Verification of Functional Binomial Queues

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

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

1 Abstract priority queues

1.1 Generic Lemmas

lemma tl-set:
  \text{distinct } q \implies \text{set } (\text{tl } q) = \text{set } q - \{\text{hd } q\}
(proof)

1.2 Type of abstract priority queues

typedef (overloaded) \(\langle a, 'b::linorder \rangle\) pq =
  \{xs :: \(\langle a \times 'b \rangle\) list. \text{distinct } (\text{map } \text{fst } xs) \wedge \text{sorted } (\text{map } \text{snd } xs)\}

morphisms alist-of Abs-pq
(proof)

lemma alist-of-Abs-pq:
  \text{assumes distinct } (\text{map } \text{fst } xs)
  \text{and sorted } (\text{map } \text{snd } xs)
  \text{shows alist-of } (\text{Abs-pq } xs) = xs
(proof)

lemma [code abstype]:
  Abs-pq (alist-of q) = q
(proof)

lemma distinct-fst-alist-of [simp]:
  \text{distinct } (\text{map } \text{fst } (\text{alist-of } q))
(proof)
lemma distinct-alist-of [simp]:
  distinct (alist-of q)
 ⟨proof⟩

lemma sorted-snd-alist-of [simp]:
  sorted (map snd (alist-of q))
 ⟨proof⟩

lemma alist-of-eqI:
  alist-of p = alist-of q ⇒ p = q
 ⟨proof⟩

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 f q = fst ′ set (alist-of q)
 ⟨proof⟩

lemma priorities-set:
  set s q = snd ′ set (alist-of q)
 ⟨proof⟩

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 ⇒ alist-of q = []
 ⟨proof⟩

lemma not-is-empty-alist-of [dest]:
  ¬ is-empty q ⇒ alist-of q ≠ []
proof

lemma alist-of-empty [simp, code abstract]:
alist-of empty = []
(proof)

lemma values-empty [simp]:
|empty| = []
(proof)

lemma priorities-empty [simp]:
∥ empty ∥ = []
(proof)

lemma values-empty-nothing [simp]:
∀ k. k \not\in \text{set} \emptyset
(proof)

lemma is-empty-empty:
is-empty q \iff q = \text{empty}
(proof)

lemma is-empty-empty-simp [simp]:
is-empty empty
(proof)

lemma map-snd-alist-of:
map (the \circ \text{priority} q) \text{values} q = map \text{snd} (alist-of q)
(proof)

lemma image-snd-alist-of:
the' priority q ' set \text{values} q = snd' set \text{alist-of} q
(proof)

lemma Min-snd-alist-of:
assumes \neg is-empty q
shows Min (snd' set \text{alist-of} q) = snd (hd \text{alist-of} q)
(proof)

lemma priority-fst:
assumes xp ∈ set \text{alist-of} q
shows priority q (fst xp) = Some (snd xp)
(proof)

lemma priority-Min:
assumes \( \neg \text{is-empty } q \)
shows priority \( q \) (min \( q \)) = Some (Min (the priority \( q \) set (values \( q \))))
⟨proof⟩

lemma priority-Min-priorities:
assumes \( \neg \text{is-empty } q \)
shows priority \( q \) (min \( q \)) = Some (Min (set \( q \)))
⟨proof⟩

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:
\( q \not= [] \implies \text{sorted } (\text{map } \text{snd } q) \implies \text{Min } (\text{snd } \cdot \text{set } q) = \text{snd } (\text{hd } q) \)
⟨proof⟩

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

lemma not-in-first-image:
x ∉ fst ‘ s \implies (x, p) ∉ s
⟨proof⟩

lemma alist-of-push [simp, code abstract]:
alist-of (push k p q) =
(if k ∉ set (values \( q \)) then insort-key snd (k, p) (alist-of q) else alist-of q)
⟨proof⟩

lemma push-values [simp]:
set | push k p q | = set | q | ∪ \{k\}
⟨proof⟩

lemma push-priorities [simp]:
\( k ∉ \text{set } | q | \implies \text{set } \| \text{push } k p q \| = \text{set } \| q \| ∪ \{p\} \)
k ∈ \text{set } | q | \implies \text{set } \| \text{push } k p q \| = \text{set } \| q \|
⟨proof⟩

lemma not-is-empty-push [simp]:
\( \neg \text{is-empty } (\text{push } k p q) \)
⟨proof⟩
lemma push-commute:
  assumes $a \neq b$ and $v \neq w$
  shows $\text{push } w b (\text{push } v a q) = \text{push } v a (\text{push } w b q)$
⟨proof⟩

