \text{xs}) \cup \{v\}
\textbf{by} (\text{simp add: insert-def vals-add-Cons})

\textbf{lemma insert-v-push:}
\text{set } (\text{vals } (\text{insert } a \ v \text{ xs})) = \text{set } |PQ. \text{push } v \ a \ (\text{pqueue } \text{xs})|
\textbf{by} (\text{simp add: vals-pqueue[symmetric]})

\textbf{lemma vals-meld:}
\text{set } (\text{dfs val } (\text{meld } \text{xs } \text{ ys})) = \text{set } (\text{dfs val } \text{xs}) \cup \text{set } (\text{dfs val } \text{ys})
\textbf{proof} (induct \textit{xs } \textit{ys} \textbf{ rule: meld.induct})
\textbf{case} (3 \textit{x \ y \ ys}) \textbf{ then show} ?\text{induct}
\textbf{using} \text{set-dfs-Cons[of val } y \text{ meld } \text{xs } \text{ ys]} \textbf{using} \text{set-dfs-Cons[of val } y \text{ ys]} \textbf{by} \textbf{auto}
\textbf{next}
\textbf{case} (4 \textit{x \ xs } \textit{ys}) \textbf{ then show} ?\text{case}
\textbf{using} \text{set-dfs-Cons[of val } x \text{ meld } \text{xs } \text{ ys]} \textbf{using} \text{set-dfs-Cons[of val } x \text{ xs]} \textbf{by} \textbf{auto}
\textbf{next}
\textbf{case} (5 \textit{x \ y \ ys}) \textbf{ then show} ?\text{case} \textbf{by} (\text{auto simp add: vals-add-Cons})
\textbf{qed} \textbf{simp-all}

28
lemma vals-meld-distinct:
  distinct (dfs val xs) \implies distinct (dfs val ys) \implies
  set (dfs val xs) \cap set (dfs val ys) = \{\} \implies
  distinct (dfs val (meld xs ys))
proof (induct xs ys rule: meld.induct)
case (3 xs y ys) then show \(?case
proof (cases y)
  case None with 3 show \(?thesis by simp
next
case (Some t)
  from 3 have \(A\): set (vals xs) \cap set (vals ys) = \{\}
  using set-dfs-Cons[of val y ys] by auto
moreover
  from Some 3 have set (bt-dfs val t) \cap set (vals xs) = \{\} by auto
moreover
  from Some 3 have set (bt-dfs val t) \cap set (vals ys) = \{\} by simp
ultimately have set (bt-dfs val t) \cap set (vals (meld xs ys)) = \{\}
  by (auto simp add: vals-meld)
with 3 Some show \(?thesis by auto
qed
next
case (4 x xs y ys) then show \(?case
proof (cases x)
  case None with 4 show \(?thesis by simp
next
case (Some t)
  from 4 have set (vals xs) \cap set (vals ys) = \{\}
  using set-dfs-Cons[of val x xs] by auto
moreover
  from Some 4 have set (bt-dfs val t) \cap set (vals xs) = \{\} by simp
moreover
  from Some 4 have set (bt-dfs val t) \cap set (vals ys) = \{\} by auto
ultimately have set (bt-dfs val t) \cap set (vals (meld xs ys)) = \{\}
  by (auto simp add: vals-meld)
with 4 Some show \(?thesis by auto
qed
next
case (5 x xs y ys) then
have \( \text{set} (\text{vals } xs) \cap \text{set} (\text{vals } ys) = {} \) by (auto simp add: set-dfs-Cons)

with 5 have \( \text{distinct} (\text{vals } (\text{meld } xs ys)) \) by simp

moreover

from 5 have \( \text{set} (\text{bt-dfs val } x) \cap \text{set} (\text{bt-dfs val } y) = {} \) by auto

with 5 have \( \text{distinct} (\text{bt-dfs val } (\text{merge } x y)) \)
  by (simp add: vals-merge-distinct)

moreover

from 5 have \( \text{set} (\text{vals } (\text{meld } xs ys)) \cap \text{set} (\text{bt-dfs val } (\text{merge } x y)) = {} \)
  by (auto simp add: vals-meld)

ultimately show \( ?\text{case} \) by (simp add: vals-add-distinct)

