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

morphisms alist-of Abs-pq
⟨proof⟩

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

lemma [code abstype]:
\[\text{Abs-pq } (\text{alist-of } q) = q\]
⟨proof⟩

lemma distinct-fst-alist-of [simp]:
\[\text{distinct } (\text{map fst } (\text{alist-of } q))\]
⟨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::linorder, 'b::linorder) pq ⇒ 'a list | '-' |
  where
  values q = map fst (alist-of q)

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

lemma values-set:
  set |q| = fst | set (alist-of q)
  ⟨proof⟩

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

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

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

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

definition empty :: ('a::linorder, '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 ≠ []
lemma alist-of-empty [simp, code abstract]:
alist-of empty = []

lemma values-empty [simp]:
\{empty\} = []

lemma priorities-empty [simp]:
\parallel empty\parallel = []

lemma values-empty-nothing [simp]:
\forall k. k \notin set \{empty\}

lemma is-empty-empty:
is-empty q \iff q = empty

lemma is-empty-empty-simp [simp]:
is-empty empty

lemma map-snd-alist-of:
map (the o priority q) (values q) = map snd (alist-of q)

lemma image-snd-alist-of:
the ' priority q ' set (values q) = snd ' set (alist-of q)

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

lemma priority-fst:
assumes xp \in set (alist-of q)
shows priority q (fst xp) = Some (snd xp)

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

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

definition push :: 'a ⇒ 'b::linorder ⇒ (\'a, \'b) pq ⇒ (\'a, \'b) pq where
push k p q = Abs-pq (if k \notin \text{set } (\text{values } q)
then \text{insort-key } \text{snd } (k, p) \ (\text{alist-of } q)
else \text{alist-of } q)

lemma Min-snd-hd:
\( q \neq [] \implies \text{sorted } (\text{map } \text{snd } q) \implies \text{Min } (\text{set } \text{snd } q) = \text{snd } (\text{hd } q) \)
⟨proof⟩

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⟩

lemma not-in-first-image:
x \notin \text{fst } s \implies (x, p) \notin s
⟨proof⟩

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

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

lemma push-priorities [simp]:
k \notin \text{set } q \implies set \ |push k p q| = set \ |q| \cup \{p\}
k \in \text{set } q \implies set \ |push k p q| = set \ |q|
⟨proof⟩

lemma not-is-empty-push [simp]:
\( \neg \text{is-empty } (\text{push } k \ p \ q) \)
⟨proof⟩

4
lemma push-commute:
 assumes a ≠ b and v ≠ w
 shows push w b (push v a q) = push v a (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) − {{min q, the (priority q (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 (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)
 ⟨proof⟩

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 \( x \in X \). P x \( \Rightarrow P (\text{None} # x) \)
  and \( x \in X \). P x \( \Rightarrow P (\text{Some} x # x) \)
  shows P x
⟨proof⟩

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 \( \Rightarrow is-bintree-list l (\text{children} t) \)
  \( \Rightarrow is-bintree-list (\text{Suc} l) (t # ts) \)

abbreviation (input) is-bintree k t \( \equiv is-bintree-list k (\text{children} t) \)
lemma is-bintree-list-triv [simp]:
  is-bintree-list 0 ts \iff ts = []
  is-bintree-list l [] \iff l = 0
⟨proof⟩

lemma is-bintree-list-simp [simp]:
  is-bintree-list (Suc l) (t \# ts) \iff
  is-bintree-list l (children t) \land is-bintree-list l ts
⟨proof⟩

lemma is-bintree-list-length [simp]:
  is-bintree-list l ts \implies length ts = l
⟨proof⟩

lemma is-bintree-list-children-last:
  assumes is-bintree-list l ts and ts \neq []
  shows children (last ts) = []
⟨proof⟩

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

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 \implies is-heap-list (priority t) (children t)
  \implies (priority t) \geq h \implies 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 \land is-heap-list (priority t) (children t) \land priority t \geq h
⟨proof⟩

lemma is-heap-list-append-dest [dest]:
  is-heap-list l (ts@rs) \implies is-heap-list l ts
  is-heap-list l (ts@rs) \implies is-heap-list l rs
⟨proof⟩

lemma is-heap-list-rev:
\[ \text{is-heap-list } l \, \text{ts} \implies \text{is-heap-list } l \, (\text{rev} \, \text{ts}) \]

\begin{proof}
\end{proof}

**Lemma**: \text{is-heap-children-larger}:
\[ \text{is-heap } t \implies \forall \, x \in \text{set} (\text{children } t), \, \text{priority } x \geq \text{priority } t \]
\begin{proof}
\end{proof}