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))
⟨proof⟩

lemma remove-min-empty [simp]:
  is-empty q ⇒ remove-min q = empty
⟨proof⟩

lemma alist-of-remove-min [simp]:
  ¬ is-empty q ⇒ alist-of (remove-min q) = tl (alist-of q)
⟨proof⟩

lemma values-remove-min [simp]:
  ¬ is-empty q ⇒ values (remove-min q) = tl (values q)
⟨proof⟩

lemma set-alist-of-remove-min:
  ¬ is-empty q ⇒ set (alist-of (remove-min q)) =
  set (alist-of q) − {((\text{min } q, \text{the } (\text{priority } q (\text{min } q))))}
⟨proof⟩

definition pop :: ('a, 'b::linorder) pq ⇒ ('a × ('a, 'b) pq) option where
  pop q = (if is-empty q then None else Some (\text{min } q, remove-min q))
lemma pop-simps [simp]:
  is-empty q ⇒ pop q = None
  ¬ is-empty q ⇒ pop q = Some (\text{min } q, remove-min q)
⟨proof⟩

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

no-notation
  PQ.values (\text{|-|})
  and PQ.priorities (\text{||-||})
2 Functional Binomial Queues

2.1 Type definition and projections

datatype \texttt{′a, ′b) bintree = Node ′a ′b (′a, ′b) bintree list}

primrec priority :: ′a, ′b) bintree ⇒ ′a where
\hspace*{1cm} priority (Node a - -) = a

primrec val :: ′a, ′b) bintree ⇒ ′b where
\hspace*{1cm} val (Node - v -) = v

primrec children :: ′a, ′b) bintree ⇒ (′a, ′b) bintree list where
\hspace*{1cm} 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]:
\hspace*{1cm} assumes P []
\hspace*{1.5cm} and \(\forall\) xs. P xs ⇒ P (None # xs)
\hspace*{1.5cm} and \(\forall\) x xs. P xs ⇒ P (Some x # xs)
\hspace*{1cm} shows P xs

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
\hspace*{1cm} is-bintree-list-Nil [simp]: is-bintree-list 0 []
\hspace*{1cm} | is-bintree-list-Cons: is-bintree-list l ts ⇒ is-bintree-list l (children t)
\hspace*{1cm} \hspace*{1cm} ⇒ 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]:
\[
is\text{-bintree-list } 0 \text{ ts } \iff \text{ts } = []
is\text{-bintree-list } l \text{ }[] \iff l = 0
\]
⟨proof⟩

**Lemma** is-bintree-list-simp [simp]:
\[
is\text{-bintree-list } (\text{Suc } l) \text{ (t } \# \text{ ts) } \iff
\text{is\text{-bintree-list } l \text{ (children t) } \land \text{is\text{-bintree-list } l \text{ ts})}
\]
⟨proof⟩

**Lemma** is-bintree-list-length [simp]:
\[
is\text{-bintree-list } l \text{ ts } = \Rightarrow \text{ length ts } = l
\]
⟨proof⟩

**Lemma** is-bintree-list-children-last:
\[
\text{assumes is\text{-bintree-list } l \text{ ts and ts } \neq []}
\text{shows children (last ts) } = []
\]
⟨proof⟩

**Lemma** is-bintree-children-length-desc:
\[
\text{assumes is\text{-bintree-list } l \text{ ts}}
\text{shows map (length } \circ \text{ children) ts } = \text{rev } [0..<l]
\]
⟨proof⟩

**Heap property**

**Inductive** is-heap-list :: 'a::linorder \Rightarrow ('a, 'b) bintree list \Rightarrow bool where
\[
is\text{-heap-list-Nil: } \text{is\text{-heap-list } h \text{ []}}
is\text{-heap-list-Cons: } \text{is\text{-heap-list } h \text{ ts } \Rightarrow \text{is\text{-heap-list } (priority t) \text{ (children t) \Rightarrow (priority t) } \geq h \Rightarrow \text{is\text{-heap-list } h \text{ (t } \# \text{ ts)}}}
\]

**Abbreviation** (input) is-heap t ≡ is-heap-list (priority t) (children t)

**Lemma** is-heap-list-simps [simp]:
\[
is\text{-heap-list } h \text{ [] } \iff \text{True}
is\text{-heap-list } h \text{ (t } \# \text{ ts) } \iff
\text{is\text{-heap-list } h \text{ ts } \land \text{is\text{-heap-list } (priority t) \text{ (children t) } \land \text{priority t } \geq h}
\]
⟨proof⟩