qed simp-all

lemma bt-augment-alist-subset:
  \( \text{set} (\text{PQ.alist-of } q) \subseteq \text{set} (\text{PQ.alist-of } (\text{bt-augment } t \ q)) \)
  \( \text{set} (\text{PQ.alist-of } q) \subseteq \text{set} (\text{PQ.alist-of } (\text{bts-augment } ts \ q)) \)

proof (induct \( t \) and \( ts \) arbitrary: \( q \) and \( q \) rule: bt-augment.induct bts-augment.induct)
  case (Node a v rs)
  show \( ?\text{case} \) using Node[of \( q \)] by (auto simp add: bt-augment-simp set-insort)

qed auto

lemma bt-augment-alist-in:
  \( (v, a) \in \text{set} (\text{PQ.alist-of } q) \implies (v, a) \in \text{set} (\text{PQ.alist-of } (\text{bt-augment } t \ q)) \)
  \( (v, a) \in \text{set} (\text{PQ.alist-of } q) \implies (v, a) \in \text{set} (\text{PQ.alist-of } (\text{bts-augment } ts \ q)) \)

using bt-augment-alist-subset[of \( q \)] by auto

lemma bt-augment-alist-union:
  distinct \( (\text{bts-dfs val } (r \ # \ [t])) \) \implies
  \( \text{set} (\text{bts-dfs val } (r \ # \ [t])) \cap \text{set } |q| = {} \implies \)
  \( \text{set} (\text{PQ.alist-of } (\text{bt-augment } t \ q)) \subseteq \text{set} (\text{PQ.alist-of } (\text{bt-augment } r \ q)) \)

proof (induct \( t \) and \( ts \) arbitrary: \( q \) \( r \) and \( q \) \( r \) rule: bt-augment.induct bts-augment.induct)
  case Nil-bintree
  from bt-augment-alist-subset[of \( q \)] show \( ?\text{case} \) by auto

next
  case (Node a v rs) then
  have \( \text{set} (\text{PQ.alist-of } (\text{bts-augment } rs \ (\text{bt-augment } r \ q))) = \)

  30
set \( (PQ.\text{alist-of (bts-augment rs q)}) \cup (PQ.\text{alist-of (bt-augment r q)}) \)

by simp

moreover
from \( \text{Node.prems have *: v \notin \text{set [bts-augment rs q] \cup set [bt-augment r q]} \) 

unfolding \( \text{bt-val-augment [symmetric]} \) by simp

hence \( v \notin \text{set [bts-augment rs (bt-augment r q)]} \) by (unfold \( \text{bt-augment-v-union} \))

moreover
from \( * \) have \( v \notin \text{set [bts-augment rs q]} \) by simp

ultimately show \( ?\text{case by (simp add: set-insert)} \)

next

case (\( \text{Cons-bintree x xs} \)) then

have — FIXME: ugly... and slow

distinct (\( \text{bts-dfs val (x \# xs)} \)) and
distinct (\( \text{bts-dfs val (r \# xs)} \)) and
distinct (\( \text{bts-dfs val [r,x]} \)) and

set (\( \text{bts-dfs val (x \# xs)} \)) \( \cap \) set (\( \text{bt-augment r q} \)) = \{\} and
set (\( \text{bts-dfs val (x \# xs)} \)) \( \cap \) set (\( \text{q} \)) = \{\} and
set (\( \text{bts-dfs val [r, x]} \)) \( \cap \) set (\( \text{q} \)) = \{\} and

unfolding \( \text{bt-val-augment [symmetric]} \) by auto

with \( \text{Cons-bintree.hyps show ?case by auto} \)

qed

lemma \( \text{bt-alist-augment:} \)

distinct (\( \text{bt-dfs val t} \)) \( \Rightarrow \)
set (\( \text{bt-dfs val t} \)) \( \cap \) set (\( \text{q} \)) = \{\} \( \Rightarrow \)
set (\( \text{bt-dfs alist t} \)) \( \cup \) set (\( PQ.\text{alist-of q} \)) = set (\( PQ.\text{alist-of (bt-augment t q)} \))

distinct (\( \text{bt-dfs val ts} \)) \( \Rightarrow \)
set (\( \text{bt-dfs val ts} \)) \( \cap \) set (\( \text{q} \)) = \{\} \( \Rightarrow \)
set (\( \text{bt-dfs alist ts} \)) \( \cup \) set (\( PQ.\text{alist-of q} \)) = set (\( PQ.\text{alist-of (bt-augment ts q)} \))