**Lemma**: \text{is-heap-Min-children-larger}:
\[ \text{is-heap } t \implies \text{children } t \neq [] \implies \text{priority } t \leq \text{Min} (\text{priority' set} (\text{children } t)) \]
\begin{proof}
\end{proof}

Combination of both: binqueue property

**Inductive**: \text{is-binqueue} :: nat \Rightarrow (\text{'a::linorder}, \ 'b) \text{binqueue} \Rightarrow \text{bool}
\begin{itemize}
  \item \text{Empty}: \text{is-binqueue} l []
  \item \text{None}: \text{is-binqueue} (\text{Suc } l) \, \text{xs} \implies \text{is-binqueue } l \, (\text{None } \# \text{xs})
  \item \text{Some}: \text{is-binqueue} (\text{Suc } l) \, \text{xs} \implies \text{is-bintree } l \, t
\end{itemize}
\[ \implies \text{is-heap } t \implies \text{is-binqueue } l \, (\text{Some } t \# \text{xs}) \]
\begin{proof}
\end{proof}

**Lemma**: \text{is-binqueue-simp} \ [\text{simp}]:
\[ \text{is-binqueue } l \, [] \iff \text{True} \]
\[ \text{is-binqueue } l \, (\text{Some } t \# \text{xs}) \iff \text{is-bintree } l \, t \land \text{is-heap } l \land \text{is-binqueue} (\text{Suc } l) \, \text{xs} \]
\[ \text{is-binqueue } l \, (\text{None } \# \text{xs}) \iff \text{is-binqueue} (\text{Suc } l) \, \text{xs} \]
\begin{proof}
\end{proof}

**Lemma**: \text{is-binqueue-trans}:
\[ \text{is-binqueue } l \, (x \# \text{xs}) \implies \text{is-binqueue} (\text{Suc } l) \, \text{xs} \]
\begin{proof}
\end{proof}

**Lemma**: \text{is-binqueue-head}:
\[ \text{is-binqueue } l \, (x \# \text{xs}) \implies \text{is-binqueue } l \, [x] \]
\begin{proof}
\end{proof}

**Lemma**: \text{is-binqueue-append}:
\[ \text{is-binqueue } l \, \text{xs} \implies \text{is-binqueue} (\text{length } \text{xs} + l) \, \text{ys} \implies \text{is-binqueue } l \, (\text{xs } \@ \text{ys}) \]
\begin{proof}
\end{proof}

**Lemma**: \text{is-binqueue-append-dest} \ [\text{dest}]:
\[ \text{is-binqueue } l \, (\text{xs } \@ \text{ys}) \implies \text{is-binqueue } l \, \text{xs} \]
\begin{proof}
\end{proof}

**Lemma**: \text{is-binqueue-children}:
\[ \text{assumes } \text{is-bintree-list } l \, \text{ts} \]
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) bintree ⇒ ('a, 'b) bintree
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 \( l \text{ } xs \)
shows is-binqueue \( l \text{ } (\text{add None } xs)\)
⟨proof⟩

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

function meld :: \( \langle \forall a : \text{linorder}, \forall b \text{ binqueue} \Rightarrow \langle a, b \rangle \text{ binqueue} \Rightarrow \langle a, b \rangle \text{ binqueue} \rangle \)
where
meld \( \langle \rangle \text{ } ys = ys \)
| meld \( xs \text{ } \langle \rangle = xs \)
| meld \( \langle \text{None} \# xs \rangle \text{ } \langle y \# ys \rangle = y \# \text{meld } xs \text{ } ys \)
| meld \( \langle x \# xs \rangle \text{ } \langle \text{None} \# ys \rangle = x \# \text{meld } xs \text{ } ys \)
| meld \( \langle \text{Some } t \# xs \rangle \text{ } \langle \text{Some } r \# ys \rangle = \)
\quad None \# \text{add } \langle \text{Some } \langle \text{merge } t \text{ } r \rangle \rangle \text{ } \langle \text{meld } xs \text{ } ys \rangle \)
⟨proof⟩ termination ⟨proof⟩

