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:
  distinct q ==> set (tl q) = set q - {hd q}
by (cases q) simp-all

1.2 Type of abstract priority queues

typedef (overloaded) ('a, 'b::linorder) pq = {xs :: ('a × 'b) list. distinct (map fst xs) ∧ sorted (map snd xs)}
morphisms alist-of Abs-pq
proof -
  have [] ∈ ?pq by simp
  then show ?thesis by blast
qed

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

lemma [code abstype]:
  Abs-pq (alist-of q) = 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} \emptyset = []
  by (simp add: empty-def Abs-pq-inverse)

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

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

lemma values-empty-nothing [simp]:
  \forall k. k \notin \text{set} \{|\emptyset|\}
  by (simp add: values-def)

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

lemma is-empty-empty-simp [simp]:
  is-empty \emptyset
  by (simp add: is-empty-empty)

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

lemma image-snd-alist-of:
  \text{the} \ \text{priority} q \ \text{set} (\text{values} q) = \text{snd} \ \text{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)
  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:
\[ q \neq [] \implies \text{sorted} \ (\text{map} \ \text{snd} \ q) \implies \text{Min} \ (\text{snd} \ \{q\}) = \text{snd} \ (\text{hd} \ q) \]

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

lemma hd-construt:
  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 alist-of q]
  by (auto simp add: priority-Min-priorities priorities-def)
  then show "?thesis" by (simp add: min-def)
qed simp

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]:
\[ \text{alist-of} \ (\text{push} \ k \ p \ q) = \begin{cases} 
\text{insort-key} \ \text{snd} \ (k, p) \ (\text{alist-of} \ q) & \text{if} \ k \notin \text{set} \ (\text{values} \ q) \\
\text{alist-of} \ q & \text{else} 
\end{cases} \]

using distinct-fst-alist-of [of q]
by (auto simp add: distinct-map set-insort-key distinct-insort not-in-first-image push-def values-def sorted-insort-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-key)

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-key)

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 \( a \neq b \ \text{and} \ v \neq w \)
  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 insort-key-left-comm)

5
definition remove-min :: ('a, 'b::linorder) pq ⇒ ('a, 'b::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, 'b::linorder) pq ⇒ ('a × ('a, 'b) 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) bintree = Node 'a 'b ('a, 'b) bintree list

primrec priority :: ('a, 'b) bintree ⇒ 'a where
priority (Node a - -) = a

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

primrec children :: ('a, 'b) bintree ⇒ ('a, 'b) bintree list where
  children (Node - - ts) = ts

type-synonym ('a, 'b) binqueue = ('a, 'b) bintree option list

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

Terminology:
- values v, w or v1, v2
- priorities a, b or a1, a2
- bintrees t, r or t1, t2
- bintree lists ts, rs or ts1, ts2
- binqueue element x, y or x1, x2
- binqueues = binqueue element lists xs, ys or xs1, xs2
- abstract priority queues q, p or q1, q2

2.2 Binomial queue properties

Binomial tree property

inductive is-bintree-list :: nat ⇒ ('a, 'b) bintree list ⇒ bool where
  is-bintree-list Nil [simp]: is-bintree-list 0 []
| is-bintree-list-Cons: is-bintree-list l ts ⇒ is-bintree-list l (children t)
  ⇒⇒ is-bintree-list (Suc l) (t # ts)

abbreviation (input) is-bintree k t ≡ is-bintree-list k (children t)
lemma is-bintree-list-triv [simp]:
  is-bintree-list 0 ts ↔ ts = []
  is-bintree-list l [] ↔ l = 0
  by (auto intro: is-bintree-list.intros elim: is-bintree-list.cases)

lemma is-bintree-list-simp [simp]:
  is-bintree-list (Suc l) (t ≠ ts) ↔
  is-bintree-list l (children t) ∧ is-bintree-list l ts
  by (auto intro: is-bintree-list.intros elim: is-bintree-list.cases)

lemma is-bintree-list-length [simp]:
  is-bintree-list l ts ⇒ length ts = l
  by (erule is-bintree-list.induct) simp-all

lemma is-bintree-list-children-last:
  assumes is-bintree-list l ts and ts ≠ []
  shows children (last ts) = []
  using assms by induct auto

lemma is-bintree-children-length-desc:
  assumes is-bintree-list l ts
  shows map (length ∘ children) ts = rev [0..<l]
  using assms by (induct ts) simp-all

Heap property

inductive is-heap-list :: 'a::linorder ⇒ ('a, 'b) bintree list ⇒ bool where
  is-heap-list-Nil: is-heap-list h []
| is-heap-list-Cons: is-heap-list h ts ⇒ is-heap-list (priority t) (children t)
  ⇒ (priority t) ≥ h ⇒ is-heap-list h (t ≠ ts)

abbreviation (input) is-heap t ≡ is-heap-list (priority t) (children t)

lemma is-heap-list-simps [simp]:
  is-heap-list h [] ↔ True
  is-heap-list h (t ≠ ts) ↔
  is-heap-list h ts ∧ is-heap-list (priority t) (children t) ∧ priority t ≥ h
  by (auto intro: is-heap-list.intros elim: is-heap-list.cases)

lemma is-heap-list-append-dest [dest]:
  is-heap-list l (ts @ rs) ⇒ is-heap-list l ts
  is-heap-list l (ts @ rs) ⇒ is-heap-list l rs
  by (induct ts) (auto intro: is-heap-list.intros elim: is-heap-list.cases)

lemma is-heap-list-rev:
is-heap-list l ts \Rightarrow is-heap-list l (\text{rev} ts) \\
\text{by} \ (\text{induct} ts \text{ rule: rev-induct}) \ \text{auto}

\text{lemma is-heap-children-larger:} \\
is-heap t \Rightarrow \forall x \in \text{set} \ (\text{children} \ t), \ \text{priority} \ x \geq \text{priority} \ t \\
\text{by} \ (\text{erule} \ \text{is-heap-list.induct}) \ \text{simp-all}

\text{lemma is-heap-Min-children-larger:} \\
is-heap t \Rightarrow \text{children} \ t \neq [] \Rightarrow \text{priority} \ t \leq \text{Min} \ (\text{priority} \ \text{set} \ (\text{children} \ t)) \\
\text{by} \ (\text{simp add: is-heap-children-larger})

\text{Combination of both: binqueue property}

\text{inductive is-binqueue :: nat} \Rightarrow ('a::linorder, 'b) \text{ binqueue} \Rightarrow \text{bool} \ \text{where} \\
\text{Empty:} \ is-binqueue \ l [] \\
| \text{None:} \ is-binqueue \ (\text{Suc} \ l) \ xs \Rightarrow \ is-binqueue \ l \ (\text{None} \ # \ xs) \\
| \text{Some:} \ is-binqueue \ (\text{Suc} \ l) \ xs \Rightarrow \ is-bintree \ l \ t \Rightarrow \ is-heap \ t \Rightarrow \ is-binqueue \ l \ (\text{Some} \ t \ # \ xs)