**Lemma** is-heap-list-append-dest [dest]:
\[
is\text{-heap-list } l \text{ ts@rs } \Rightarrow \text{is\text{-heap-list } l \text{ ts}
is\text{-heap-list } l \text{ ts@rs } \Rightarrow \text{is\text{-heap-list } l \text{ rs}
\]
⟨proof⟩

**Lemma** is-heap-list-rev:
is-heap-list l ts \implies is-heap-list l (rev ts)
⟨proof⟩

lemma is-heap-children-larger:
\text{is-heap } t \implies \forall x \in \text{set (children } t) . \text{priority } x \geq \text{priority } t
⟨proof⟩

lemma is-heap-Min-children-larger:
\text{is-heap } t \implies \text{children } t \neq [] \implies
\text{priority } t \leq \text{Min (priority } \cdot \text{ set (children } t))
⟨proof⟩

Combination of both: binqueue property

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

lemma is-binqueue-simp [simp]:
is-binqueue l [] \iff \text{True}
is-binqueue l (\text{Some } t \# \text{ xs}) \iff
\text{is-bintree } l t \land \text{is-heap } t \land \text{is-binqueue } (\text{Suc } l) \text{ xs}
is-binqueue l (\text{None } \# \text{ xs}) \iff \text{is-binqueue } (\text{Suc } l) \text{ xs}
⟨proof⟩

lemma is-binqueue-trans:
is-binqueue l (x \# \text{xs}) \implies \text{is-binqueue } (\text{Suc } l) \text{ xs}
⟨proof⟩

lemma is-binqueue-head:
is-binqueue l (x \# \text{xs}) \implies \text{is-binqueue } l [x]
⟨proof⟩

lemma is-binqueue-append:
is-binqueue l \text{ xs } \implies \text{is-binqueue } (\text{length } \text{xs} + l) \text{ ys } \implies \text{is-binqueue } l \text{ (xs } \@ \text{ ys)}
⟨proof⟩

lemma is-binqueue-append-dest [dest]:
is-binqueue l (\text{xs } \@ \text{ ys}) \implies \text{is-binqueue } l \text{ xs}
⟨proof⟩

lemma is-binqueue-children:
\text{assumes is-bintree-list } l \text{ ts}
and is-heap-list t ts
shows is-binqueue 0 (map Some (rev ts))
(proof)

lemma is-binqueue-select:
  is-binqueue l xs ⇒ Some t ∈ set xs ⇒ ∃ k. is-bintree k t ∧ is-heap t
(proof)

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)

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

lemma normalized-map-Some [simp]:
  normalized (map Some xs)
(proof)

lemma normalized-Cons:
  normalized (x#xs) ⇒ normalized xs
(proof)

lemma normalized-append:
  normalized xs ⇒ normalized ys ⇒ normalized (xs@ys)
(proof)

lemma normalized-not-None:
  normalized xs ⇒ set xs ≠ {None}
(proof)

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⟩

lemma is-binqueue-normalize:
    is-binqueue l xs ⇒ is-binqueue l (normalize xs)
⟨proof⟩

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)
⟨proof⟩

lemma is-heap-merge:
    assumes is-heap t1 is-heap t2
    shows is-heap (merge t1 t2)
⟨proof⟩

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 ≠ []
⟨proof⟩

lemma normalized-add:
assumes normalized \(xs\)
shows normalized \((\text{add } x \text{ } xs)\)
⟨proof⟩

lemma is-binqueue-add-None:
assumes is-binqueue \(l\) \(xs\)
shows is-binqueue \(l\) \((\text{add } \text{None } xs)\)
⟨proof⟩

lemma is-binqueue-add-Some:
assumes is-binqueue \(l\) \(xs\)
and is-bintree \(l\) \(t\)
and is-heap \(t\)
shows is-binqueue \(l\) \((\text{add } \text{Some } t \text{ } xs)\)
⟨proof⟩

function meld :: \( ('a::\text{linorder}, 'b) \text{ binqueue} \Rightarrow ('a, 'b) \text{ binqueue} \Rightarrow ('a, 'b) \text{ binqueue} \)
where
meld \(\text{[]}\) \(ys\) = \(ys\)
meld \(xs\) \(\text{[]}\) = \(xs\)
meld \((\text{None } \neq xs)\) \((y \# ys)\) = \(y \# \text{meld } xs \text{ } ys\)
meld \((x \# xs)\) \((\text{None } \neq ys)\) = \(x \# \text{meld } xs \text{ } ys\)
meld \((\text{Some } t \# xs)\) \((\text{Some } r \# ys)\) =
\(\text{None } \neq \text{add } (\text{Some } (\text{merge } t \text{ } r))\) \((\text{meld } xs \text{ } ys)\)
⟨proof⟩ termination ⟨proof⟩