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

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

lemma nonempty-meld-commute:
meld \( xs \text{ } ys \neq \langle \rangle \implies \text{meld } xs \text{ } ys \neq \langle \rangle \)
⟨proof⟩

lemma is-binqueue-meld:
assumes is-binqueue \( l \text{ } xs \)
and is-binqueue \( l \text{ } ys \)
shows is-binqueue \( l \text{ } (\text{meld } xs \text{ } ys)\)
⟨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:
\[ \text{min} \ [x] = \text{Some} \ a \implies \text{priority} \ (\text{the} \ x) = a \]
\[ \text{min} \ [x] = \text{None} \implies x = \text{None} \]
\langle proof \rangle

lemma min-Some-not-None:
\[ \text{min} \ (\text{Some} \ t \neq xs) \neq \text{None} \]
\langle proof \rangle

lemma min-None-trans:
assumes \( \text{min} \ (x \# xs) = \text{None} \)
shows \( \text{min} \ xs = \text{None} \)
\langle proof \rangle

lemma min-None-None:
\[ \text{min} \ xs = \text{None} \iff xs = [] \lor \text{set} \ xs = \{\text{None}\} \]
\langle proof \rangle

lemma normalized-min-not-None:
\[ \text{normalized} \ xs \implies xs \neq [] \implies \text{min} \ xs \neq \text{None} \]
\langle proof \rangle

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

lemma min-exists:
assumes \( \text{min} \ xs = \text{Some} \ a \)
shows \( \text{Some} \ a \in \text{map}-\text{option} \ \text{priority} \ \cdot \ \text{set} \ xs \)
\langle proof \rangle

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

declare find.simps [simp del]

lemma find-simps [simp, code]:
\[ \text{find} \ a [] = \text{None} \]
\[ \text{find} \ a (\text{None} \# xs) = \text{find} \ a \ xs \]
\begin{verbatim}
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) bqueue ⇒ (′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
(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)
\end{verbatim}
**Lemma** `find-min-is-min`:

**Assumes**

- `normalized xs`  
- `xs ≠ []`  
- `find-min xs = Some t`

**Shows**

\[ ∀ x ∈ \text{set } xs. \ x = \text{None} ∨ (\text{priority } t) ≤ \text{priority } (\text{the } x) \]

(Proof)

**Lemma** `normalized-find-min-exists`:

\[ \text{normalized } xs ⇒ xs ≠ [] ⇒ \exists t. \ \text{find-min } xs = \text{Some } t \]

(Proof)

**Primrec**

**Match**

\[ \text{match } :: \ ⟦ \text{a:linorder} ⇒ (\text{a, 'b) bintree option} ⇒ (\text{a, 'b) bintree option} \]

**Where**

- `match a None = None`
- `match a (Some t) = (\text{if priority } t = a \text{ then None else Some } t)`

**Definition** `delete-min`::

\[ \text{delete-min } :: \ ⟦ \text{a:linorder, 'b) binqueue ⇒ (\text{a, 'b) binqueue} \]

**Where**

- `delete-min xs = (\text{case } \text{find-min } xs`
- `\text{of Some } (\text{Node } a \ v \ ts) ⇒ `  
- `\text{normalize } (\text{meld } (\text{map Some } (\text{rev } ts)) \ (\text{map } (\text{match } a) \ xs))`
- `\text{None ⇒ []})`

**Lemma** `delete-min-empty` [simp]:

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

(Proof)

**Lemma** `delete-min-nonempty` [simp]:

\[ \text{normalized } xs ⇒ xs ≠ [] ⇒ \text{find-min } xs = \text{Some } t \]

\[ ⇒ \text{delete-min } xs = \text{normalize} \]

\[ (\text{meld } (\text{map Some } (\text{rev } (\text{children } t))) \ (\text{map } (\text{match } (\text{priority } t)) \ xs)) \]

(Proof)