proof (\( \text{induct t and ts rule: bt-augment.induct bt-augment.induct} \))

case \( \text{Nil-bintree then show ?case by simp} \)

next

case (\( \text{Node a v rs} \))

hence \( v \notin \text{set [bts-augment rs q]} \)

unfolding \( \text{bt-val-augment [symmetric]} \) by simp

with \( \text{Node show ?case by (simp add: set-insert)} \)

next

case (\( \text{Cons-bintree r rs} \)) then

have set (\( PQ.\text{alist-of (bs-augment (r \# rs) q)} \)) =
set \((PQ.\text{alist-of} \ (\text{bts-augment} \ rs \ q)) \cup set \ (PQ.\text{alist-of} \ (\text{bts-augment} \ r \ q))\)

using \(\text{bt-augment-alist-union}\) by simp

with Cons-bintree \(\text{bt-augment-alist-subset}\) show \(\text{?pcase}\) by auto

qed

lemma alist-pqueue:
\[\text{distinct} \ (\text{vals} \ xs) \implies set \ (\text{dfs} \ alist \ xs) = set \ (PQ.\text{alist-of} \ (\text{pqueue} \ xs))\]

by (induct xs) (simp-all add: vals-pqueue bt-alist-augment)

lemma alist-pqueue-priority:
\[\text{distinct} \ (\text{vals} \ xs) \implies (v, a) \in set \ (\text{dfs} \ alist \ xs) \implies PQ.\text{priority} \ (\text{pqueue} \ xs) \ v = \text{Some} \ a\]

by (simp add: alist-pqueue PQ.priority-def)

lemma prios-pqueue:
\[\text{distinct} \ (\text{vals} \ xs) \implies set \ (\text{prios} \ xs) = set \ ||\text{pqueue} \ xs||\]

by (auto simp add: alist-pqueue priorities-set alist-split-set)

lemma alist-merge [simp]:
\[\text{distinct} \ (\text{bt-dfs} \ val \ t) \implies \text{distinct} \ (\text{bt-dfs} \ val \ r) \implies set \ (\text{bt-dfs} \ val \ t) \cap set \ (\text{bt-dfs} \ val \ r) = \{\} \implies set \ (\text{bt-dfs} \ alist \ (\text{merge} \ t \ r)) = set \ (\text{bt-dfs} \ alist \ t) \cup set \ (\text{bt-dfs} \ alist \ r)\]

by (auto simp add: bt-dfs-simp merge-def alist-split)

lemma alist-add-Cons:
\[\text{assumes distinct} \ (\text{vals} \ (x\#xs))\]
\[\text{shows set} \ (\text{dfs} \ alist \ (\text{add} \ x \ xs)) = set \ (\text{dfs} \ alist \ (x \#xs))\]

using assms proof (induct xs arbitrary: x)

case Empty then show \(\text{?case}\) by (cases x) simp-all

next
case None then show \(\text{?case}\) by (cases x) simp-all

next
case (Some y ys) then

show \(\text{?case}\)

proof (cases x)

case (Some t)

note prem = Some.prems Some

from prem have distinct \(\text{bt-dfs} \ val \ (\text{merge} \ t \ y)\)

by (auto simp add: bt-dfs-simp merge-def)

with prem have distinct \(\text{vals} \ (\text{Some} \ (\text{merge} \ t \ y) \# \ ys)\) by auto

with prem Some.hyps

have set \(\text{dfs} \ alist \ (\text{add} \ (\text{Some} \ (\text{merge} \ t \ y)) \ ys)) = \)

set \(\text{dfs} \ alist \ (\text{Some} \ (\text{merge} \ t \ y) \# \ ys))\) by simp