lemma meld-singleton-add [simp]:
meld \([\text{Some } t]\) \(xs\) = \(\text{add } (\text{Some } t) \text{ } xs\)
⟨proof⟩

lemma nonempty-meld [simp]:
\(xs \neq \text{[]}\) \(\Rightarrow\) meld \(xs\) \(ys\) \(\neq \text{[]}\)
\(ys \neq \text{[]}\) \(\Rightarrow\) meld \(xs\) \(ys\) \(\neq \text{[]}\)
⟨proof⟩

lemma nonempty-meld-commute:
meld \(xs\) \(ys\) \(\neq \text{[]}\) \(\Rightarrow\) meld \(xs\) \(ys\) \(\neq \text{[]}\)
⟨proof⟩

lemma is-binqueue-meld:
assumes is-binqueue \(l\) \(xs\)
and is-binqueue \(l\) \(ys\)
shows is-binqueue \(l\) \((\text{meld } xs \text{ } ys)\)
⟨proof⟩
lemma normalized-meld:
assumes normalized xs
and normalized ys
shows normalized (meld xs ys)
⟨proof⟩

lemma normalized-meld-weak:
assumes normalized xs
and length ys ≤ length xs
shows normalized (meld xs ys)
⟨proof⟩

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 Some 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')
⟨proof⟩

lemma least-split:
assumes least x y = Some z
shows x = Some z ∨ y = Some z
⟨proof⟩

interpretation least: semilattice least ⟨proof⟩

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)
⟨proof⟩

lemma [code]:
min xs = fold (λ x. least (map-option priority x)) xs None
⟨proof⟩
lemma min-single:
min \{x\} = Some a \implies priority (the x) = a
min \{x\} = None \implies x = None
⟨proof⟩

lemma min-Some-not-None:
min (Some t \# xs) \neq None
⟨proof⟩

lemma min-None-trans:
assumes min (x\#xs) = None
shows min xs = None
⟨proof⟩

lemma min-None-None:
min xs = None \iff xs = [] \lor set xs = \{None\}
⟨proof⟩

lemma normalized-min-not-None:
normalized xs \implies xs \neq [] \implies min xs \neq None
⟨proof⟩

lemma min-is-min:
assumes normalized xs
and xs \neq []
and min xs = Some a
shows \forall x \in set xs. x = None \lor a \leq priority (the x)
⟨proof⟩

lemma min-exists:
assumes min xs = Some a
shows Some a \in map-option priority ' set xs
⟨proof⟩

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

declare find.simps [simp del]

lemma find-simps [simp, code]:
find a [] = None
find a (None \# xs) = find a xs
find a (Some t ≠ xs) = (if priority t = a then Some t else find a xs)
⟨proof⟩

lemma find-works:
  assumes Some a ∈ set (map (map-option priority) xs)
  shows ∃ t. find a xs = Some t ∧ priority t = a
⟨proof⟩

lemma find-works-not-None:
  Some a ∈ set (map (map-option priority) xs) ⇒ find a xs ≠ None
⟨proof⟩

lemma find-None:
  find a xs = None ⇒ Some a ∉ set (map (map-option priority) xs)
⟨proof⟩

lemma find-exist:
  find a xs = Some t ⇒ Some t ∈ set xs
⟨proof⟩

definition find-min :: ('a::linorder, 'b) binqueue ⇒ ('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 (Some None ≠ xs) = find-min xs
⟨proof⟩

lemma find-min-single:
  find-min [x] = x
⟨proof⟩

lemma min-eq-find-min-None:
  min xs = None ↔ find-min xs = None
⟨proof⟩

lemma min-eq-find-min-Some:
  min xs = Some a ↔ (∃ t. find-min xs = Some t ∧ priority t = a)
⟨proof⟩

lemma find-min-exist:
  assumes find-min xs = Some t
  shows Some t ∈ set xs
⟨proof⟩
**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)  
⟨proof⟩

**Lemma** normalized-find-min-exists:  
normalized xs ⇒ xs ≠ [] ⇒ ∃ t. find-min xs = Some t  
⟨proof⟩

**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 [] = []  
⟨proof⟩

**Lemma** delete-min-nonempty [simp]:  
normalized xs ⇒ xs ≠ [] ⇒ find-min xs = Some t  
⇒ delete-min xs = normalize  
(meld (map Some (rev (children t))) (map (match (priority t)) xs))  
⟨proof⟩