\text{lemma is-binqueue-simp [simp]:} \\
is-binqueue \ l [] \longleftrightarrow \text{True} \\
is-binqueue \ l \ (\text{Some} \ t \ # \ xs) \longleftrightarrow \is-binqueue \ (\text{Suc} \ l) \ xs \\
is-binqueue \ l \ (\text{None} \ # \ xs) \longleftrightarrow \is-binqueue \ (\text{Suc} \ l) \ xs \\
\text{by} \ (\text{auto intro: is-binqueue.intros elim: is-binqueue.cases})

\text{lemma is-binqueue-trans:} \\
is-binqueue \ l \ (x\#xs) \Rightarrow \ is-binqueue \ (\text{Suc} \ l) \ xs \\
\text{by} \ (\text{cases} \ x) \ \text{simp-all}

\text{lemma is-binqueue-head:} \\
is-binqueue \ l \ (x\#xs) \Rightarrow \ is-binqueue \ l \ [x] \\
\text{by} \ (\text{cases} \ x) \ \text{simp-all}

\text{lemma is-binqueue-append:} \\
is-binqueue \ l \ xs \Rightarrow \ is-binqueue \ \text{(length} \ xs + l) \ ys \Rightarrow \ is-binqueue \ l \ (xs @ ys) \\
\text{by} \ (\text{induct} \ xs \ \text{arbitrary:} \ l) \ (\text{auto intro: is-binqueue.intros elim: is-binqueue.cases})

\text{lemma is-binqueue-append-dest [dest]:} \\
is-binqueue \ l \ (xs @ ys) \Rightarrow \ is-binqueue \ l \ xs \\
\text{by} \ (\text{induct} \ xs \ \text{arbitrary:} \ l) \ (\text{auto intro: is-binqueue.intros elim: is-binqueue.cases})

\text{lemma is-binqueue-children:} \\
\text{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 ⟷ Some t ∈ set xs ⟷ ∃ k. is-bintree k t ∧ is-heap t
by (induct xs arbitrary: l) (auto intro: is-binqueue.intros elim: is-binqueue.cases)

Normalized representation

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

lemma normalized-last-not-None:
  — sometimes the inductive definition might work better
normalized xs ⟷ xs = [] ∨ last xs ≠ None
proof
  assume normalized xs
  then show xs = [] ∨ last xs ≠ None
    by (rule normalized.induct) simp-all
next
  assume *: xs = [] ∨ last xs ≠ None
  show normalized xs proof (cases xs rule: rev-cases)
    case Nil then show ?thesis by (simp add: normalized.intros)
  next
    case (snoc ys x) with * obtain t where last xs = Some t by auto
    with snoc have xs = ys @ [Some t] by simp
    then show ?thesis by (simp add: normalized.intros)
  qed
qed

lemma normalized-simps [simp]:
  normalized [] ⟷ True
  normalized (Some t # xs) ⟷ normalized xs
  normalized (None # xs) ⟷ xs ≠ [] ∧ 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) ⟷ normalized xs

by (auto simp add: normalized-last-not-None)

lemma normalized-append:
  normalized xs → normalized ys → normalized (xs@ys)
by (cases ys) (simp-all add: normalized-last-not-None)

lemma normalized-not-None:
  normalized xs → set xs ≠ {None}
by (induct xs) (auto simp add: normalized-Cons [of - ts] dest: subset-singletonD)

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

definition normalize :: ('a, 'b) binqueue ⇒ ('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 ⇒ 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 ⇒ ('a, 'b) bintree ⇒ ('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
  add :: ('a::linorder, 'b) bintree option ⇒ ('a, 'b) binqueue ⇒ ('a, 'b) binqueue
where
  add None xs = xs
| add (Some t) [] = [Some t]
| add (Some t) (None # xs) = Some t # xs
| add (Some t) (Some r # xs) = None # add (Some (merge t r)) xs

lemma add-Some-not-Nil [simp]:
  add (Some t) xs ≠ []
  by (induct Some t xs rule: add.induct) simp-all

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

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

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

function
  meld :: ('a::linorder, 'b) binqueue ⇒ ('a, 'b) binqueue ⇒ ('a, 'b) binqueue
where
  meld [] ys = ys
| meld xs [] = xs
| meld (None # xs) (y # ys) = y # meld xs ys
| meld (x # xs) (None # ys) = x # meld xs ys
| meld (Some t # xs) (Some r # ys) =
  None # add (Some (merge t r)) (meld xs ys)
  by pat-completeness auto termination by lexicographic-order
lemma meld-singleton-add [simp]:
meld [Some t] xs = add (Some t) xs
by (induct Some t xs rule: add.induct) simp-all

lemma nonempty-meld [simp]:
xs ≠ [] ⇒ meld xs ys ≠ []
ys ≠ [] ⇒ meld xs ys ≠ []
by (induct xs ys rule: meld.induct) auto

lemma nonempty-meld-commute:
meld xs ys ≠ [] ⇒ meld xs ys ≠ []
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) bintree
fix y :: ('a', 'b) bintree
fix l :: nat
assume ⋀ l. is-binqueue l xs ⇒ is-binqueue l ys
⇒ is-binqueue l (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) bintree
fix x :: ('a', 'b) bintree
fix l :: nat
assume ⋀ l. is-binqueue l xs ⇒ is-binqueue l ys
⇒ is-binqueue l (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)
lemma normalized-meld-weak:
  assumes normalized xs
  and length ys ≤ length xs
  shows normalized (meld xs ys)
using assms
proof (induct xs ys rule: meld.induct)
  fix xs ys :: (′a, ′b) bintree option
  fix y :: (′a, ′b) bintree option
  assume normalized xs ⇒ length ys ≤ length xs ⇒ normalized (meld xs ys)
  and normalized (x # xs)
  and length (y # ys) ≤ length (None # xs)
  then show normalized (meld (None # xs) (y # ys)) by (cases y) simp-all
next
  fix xs ys :: (′a, ′b) bintree option
  fix x :: (′a, ′b) bintree option
  assume normalized xs ⇒ length ys ≤ length xs ⇒ normalized (meld xs ys)
  and normalized (x # xs)
  and length (None # ys) ≤ length (x # xs)
  then show normalized (meld (x # xs) (None # ys)) by (cases x) simp-all
qed (simp-all add: normalized-add)

definition least :: ′a::linorder option ⇒ ′a option ⇒ ′a option where
  least x y = (case x of
    None ⇒ y
| Some x' ⇒ (case y of
      None ⇒ x
| Some y' ⇒ if x' ≤ y' then x else y))

lemma least-simps [simp, code]:
  least None x = x
least x None = x
least (Some x') (Some y') = (if x' ≤ y' then Some x' else Some y')

unfolding least-def by (simp-all) (cases x, simp-all)

lemma least-split:
  assumes least x y = Some z
  shows x = Some z ∨ 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)

definition min :: ('a::linorder, 'b) binqueue ⇒ '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: least fold-commute-apply [symmetric]
  fun-eq-iff least.left-commute del: least-simps)