**Lemma** `is-binqueue-delete-min`:

**Assumes**

- `is-binqueue 0 xs`

**Shows**

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

(Proof)

**Lemma** `normalized-delete-min`:

\[ \text{normalized } (\text{delete-min } xs) \]

(Proof)

Dedicated grand unified operation for generated program

**Definition**
meld' :: ('a, 'b) bintree option ⇒ ('a::linorder, 'b) binqueue ⇒ ('a, 'b) binqueue ⇒ ('a, 'b) binqueue

where
meld' z xs ys = add z (meld xs ys)

lemma [code]:
  add z xs = meld' z [] xs
  meld xs ys = meld' None xs ys
  ⟨proof⟩

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
  ⟨proof⟩

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
  ⟨proof⟩

lemma is-binqueue-insert:
is-binqueue 0 xs ⇒ is-binqueue 0 (insert a v xs)
  ⟨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) # bt-dfs f ts
  | bts-dfs f [] = []
\[ \text{bts-dfs } f \ (t \ # \ ts) = \text{bt-dfs } f \ t \ # \ \text{bts-dfs } f \ ts \]

**lemma** \text{bts-dfs-simp}:
\[ \text{bt-dfs } f \ t = f \ t \ # \ \text{bts-dfs } f \ (\text{children } t) \]
\langle proof \rangle

**lemma** \text{bts-dfs-append } [simp]:
\[ \text{bts-dfs } f \ (ts \ @ \ rs) = \text{bts-dfs } f \ ts \ @ \ \text{bts-dfs } f \ rs \]
\langle proof \rangle

**lemma** \text{set-bts-dfs-rev}:
\[ \text{set } (\text{bts-dfs } f \ (\text{rev } ts)) = \text{set } (\text{bts-dfs } f \ ts) \]
\langle proof \rangle

**lemma** \text{bts-dfs-rev-distinct}:
\[ \text{distinct } (\text{bts-dfs } f \ ts) \implies \text{distinct } (\text{bts-dfs } f \ (\text{rev } ts)) \]
\langle proof \rangle

**lemma** \text{bt-dfs-comp}:
\[ \text{bt-dfs } (f \circ g) \ t = \text{map } f \ (\text{bt-dfs } g \ t) \]
\[ \text{bts-dfs } (f \circ g) \ ts = \text{map } f \ (\text{bts-dfs } g \ ts) \]
\langle proof \rangle

**lemma** \text{bt-dfs-comp-distinct}:
\[ \text{distinct } (\text{bt-dfs } (f \circ g) \ t) \implies \text{distinct } (\text{bt-dfs } g \ t) \]
\[ \text{distinct } (\text{bts-dfs } (f \circ g) \ ts) \implies \text{distinct } (\text{bts-dfs } g \ ts) \]
\langle proof \rangle

**lemma** \text{bt-dfs-distinct-children}:
\[ \text{distinct } (\text{bt-dfs } f \ x) \implies \text{distinct } (\text{bts-dfs } f \ (\text{children } x)) \]
\langle proof \rangle

**fun** dfs :: \((\text{a::inorder, } \text{b} \ \text{bintree} \Rightarrow \text{'}c) \Rightarrow \text{'}a, \text{'}b \ \text{bintree} \Rightarrow \text{'}c \ \text{list}) \text{ where}
\[ \text{dfs } f \ [\] = [] \]
\[ | \text{dfs } f \ (\text{None } \# \ xs) = \text{dfs } f \ xs \]
\[ | \text{dfs } f \ (\text{Some } t \ # \ xs) = \text{bt-dfs } f \ t \ @ \ \text{dfs } f \ xs \]

**lemma** dfs-append:
\[ \text{dfs } f \ (xs \ @ \ ys) = (\text{dfs } f \ xs) \ @ \ (\text{dfs } f \ ys) \]
\langle proof \rangle

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

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

lemma dfs-comp-distinct:
\[ \text{distinct} \ (\text{dfs} \ (f \circ g) \ xs) \implies \text{distinct} \ (\text{dfs} \ g \ xs) \]
\langle proof \rangle