**Lemma** is-binqueue-delete-min:  
assumes is-binqueue 0 xs  
shows is-binqueue 0 (delete-min xs)  
⟨proof⟩

**Lemma** normalized-delete-min:  
normalized (delete-min xs)  
⟨proof⟩

Dedicated grand unified operation for generated program

definition
meld' :: ('a, 'b) binqueue option ⇒ ('a::linorder, 'b) binqueue
⇒ ('a, 'b) binqueue ⇒ ('a, 'b) binqueue

where
meld' x y = add z (meld x y)

lemma [code]:
add z xs = meld' z [] xs
meld xs y = meld' None xs y

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 [] = meld' z []
meld' z [] (y # ys) = meld' None [z] (y # ys)
meld' (Some t) [] ys = meld' None [Some t] ys
meld' None [] = ys

Interface operations

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

definition
insert :: 'a::linorder ⇒ 'b ⇒ ('a, 'b) binqueue ⇒ ('a, 'b) binqueue

where
insert x y = add (Some (Node x y [])) y

lemma insert-simps [simp]:
insert x y [] = [Some (Node x y [])]
insert x y (None # xs) = Some (Node x y []) # xs
insert x y (Some t # xs) = add (Some (merge (Node x y [])) t) xs

lemma is-binqueue-insert:
is-binqueue 0 xs ⇒ is-binqueue 0 (insert x y)

⟨proof⟩
lemma normalized-insert:
  normalized xs ⇒ normalized (insert a v xs)
⟨proof⟩

definition pop :: ('a::linorder, 'b) binqueue ⇒ ('b × 'a) option × ('a, 'b) binqueue
where
  pop xs = (case find-min xs of
    None ⇒ (None, xs)
  | Some t ⇒ (Some (val t, priority t), delete-min xs))

lemma pop-empty [simp]:
  pop empty = (None, empty)
⟨proof⟩

lemma pop-nonempty [simp]:
  normalized xs ⇒ xs ≠ [] ⇒ find-min xs = Some t
  ⇒ pop xs = (Some (val t, priority t), normalize
    (meld (map Some (rev (children t))) (map (match (priority t)) xs))))
⟨proof⟩

lemma pop-code [code]:
  pop xs = (case find-min xs of
    None ⇒ (None, xs)
  | Some t ⇒ (Some (val t, priority t), normalize
    (meld (map Some (rev (children t))) (map (match (priority t)) xs))))
⟨proof⟩

3 Relating Functional Binomial Queues To The Abstract Priority Queues

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

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

primrec bt-dfs :: ('a::linorder, 'b) bintree ⇒ 'c ⇒ ('a, 'b) bintree ⇒ 'c list
and bts-dfs :: ('a::linorder, 'b) bintree ⇒ 'c ⇒ ('a, 'b) bintree list ⇒ 'c list
where
  bt-dfs f (Node a v ts) = f (Node a v ts) # bts-dfs f ts
  | bts-dfs f [] = []
| bts-dfs f (t # ts) = bt-dfs f t @ bts-dfs f ts

**lemma** bts-dfs-simp:
bt-dfs f t = f t # bts-dfs f (children t)
⟨proof⟩

**lemma** bts-dfs-append [simp]:
bts-dfs f (ts @ rs) = bts-dfs f ts @ bts-dfs f rs
⟨proof⟩

**lemma** set-bts-dfs-rev:
set (bts-dfs f (rev ts)) = set (bts-dfs f ts)
⟨proof⟩

**lemma** bts-dfs-rev-distinct:
distinct (bts-dfs f ts) ⇒ distinct (bts-dfs f (rev ts))
⟨proof⟩

**lemma** bts-dfs-comp:
bts-dfs (f ◦ g) t = map f (bt-dfs g t)
bts-dfs (f ◦ g) ts = map f (bts-dfs g ts)
⟨proof⟩

**lemma** bts-dfs-comp-distinct:
distinct (bt-dfs (f ◦ g) t) ⇒ distinct (bt-dfs g t)
distinct (bts-dfs (f ◦ g) ts) ⇒ distinct (bts-dfs g ts)
⟨proof⟩