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

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

lemma min-Some-not-None:
  min (Some t # xs) ≠ 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:
  \(\min xs = \text{None} \iff xs = [] \lor \text{set} \, xs = \{\text{None}\}\)
proof (rule iffI)
  have splitQ: \(\forall \, xs. \, xs \subseteq \{\text{None}\} \implies xs = [] \lor \text{set} \, xs = \{\text{None}\}\) by auto
  assume \(\min xs = \text{None}\)
  then have \(\text{set} \, xs \subseteq \{\text{None}\}\)
  proof (induct \(xs\))
    case (None \(ys\)) thus \(?\)case using min-None-trans[of - \(ys\)] by simp
  next
    case (Some \(t\) \(ys\)) thus \(?\)case using min-Some-not-None[of \(t\) \(ys\)] by simp
  qed simp
  with splitQ show \(xs = [] \lor \text{set} \, xs = \{\text{None}\}\) by auto
next
  show \(xs = [] \lor \text{set} \, xs = \{\text{None}\} \implies \min xs = \text{None}\)
  by (induct \(xs\)) (auto dest: subset-singletonD)
qed

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

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

**Assumes** `min xs = Some a`  
**Shows** `Some a ∈ map-option priority ' set xs`

**Proof** (rule `contr`)  
**Assume** `Some a ∉ map-option priority ' set xs`  
**Then have** `∀ x ∈ set xs. x = None ∨ priority (the x) ≠ a` by (induct `xs`) auto

**Proof** (induct `xs` arbitrary: `a`)  
**Case** `(Some t ys)`  
**Hence** `priority t ≠ a` and `min ys ≠ Some a` by (induct `xs`)

**Show** ?case  
**Proof** (rule `contr`, `simp`)

**Assume** `least (Some (priority t)) (min ys) = Some a`  
**Hence** `Some (priority t) = Some a ∨ min ys = Some a` by (rule `least-split`)

**With** `min ys ≠ Some a` **have** `priority t = a` by `simp`

**With** `priority t ≠ a` **show** `False` by `simp`

**Qed**

**Qed**

With `assms` **show** `False` by `simp`

**Qed**

Primrec `find :: ('a::linorder ⇒ 'b) binqueue ⇒ ('a,'b) bintree option` where

- `find a [] = None`
- `find a (x#xs) = (case x of None ⇒ find a xs |
  Some t ⇒ 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 ∈ set (map (map-option priority) xs)`  
**Shows** `∃ t. find a xs = Some t ∧ priority t = a`

**Using** `assms` **by** (induct `xs`) auto

Lemma `find-works-not-None`:

- `Some a ∈ set (map (map-option priority) xs) → find a xs ≠ None`  
  **By** (drule `find-works`) auto

Lemma `find-None`:

- `find a xs = None → Some a ∉ set (map (map-option priority) xs)`
by (auto simp add: find-works-not-None)

lemma find-exist:
find a xs = Some t ==> Some t ∈ set xs
by (induct xs) (simp-all add: eq-commute)

definition find-min :: ('a::linorder, 'b) binqueue ⇒ ('a, 'b) bin queue option where
find-min xs = (case min xs of None ⇒ None | Some a ⇒ find a xs)

lemma find-min-simps [simp]:
find-min [] = None
find-min (None # xs) = find-min xs
by (auto simp add: find-min-def split: option.split)

lemma find-min-single:
find-min [x] = x
by (cases x) (auto simp add: find-min-def)

lemma min-eq-find-min-None:
min xs = None =⇒ find-min xs = None
proof (rule iffI)
show min xs = None =⇒ find-min xs = None
proof (simp add: find-min-def)
next
assume *: find-min xs = None
show min xs = None
proof (rule ccontr)
assume min xs ≠ None
then obtain a where min xs = Some a by auto
hence find-min xs ≠ None
by (simp add: find-min-def min-exists find-works-not-None)
with * show False by simp
qed

lemma min-eq-find-min-Some:
min xs = Some a =⇒ (∃ t. find-min xs = Some t ∧ priority t = a)
proof (rule iffI)
show D1: ∀ a. min xs = Some a
implies (∃ t. find-min xs = Some t ∧ priority t = a)
by (simp add: find-min-def find-works min-exists)
assume *: ∃ t. find-min xs = Some t ∧ priority t = a
show min xs = Some a
proof (rule ccontr)
assume min xs ≠ Some a thus False
proof (cases min xs)
case None
hence find-min xs = None by (simp only: min-eq-find-min-None)
with * show False by simp
next
case (Some b)
with :min xs ≠ Some a: have a ≠ b by simp
with * Some 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 (auto 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) binqueue ⇒ ('a, 'b) binqueue 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]:
normalized xs ⇒ \(\text{find-min} \; \text{xs} = \text{Some} \; t\)
⇒ \(\text{delete-min} \; \text{xs} = \text{normalize} \; (\text{meld} \; (\text{map} \; \text{Some} \; (\text{rev} \; (\text{children} \; t))) \; (\text{map} \; (\text{match} \; (\text{priority} \; t)) \; \text{xs}))\)

unfolding delete-min-def by (cases t) simp

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

lemma normalized-delete-min:
normalized (delete-min xs)
by (cases find-min xs)
(auto simp add: delete-min-def normalized-normalize split: bintree.split)

Dedicated grand unified operation for generated program

definition
meld' :: ('a, 'b) bintree option ⇒ ('a::linorder, 'b) bqueue
⇒ ('a, 'b) bqueue ⇒ ('a, 'b) bqueue
where
meld' z xs ys = add z (meld xs ys)

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

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

\text{Interface operations}

\text{abbreviation (input) empty :: ('a,'b) binqueue where}
empty \equiv []

\text{definition}
in\text{sert} :: 'a::linorder \Rightarrow 'b \Rightarrow ('a, 'b) binqueue \Rightarrow ('a, 'b) binqueue
\text{where}
in\text{sert a v xs} = \text{add (Some (Node a v [])) xs}