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

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

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

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

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

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

lemma in-set-in-alist:
\[ \text{assumes} \ \text{Some} \ t \in \text{set} \ xs \]
\[ \text{shows} \ (\text{val} \ t, \text{priority} \ t) \in \text{set} \ (\text{dfs} \ \text{alist} \ xs) \]
\langle proof \rangle
abbreviation vals where vals ≡ dfs val
abbreviation prios where prios ≡ dfs priority
abbreviation elements where elements ≡ dfs alist

primrec
   bt-augment :: (′a::linorder, ′b) bintree ⇒ (′b, ′a) PQ.pq
and
   bts-augment :: (′a::linorder, ′b) bintree list ⇒ (′b, ′a) PQ.pq ⇒ (′b, ′a) PQ.pq
where
   bt-augment (Node a v ts) q = PQ.push v a (bts-augment ts q)
| bts-augment [] q = q
| bts-augment (t # ts) q = bts-augment ts (bt-augment t q)

lemma bts-augment [simp]:
   bts-augment = fold bt-augment
⟨proof⟩

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|
⟨proof⟩

lemma bt-augment-v-in:
   v ∈ set |q| ⇒ v ∈ set |bt-augment t 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|

20
set \{bt-augment ts (bt-augment r q)\} = 
set \{bt-augment ts q\} \cup set \{bt-augment r q\}
⟨proof⟩

**lemma** bt-val-augment:
**shows** set (bt-dfs val t) \cup set \{q\} = set \{bt-augment t q\}
**and** set (bts-dfs val ts) \cup set \{q\} = set \{bts-augment ts q\}
⟨proof⟩

**lemma** vals-pqueue:
set (vals xs) = set \{pqueue xs\}
⟨proof⟩

**lemma** bt-augment-v-push:
set \{bt-augment t \(PQ\).push v a q\} = set \{bt-augment t q\} \cup \{v\}
set \{bts-augment ts \(PQ\).push v a q\} = set \{bts-augment ts q\} \cup \{v\}
⟨proof⟩

**lemma** bt-augment-v-push-commute:
set \{bt-augment t \(PQ\).push v a q\} = set \(PQ\).push v a (bt-augment t q)\}
set \{bts-augment ts \(PQ\).push v a q\} = set \(PQ\).push v a (bts-augment ts q)\}
⟨proof⟩

**lemma** bts-augment-v-union:
set \{bt-augment t (bts-augment rs q)\} = 
set \{bt-augment t q\} \cup set \{bts-augment rs q\}
set \{bts-augment ts (bts-augment rs q)\} = 
set \{bts-augment ts q\} \cup set \{bts-augment rs q\}
⟨proof⟩

**lemma** bt-augment-v-commute:
set \{bt-augment t (bt-augment r q)\} = set \{bt-augment r (bt-augment t q)\}
set \{bt-augment t (bts-augment rs q)\} = set \{bts-augment rs (bt-augment t q)\}
set \{bts-augment ts (bts-augment rs q)\} = 
set \{bts-augment rs (bts-augment ts q)\}
⟨proof⟩

**lemma** bt-augment-v-merge:
set \{bt-augment (merge t r) q\} = set \{bt-augment t (bt-augment r q)\}
⟨proof⟩

**lemma** vals-merge [simp]:
set (bt-dfs val (merge t r)) = set (bt-dfs val t) \cup set (bt-dfs val r)
⟨proof⟩

21
lemma vals-merge-distinct:
distinct (bt-dfs val t) \implies\ distinct (bt-dfs val r) \implies
set (bt-dfs val t) \cap set (bt-dfs val r) = \{\} \implies
distinct (bt-dfs val (merge t r))
⟨proof⟩

lemma vals-add-Cons:
set (vals (add x xs)) = set (vals (x # xs))
⟨proof⟩

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

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

lemma insert-v-push:
set (vals (insert a v xs)) = set |PQ.push v a (pqueue xs)|
⟨proof⟩

lemma vals-meld:
set (dfs val (meld xs ys)) = set (dfs val xs) \cup set (dfs val ys)
⟨proof⟩

lemma vals-meld-distinct:
distinct (dfs val xs) \implies\ distinct (dfs val ys) \implies
set (dfs val xs) \cap set (dfs val ys) = \{\} \implies
distinct (dfs val (meld xs ys))
⟨proof⟩