**lemma** bts-dfs-distinct-children:
distinct (bt-dfs f x) ⇒ distinct (bts-dfs f (children x))
⟨proof⟩

**fun** dfs :: ('a::linorder, 'b) bintree ⇒ '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)
⟨proof⟩

**lemma** set-dfs-rev:
set (dfs f (rev xs)) = set (dfs f xs)
⟨proof⟩
lemma set-dfs-Cons:
\[
\text{set} \{(dfs \ f \ (x \ # \ xs))\} = \text{set} \((dfs \ f \ xs) \cup \text{set} \((dfs \ f \ [x])\)
\]
⟨proof⟩

lemma dfs-comp:
\[
dfs \ ((f \circ g) \ xs) = \text{map} \ f \ ((dfs \ g \ xs)
\]
⟨proof⟩

lemma dfs-comp-distinct:
\[
distinct \ ((dfs \ f \ g) \ xs) \implies \text{distinct} \ ((dfs \ g \ xs)
\]
⟨proof⟩

lemma dfs-distinct-member:
\[
distinct \ ((dfs \ f \ xs)) \implies
Some \ x \in \text{set} \ xs \implies
\text{distinct} \ ((bt-dfs \ f \ x)
\]
⟨proof⟩

lemma dfs-map-Some-idem:
\[
dfs \ f \ ((\text{map} \ Some \ xs) = \text{bts-dfs} \ f \ xs
\]
⟨proof⟩

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

lemma alist-split-pre:
\[
\text{val} \ t = (\text{fst} \circ \text{alist}) \ t
\]
\[
\text{priority} \ t = (\text{snd} \circ \text{alist}) \ t
\]
⟨proof⟩

lemma alist-split:
\[
\text{val} = \text{fst} \circ \text{alist}
\]
\[
\text{priority} = \text{snd} \circ \text{alist}
\]
⟨proof⟩

lemma alist-split-set:
\[
\text{set} \ ((dfs \ \text{val} \ xs) = \text{fst} \ ' \ \text{set} \ ((dfs \ \text{alist} \ xs)
\]
\[
\text{set} \ ((dfs \ \text{priority} \ xs) = \text{snd} \ ' \ \text{set} \ ((dfs \ \text{alist} \ xs)
\]
⟨proof⟩

lemma in-set-in-alist:
\[
\text{assumes} \ Some \ t \in \text{set} \ xs
\]
\[
\text{shows} \ ((\text{val} \ t, \text{priority} \ t) \in \text{set} \ ((dfs \ \text{alist} \ xs)
\]
⟨proof⟩
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⟩

lemma bt-augment-Node [simp]:
  bt-augment (Node a v ts) q = PQ.push v a (fold bt-augment ts q)
⟨proof⟩

lemma bt-augment-simp:
  bt-augment t q = PQ.push (val t) (priority t) (fold bt-augment (children t) q)
⟨proof⟩

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|
⟨proof⟩

lemma bt-augment-v-in:
  v ∈ set |q| ⇒ v ∈ set |bt-augment t q|
  v ∈ set |q| ⇒ v ∈ set |bts-augment ts q|
⟨proof⟩

lemma bt-augment-v-union:
  set |bt-augment t (bt-augment r q)| =
  set |bt-augment t q| ∪ set |bt-augment r q|
\[
\begin{align*}
\text{set } &|\text{bt-augment ts (bt-augment r q)}| = \\
&\text{set } |\text{bt-augment ts q}| \cup \text{set } |\text{bt-augment r q}|
\end{align*}
\]

\text{lemma } \text{bt-val-augment:}
\text{shows } \text{set } (\text{bt-dfs val t}) \cup \text{set } |q| = \text{set } |\text{bt-augment t q}|
\text{and } \text{set } (\text{bts-dfs val ts}) \cup \text{set } |q| = \text{set } |\text{bt-augment ts q}|

\text{lemma } \text{vals-pqueue:}
\text{set } (\text{vals xs}) = \text{set } |\text{pqueue xs}|

\text{lemma } \text{bt-augment-v-push:}
\text{set } |\text{bt-augment t (PQ.push v a q)}| = \text{set } |\text{bt-augment t q}| \cup \{v\}
\text{set } |\text{bts-augment ts (PQ.push v a q)}| = \text{set } |\text{bts-augment ts q}| \cup \{v\}

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

\text{lemma } \text{bts-augment-v-union:}
\text{set } |\text{bt-augment t (bts-augment rs q)}| = \\
\text{set } |\text{bt-augment t q}| \cup \text{set } |\text{bts-augment rs q}|
\text{set } |\text{bts-augment ts (bts-augment rs q)}| = \\
\text{set } |\text{bts-augment ts q}| \cup \text{set } |\text{bts-augment rs q}|