\text{lemma insert-simps [simp]:}
in\text{sert a v []} = \text{[Some (Node a v [])]}
in\text{sert a v (\text{None } \# \text{ xs})} = \text{Some (Node a v [])} \# \text{ xs}
in\text{sert a v (\text{Some t } \# \text{ xs})} = \text{None } \# \text{add (Some (merge (Node a v [])) t)} \text{ xs}
\text{by (simp-all add: insert-def)}

\text{lemma is-binqueue-insert:}
is\text-binqueue 0 \text{ xs} \Rightarrow is\text-binqueue 0 (\text{insert a v xs})
\text{by (simp add: is-binqueue-add-Some insert-def)}

\text{lemma normalized-insert:}
normalized \text{ xs} \Rightarrow normalized (\text{insert a v xs})
\text{by (simp add: normalized-add insert-def)}

\text{definition}
\text{pop} :: ('a::linorder, 'b) binqueue \Rightarrow (('b } \times 'a) option \times ('a, 'b) binqueue)
where
\[
\begin{align*}
\text{pop } \text{xs} &= \begin{cases} \\
  \text{None} \Rightarrow (\text{None}, \text{xs}) \\
  \text{Some } t \Rightarrow (\text{Some (val } t, \text{ priority } t), \text{ delete-min } \text{xs})
\end{cases}
\end{align*}
\]

lemma \text{pop-empty [simp]}:
\[
\begin{align*}
\text{pop empty} &= (\text{None, empty}) \\
\text{by (simp add: pop-def empty-def)}
\end{align*}
\]

lemma \text{pop-nonempty [simp]}:
\[
\begin{align*}
\text{normalized } \text{xs} \Rightarrow \text{xs} \neq [] \Rightarrow \text{find-min } \text{xs} &= \text{Some } t \\
\Rightarrow \text{pop } \text{xs} &= (\text{Some (val } t, \text{ priority } t), \text{ normalize} \\
& (\text{meld (map Some (rev (children } t))) (\text{map (match (priority } t))) \text{ xs})))
\end{align*}
\]

by (simp add: pop-def)

lemma \text{pop-code [code]}:
\[
\begin{align*}
\text{pop } \text{xs} &= \begin{cases} \\
  \text{None} \Rightarrow (\text{None, xs}) \\
  \text{Some } t \Rightarrow (\text{Some (val } t, \text{ priority } t), \text{ normalize} \\
& (\text{meld (map Some (rev (children } t))) (\text{map (match (priority } t))) \text{ xs})))
\end{cases}
\end{align*}
\]

by (cases \text{find-min } \text{xs}) (simp-all add: pop-def delete-min-def split: bintree.split)

3 Relating Functional Binomial Queues To The Abstract Priority Queues

notation
\[
\begin{align*}
PQ.\text{values} &= \text{\texttt{|\texttt{(-)|}}} \\
\text{and } PQ.\text{priorities} &= \text{\texttt{\texttt{\texttt{\|(-)||}}}}
\end{align*}
\]

Naming convention: prefix \text{bt-} for bintrees, \text{bts-} for bintree lists, no prefix for binqueues.

primrec \text{bt-dfs :: (\texttt{('a::linorder, 'b) bintree => 'c) => (\texttt{('a, 'b) bintree => 'c list})}}
\text{ and } \text{bts-dfs :: (\texttt{('a::linorder, 'b) bintree => 'c list) => (\texttt{('a, 'b) bintree list => 'c list)}}

where
\[
\begin{align*}
\text{bt-dfs } f \ (\text{Node } a \ v \ ts) &= f \ (\text{Node } a \ v \ ts) \# \text{ bts-dfs } f \ ts \\
\text{bts-dfs } f \ [] &= [] \\
\text{bts-dfs } f \ (t \ # \ ts) &= \text{bt-dfs } f \ t \ @ \text{ bts-dfs } f \ ts
\end{align*}
\]

lemma \text{bt-dfs-simp}:
\[
\begin{align*}
\text{bt-dfs } f \ t &= f \ t \ # \text{ bts-dfs } f \ (\text{children } t) \\
\text{by (cases } t\text{) simp-all}
\end{align*}
\]

lemma \text{bts-dfs-append [simp]}:
lemma set-bts-dfs-rev:
set (bts-dfs f (rev ts)) = set (bts-dfs f ts)
by (induct ts) simp-all

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 (bt-dfs (f \circ g) t) \implies distinct (bt-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 (bt-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:
  \( \text{dfs} (f \circ g) \, \text{xs} = \text{map} f (\text{dfs} \, g \, \text{xs}) \)
  by (induct \text{xs}) (simp-all add: bl-dfs-comp del: o-apply)

lemma dfs-comp-distinct:
  distinct (\text{dfs} (f \circ g) \, \text{xs}) \implies \text{distinct} (\text{dfs} \, g \, \text{xs})
  by (simp add: dfs-comp distinct-map[of \, f])

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

lemma dfs-map-Some-idem:
  \( \text{dfs} \, f \, (\text{map} \, \text{Some} \, \text{xs}) = \text{bts-dfs} \, f \, \text{xs} \)
  by (induct \text{xs}) simp-all

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

lemma alist-split-pre:
  \( \text{val} \, t = (\text{fst} \circ \text{alist}) \, t \)
  \( \text{priority} \, t = (\text{snd} \circ \text{alist}) \, t \)
  by (cases \( t \), simp)+

lemma alist-split:
  \( \text{val} = \text{fst} \circ \text{alist} \)
  \( \text{priority} = \text{snd} \circ \text{alist} \)
  by (auto intro!: ext simp add: alist-split-pre)

lemma alist-split-set:
  \( \text{set} \, (\text{dfs} \, \text{val} \, \text{xs}) = \text{fst} \cdot \, \text{set} \, (\text{dfs} \, \text{alist} \, \text{xs}) \)
  \( \text{set} \, (\text{dfs} \, \text{priority} \, \text{xs}) = \text{snd} \cdot \, \text{set} \, (\text{dfs} \, \text{alist} \, \text{xs}) \)
  by (auto simp add: dfs-comp alist-split)

lemma in-set-in-alist:
  assumes \( \text{Some} \, t \in \text{set} \, \text{xs} \)
  shows \( (\text{val} \, t, \text{priority} \, t) \in \text{set} \, (\text{dfs} \, \text{alist} \, \text{xs}) \)
  using assms
  proof (induct \text{xs})
    case (\text{Some} \, x \, \text{xs})
    then show \(?\) by

proof (cases Some t ∈ set xs)
  case False with Some show thesis by (cases t) (auto simp add: bt-dfs-simp)
qed simp
qed simp-all

