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

1.2 Type of abstract priority queues

typedef (overloaded) ('a, 'b:linorder) pq = 
\{xs :: ('a \times 'b) list. distinct (map fst xs) \land \text{sorted (map snd xs)}\}
morphisms alist-of Abs-pq
proof –
have \([] \in ?pq\) 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]:
\[ \text{is-empty } q \implies \text{alist-of } q = \[] \]
by \((\text{simp add: is-empty-def})\)

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

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

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

**Lemma** priorities-empty [simp]:
\[ \|\text{empty}\| = \[] \]
by \((\text{simp add: priorities-def})\)

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

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

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

**Lemma** map-snd-alist-of:
\[ \text{map (the o priority } q) (\text{values } q) = \text{map snd (alist-of } q) \]
by \((\text{auto simp add: values-def priority-def})\)

**Lemma** image-snd-alist-of:
\[ \text{the } \text{priority } q \cdot \text{set (values } q) = \text{snd } \cdot \text{set (alist-of } q) \]
**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)
    then
lemma Min-snd-hd:
\[ q \neq [] \quad \Rightarrow \quad \text{sorted} (\text{map} \ q) \quad \Rightarrow \quad \text{Min} (\text{snd} \ \text{set} \ q) = \text{snd} (\text{hd} \ q) \]
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:
\[ x \notin \text{fst \ ' \ s} \quad \Rightarrow \quad (x, p) \notin s \]
by (auto simp add: image-def)

lemma alist-of-push [simp, code abstract]:
\[ \text{alist-of} \ (\text{push} \ k \ p \ q) = \]
\[ \begin{cases} \text{if} \ k \notin \text{set} \ (\text{values} \ q) \quad \text{then insort-key} \ \text{snd} \ (k, p) \ \text{alist-of} \ q \quad \text{else} \ \text{alist-of} \ q \end{cases} \]
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]:
\[ k \notin \text{set} \ |q| \quad \Rightarrow \quad \text{set} \ |\text{push} \ k \ p \ q| = \text{set} \ |q| \cup \{p\} \]
\[ k \in \text{set} \ |q| \quad \Rightarrow \quad \text{set} \ |\text{push} \ k \ p \ q| = \text{set} \ |q| \]
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)
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) ('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 ∃ xs. P xs ⇒ P (None # xs)
and ∃ x xs. P xs ⇒ P (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 \iff ts = []
  is-bintree-list l [] \iff 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) \iff 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 = \Rightarrow 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 \neq []
  shows children (last ts) = []
  using assms by (induct auto)

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

Heap property

inductive is-heap-list :: \'a::linorder \Rightarrow ('a, 'b) bintree list \Rightarrow bool where
  is-heap-list-Nil: is-heap-list h []
  | is-heap-list-Cons: is-heap-list h ts =\Rightarrow is-heap-list (priority t) (children t)
  \Rightarrow (priority t) \geq h =\Rightarrow is-heap-list h (t # ts)

abbreviation (input) is-heap t \equiv is-heap-list (priority t) (children t)

lemma is-heap-list-simps [simp]:
  is-heap-list h [] \iff True
  is-heap-list h (t # ts) \iff is-heap-list h ts \&\& is-heap-list (priority t) (children t) \&\& priority t \geq 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) =\Rightarrow is-heap-list l ts
  is-heap-list l (ts@rs) =\Rightarrow 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 \( l \) (rev \( ts \))
by (induct \( ts \) rule: rev-induct) auto

**Lemma** is-heap-children-larger:

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

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

\( is-heap \ t \ \Rightarrow \ \text{children} \ t \ \neq \ [] \ \Rightarrow \ \text{priority} \ t \ \leq \ \text{Min} \ (\text{priority} \ ' \ \text{set} \ (\text{children} \ t)) \)
by (simp add: is-heap-children-larger)