32
moreover
from prem have set (bt-dfs val t) ∩ set (bt-dfs val y) = {} by auto
with prem
have set (bt-dfs alist (merge t y)) =
  set (bt-dfs alist t) ∪ set (bt-dfs alist y)
by simp

moreover note prem and Un-assoc

ultimately
  show ?thesis by simp
qed simp

lemma alist-insert [simp]:
distinct (vals xs) ⟹
v ∉ set (vals xs) ⟹
set (dfs alist (insert a v xs)) = set (dfs alist xs) ∪ {(v,a)}
by (simp add: insert-def alist-add-Cons)

lemma insert-push:
distinct (vals xs) ⟹
v ∉ set (vals xs) ⟹
set (dfs alist (insert a v xs)) = set (PQ.alist-of (PQ.push v a (pqueue xs)))
by (simp add: alist-pqueue vals-pqueue set-insort)

lemma insert-p-push:
  assumes distinct (vals xs)
  and v ∉ set (vals xs)
  shows set (prios (insert a v xs)) = set (PQ.push v a (pqueue xs))
proof –
  from assms
  have set (dfs alist (insert a v xs)) =
    set (PQ.alist-of (PQ.push v a (pqueue xs)))
  by (rule insert-push)
  thus ?thesis by (simp add: alist-split-set priorities-set)
qed

lemma empty-empty:
normalized xs ⟹ xs = empty ⟷ PQ.is-empty (pqueue xs)
proof (rule iffI)
  assume xs = [] then show PQ.is-empty (pqueue xs) by simp
next
  assume N: normalized xs and E: PQ.is-empty (pqueue xs)
  show xs = []
proof (rule ccontr)
assume \( xs \neq [] \)
with \( N \) have \( \text{set} (\text{vals} \ xs) \neq \{\} \)
by (induct \( xs \)) (simp-all add: bt-dfs-simp dfs-append)
hence \( \text{set} (|pqueue \ xs|) \neq \{\} \) by (simp add: vals-pqueue)
mOREOVER
from \( E \) have \( \text{set} \ |pqueue \ xs| = \{\} \) by (simp add: is-empty-empty)
ultimately show \( \text{False} \) by simp
qed

lemma bt-dfs-Min-priority:
assumes is-heap \( t \)
shows \( \text{priority} \ t = \text{Min} (\text{set} (\text{bt-dfs priority} \ t)) \)
using assms
proof (induct priority \( t \) children \( t \) arbitrary: \( t \))
  case is-heap-list-Nil then show \( \text{?case} \) by (simp add: bt-dfs-simp)
next
  case (is-heap-list-Cons \( rs \) \( r \) \( t \))
  note cons = this
  let \( ?M = \text{Min} (\text{set} (\text{bt-dfs priority} \ t)) \)
  obtain \( t' \) where \( t' = \text{Node} (\text{priority} \ t) (\text{val} \ t) \) \( rs \) by auto
  hence \( \text{ot: \( rs = children \ t' \) \( \text{priority} \ t' = \text{priority} \ t \) by simp-all} \)
  with is-heap-list-Cons have \( \text{priority} \ t = \text{Min} (\text{set} (\text{bt-dfs priority} \ t')) \)
   by simp
  with \( \text{ot} \)
  have \( \text{priority} \ t = \text{Min} (\text{Set.insert} (\text{priority} \ t) (\text{set} (\text{bts-dfs priority} \ rs)))) \)
   by (simp add: bt-dfs-simp)
mOREOVER
from \( \text{cons} \) have \( \text{priority} \ r = \text{Min} (\text{set} (\text{bt-dfs priority} \ r)) \) by simp
mOREOVER
from \( \text{cons} \) have children \( t = r \neq \) \( rs \) by simp
then have \( \text{bts-dfs priority} \ (\text{children} \ t) = \)
  (\text{bt-dfs priority} \ r) @ (\text{bts-dfs priority} \ rs) by simp
hence \( \text{bt-dfs priority} \ t = \)
  \( \text{priority} \ t \neq (\text{bt-dfs priority} \ r @ \text{bts-dfs priority} \ rs) \)
  by (simp add: bt-dfs-simp)
hence \( A: \ ?M = \text{Min} \)
  (\text{Set.insert} (\text{priority} \ t) (\text{set} (\text{bt-dfs priority} \ r) \cup \text{set} (\text{bts-dfs priority} \ rs))))
  by simp
have \( \text{Set.insert} (\text{priority } t) (\text{set} (\text{bt-dfs priority } r)) \)
\( \cup \text{set} (\text{bt-dfs priority } rs)) = \text{Set.insert} (\text{priority } t) (\text{set} (\text{bt-dfs priority } rs)) \cup \text{set} (\text{bt-dfs priority } r) \)
by auto

with \( A \) have \( ?M = \text{Min} \)
\( (\text{Set.insert} (\text{priority } t) (\text{set} (\text{bt-dfs priority } rs)) \cup \text{set} (\text{bt-dfs priority } r)) \)
by simp