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 ⇒ ('a, 'b) PQ.pq
  and
  bts-augment :: ('a::linorder, 'b) bintree list ⇒ ('a, 'b) PQ.pq ⇒ ('a, 'b) 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: bt-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 $r$ rule: $bt$-augment.induct $bts$-augment.induct)
auto

lemma $bt$-augment-v-in:
$v \in \text{set } |q| \Rightarrow v \in \text{set } |bt$-augment $t$ $q|
$v \in \text{set } |q| \Rightarrow v \in \text{set } |bts$-augment $ts$ $q|
using$ $bt$-augment-v-subset[of $q$] by auto

lemma $bt$-augment-v-union:
$\text{set } |bt$-augment $t$ ($bt$-augment $r$ $q)| =
\text{set } |bt$-augment $t$ $q| \cup \text{set } |bt$-augment $r$ $q|
$\text{set } |bts$-augment $ts$ ($bt$-augment $r$ $q)| =
\text{set } |bts$-augment $ts$ $q| \cup \text{set } |bt$-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-v-subset[of $q$] show ?case by auto
qed auto

lemma $bt$-val-augment:
shows $\text{set } (bt$dfs $val$ $t) \cup \text{set } |q| = \text{set } |bt$-augment $t$ $q|
and $\text{set } (bts$dfs $val$ $ts) \cup \text{set } |q| = \text{set } |bts$-augment $ts$ $q|
proof (induct $t$ and $ts$ rule: $bt$-augment.induct $bts$-augment.induct)
case ($Cons$-bintree $r$ $rs$)
have $\text{set } |bts$-augment $rs$ ($bt$-augment $r$ $q)| =
\text{set } |bts$-augment $rs$ $q| \cup \text{set } |bt$-augment $r$ $q|
by (simp only: $bt$-augment-v-union)
with $bt$-augment-v-subset[of $q$]
have $\text{set } |bts$-augment $rs$ ($bt$-augment $r$ $q)| =
\text{set } |bts$-augment $rs$ $q| \cup \text{set } |bt$-augment $r$ $q| \cup \text{set } |q|
by auto
with $Cons$-bintree show ?case by auto
qed auto

lemma vals-pqueue:
$\text{set } (vals$ $xs) = \text{set } |pqueue$ $xs|
by (induct $xs$) (simp-all add: $bt$-val-augment)

lemma $bt$-augment-v-push:
$\text{set } |bt$-augment $t$ ($PQ$.push $v$ $a$ $q)| = \text{set } |bt$-augment $t$ $q| \cup \{v\}
$\text{set } |bts$-augment $ts$ ($PQ$.push $v$ $a$ $q)| = \text{set } |bts$-augment $ts$ $q| \cup \{v\}
using $bt$-val-augment(where $q = PQ$.push $v$ $a$ $q$) by (simp-all add: $bt$-val-augment)

lemma $bt$-augment-v-push-commute:
$\text{set } |bt$-augment $t$ ($PQ$.push $v$ $a$ $q)| = \text{set } |PQ$.push $v$ $a$ ($bt$-augment $t$ $q)|
lemma bts-augment-v-push:
set | bts-augment ts (PQ.push v a q)| = set | PQ.push v a (bts-augment ts q)|
by (simp-all add: bts-augment-v-push del: bts-augment)

lemma bts-augment-v-union:
set | bts-augment ts (PQ.push v a q)| =
set | bts-augment ts q| ∪ set | bts-augment rs q|
set | bts-augment ts (bts-augment rs q)| =
set | bts-augment ts q| ∪ set | bts-augment rs q|
proof (induct t and ts arbitrary: q rs and q rs rule: bts-augment.induct bts-augment.induct)
case Nil-bintree
from bts-augment-v-subset[of q] show ?case by auto
next
case (Cons-bintree x xs)
let ?L = set | bts-augment xs (bts-augment x (bts-augment rs q))|
from bts-augment-v-union
have *: ∀ q. set | bts-augment xs (bts-augment x q)| =
set | bts-augment xs q| ∪ set | bts-augment x q| by simp
with Cons-bintree
have ?L =
set | bts-augment xs q| ∪ set | bts-augment rs q| ∪ set | bts-augment x q|
by auto
with * show ?case by auto
qed simp

lemma bts-augment-v-commute:
set | bts-augment t (bts-augment r q)| = set | bts-augment r (bts-augment t q)|
set | bts-augment t (bts-augment rs q)| = set | bts-augment rs (bts-augment t q)|
set | bts-augment ts (bts-augment rs q)| =
set | bts-augment rs (bts-augment ts q)|
unfolding bts-augment-v-union bts-augment-v-union by auto

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

lemma vals-merge [simp]:
set (bt-dfs val (merge t r)) = set (bt-dfs val t) ∪ set (bt-dfs val r)
by (auto simp add: bt-dfs-simp merge-def)

lemma vals-merge-distinct:
distinct (bt-dfs val t) ⇒ distinct (bt-dfs val r) ⇒
\begin{align*}
\text{set} \ (\text{bt-dfs val} \ t) \cap \text{set} \ (\text{bt-dfs val} \ r) &= \{\} \\
\Rightarrow \text{distinct} \ (\text{bt-dfs val} \ (\text{merge} \ t \ r)) \\
\text{by} \ (\text{auto simp add: bt-dfs-simp merge-def})
\end{align*}

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

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

\textbf{lemma vals-add-distinct:} \\
\text{assumes} \ \text{distinct} \ (\text{vals} \ xs) \\
\text{and} \ \text{distinct} \ (\text{dfs val} \ [x]) \\
\text{and} \ \text{set} \ (\text{vals} \ xs) \cap \text{set} \ (\text{dfs val} \ [x]) = \{\} \\
\text{shows} \ \text{distinct} \ (\text{vals} \ (\text{add} \ x \ xs))

\text{using} \ \text{assms}

\textbf{proof (cases x)} \\
\text{case} (\text{Some} \ t) \text{ with} \ \text{assms show} \ ?\text{thesis}

\text{proof (induct} \ xs \ \text{arbitrary:} \ x \ t) \\
\text{case} (\text{Some} \ r \ xs) \\
\text{then have} \ \text{set} \ (\text{bt-dfs val} \ t) \cap \text{set} \ (\text{bt-dfs val} \ r) = \{\} \text{ by auto}

\text{with} \ \text{Some have} \ \text{distinct} \ (\text{bt-dfs val} \ (\text{merge} \ t \ r)) \text{ by (simp add: vals-merge-distinct)}

\text{moreover}
\text{with} \ \text{Some have} \ \text{set} \ (\text{vals} \ xs) \cap \text{set} \ (\text{bt-dfs val} \ (\text{merge} \ t \ r)) = \{\} \text{ by auto}