**Combination of both: binqueue property**

**Inductive** is-binqueue :: nat ⇒ ('a::linorder, 'b) binqueue ⇒ bool where
  - Empty: is-binqueue \( l \) []
  - None: is-binqueue (Suc \( l \)) \( xs \) \( \Rightarrow \) is-binqueue \( l \) (None # \( xs \))
  - Some: is-binqueue (Suc \( l \)) \( xs \) \( \Rightarrow \) is-bintree \( l \) \( t \)
    \( \Rightarrow \) is-heap \( t \) \( \Rightarrow \) is-binqueue \( l \) (Some \( t \) # \( xs \))

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

\( is-binqueue \ l \) [] \( \leftrightarrow \) True
\( is-binqueue \ l \) (Some \( t \) # \( xs \)) \( \leftrightarrow \)
\( is-bintree \ l \) \( t \) \( \land \) is-heap \( l \) \( \land \) is-binqueue (Suc \( l \)) \( xs \)
\( is-binqueue \ l \) (None # \( xs \)) \( \leftrightarrow \) is-binqueue (Suc \( l \)) \( xs \)
by (auto intro: is-binqueue.intros elim: is-binqueue.cases)

**Lemma** is-binqueue-trans:

\( is-binqueue \ l \) \( (x\#xs) \) \( \Rightarrow \) is-binqueue (Suc \( l \)) \( xs \)
by (cases \( x \)) simp-all

**Lemma** is-binqueue-head:

\( is-binqueue \ l \) \( (x\#xs) \) \( \Rightarrow \) is-binqueue \( l \) [\( x \)]
by (cases \( x \)) simp-all

**Lemma** is-binqueue-append:

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

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

\( is-binqueue \ l \) \( (xs \@ys) \) \( \Rightarrow \) is-binqueue \( l \) \( xs \)
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 ⇒ 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:
\[
\text{normalized } xs \implies \text{normalized } ys \implies \text{normalized } (xs @ ys)
\]
by (cases ys) (simp-all add: normalized-last-not-None)

**lemma** normalized-not-None:
\[
\text{normalized } xs \implies \text{set } xs \neq \{ \text{None} \}
\]
by (induct xs) (auto simp add: normalized-Cons [of - ts] dest: subset-singletonD)

**primrec** normalize' :: (′a ‚ ′b) binqueue ⇒ (′a ‚ ′b) binqueue where
\[
\text{normalize' } [] = []
\]
[ normaquoteze' (3 ≠ xs) =
\[
\text{(case } x \text{ of } \text{None } \Rightarrow \text{normalize' } xs | \text{Some } t \Rightarrow (x ≠ xs))
\]

**definition** normalize :: (′a ‚ ′b) binqueue ⇒ (′a ‚ ′b) binqueue where
\[
\text{normalize } xs = \text{rev } (\text{normalize' } (\text{rev } xs))
\]

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

**lemma** is-bintree-normalize:
\[
\text{is-bintree } l xs \implies \text{is-bintree } l (\text{normalize } xs)
\]
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
\[
\text{merge } t1 t2 = (\text{if } \text{priority } t1 < \text{priority } t2 \text{ then } \text{Node } (\text{priority } t1) (\text{val } t1) (t2 # \text{children } t1) \text{ else } \text{Node } (\text{priority } t2) (\text{val } t2) (t1 # \text{children } t2))
\]

**lemma** is-bintree-list-merge:
\[
\text{assumes } \text{is-bintree } l t1 \text{ is-bintree } l t2
\]
\[
\text{shows } \text{is-bintree } (\text{Suc } l) (\text{merge } t1 t2)
\]
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

12
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) binqueue
fix y :: ('a, 'b) bintree option
fix l :: nat
assume \( \wedge \ l. \text{is-binqueue } l \text{ xs } \implies \text{is-binqueue } l \text{ ys } \)
⇒ is-binqueue l (meld xs ys)
and is-binqueue (None # ys)
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) binqueue
fix x :: ('a, 'b) bintree option
fix l :: nat
assume \( \wedge \ l. \text{is-binqueue } l \text{ xs } \implies \text{is-binqueue } l \text{ ys } \)
⇒ 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)
fix \(xs\) \(ys\) :: \(\langle a, b \rangle\) binqueue
fix \(y\) :: \(\langle a, b \rangle\) bintree option
assume normalized \(xs\) \\(\Rightarrow\) normalized \(ys\) \\(\Rightarrow\) normalized \((meld xs ys)\)
and normalized \((None \# xs)\)
and normalized \((y \# ys)\)
then show normalized \((meld (None \# xs) (y \# ys))\) by \((\text{cases } y)\) simp-all