with \( \text{Min-Un} \)
\( [\text{of } \text{Set.insert} (\text{priority } t) (\text{set} (\text{bt-dfs priority } rs)) \cup \text{set} (\text{bt-dfs priority } r)] \)
have \( ?M = \text{ord-class.min} (\text{Min} (\text{set} (\text{bt-dfs priority } rs))) \cup \text{set} (\text{bt-dfs priority } r)) \)
by (auto simp add: \( \text{bt-dfs-simp} \))

ultimately
have \( ?M = \text{ord-class.min} (\text{priority } t) (\text{priority } r) \) by simp

with \( \text{priority } t \leq \text{priority } r \) show \( ?\text{case} \) by (auto simp add: \( \text{ord-class.min-def} \))
qed

lemma \( \text{is-binqueue-min-Min-prios} \):
assumes \( \text{is-binqueue } l \text{ xs} \)
and \( \text{normalized } xs \)
and \( xs \neq [] \)
shows \( \text{min } xs = \text{Some} (\text{Min} (\text{set} (\text{prios } xs))) \)
using \( \text{assms} \)
proof (induct \( xs \))
case \( \text{Some } l \text{ xs } x \) then show \( ?\text{case} \)
proof (cases \( xs \neq [] \))
case False with \( \text{Some} \) show \( ?\text{thesis} \)
using \( \text{bt-dfs-Min-priority}[\text{of } x] \) by (simp add: \( \text{min-single} \))
next
case True note \( T = \text{this } \text{Some} \)

from \( T \) have \( \text{normalized } xs \) by simp
with \( \langle xs \neq [] \rangle \) have \( \text{prios } xs \neq [] \) by (induct \( xs \)) (simp-all add: \( \text{bt-dfs-simp} \))
with \( T \) show \( ?\text{thesis} \)
using \( \text{Min-Un}[\text{of } \text{set} (\text{bt-dfs priority } x) \text{ set} (\text{prios } xs)] \)
using \( \text{bt-dfs-Min-priority}[\text{of } x] \)
by (auto simp add: \( \text{bt-dfs-simp} \text{ ord-class.min-def} \))
qed
qed simp-all

lemma \( \text{min-p-min} \):
assumes is-binqueue l xs
and xs ≠ []
and normalized xs
and distinct (vals xs)
and distinct (prios xs)
shows min xs = PQ.priority (pqueue xs) (PQ.min (pqueue xs))
proof –
from (xs ≠ []); (normalized xs) have ¬PQ.is-empty (pqueue xs)
by (simp add: empty-empty)
moreover
from assms have min xs = Some (Min (set (prios xs)))
by (simp add: is-binqueue-min-Min-prios)
with (distinct (vals xs); have min xs = Some (Min (set ||pqueue xs|| )))
by (simp add: prios-pqueue)
ultimately show ?thesis
by (simp add: priority-Min-priorities [where q = pqueue xs] )
qed

lemma find-min-p-min:
assumes is-binqueue l xs
and xs ≠ []
and normalized xs
and distinct (vals xs)
and distinct (prios xs)
shows priority (the (find-min xs)) =
the (PQ.priority (pqueue xs) (PQ.min (pqueue xs)))
proof –
from assms have min xs ≠ None by (simp add: normalized-min-not-None)
from assms have min xs = PQ.priority (pqueue xs) (PQ.min (pqueue xs))
by (simp add: min-p-min)
with (min xs ≠ None) show ?thesis by (auto simp add: min-eq-find-min-Some)
qed

lemma find-min-v-min:
assumes is-binqueue l xs
and xs ≠ []
and normalized xs
and distinct (vals xs)
and distinct (prios xs)
shows val (the (find-min xs)) = PQ.min (pqueue xs)
proof –
from assms have min xs ≠ None by (simp add: normalized-min-not-None)
then obtain a where oa: Some a = min xs by auto
then obtain \( t \) where \( \text{ot} : \text{find-min} \, \text{xs} = \text{Some} \, t \) priority \( t = a \) using \( \text{min-eq-find-min-Some} \, \text{[af} \, \text{xs} \, a] \) by auto

hence \(*\) : (val \( t, a \)) \( \in \) set (dfs alist \, \text{xs})
by (auto simp add: find-min-exist in-set-alist)