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

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

\text{by (simp add: insert-def vals-add-Cons)}

\textbf{lemma insert-v-push:} \\
\text{set} \ (\text{vals} \ (\text{insert} \ a \ v \ xs)) = \text{set} \ |\text{PQ.push} \ a \ (\text{pqueue} \ xs)|

\text{by (simp add: vals-pqueue[ symmetric])}

\textbf{lemma vals-meld:} \\
\text{set} \ (\text{dfs val} \ (\text{meld} \ xs \ ys)) = \text{set} \ (\text{dfs val} \ xs) \cup \text{set} \ (\text{dfs val} \ ys)

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

lemma vals-meld-distinct:
  distinct (dfs val xs) ⟷ distinct (dfs val ys) ⟷
  set (dfs val xs) ∩ set (dfs val ys) = {}
  distinct (dfs val (meld xs ys))
proof (induct xs y rule: meld.induct)
  case (3 x 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) ∩ set (vals ys) = {}
      using set-dfs-Cons[of val y ys] by auto
    moreover
    from Some 3 have set (bt-dfs val t) ∩ set (vals xs) = {}
      by auto
    moreover
    from Some 3 have set (bt-dfs val t) ∩ set (vals ys) = {}
      by simp
    ultimately have set (bt-dfs val t) ∩ set (vals (meld xs ys)) = {}
      by (auto simp add: vals-meld)
    with 3 Some show ?thesis by auto
  qed
next
  case (4 x xs 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) ∩ set (vals ys) = {}
      using set-dfs-Cons[of val x xs] by auto
    moreover
    from Some 4 have set (bt-dfs val t) ∩ set (vals xs) = {}
      by simp
    moreover
    from Some 4 have set (bt-dfs val t) ∩ set (vals ys) = {}
      by auto
    ultimately have set (bt-dfs val t) ∩ set (vals (meld xs ys)) = {}
  qed
by (auto simp add: vals-meld)
with 4 Some show ?thesis by auto
qed

next

case (5 x xs y ys) then
have set (vals xs) ∩ set (vals ys) = {} by (auto simp add: set-dfs-Cons)
with 5 have distinct (vals (meld xs ys)) by simp

moreover
from 5 have set (bt-dfs val x) ∩ set (bt-dfs val y) = {} by auto
with 5 have distinct (bt-dfs val (merge x y))
  by (simp add: vals-merge-distinct)

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

ultimately show ?case by (simp add: vals-add-distinct)
qed simp-all

lemma bt-augment-alist-subset:
set (PQ.alist-of q) ⊆ set (PQ.alist-of (bt-augment t q))
set (PQ.alist-of q) ⊆ set (PQ.alist-of (bts-augment ts q))

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

lemma bt-augment-alist-in:
(v,a) ∈ set (PQ.alist-of q) ⇒ (v,a) ∈ set (PQ.alist-of (bt-augment t q))
(v,a) ∈ set (PQ.alist-of q) ⇒ (v,a) ∈ set (PQ.alist-of (bts-augment ts q))
using bt-augment-alist-subset[of q] by auto

lemma bt-augment-alist-union:
distinct (bt-dfs val (r # [t])) ⇒
set (bt-dfs val (r # [t])) ∩ set [q] = {} ⇒
set (PQ.alist-of (bt-augment t (bt-augment r q))) =
  set (PQ.alist-of (bt-augment t q)) ∪ set (PQ.alist-of (bt-augment r q))

distinct (bt-dfs val (r # ts)) ⇒
set (bt-dfs val (r # ts)) ∩ set [q] = {} ⇒
set (PQ.alist-of (bt-augment ts (bt-augment r q))) =
  set (PQ.alist-of (bts-augment ts q)) ∪ set (PQ.alist-of (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 ?case by auto

next

  case (Node a v rs) then
  have
    set (PQ-alist-of (bts-augment rs (bt-augment r q))) =
    set (PQ-alist-of (bts-augment rs q)) ∪ set (PQ-alist-of (bt-augment r q))
    by simp

  moreover
  from Node.prems have v ∉ set |bts-augment rs q| ∪ set |bt-augment r q| by simp
  unfolding bt-val-augment[symmetric] by auto
  hence v ∉ set |bts-augment rs (bt-augment r q)| by (unfold bt-augment-v-union)

  moreover
  from * have v ∉ set |bts-augment rs q| by simp

  ultimately show ?case by (simp add: set-insort-key)

next

  case (Cons-bintree x xs) then
  have — FIXME: ugly... and slow
    distinct (bts-dfs val (x # xs)) and
    distinct (bts-dfs val (r # xs)) and
    distinct (bts-dfs val [r,x]) and
    set (bts-dfs val (x # xs)) ∩ set |bt-augment r q| = {} and
    set (bts-dfs val (x # xs)) ∩ set |q| = {} and
    set (bts-dfs val [r,x]) ∩ set |q| = {} and
    set (bts-dfs val (r # xs)) ∩ set |q| = {}
  unfolding bt-val-augment[symmetric] by auto
  with Cons-bintree.hyps show ?case by auto

qed

lemma bt-alist-augment:
  distinct (bt-dfs val t) ⇒
  set (bt-dfs val t) ∩ set |q| = {} ⇒
  set (bt-dfs alist t) ∪ set (PQ-alist-of q) = set (PQ-alist-of (bt-augment t q))

  distinct (bts-dfs val ts) ⇒
  set (bts-dfs val ts) ∩ set |q| = {} ⇒
  set (bts-dfs alist ts) ∪ set (PQ-alist-of q) =
  set (PQ-alist-of (bt-augment ts q))

proof (induct t and ts rule: bt-augment.induct bts-augment.induct)

  case Nil-bintree then show ?case by simp

next

  case (Node a v rs)
  hence v ∉ set |bts-augment rs q|
unfolding bt-val-augment[ symmetric] by simp
with Node show ?case by (simp add: set-insert-key)
next
case (Cons-bintree r rs) then
have set (PQ.alist-of (bts-augment (r # rs) q)) =
  set (PQ.alist-of (bts-augment rs q)) ∪ set (PQ.alist-of (bt-augment r q))
  using bt-augment-alist-union by simp
with Cons-bintree bt-augment-alist-subset show ?case by auto
qed

lemma alist-pqueue:
distinct (vals xs) ⇒ set (dfs alist xs) = set (PQ.alist-of (pqueue xs))
by (induct xs) (simp-all add: vals-pqueue bt-alist-augment)

lemma alist-pqueue-priority:
distinct (vals xs) ⇒ (v, a) ∈ set (dfs alist xs)
  ⇒ PQ.priority (pqueue xs) v = Some a
by (simp add: alist-pqueue PQ.priority-def)