next
fix \(xs\) \(ys\) :: \(\langle a, b \rangle\) binqueue
fix \(x\) :: \(\langle a, b \rangle\) bintree option
assume normalized \(xs\) \\(\Rightarrow\) normalized \(ys\) \\(\Rightarrow\) normalized \((meld xs ys)\)
and normalized \((x \# xs)\)
and normalized \((None \# ys)\)
then show normalized \((meld (x \# xs) (None \# ys))\) by \((\text{cases } x)\) simp-all

qed (simp-all add: normalized-add)

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

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

lemma least-simps [simp, code]:
least None \(x\) = \(x\)
least \( x \) None = \( x \)
least (Some \( x' \)) (Some \( y' \)) = (if \( x' \leq y' \) 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 \( z \) \lor y = Some \( z \)
using assms proof 
  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 \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.left-commute del: 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 \( a \) \Rightarrow priority (the x) = \( a \)
min [x] = None \Rightarrow 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:
  \[ \text{min } xs = \text{None} \iff xs = [] \lor \text{set } xs = \{\text{None}\} \]

proof (rule iffI)
  have splitQ: \( \forall xs. xs \subseteq \{\text{None}\} \implies xs = \{\}\lor \text{set } xs = \{\text{None}\} \) by auto

  assume \( \text{min } xs = \text{None} \)
  then have \( \text{set } xs \subseteq \{\text{None}\} \)
    proof (induct xs)
      case (None ys)
      thus \(?\) by simp
    next
      case (Some t ys)
      thus \(?\) by simp
    qed simp

  with splitQ show \( xs = [] \lor \text{set } xs = \{\text{None}\} \) by auto

next
  show \( xs = [] \lor \text{set } xs = \{\text{None}\} \implies \text{min } xs = \text{None} \)
    by (induct xs) (auto dest: subset-singletonD)
  qed

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

lemma min-is-min:
  assumes \( \text{normalized } xs \)
  and \( xs \neq [] \)
  and \( \text{min } xs = \text{Some } a \)
  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: binqueue-induct)
    case (Some t ys)
    thus \(?\) by simp
      proof (cases a \leq \text{priority } t)
        case False
        with \( \text{Some } N \) have \( \text{N} \)
          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 \( \text{a’} \): \( \text{min } ys = \text{Some } a’ \) 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
      qed simp
    qed simp
  qed simp-all
lemma min-exists:
assumes min xs = Some a
shows Some a ∈ map-option priority " set xs
proof (rule ccontr)
  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)
  show ?case
  proof (rule ccontr)
    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 simp-all
with assms show False by simp
qed

primrec find :: 'a::linorder ⇒ ('a, '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) bintree ⇒ ('a, 'b) bintree 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
  by (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
  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
     ⇒ (∃ 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-None 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]:**

\[ \text{delete-min} \; [] = [] \]

**by** (simp add: delete-min-def)

**lemma delete-min-nonempty [simp]:**

\( \text{normalized} \; \text{xs} \Rightarrow \text{xs} \neq [] \Rightarrow \text{find-min} \; \text{xs} = \text{Some} \; t \)

\[ \Rightarrow \text{delete-min} \; \text{xs} = \text{normalize} \; (\text{meld} \; (\text{map} \; \text{Some} \; (\text{rev} \; \text{children} \; t)) \; (\text{map} \; (\text{match} \; (\text{priority} \; t)) \; \text{xs})) \]

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

**lemma is-binqueue-delete-min:**

assumes is-binqueue 0 \( \text{xs} \)

shows is-binqueue 0 (delete-min \( \text{xs} \))

**proof** (cases find-min \( \text{xs} \))

**case** (\text{Some} \; t)

from \text{assms} have is-binqueue 0 (map (match (priority \; t)) \; \text{xs})