lemma bt-augment-alist-subset:
set (PQ.alist-of q) \subseteq set (PQ.alist-of (bt-augment t q))
set (PQ.alist-of q) \subseteq set (PQ.alist-of (bts-augment ts q))
⟨proof⟩

lemma bt-augment-alist-in:
(v,a) \in set (PQ.alist-of q) \implies (v,a) \in set (PQ.alist-of (bt-augment t q))
(v,a) \in set (PQ.alist-of q) \implies (v,a) \in set (PQ.alist-of (bts-augment ts q))
⟨proof⟩
\textbf{lemma} \textit{bt-augment-alist-union}: \\
\textit{distinct (\texttt{bts-dfs val (r \# [t])) \Rightarrow}} \\
\text{set (\texttt{bts-dfs val (r \# [t])) \cap \set q = {}} \Rightarrow \\
\text{set (\texttt{PQ.alist-of (bt-augment t (bt-augment r q))}) =} \\
\text{set (\texttt{PQ.alist-of (bt-augment t q)}) \cup set (PQ.alist-of (bt-augment r q))} \\

\text{distinct (\texttt{bts-dfs val (r \# ts)}) \Rightarrow} \\
\text{set (\texttt{bts-dfs val (r \# ts)}) \cap \set q = {}} \Rightarrow \\
\text{set (\texttt{PQ.alist-of (bt-augment ts (bt-augment r q))}) =} \\
\text{set (\texttt{PQ.alist-of (bt-augment ts q)}) \cup set (\texttt{PQ.alist-of (bt-augment r q)})} \\
\langle proof \rangle \\

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

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

\textbf{lemma} \textit{alist-pqueue}: \\
\textit{distinct (\texttt{vals xs}) \Rightarrow set (dfs alist xs) = set (PQ.alist-of (pqueue xs))} \\
\langle proof \rangle \\

\textbf{lemma} \textit{alist-pqueue-priority}: \\
\textit{distinct (\texttt{vals xs}) \Rightarrow (v, a) \in set (dfs alist xs) \Rightarrow PQ.priority (pqueue xs) v = Some a} \\
\langle proof \rangle \\

\textbf{lemma} \textit{prios-pqueue}: \\
\textit{distinct (\texttt{vals xs}) \Rightarrow set (prios xs) = set ||pqueue xs||} \\
\langle proof \rangle \\

\textbf{lemma} \textit{alist-merge [simp]}: \\
\textit{distinct (\texttt{bt-dfs val t}) \Rightarrow distinct (bt-dfs val r) \Rightarrow} \\
\text{set (\texttt{bt-dfs val t}) \cap set (\texttt{bt-dfs val r}) = {}} \Rightarrow \\
\text{set (\texttt{bt-dfs alist (merge t r)}) = set (\texttt{bt-dfs alist t}) \cup set (\texttt{bt-dfs alist r})} \\
\langle proof \rangle \\

\textbf{lemma} \textit{alist-add-Cons}: \\
\textbf{assumes} \textit{distinct (\texttt{vals (x\#xs)})} \\
\textbf{shows} set (\texttt{dfs alist (add x xs)}) = set (\texttt{dfs alist (x \# xs)})
\textbf{lemma \texttt{alist-insert}} [simp]:
\begin{align*}
distinct (\texttt{vals} \ \texttt{xs}) & \rightarrow \\
v \not\in \texttt{set} (\texttt{vals} \ \texttt{xs}) & \rightarrow \\
\texttt{set} (\texttt{dfs} \ \texttt{alist} \ (\texttt{insert} \ a \ v \ \texttt{xs})) & = \texttt{set} (\texttt{dfs} \ \texttt{alist} \ \texttt{xs}) \cup \{(v,a)\}
\end{align*}
\textit{proof}

\textbf{lemma \texttt{insert-push}}:
\begin{align*}
distinct (\texttt{vals} \ \texttt{xs}) & \rightarrow \\
v \not\in \texttt{set} (\texttt{vals} \ \texttt{xs}) & \rightarrow \\
\texttt{set} (\texttt{dfs} \ \texttt{alist} \ (\texttt{insert} \ a \ v \ \texttt{xs})) & = \texttt{set} (\texttt{PQ-alist-of} \ (\texttt{PQ-push} \ v \ a \ (\texttt{pqueue} \ \texttt{xs})))
\end{align*}
\textit{proof}