\text{lemma } \text{bt-augment-v-commute:}
\text{set } |\text{bt-augment t (bt-augment r q)}| = \text{set } |\text{bt-augment r (bt-augment t q)}|
\text{set } |\text{bt-augment t (bts-augment rs q)}| = \text{set } |\text{bts-augment rs (bt-augment t q)}|
\text{set } |\text{bts-augment ts (bts-augment rs q)}| = \\
\text{set } |\text{bts-augment rs (bts-augment ts q)}|

\text{lemma } \text{bt-augment-v-merge:}
\text{set } |\text{bt-augment (merge t r) q}| = \text{set } |\text{bt-augment t (bt-augment r q)}|

\text{lemma } \text{vals-merge [simp]:}
\text{set } (\text{bt-dfs val (merge t r)}) = \text{set } (\text{bt-dfs val t}) \cup \text{set } (\text{bt-dfs val r})
**Lemma vals-merge-distinct:**
\[
\text{distinct} (\text{bt-dfs val } t) \implies \text{distinct} (\text{bt-dfs val } r) \implies \\
\text{set} (\text{bt-dfs val } t) \cap \text{set} (\text{bt-dfs val } r) = \{\} \implies \\
\text{distinct} (\text{bt-dfs val } (\text{merge } t r))
\]
⟨proof⟩

**Lemma vals-add-Cons:**
\[
\text{set} (\text{vals } (\text{add } x xs)) = \text{set} (\text{vals } (x \# xs))
\]
⟨proof⟩

**Lemma vals-add-distinct:**
\[
\text{assumes } \text{distinct } (\text{vals } xs) \text{ and } \text{distinct } (\text{dfs val } [x]) \text{ and } \\
\text{set} (\text{vals } xs) \cap \text{set} (\text{dfs val } [x]) = \{\} \text{ shows } \text{distinct } (\text{vals } (\text{add } x xs))
\]
⟨proof⟩

**Lemma vals-insert [simp]:**
\[
\text{set} (\text{vals } (\text{insert } a v xs)) = \text{set} (\text{vals } xs) \cup \{v\}
\]
⟨proof⟩

**Lemma insert-v-push:**
\[
\text{set} (\text{vals } (\text{insert } a v xs)) = \text{set } |PQ.\text{push } v a (pqueue xs)|
\]
⟨proof⟩

**Lemma vals-meld:**
\[
\text{set} (\text{dfs val } (\text{meld } xs ys)) = \text{set} (\text{dfs val } xs) \cup \text{set} (\text{dfs val } ys)
\]
⟨proof⟩

**Lemma vals-meld-distinct:**
\[
\text{distinct } (\text{dfs val } xs) \implies \text{distinct } (\text{dfs val } ys) \implies \\
\text{set} (\text{dfs val } xs) \cap \text{set} (\text{dfs val } ys) = \{\} \implies \\
\text{distinct } (\text{dfs val } (\text{meld } xs ys))
\]
⟨proof⟩

**Lemma bt-augment-alist-subset:**
\[
\text{set } (PQ.\text{alist-of } q) \subseteq \text{set } (PQ.\text{alist-of } (\text{bt-augment } t q)) \\
\text{set } (PQ.\text{alist-of } q) \subseteq \text{set } (PQ.\text{alist-of } (\text{bts-augment } ts q))
\]
⟨proof⟩

**Lemma bt-augment-alist-in:**
\[
(v,a) \in \text{set } (PQ.\text{alist-of } q) \implies (v,a) \in \text{set } (PQ.\text{alist-of } (\text{bt-augment } t q)) \\
(v,a) \in \text{set } (PQ.\text{alist-of } q) \implies (v,a) \in \text{set } (PQ.\text{alist-of } (\text{bts-augment } ts q))
\]
⟨proof⟩
lemma bt-augment-alist-union:
\[
\begin{aligned}
\text{distinct } \big( \text{bts-dfs val } (r \neq [t]) \big) \implies \\
\text{set } \big( \text{bts-dfs val } (r \neq [t]) \big) \cap \text{set } |q| = \{\} \implies \\
\text{set } \big( \text{PQ-alist-of } \big( \text{bt-augment } t \ (\text{bt-augment } r \ q) \big) \big) = \\
\text{set } \big( \text{PQ-alist-of } \big( \text{bt-augment } t \ q \big) \big) \cup \text{set } \big( \text{PQ-alist-of } \big( \text{bt-augment } r \ q \big) \big)
\end{aligned}
\]

lemma bt-alist-augment:
\[
\begin{aligned}
\text{distinct } \big( \text{bts-dfs val } ts \big) \implies \\
\text{set } \big( \text{bts-dfs val } ts \big) \cap \text{set } |q| = \{\} \implies \\
\text{set } \big( \text{PQ-alist-of } \big( \text{bt-augment } ts \ (\text{bt-augment } r \ q) \big) \big) = \\
\text{set } \big( \text{PQ-alist-of } \big( \text{bt-augment } ts \ q \big) \big) \cup \text{set } \big( \text{PQ-alist-of } \big( \text{bt-augment } r \ q \big) \big)
\end{aligned}
\]