**by** (induct \( \text{xs} \)) simp-all

moreover

from \text{Some} \; \text{have} Some \; t \in \text{set} \; \text{xs} **by** (rule find-min-exist)

with \text{assms} have \( \exists l. \; \text{is-bintree} \; l \; t \; \text{and} \; \text{is-heap} \; t \)

**using** is-binqueue-select[of 0 \; \text{xs} \; t] **by** auto

with \text{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 \( \text{xs} \))

**by** (cases find-min \( \text{xs} \))

(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) bintree

⇒ ('a, 'b) bintree ⇒ ('a, 'b) bintree

**where**

meld' \; z \; \text{xs} \; \text{ys} = \text{add} \; z \; (\text{meld} \; \text{xs} \; \text{ys})

**lemma [code]:**

\[ \text{add} \; z \; \text{xs} = \text{meld'} \; z \; [] \; \text{xs} \]

20
meld \(xs\) \(ys\) = meld' \(None\) \(xs\) \(ys\)

by (simp-all add: meld'-def)

lemma [code]:
meld' \(z\) (Some \(t\) \# \(xs\)) (Some \(r\) \# \(ys\)) =
\(z\) \# (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 insert-def)

definition
pop :: ('a::linorder,'b) binqueue ⇒ ('('b × 'a) option × ('a,'b) binqueue)
where
\[
\text{pop } xs = (\text{case find-min } xs \text{ of} \\
\quad \text{None } \Rightarrow (\text{None, } xs) \\
\quad | \text{Some } t \Rightarrow (\text{Some } (\text{val } t, \text{priority } t), \text{delete-min } xs))
\]

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

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

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

3 Relating Functional Binomial Queues To The Abstract Priority Queues

\[\text{notation} \]

\[\text{PQ.values } (|(-)|) \]
\[\text{and PQ.priorities } (||(-)||)\]

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

\[\text{primrec bt-dfs :: } (\text{a::linorder, 'b} \text{ bintree } \Rightarrow 'c) \Rightarrow (\text{a, 'b}) \text{ bintree } \Rightarrow 'c \text{ list} \]
\[\text{and bts-dfs :: } (\text{a::linorder, 'b} \text{ bintree } \Rightarrow 'c) \Rightarrow (\text{a, 'b}) \text{ bintree list } \Rightarrow 'c \text{ list} \]

\[\text{where} \]
\[\text{bt-dfs } f \ (\text{Node } a \ v \ ts) = f \ (\text{Node } a \ v \ ts) \# \text{ bt-dfs } f ts \]
\[| \text{ bt-dfs } f \ [] = [] \]
\[| \text{ bt-dfs } f \ (t \ # \ ts) = \text{ bt-dfs } f \ t \ @ \text{ bt-dfs } f \ ts \]

\[\text{lemma bt-dfs-simp:} \]
\[\text{bt-dfs } f \ t = f \ t \# \text{ bt-dfs } f \ (\text{children } t) \]
\[\text{by (cases } t \text{ simp-all)}\]

\[\text{lemma bts-dfs-append [simp]:} \]
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 (bts-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) bintree \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:
  \[ \text{dfs} \left( f \circ g \right) \text{xs} = \text{map f} \left( \text{dfs} \ g \text{xs} \right) \]
  by \((\text{induct xs}) \ (\text{simp-all add: bt-dfs-comp del: o-apply})\)

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

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

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

primrec alist :: \((a, b) \text{ bintree} \Rightarrow (b \times a)\) where
  \[ \text{alist} \left( \text{Node } a \ v - \right) = \left( v, a \right) \]