lemma prios-pqueue:
distinct (vals xs) ⇒ set (prios xs) = set ||pqueue xs||
by (auto simp add: alist-pqueue priorities-set alist-split-set)

lemma alist-merge [simp]:
distinct (bt-dfs val t) ⇒ distinct (bt-dfs val r) ⇒
  set (bt-dfs val t) ∩ set (bt-dfs val r) = {} ⇒
  set (bt-dfs alist (merge t r)) = set (bt-dfs alist t) ∪ set (bt-dfs alist r)
by (auto simp add: bt-dfs-simp merge-def alist-split)

lemma alist-add-Cons:
assumes distinct (vals (x # xs))
shows set (dfs alist (add x xs)) = set (dfs alist (x # xs))
using assms proof (induct xs arbitrary: x)
case Empty then show ?case by (cases x) simp-all
next
case None then show ?case by (cases x) simp-all
next
case (Some y ys) then
show ?case
proof (cases x)
case (Some t)
  note prem = Some.prems Some

from prem have distinct (bt-dfs val (merge t y))
  by (auto simp add: bt-dfs-simp merge-def)
with prem have distinct (vals (Some (merge t y) ≠ ys)) by auto
with prem Some.hyps
  have set (dfs alist (add (Some (merge t y)) ys)) =
    set (dfs alist (Some (merge t y) ≠ ys)) by simp
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
qed

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-key)

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 \(set \; (vals \; xs) \neq \{\}\)
by (induct \(xs\)) (simp-all add: bt-dfs-simp dfs-append)

hence \(set \; |pqueue \; xs| \neq \{\}\) by (simp add: vals-pqueue)

moreover
from \(E\) have \(set \; |pqueue \; xs| = \{\}\) by (simp add: is-empty-empty)

ultimately show False by simp
qed
qed

lemma bt-dfs-Min-priority:
  assumes is-heap \(t\)
  shows priority \(t\) = Min \((set \; (bt-dfs \; priority \; t))\)
using assms
proof (induct priority \(t\) children \(t\) arbitrary: \(t\))
case is-heap-list-Nil then show \(?case\) by (simp add: bt-dfs-simp)
next
case (is-heap-list-Cons \(rs\) \(r\) \(t\)) note cons = this
let \(?M\) = Min \((set \; (bt-dfs \; priority \; t))\)

obtain \(t'\) where \(t' = Node \; (priority \; t) \; (val \; t) \; rs\) by auto
hence \(ot: rs = children \; t'\) priority \(t' = priority \; t\) by simp-all
with is-heap-list-Cons have priority \(t\) = Min \((set \; (bt-dfs \; priority \; t'))\)
  by simp
with \(ot\)
have priority \(t\) = Min \((Set.insert \; (priority \; t) \; (set \; (bts-dfs \; priority \; rs))))\)
  by (simp add: bt-dfs-simp)

moreover
from \(cons\) have priority \(r\) = Min \((set \; (bt-dfs \; priority \; r))\) by simp
moreover
from \(cons\) have children \(t = r \neq rs\) by simp
then have bts-dfs priority \((children \; t)\) =
  \((bt-dfs \; priority \; r) @ (bt-dfs \; priority \; rs)\) by simp
hence bt-dfs priority \(t\) =
  priority \(t \neq (bt-dfs \; priority \; r) @ (bt-dfs \; priority \; rs)\)
by (simp add: bt-dfs-simp)

**hence** \( A \): \( \forall M = \text{Min} (\text{Set.insert (priority } t) (\text{set (bt-dfs priority } r) \cup \text{set (bts-dfs priority } rs))) \)
by simp

**have** \( \text{Set.insert (priority } t) (\text{set (bt-dfs priority } r) \cup \text{set (bts-dfs priority } rs)) = \text{Set.insert (priority } t) (\text{set (bts-dfs priority } rs)) \cup \text{set (bt-dfs priority } r) \)
by auto

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

**with Min-Un**
[of \( \text{Set.insert (priority } t) (\text{set (bts-dfs priority } rs)) \cup \text{set (bt-dfs priority } r) \)]

**have** \( \forall M = \text{ord-class.min (priority } t) (\text{priority } r) \)
by (auto simp add: bt-dfs-simp)

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

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

**qed**

**lemma** is-binqueue-min-Min-prios:
**assumes** is-binqueue \( l \) \( xs \)
and \( \text{normalized } xs \)
and \( xs \neq [] \)
**shows** \( \text{min } xs = \text{Some (Min (set (prios } xs)))} \)
using assms

**proof** (induct \( xs \))
**case** (Some \( l \) \( xs \) \( x \)) **then show** \( \text{?case} \)
**proof** (cases \( xs \neq [] \))
**case** False with Some **show** \( \text{?thesis} \)
using bt-dfs-Min-priority[of \( x \)] **by** (simp add: min-single)
next
**case** True note \( T = \text{this Some} \)

from \( T \) **have** \( \text{normalized } xs \)
by simp

**with** \( xs \neq [] \) **have** \( \text{prios } xs \neq [] \)
by (induct \( xs \)) (simp-all add: bt-dfs-simp)

**with** \( T \) **show** \( \text{?thesis} \)
using Min-Un[of \( \text{set (bt-dfs priority } x) \) \( \text{set (prios } xs) \)]
using bt-dfs-Min-priority[of \( x \)]

35
by (auto simp add: bt-dfs-simp ord-class.min-def)
qed
qed simp-all

lemma 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 ot: find-min xs = Some t priority t = a
  using min-eq-find-min-Some [of xs a] by auto

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

have PQ.min (pqueue xs) = val t
proof (rule ccontr)
  assume A: PQ.min (pqueue xs) ≠ val t
  then obtain t’ where ot’:PQ.min (pqueue xs) = t’ by simp
  with A have NE: val t ≠ t’ by simp

from ot’ oa assms have (t’, a) ∈ set (dfs alist xs)
  by (simp add: alist-pqueue PQ.priority-def min-p-min)

with * NE have ¬ distinct (prios xs)
  unfolding alist-split(2)
  unfolding dfs-comp
  by (induct (dfs alist xs)) (auto simp add: rev-image-eqI)
with (distinct (prios xs)): show False by simp
qed
with ot show ?thesis by auto
qed

lemma alist-normalize-idem:
dfs alist (normalize xs) = dfs alist xs
unfolding normalize-def
proof (induct xs rule: rev-induct)
case (snoc x xs) then show ?case by (cases x) (simp-all add: dfs-append)
qed simp

lemma dfs-match-not-in:
(∀ t. Some t ∈ set xs → priority t ≠ a) ⇒
  set (dfs f (map (match a) xs)) = set (dfs f xs)
by (induct xs) simp-all

lemma dfs-match-subset:
set (dfs f (map (match a) xs)) ⊆ set (dfs f xs)
proof (induct xs rule: list.induct)
case (Cons x xs) then show ?case by (cases x) auto