\textbf{lemma \texttt{insert-p-push}}:
\begin{align*}
\textbf{assumes} \ & \texttt{distinct} \ (\texttt{vals} \ \texttt{xs}) \\
\textbf{and} \ & v \not\in \texttt{set} \ (\texttt{vals} \ \texttt{xs}) \\
\textbf{shows} \ & \texttt{set} \ (\texttt{prios} \ (\texttt{insert} \ a \ v \ \texttt{xs})) = \texttt{set} \ (\texttt{PQ-push} \ v \ a \ (\texttt{pqueue} \ \texttt{xs}))
\end{align*}
\textit{proof}

\textbf{lemma \texttt{empty-empty}}:
\begin{align*}
\texttt{normalized} \ \texttt{xs} & \rightarrow \ \texttt{xs} = \texttt{empty} \leftarrow \ P\texttt{Q-is-empty} \ (\texttt{pqueue} \ \texttt{xs})
\end{align*}
\textit{proof}

\textbf{lemma \texttt{bt-dfs-Min-priority}}:
\begin{align*}
\textbf{assumes} \ & \texttt{is-heap} \ t \\
\textbf{shows} \ & \texttt{priority} \ t = \texttt{Min} \ (\texttt{set} \ (\texttt{bt-dfs} \ \texttt{priority} \ t))
\end{align*}
\textit{proof}

\textbf{lemma \texttt{is-binqueue-min-Min-prios}}:
\begin{align*}
\textbf{assumes} \ & \texttt{is-binqueue} \ l \ \texttt{xs} \\
\textbf{and} \ & \texttt{normalized} \ \texttt{xs} \\
\textbf{and} \ & \texttt{xs} \not= \texttt{[]} \\
\textbf{shows} \ & \texttt{min} \ \texttt{xs} = \texttt{Some} \ (\texttt{Min} \ (\texttt{set} \ \texttt{prios} \ \texttt{xs}))
\end{align*}
\textit{proof}

\textbf{lemma \texttt{min-p-min}}:
\begin{align*}
\textbf{assumes} \ & \texttt{is-binqueue} \ l \ \texttt{xs} \\
\textbf{and} \ & \texttt{xs} \not= \texttt{[]} \\
\textbf{and} \ & \texttt{normalized} \ \texttt{xs} \\
\textbf{and} \ & \texttt{distinct} \ (\texttt{vals} \ \texttt{xs}) \\
\textbf{and} \ & \texttt{distinct} \ (\texttt{prios} \ \texttt{xs}) \\
\textbf{shows} \ & \texttt{min} \ \texttt{xs} = \texttt{PQ-priority} \ (\texttt{pqueue} \ \texttt{xs}) \ (\texttt{PQ-min} \ (\texttt{pqueue} \ \texttt{xs}))
\end{align*}
\textit{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 (queue xs) (PQ.min (queue 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 (queue 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:
\[ \text{distinct (dfs val xs) } \implies \text{distinct (dfs val ys) } \implies \\
\text{set (dfs val xs) } \cap \text{set (dfs val ys) } = \{ \} \implies \\
\text{set (dfs alist (meld xs ys)) } = \text{set (dfs alist xs) } \cup \text{set (dfs alist ys)} \]

\langle \text{proof} \rangle

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

\textbf{lemma} \text{alist-remove-min}:
\begin{itemize}
\item \textbf{assumes} \text{is-binqueue l xs}
\item \textbf{and} \text{distinct (vals xs)}
\item \textbf{and} \text{distinct (prios xs)}
\item \textbf{and} \text{normalized xs}
\item \textbf{and} \text{xs } \neq \text{[]}\n\item \textbf{shows} \text{set (dfs alist (delete-min xs)) } = \\
\text{set (PQ-alist-of (PQ-remove-min (pqueue xs)))}
\end{itemize}
\langle \text{proof} \rangle

\textbf{no-notation}
\begin{itemize}
\item \textbf{PQ-values} \{(-)\}
\item \textbf{and} \textbf{PQ-priorities} \{\|(-)\|\}
\end{itemize}