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

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

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

lemma in-set-in-alist:
  assumes \(\text{Some } t \in \text{set xs}\)
  shows \((\text{val } t, \text{priority } t) \in \text{set } (\text{dfs} \text{alist } \text{xs})\)
  using \assms\nproof \((\text{induct xs})\)
  case \((\text{Some } x \text{xs})\) then show \(?case\)
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
  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 ⇒ ('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: 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 q rule; bt-augment.induct bts-augment.induct)
auto

lemma bt-augment-v-in:
\[ v \in \text{set} | q \implies v \in \text{set} | \text{bt-augment} t q | \]
\[ v \in \text{set} | q \implies v \in \text{set} | \text{bts-augment} ts q | \]
using 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 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 \[ (\text{set} | \text{bt-dfs} \text{val} t) \cup \text{set} | q | = \text{set} | \text{bt-augment} t q | \]
and \[ (\text{set} | \text{bts-dfs} \text{val} ts) \cup \text{set} | q | = \text{set} | \text{bts-augment} ts q | \]
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 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 ?case by auto
qed auto

lemma vals-pqueue:
\[ \text{set} (\text{vals} \text{xs}) = \text{set} | \text{pqueue} \text{xs} | \]
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 bt-val-augment|where q = PQ.push v a q by (simp-all add: bt-val-augment)

lemma bt-augment-v-push-commute:
\[ \text{set} | \text{bt-augment} t (\text{PQ}.\text{push} v a q) | = \text{set} | \text{PQ}.\text{push} v a (\text{bt-augment} t q) | \]
lemma bts-augment-v-union:
set | bts-augment t (bts-augment rs q) | =
set | bts-augment t 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: bt-augment.induct bts-augment.induct)
case Nil-bintree
from bt-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 bt-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) ⇒
\[
\begin{align*}
set (bt-dfs \ val \ t) \cap \set (bt-dfs \ val \ r) &= \{\} \implies \\
\text{distinct} (bt-dfs \ val \ (merge \ t \ r)) &\quad \text{by (auto simp add: bt-dfs-simp merge-def)}
\end{align*}
\]

**lemma** vals-add-Cons:
\[
set (vals \ (add \ x \ xs)) = \set (vals \ (x \ # \ xs))
\]
**proof** (cases \(x\))
\[
\begin{align*}
case \ (\text{Some} \ t) &\quad \text{then show \ ?thesis} \\
&\quad \text{by (induct \(xs\) arbitrary; \(x \ t\) auto)
\end{align*}
\]
**qed simp**

**lemma** vals-add-distinct:
\[
\begin{align*}
&\text{assumes distinct} (vals \ xs) \\
&\text{and distinct} (dfs val [x]) \\
&\text{and set} (vals \ xs) \cap \set (dfs val [x]) = \{\} \\
&\text{shows distinct} (vals \ (add \ x \ xs))
\end{align*}
\]
**using** assms
**proof** (cases \(x\))
\[
\begin{align*}
case \ (\text{Some} \ t) &\quad \text{with assms show \ ?thesis} \\
&\quad \text{proof (induct \(xs\) arbitrary; \(x \ t\) auto)
\end{align*}
\]
**qed simp**

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

**lemma** insert-v-push:
\[
set (vals \ (insert \ a \ v \ xs)) = \set [PQ.push \ v \ a \ (pqueue \ xs)]
\]
**by (simp add: vals-pqueue[symmetric])**

**lemma** vals-meld:
\[
set (dfs \ val \ (meld \ xs \ ys)) = \set (dfs \ val \ xs) \cup \set (dfs \ val \ ys)
\]
**proof** (induct \(xs \ ys\) rule: meld.induct)
\[
\begin{align*}
case \ (3 \ xs \ y \ ys) &\quad \text{then show \ ?case} \\
&\quad \text{using \(set-dfs-Cons[of \ val \ y \ meld \ xs \ ys]\ using \(set-dfs-Cons[of \ val \ y \ ys]\ by \ auto
\end{align*}
\]
**next**
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 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) ∩ 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)) = {}
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 bts-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 (bts-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 and q rule: bt-augment.induct bts-augment.induct)

case Nil-bintree
from \texttt{bt-augment-alist-subset[of q]} show \texttt{?case by auto} 