37
qed simp

lemma dfs-match-distinct:
  distinct (dfs f xs) \implies distinct (dfs f (map (match a) xs))
proof (induct xs rule: list.induct)
  case (Cons x xs) then show ?case
    using dfs-match-subset[of f 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 \( \text{set} (\text{bt-dfs} f r) - \text{set} (\text{bt-dfs} f t) = \text{set} (\text{bt-dfs} f r) \) by \text{auto}

ultimately show \(?thesis by \text{auto}

qed

\text{simp-all}

\text{lemma} \ \text{alist-meld:}
\quad \text{distinct} (\text{dfs val} \ xs) \implies \text{distinct} (\text{dfs val} \ ys) \implies
\quad \text{set} (\text{dfs val} \ xs) \cap \text{set} (\text{dfs val} \ ys) = \{\} \implies
\quad \text{set} (\text{dfs alist} (\text{meld} \ xs \ ys)) = \text{set} (\text{dfs alist} \ xs) \cup \text{set} (\text{dfs alist} \ ys)

\text{proof (induct} \ xs \ ys \ \text{rule: meld.induct)}
\quad \text{case} (3 \ xs \ y \ ys)
\quad \text{have} \quad \text{set} (\text{dfs alist} \ (y \ # \ \text{meld} \ xs \ ys)) =
\quad \text{set} (\text{dfs alist} \ xs) \cup \text{set} (\text{dfs alist} \ (y \ # \ ys))

\text{proof –}
\quad \text{note} \ \text{assms} = 3
\quad \text{from} \ \text{assms} \ \text{have} \ (\text{vals} \ xs) \cap \text{set} \ (\text{vals} \ ys) = \{\}
\quad \text{using} \ \text{set-dfs-Cons[of val y ys]} \ \text{by auto}

moreover
\quad \text{from} \ \text{assms} \ \text{have} \ \text{distinct} (\text{vals} \ ys) \ \text{by (cases} y) \ \text{simp-all}

moreover
\quad \text{from} \ \text{assms} \ \text{have} \ \text{distinct} (\text{vals} \ xs) \ \text{by simp}

moreover \ \text{note} \ \text{assms}
\quad \text{ultimately have} \quad \text{set} (\text{dfs alist} \ (\text{meld} \ xs \ ys)) =
\quad \text{set} (\text{dfs alist} \ xs) \cup \text{set} (\text{dfs alist} \ ys) \ \text{by simp}

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

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

\text{then show} \quad ?thesis \ \text{using} \ \text{set-dfs-Cons[of alist y ys]} \ \text{by auto}

\text{qed}

\text{thus} \quad ?case \ \text{by simp}

\text{next}
\quad \text{case} (4 \ x \ xs \ ys)
\quad \text{have} \quad \text{set} (\text{dfs alist} \ (x \ # \ \text{meld} \ xs \ ys)) =
\quad \text{set} (\text{dfs alist} \ (x \ # \ xs)) \cup \text{set} (\text{dfs alist} \ ys)

\text{proof –}
\quad \text{note} \ \text{assms} = 4
\quad \text{from} \ \text{assms} \ \text{have} \ (\text{vals} \ xs) \cap \text{set} \ (\text{vals} \ ys) = \{\}
\quad \text{using} \ \text{set-dfs-Cons[of val x xs]} \ \text{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
set
(dfs
alist
(meld
xs
ys))
=
set
(dfs
alist
xs)
∪
set
(dfs
alist
ys)
by
simp

hence
set
(dfs
alist
(x
meld
xs
ys))
=
set
(dfs
alist
x)
∪
set
(dfs
alist
xs)
∪
set
(dfs
alist
ys)
using
set-dfs-Cons[of
alist
x
meld
xs
ys]
by
auto

then
show
?thesis
using
set-dfs-Cons[of
alist
x
xs]
by
auto
qed

thus
?case
by
simp

next

case
(5
x
xs
y
ys)

have
set
(dfs
alist
(add
(Some
(merge
x
y))
(meld
xs
ys)))
=
set
(bt-dfs
alist
x)
∪
set
(dfs
alist
xs)
∪
set
(bt-dfs
alist
y)
∪
set
(dfs
alist
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
x
yint:
set
(bt-dfs
val
x)
∩
set
(bt-dfs
val
y)
= \{\}
by
(auto
simp
add:
set-dfs-Cons)

ultimately
have
∗:
set
(dfs
alist
(Some
(merge
x
y)))
=
set
(bt-dfs
alist
x)
∪
set
(bt-dfs
alist
y)
by
auto

moreover
from
assms
have
∗∗:
set
(dfs
alist
(meld
xs
ys))
=
set
(dfs
alist
xs)
∪
set
(dfs
alist
ys)
by
(auto
simp
add:
set-dfs-Cons)

moreover
from
assms
have
distinct
(vals
(Some
(merge
x
y)
meld
xs
ys))

proof
−

from
assms
x
yint
have
distinct
(bt-dfs
val
(Some
(merge
x
y)))
by
(simp
add:
vals-merge-distinct)

moreover

from
assms
have
distinct
(vals
xs)

40
and distinct (vals ys)
and set (vals xs) ∩ 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 (btdfs val (merge x y)) ∩ 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) ∈ set xs
by (simp add: find-min-exist)
hence sub: set (btdfs alist ts) ⊆ set (dfs alist xs)
by (induct xs) (auto simp add: btdfs-simp)

from d IN have (v,a) /∈ set (btdfs alist ts)
using dfs-distinct-member[of alist xs Node a v ts] by simp
with sub have set (btdfs alist ts) ⊆ set (dfs alist xs) − {(v, a)} by blast
hence nu: set (btdfs alist ts) ∪ (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 (btdfs 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-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
by (simp add: delete-min-def alist-normalize-idem set-bts-dfs-rev dfs-map-Some-idem
dfs-match Diff-insert2 [of set (dfs alist xs) (v,a) set (bts-dfs alist ts)])
qed

lemma alist-remove-min:
assumes is-binqueue l xs
and distinct (vals xs)
and distinct (prios xs)
and normalized xs
and xs ≠ []
shows set (dfs alist (delete-min xs)) =
set (PQ.alist-of (PQ.remove-min (pqueue xs)))

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)

cf.
moreover
from Node ot 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 (\{\cdot\}\}
and PQ.priorities (\|\cdot\|)