have \( \text{PQ.min} \, (pqueue \, \text{xs}) = \text{val} \, t \)
proof (rule ccontr)
assume \( \text{A}: \text{PQ.min} \, (pqueue \, \text{xs}) \neq \text{val} \, t \)
then obtain \( t' \) where \( \text{ot'}: \text{PQ.min} \, (pqueue \, \text{xs}) = \text{t'} \) by simp
with \( \text{A} \) have \( \text{NE}: \text{val} \, t \neq \text{t'} \) by simp

from \( \text{ot'} \, \text{oa} \) assms have \( (t', a) \in \text{set} \, (dfs \, \text{alist} \, \text{xs}) \)
by (simp add: alist-pqueue PQ.priority-def min-p-min)

with \( \text{NE} \) have \( \neg \) distinct (prios \, \text{xs})
unfolding alist-split (2)
unfolding dfs-comp
by (induct (dfs \, \text{alist} \, \text{xs})) (auto simp add: rev-image-eqI)

with \( \langle \text{distinct} \, (prios \, \text{xs}) \rangle \) show \( \text{False} \) by simp
qed

with \( \text{ot} \) show \( \text{thesis} \) by auto
qed

lemma \( \text{alist-normalize-idem} \):
\( \text{dfs} \, \text{alist} \, (\text{normalize} \, \text{xs}) = \text{dfs} \, \text{alist} \, \text{xs} \)
unfolding normalize-def
proof (induct \, \text{xs} \, \text{rule: rev-induct})
case (snoc \, x \, \text{xs}) then show \( \langle \text{case} \, \text{by} \, (\text{cases} \, x) \, \text{(simp-all add: dfs-append)} \rangle \)
qed simp

lemma \( \text{dfs-match-not-in} \):
(\( \forall \, t. \, \text{Some} \, t \in \text{set} \, \text{xs} \rightarrow \text{priority} \, t \neq a \) \( \implies \)
set (dfs \, f \, (\text{map} \, (\text{match} \, a) \, \text{xs})) = set (dfs \, f \, \text{xs})
by (induct \, \text{xs} \, \text{simp-all})

lemma \( \text{dfs-match-subset} \):
set (dfs \, f \, (\text{map} \, (\text{match} \, a) \, \text{xs})) \subseteq set (dfs \, f \, \text{xs})
proof (induct \, \text{xs} \, \text{rule: list-induct})
case (Cons \, x \, \text{xs}) then show \( \langle \text{case} \, \text{by} \, (\text{cases} \, x) \, \text{auto} \rangle \)
qed simp

lemma \( \text{dfs-match-distinct} \):
distinct (dfs \, f \, \text{xs}) \( \implies \) distinct (dfs \, f \, (\text{map} \, (\text{match} \, a) \, \text{xs}))
proof (induct \, \text{xs} \, \text{rule: list-induct})

37
case (Cons x xs) then show \(?\)case
  using dfs-match-subset[of a xs]
  by (cases x, auto)
qed simp

lemma dfs-match:
  distinct (prios xs) \implies
  distinct (dfs f xs) \implies
  Some t \in set xs \implies
  priority t = a \implies
  set (dfs f (map (match a) xs)) = set (dfs f xs) - set (bt-dfs f t)
proof (induct xs arbitrary: t)
case (Some r xs t) then show \(?\)case
  proof (cases t = r)
    case True
    from Some have priority r \notin set (prios xs) by (auto simp add: bt-dfs-simp)
    with Some True have a \notin set (prios xs) by simp
    hence \forall s. Some s \in set xs \implies priority s \neq a
      by (induct xs) (auto simp add: bt-dfs-simp)
    hence set (dfs f (map (match a) xs)) = set (dfs f xs)
      by (simp add: dfs-match-not-in)
    with True Some show \(?\)thesis by auto
  next
    case False
    with Some.prems have Some t \in set xs by simp
    with priority t = a have a \in set (prios xs)
    proof (induct xs)
      case (Some x xs) then show \(?\)case
        by (cases t = x) (simp-all add: bt-dfs-simp)
    qed simp-all
    with False Some have priority r \neq a by (auto simp add: bt-dfs-simp)
moreover
  from Some False
  have set (dfs f (map (match a) xs)) = set (dfs f xs) - set (bt-dfs f t)
    by simp
moreover
  from Some.prems False have set (bt-dfs f t) \cap set (bt-dfs f r) = {}
    by (induct xs) auto
  hence set (bt-dfs f r) - set (bt-dfs f t) = set (bt-dfs f r) by auto
ultimately show \(?\)thesis by auto
qed simp-all
lemma alist-meld:
  distinct (dfs val xs) \implies distinct (dfs val ys) \implies
  set (dfs val xs) \cap set (dfs val ys) = \{\} \implies
  set (dfs alist (meld xs ys)) = set (dfs alist xs) \cup set (dfs alist ys)