next

\begin{itemize}
\item[case (Node a v rs)]
\begin{itemize}
\item have 
\begin{itemize}
\item set (PQ.alist-of (bts-augment rs (bt-augment r q))) = 
\begin{itemize}
\item set (PQ.alist-of (bts-augment rs q)) ∪ set (PQ.alist-of (bt-augment r q))
\end{itemize}
\end{itemize}
\begin{itemize}
\item by simp
\end{itemize}
moreover
\begin{itemize}
\item from \texttt{Node.prems} have \texttt{*}: \(v \notin \text{set |bts-augment rs q| ∪ set |bt-augment r q|}\)
\end{itemize}
\begin{itemize}
\item unfolding \texttt{bt-val-augment[symmetric]} by simp
\end{itemize}
hence
\item[v \notin \text{set |bts-augment rs q|}]
by \texttt{(unfold bt-augment-v-union)}
\end{itemize}
moreover
\begin{itemize}
\item from \texttt{*} have \texttt{v \notin set |bt-augment rs q|} by simp
\end{itemize}
ultimately show \texttt{?case by (simp add: set-insort-key)}
\end{itemize}

next

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

qed

\textbf{lemma} \texttt{bt-alist-augment}:
\begin{itemize}
\item distinct (bts-dfs val t) \implies set (bts-dfs val t) ∩ set \(\text{|q| = \{\}}\) \implies set (bts-dfs alist t) ∪ set (PQ.alist-of q) = set (PQ.alist-of (bt-augment t q))
\end{itemize}

\begin{itemize}
\item distinct (bts-dfs val ts) \implies set (bts-dfs val ts) ∩ set \(\text{|q| = \{\}}\) \implies set (bts-dfs alist ts) ∪ set (PQ.alist-of q) = set (PQ.alist-of (bt-augment ts q))
\end{itemize}
\textbf{proof (induct t and ts rule: bt-augment.induct bts-augment.induct)}

\begin{itemize}
\item case \texttt{Nil-bintree} then show \texttt{?case by simp}
\end{itemize}

next

\begin{itemize}
\item[case (Node a v rs)]
\item hence \texttt{v \notin set |bt-augment rs q|}
\end{itemize}
unfolding $bt$-val-augment[symmetric] by simp

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

next
case (Cons-bintree \(r\) \(rs\)) then
  have \(\text{set (PQ.alist-of (bts-augment (r ≠ rs) q))} = \text{set (PQ.alist-of (bts-augment rs q))} ∪ \text{set (PQ.alist-of (bt-augment r q))}\)
  using bt-augment-alist-union by simp
  with Cons-bintree bt-augment-alist-subset show \texttt{?case} by auto
qed

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

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

lemma prios-pqueue:
  \(\text{distinct (vals \(xs\))} ⇒ \text{set (prios \(xs\))} = \text{set (pqueue \(xs\))}\]
  by \texttt{(auto simp add: alist-pqueue priorities-set alist-split-set)}

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

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

  from \(\text{prem}\) have \(\text{distinct (bt-dfs val (merge \(t\) \(y\)))}\)
  by \texttt{(auto simp add: bt-dfs-simp merge-def)}
with prem have distinct (vals (Some (merge t y) # y)) 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 \(\text{PQ.is-empty} (\text{pq} \text{ue} \; xs)\) by simp
next
assume \(N: \text{normalized} \; xs\) and \(E: \text{PQ.is-empty} (\text{pq} \text{ue} \; xs)\)
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 \(\text{set} \; |\text{pq} \text{ue} \; xs| \neq \{\}\) by (simp add: vals-pqueue)
moreover
from \(E\) have \(\text{set} \; |\text{pq} \text{ue} \; xs| = \{\}\) by (simp add: is-empty-empty)
ultimately show \(\text{False}\) by simp
qed
qed

lemma bt-dfs-Min-priority:
  assumes \(\text{is-heap} \; t\)
  shows \(\text{priority} \; t = \text{Min} (\text{set} \; (\text{bt-dfs} \; \text{priority} \; t))\)
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 \(?M = \text{Min} (\text{set} \; (\text{bt-dfs} \; \text{priority} \; t))\)