⟨proof⟩

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

⟨proof⟩

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

⟨proof⟩

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

⟨proof⟩

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

⟨proof⟩

lemma alist-add-Cons:
\[
\begin{aligned}
\text{assumes } \text{distinct } (\text{vals } (x \# xs)) \\
\text{shows } \text{set } \big( \text{dfs } \text{alist } (\text{add } x \ xs) \big) = \text{set } \big( \text{dfs } \text{alist } (x \ # \ xs) \big)
\end{aligned}
\]
lemma alist-insert [simp]:
\[
\text{distinct (vals xs) } \implies \\
v \notin \text{set (vals xs)} \implies \\
\text{set (dfs alist (insert a v xs))} = \text{set (dfs alist xs) } \cup \{(v,a)\}
\]
⟨proof⟩

lemma insert-push:
\[
\text{distinct (vals xs) } \implies \\
v \notin \text{set (vals xs)} \implies \\
\text{set (dfs alist (insert a v xs))} = \text{set (PQ.alist-of (PQ.push v a (pqueue xs)))}
\]
⟨proof⟩

lemma insert-p-push:
\[
\text{assumes distinct (vals xs)} \\
\text{and } v \notin \text{set (vals xs)} \\
\text{shows set (prios (insert a v xs))} = \text{set (PQ.push v a (pqueue xs))}
\]
⟨proof⟩

lemma empty-empty:
\[
\text{normalized xs } \implies \text{xs } = \text{empty } \iff \text{PQ.is-empty (pqueue xs)}
\]
⟨proof⟩

lemma bt-dfs-Min-priority:
\[
\text{assumes is-heap t} \\
\text{shows priority t } = \text{Min (set (bt-dfs priority t))}
\]
⟨proof⟩

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

lemma min-p-min:
\[
\text{assumes is-binqueue l xs} \\
\text{and xs } \neq [] \\
\text{and normalized xs} \\
\text{and distinct (vals xs)} \\
\text{and distinct (prios xs)} \\
\text{shows min xs } = \text{PQ.priority (pqueue xs) (PQ.min (pqueue xs))}
\]
⟨proof⟩
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⟩

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⟩

lemma alist-normalize-idem:
  dfs alist (normalize xs) = dfs alist xs
⟨proof⟩

lemma dfs-match-not-in:
  (∀ t. Some t ∈ set xs → priority t ≠ a) →
  set (dfs f (map (match a) xs)) = set (dfs f xs)
⟨proof⟩

lemma dfs-match-subset:
  set (dfs f (map (match a) xs)) ⊆ set (dfs f xs)
⟨proof⟩

lemma dfs-match-distinct:
  distinct (dfs f xs) → distinct (dfs f (map (match a) xs))
⟨proof⟩

lemma dfs-match:
  distinct (prios xs) →
  distinct (dfs f xs) →
  Some t ∈ set xs →
  priority t = a →
  set (dfs f (map (match a) xs)) = set (dfs f xs) − set (bt-dfs f t)
⟨proof⟩

lemma alist-meld:
distinct (dfs val xs) \Rightarrow distinct (dfs val ys) \Rightarrow
set (dfs val xs) \cap set (dfs val ys) = \{\} \Rightarrow
set (dfs alist (meld xs ys)) = set (dfs alist xs) \cup set (dfs alist ys)
\langle proof \rangle

\text{lemma \ alost-delete-min:}
\text{assumes \ distinct (vals xs)}
\text{and \ distinct (prios xs)}
\text{and \ find-min xs = Some (Node a v ts)}
\text{shows \ set (dfs alist (delete-min xs)) = set (dfs alist xs) - \{(v, a)\}}
\langle proof \rangle

\text{lemma \ alost-remove-min:}
\text{assumes \ is-binqueue l xs}
\text{and \ distinct (vals xs)}
\text{and \ distinct (prios xs)}
\text{and \ normalized xs}
\text{and \ xs \neq []}
\text{shows \ set (dfs alist (delete-min xs)) =}
\text{set (PQ.alist-of (PQ.remove-min (pqqueue xs)))}
\langle proof \rangle

\text{no-notation}
\text{PQ.values (\{(-)\})}
\text{and \ PQ.priorities (\{\parallel(-)\\})}