proof (induct xs ys rule: meld.induct)
  case (3 xs y ys)
  have set (dfs alist (y \# meld xs ys)) =
    set (dfs alist xs) \cup set (dfs alist (y \# ys))
  proof
    note assms = 3
    from assms have set (vals xs) \cap set (vals ys) = \{
      using set-dfs-Cons[of val y ys] by auto
    }
    moreover
    from assms have distinct (vals ys) by (cases y) simp-all
  moreover
  from assms have distinct (vals xs) by simp
  moreover note assms
  ultimately have set (dfs alist (meld xs ys)) =
    set (dfs alist xs) \cup set (dfs alist ys) by simp
  hence set (dfs alist (y \# meld xs ys)) =
    set (dfs alist (y)) \cup set (dfs alist xs) \cup set (dfs alist ys)
    using set-dfs-Cons[of alist y meld xs ys] by auto
  then show ?thesis using set-dfs-Cons[of alist y ys] by auto
  qed
  thus ?case by simp
next
  case (4 x xs ys)
  have set (dfs alist (x \# meld xs ys)) =
    set (dfs alist (x \# xs)) \cup set (dfs alist ys)
  proof
    note assms = 4
    from assms have set (vals xs) \cap set (vals ys) = \{
      using set-dfs-Cons[of val x xs] by auto
    }
    moreover
    from assms have distinct (vals xs) by (cases x) simp-all
  moreover
  from assms have distinct (vals ys) by simp
moreover note assms
ultimately have \( \text{set } (\text{dfs } \text{alist } (\text{meld } \text{xs } \text{ys})) = \text{set } (\text{dfs } \text{alist } \text{xs}) \cup \text{set } (\text{dfs } \text{alist } \text{ys}) \) by simp

hence \( \text{set } (\text{dfs } \text{alist } (x \# \text{meld } \text{xs } \text{ys})) = \text{set } (\text{dfs } \text{alist } [x]) \cup \text{set } (\text{dfs } \text{alist } \text{xs}) \cup \text{set } (\text{dfs } \text{alist } \text{ys}) \)

using \text{set-dfs-Cons[of } \text{alist } x \text{ meld } \text{xs } \text{ys}] \) by auto

then show ?thesis using \text{set-dfs-Cons[of } \text{alist } x \text{ xs}] \) by auto
qed
thus ?case by simp
next
case (5 x xs y ys)
have \( \text{set } (\text{dfs } \text{alist } (\text{add } (\text{Some } (\text{merge } x y)) (\text{meld } \text{xs } \text{ys}))) = \text{set } (\text{bt-dfs } \text{alist } x) \cup \text{set } (\text{dfs } \text{alist } \text{xs}) \cup \text{set } (\text{bt-dfs } \text{alist } y) \cup \text{set } (\text{dfs } \text{alist } \text{ys}) \)
proof –

note assms = 5

from assms have distinct (bt-dfs val x) distinct (bt-dfs val y) by simp-all
moreover from assms have xyint:
set (bt-dfs val x) \cap set (bt-dfs val y) = \{\} by (auto simp add: set-dfs-Cons)
ultimately have *: \( \text{set } (\text{dfs } \text{alist } (\text{Some } (\text{merge } x y))) = \text{set } (\text{bt-dfs } \text{alist } x) \cup \text{set } (\text{bt-dfs } \text{alist } y) \) by auto

moreover
from assms
have **: \( \text{set } (\text{dfs } \text{alist } (\text{meld } \text{xs } \text{ys})) = \text{set } (\text{dfs } \text{alist } \text{xs}) \cup \text{set } (\text{dfs } \text{alist } \text{ys}) \) by (auto simp add: set-dfs-Cons)