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 \(\text{priority} \; t = \text{Min} (\text{set} \; (\text{bt-dfs} \; \text{priority} \; t'))\)
by simp
with \(\text{ot}\)
have \(\text{priority} \; t = \text{Min} (\text{Set.insert} (\text{priority} \; t) \; (\text{set} \; (\text{bts-dfs} \; \text{priority} \; rs)))\)
by (simp add: bt-dfs-simp)
moreover
from cons have \(\text{priority} \; r = \text{Min} (\text{set} \; (\text{bt-dfs} \; \text{priority} \; r))\) by simp
moreover
from cons have \(\text{children} \; t = r \neq rs\) by simp
then have \(\text{bts-dfs} \; \text{priority} \; (\text{children} \; t) =\)
\((\text{bt-dfs} \; \text{priority} \; r) @ (\text{bts-dfs} \; \text{priority} \; rs)\) by simp
hence \(\text{bt-dfs} \; \text{priority} \; t =\)
\(\text{priority} \; t \neq (\text{bt-dfs} \; \text{priority} \; r @ \text{bts-dfs} \; \text{priority} \; rs)\)

34
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** Set.insert (priority t) (set (bt-dfs priority r))

\( \cup \) set (bts-dfs priority rs) =

Set.insert (priority t) (set (bts-dfs priority rs)) \cup set (bt-dfs priority r)

by auto

with A have \( ?M = \text{Min} \)

(\(\text{Set.insert} \ (\text{priority} \ t) \ (\text{set} \ (\text{bts-dfs priority} \ rs)) \cup \text{set} \ (\text{bt-dfs priority} \ r)\))

by simp

with Min-Un

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

have \( ?M = \text{ord-class.min} \ (\text{Min} \ (\text{Set.insert} \ (\text{priority} \ t) \ (\text{set} \ (\text{bts-dfs priority} \ rs)))))) \)

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

ultimately

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

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

qed

**lemma** is-biqueue-min-Min-prios:

assumes is-biqueue l xs

and normalized xs

and xs \( \neq \) []

shows min xs = Some (Min (set (prios xs)))

using assms

**proof** (induct xs)

case (Some l xs x) then show ?case

proof (cases xs \( \neq \) [])

case False with Some show ?thesis

using bt-dfs-Min-priority[of x] by (simp add: min-single)

next

case True note \( T = \text{this} \) Some

from \( T \) have normalized xs by simp

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

with \( T \) show ?thesis

using Min-Un[of set (bt-dfs priority x) set (prios xs)]

using bt-dfs-Min-priority[of x]
lemma min-p-min:
assumes is-binqueue l xs
and \( xs \neq [] \)
and normalized xs
and distinct (vals xs)
and distinct (prios xs)
sshows \( \min xs = \text{PQ}.\text{priority} (\text{pqueue xs}) \ (\text{PQ}.\min (\text{pqueue xs})) \)
proof –
from \( xs \neq [] \) \( \text{normalized xs} \) have \( \neg \text{PQ}.\text{is-empty} (\text{pqueue xs}) \)
by (simp add: empty-empty)
moreover
from assms have \( \min xs = \text{Some} (\text{Min} (\text{set} (\text{prios xs}))) \)
by (simp add: is-binqueue-min-Min-prios)
with (distinct (vals xs)) have \( \min xs = \text{Some} (\text{Min} (\text{set} \ \parallel \text{pqueue xs} \parallel )) \)
by (simp add: prios-pqueue)
ultimately show \(?thesis \)
by (simp add: priority-Min-priorities [where \( q = \text{pqueue xs} \] )
qed

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

lemma find-min-v-min:
assumes is-binqueue l xs
and \( xs \neq [] \)
and normalized xs
and distinct (vals xs)
and distinct (prios xs)
s shows \( \text{val (the (find-min xs))} = \text{PQ.min (pqueue xs)} \)

proof
from assms have \( \text{min xs} \neq \text{None} \) by (simp add: normalized-min-not-None)
then obtain a where oo: 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 \( \ast \): (val t, a) \( \in \) set (dfs alist xs)
    by (auto simp add: find-min-exist in-set-in-alist)

have \( \text{PQ.min (pqueue xs)} = \text{val t} \)
    proof (rule ccontr)
      assume A: \( \text{PQ.min (pqueue xs)} \neq \text{val t} \)
      then obtain t' where ot':PQ.min (pqueue xs) = t' by simp
      with A have NE: \( \text{val t} \neq t' \) by simp
      from ot' oo assms have (t', a) \( \in \) set (dfs alist xs)
          by (simp add: alist-pqueue PQ.priority-def min-p-min)
      with \( \ast \) NE have \( \neg \) distinct (prios xs)
          unfolding alist-split(2)
          unfolding dfs-comp
          by (induct (dfs alist xs)) (auto simp add: rev-image-eqI)
      with \( \neg \) 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 (cases x) (simp-all add: dfs-append)
  qed simp

lemma dfs-match-not-in:
  \( \forall t. \text{Some t} \in \text{set xs} \rightarrow \text{priority t \neq a} \) \( \rightarrow \)
  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)) \( \subseteq \) set (dfs f xs)
proof (induct xs rule: list.induct)
case (Cons x xs)
  then show \( \? \)case (cases x) auto
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) \setminus 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)
    next
      case False
      from Some False
      have set (dfs f (map (match a) xs)) = set (dfs f xs) \setminus set (bt-dfs f t)
        by simp
    next
      from Some.prems False have set (bt-dfs f t) \cap set (bt-dfs f r) = {}
        by (induct xs) auto
  qed simp
hence \( \text{set} (\text{bt-dfs } f \ r) \setminus \text{set} (\text{bt-dfs } f \ t) = \text{set} (\text{bt-dfs } f \ r) \) by auto

ultimately show \(?thesis by auto

qed

qed simp-all

\textbf{lemma} \texttt{alist-meld:}

\texttt{distinct (dfs val xs) \Rightarrow \text{distinct (dfs val ys)}} \Rightarrow

\texttt{set (dfs val xs) \cap set (dfs val ys) = \{\} \Rightarrow

\texttt{set (dfs alist (meld xs ys)) = set (dfs alist xs) \cup set (dfs alist ys)}}

\textbf{proof} (induct \(xs\) \(ys\) rule: \texttt{meld.induct})

\texttt{case (3 xs y ys)}

\texttt{have set (dfs alist (y \# meld xs ys)) =}

\texttt{set (dfs alist xs) \cup set (dfs alist (y \# ys))}

\textbf{proof –}

\texttt{note assms = 3}

\texttt{from assms have set (vals xs) \cap set (vals ys) = \{\}}

\texttt{using set-dfs-Cons[of val y ys] by auto}

moreover

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

moreover

\texttt{from assms have distinct (vals xs) by simp}

moreover \texttt{note assms}

ultimately \texttt{have set (dfs alist (meld xs ys)) =}

\texttt{set (dfs alist xs) \cup set (dfs alist ys) by simp}

\texttt{hence set (dfs alist (y \# meld xs ys)) =}

\texttt{set (dfs alist [y]) \cup set (dfs alist xs) \cup set (dfs alist ys)}

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

\texttt{then show \(?thesis using set-dfs-Cons[of alist y y ys] by auto

qed

thus \(?case by simp

next

\texttt{case (4 x xs ys)}

\texttt{have set (dfs alist (x \# meld xs ys)) =}

\texttt{set (dfs alist (x \# xs)) \cup set (dfs alist ys)}

\textbf{proof –}

\texttt{note assms = 4}

\texttt{from assms have set (vals xs) \cap set (vals ys) = \{\}}

\texttt{using set-dfs-Cons[of val x xs] by auto

39
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 xyint:
  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 xyint have distinct (bt-dfs val (merge x y))
    by (simp add: vals-merge-distinct)

moreover
from assms have
  distinct (vals xs)
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 (bt-dfs 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 (bts-dfs alist ts) ⊆ set (dfs alist xs)
by (induct xs) (auto simp add: bt-dfs-simp)

from d IN have (v,a) /∈ 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) ⊆ set (dfs alist xs) − {(v,a)} by blast
hence nu: set (bts-dfs 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 (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-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 (bt-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 at have priority (the (find-min xs)) = a by simp
with assms have a = the (PQ.priority (pq queue xs) (PQ.min (pq queue 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 (pq queue 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 (\(||\(-\)||\)\)