moreover
from assms have distinct (vals (\text{Some } (\text{merge } x y) \# \text{meld } \text{xs } \text{ys}))
proof –
from assms xyint have distinct (bt-dfs val (\text{merge } x y))
by (simp add: vals-merge-distinct)

moreover
from assms have
distinct (vals xs)
and distinct (vals ys)
and set (vals xs) \cap set (vals ys) = \{
by (auto simp add: set-dfs-Cons)
hence distinct (vals (meld xs ys)) by (rule vals-meld-distinct)
moreover
from assms
have set (bt-dfs val (merge x y)) \cap set (vals (meld xs ys)) = {}
by (auto simp add: vals-meld)

ultimately show ?thesis by simp
qed

ultimately show ?thesis by (auto simp add: alist-add-Cons)
qed
thus ?case by auto
qed simp-all

lemma alist-delete-min:
assumes distinct (vals xs)
and distinct (prios xs)
and find-min xs = Some (Node a v ts)
shows set (dfs alist (delete-min xs)) = set (dfs alist xs) - {{v, a}}
proof
from ⟨distinct (vals xs)⟩ have d: distinct (dfs alist xs)
using dfs-comp-distinct[of fst alist xs]
by (simp only: alist-split)
from assms have IN: Some (Node a v ts) \in set xs
by (simp add: find-min-exist)
hence sub: set (bts-dfs alist ts) \subseteq set (dfs alist xs)
by (induct xs) (auto simp add: bt-dfs-simp)
from d IN have (v,a) \notin set (bts-dfs alist ts)
using dfs-distinct-member[of alist xs Node a v ts] by simp
with sub have set (bts-dfs alist ts) \subseteq set (dfs alist xs) - {{v,a}} by blast
hence ra: set (bts-dfs alist ts) \cup (set (dfs alist xs) - {{v,a}}) =
set (dfs alist xs) - {{v,a}} by auto
from assms have distinct (vals (map (match a) xs))
by (simp add: dfs-match-distinct)
moreover
from IN assms have distinct (bts-dfs val ts)
using dfs-distinct-member[of val xs Node a v ts]
by (simp add: bt-dfs-distinct-children)
hence distinct (vals (map Some (rev ts)))
by (simp add: bts-dfs-rev-distinct dfs-map-Some-idem)
moreover
from assms IN have set (dfs val (map (match a) xs)) = 
  set (dfs val xs) - set (bt-dfs val (Node a v ts))
by (simp add: dfs-match)
hence set (vals (map (match a) xs)) ∩ set (vals (map Some (rev ts))) = {}
by (auto simp add: dfs-match 
ultimately have set (vals (map (match a) xs)) ∩ set (vals (map Some (rev ts))) = {}
by (simp add: dfs-map- Some-idem set-bts-dfs-rev)
ultimately have set (dfs alist (meld (map Some (rev ts)) (map (match a) xs))) = 
  set (dfs alist (map Some (rev ts))) ∪ set (dfs alist (map (match a) xs))
using alist-meld by auto
with assms d IN nu show ?thesis
proof -
  from assms obtain t where ot: find-min xs = Some t 
  using normalized-find-min-exists by auto
with assms show ?thesis 
proof (cases t)
case (Node a v ys)
from assms have ¬ PQ.is-empty (pqueue xs) by (simp add: empty-empty)
hence set (PQ.alist-of (PQ.remove-min (pqueue xs))) = 
  set (PQ.alist-of (pqueue xs)) - {{PQ.min (pqueue xs),
  the (PQ.priority (pqueue xs) (PQ.min (pqueue xs)))}}
by (simp add: set-alist-of-remove-min[of pqueue xs] del: alist-of-remove-min)
moreover
from assms ot Node 
have set (dfs alist (delete-min xs)) = set (dfs alist xs) - {(v, a)}
  using alist-delete-min[of xs] by simp
moreover
from Node ot have priority (the (find-min xs)) = a by simp
with assms have a = the (PQ.priority (pqueue xs) (PQ.min (pqueue xs)))
  by (simp add: find-min-p-min)
moreover
from Node at have val (the (find-min xs)) = v by simp
with assms have v = PQ.min (pqueue xs) by (simp add: find-min-v-min)

moreover note ⟨distinct (vals xs)⟩
ultimately show ?thesis by (simp add: alist-pqueue)
qed
qed

no-notation
PQ.values ([(-)])
and PQ.priorities (||(-)||)