

Verification of Functional Binomial Queues

René Neumann

Technische Universität München, Institut für Informatik
<http://www.in.tum.de/~neumannr/>

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

1 Abstract priority queues

1.1 Generic Lemmas

lemma *tl-set*:

$distinct\ q \implies set\ (tl\ q) = set\ q - \{hd\ q\}$
<proof>

1.2 Type of abstract priority queues

typedef (**overloaded**) (*'a, 'b::linorder*) *pq* =
 $\{xs :: ('a \times 'b)\ list.\ distinct\ (map\ fst\ xs) \wedge sorted\ (map\ snd\ xs)\}$
morphisms *alist-of Abs-pq*
<proof>

lemma *alist-of-Abs-pq*:

assumes $distinct\ (map\ fst\ xs)$
and $sorted\ (map\ snd\ xs)$
shows $alist-of\ (Abs-pq\ xs) = xs$
<proof>

lemma [*code abstype*]:

$Abs-pq\ (alist-of\ q) = q$
<proof>

lemma *distinct-fst-alist-of* [*simp*]:

$distinct\ (map\ fst\ (alist-of\ q))$
<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* \implies *p* = *q*
 ⟨*proof*⟩

definition *values* :: ('a, 'b::linorder) pq \Rightarrow 'a list (|(-)|) **where**
 values *q* = *map fst* (*alist-of* *q*)

definition *priorities* :: ('a, 'b::linorder) pq \Rightarrow '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, 'b::linorder) pq \Rightarrow bool **where**
 is-empty *q* \longleftrightarrow *alist-of* *q* = []

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

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

definition *empty* :: ('a, 'b::linorder) pq **where**
 empty = *Abs-pq* []

lemma *is-empty-alist-of* [*dest*]:
 is-empty *q* \implies *alist-of* *q* = []
 ⟨*proof*⟩

lemma *not-is-empty-alist-of* [*dest*]:
 \neg *is-empty* *q* \implies *alist-of* *q* \neq []

$\langle proof \rangle$

lemma *alist-of-empty* [*simp*, *code abstract*]:

$alist\ of\ empty = []$

$\langle proof \rangle$

lemma *values-empty* [*simp*]:

$|empty| = []$

$\langle proof \rangle$

lemma *priorities-empty* [*simp*]:

$||empty|| = []$

$\langle proof \rangle$

lemma *values-empty-nothing* [*simp*]:

$\forall k. k \notin set\ |empty|$

$\langle proof \rangle$

lemma *is-empty-empty*:

$is\ empty\ q \longleftrightarrow q = empty$

$\langle proof \rangle$

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

$is\ empty\ empty$

$\langle proof \rangle$

lemma *map-snd-alist-of*:

$map\ (the\ \circ\ priority\ q)\ (values\ q) = map\ snd\ (alist\ of\ q)$

$\langle proof \rangle$

lemma *image-snd-alist-of*:

$the\ 'priority\ q\ 'set\ (values\ q) = snd\ 'set\ (alist\ of\ q)$

$\langle proof \rangle$

lemma *Min-snd-alist-of*:

assumes $\neg is\ empty\ q$

shows $Min\ (snd\ 'set\ (alist\ of\ q)) = snd\ (hd\ (alist\ of\ q))$

$\langle proof \rangle$

lemma *priority-fst*:

assumes $xp \in set\ (alist\ of\ q)$

shows $priority\ q\ (fst\ xp) = Some\ (snd\ xp)$

$\langle proof \rangle$

lemma *priority-Min*:

assumes \neg *is-empty* q
shows $\text{priority } q (\text{min } q) = \text{Some } (\text{Min } (\text{the } \text{'priority } q \text{' set } (\text{values } q)))$
 $\langle \text{proof} \rangle$

lemma *priority-Min-priorities*:

assumes \neg *is-empty* q
shows $\text{priority } q (\text{min } q) = \text{Some } (\text{Min } (\text{set } \|q\|))$
 $\langle \text{proof} \rangle$

definition *push* :: $'a \Rightarrow 'b::\text{linorder} \Rightarrow ('a, 'b) \text{pq} \Rightarrow ('a, 'b) \text{pq}$ **where**

$\text{push } k \ p \ q = \text{Abs-pq}$ (if $k \notin \text{set } (\text{values } q)$
then $\text{insort-key snd } (k, p)$ (*alist-of* q)
else *alist-of* q)

lemma *Min-snd-hd*:

$q \neq [] \Longrightarrow \text{sorted } (\text{map } \text{snd } q) \Longrightarrow \text{Min } (\text{snd } \text{'set } q) = \text{snd } (\text{hd } q)$
 $\langle \text{proof} \rangle$

lemma *hd-construct*:

assumes \neg *is-empty* q
shows $\text{hd } (\text{alist-of } q) = (\text{min } q, \text{the } (\text{priority } q (\text{min } q)))$
 $\langle \text{proof} \rangle$

lemma *not-in-first-image*:

$x \notin \text{fst } \text{'s} \Longrightarrow (x, p) \notin s$
 $\langle \text{proof} \rangle$

lemma *alist-of-push* [*simp*, *code abstract*]:

$\text{alist-of } (\text{push } k \ p \ q) =$
(if $k \notin \text{set } (\text{values } q)$ then $\text{insort-key snd } (k, p)$ (*alist-of* q) else *alist-of* q)
 $\langle \text{proof} \rangle$

lemma *push-values* [*simp*]:

$\text{set } |\text{push } k \ p \ q| = \text{set } |q| \cup \{k\}$
 $\langle \text{proof} \rangle$

lemma *push-priorities* [*simp*]:

$k \notin \text{set } |q| \Longrightarrow \text{set } \|\text{push } k \ p \ q\| = \text{set } \|q\| \cup \{p\}$
 $k \in \text{set } |q| \Longrightarrow \text{set } \|\text{push } k \ p \ q\| = \text{set } \|q\|$
 $\langle \text{proof} \rangle$

lemma *not-is-empty-push* [*simp*]:

\neg *is-empty* ($\text{push } k \ p \ q$)
 $\langle \text{proof} \rangle$

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* :: $('a, 'b::\text{linorder}) \text{pq} \Rightarrow ('a, 'b::\text{linorder}) \text{pq}$ **where**

$\text{remove-min } q = (\text{if is-empty } q \text{ then empty else Abs-pq } (\text{tl } (\text{alist-of } q)))$

lemma *alift-of-remove-min-if* [*code abstract*]:

$\text{alist-of } (\text{remove-min } q) = (\text{if is-empty } q \text{ then } [] \text{ else } \text{tl } (\text{alist-of } q))$

<proof>

lemma *remove-min-empty* [*simp*]:

$\text{is-empty } q \Longrightarrow \text{remove-min } q = \text{empty}$

<proof>

lemma *alist-of-remove-min* [*simp*]:

$\neg \text{is-empty } q \Longrightarrow \text{alist-of } (\text{remove-min } q) = \text{tl } (\text{alist-of } q)$

<proof>

lemma *values-remove-min* [*simp*]:

$\neg \text{is-empty } q \Longrightarrow \text{values } (\text{remove-min } q) = \text{tl } (\text{values } q)$

<proof>

lemma *set-alist-of-remove-min*:

$\neg \text{is-empty } q \Longrightarrow \text{set } (\text{alist-of } (\text{remove-min } q)) =$

$\text{set } (\text{alist-of } q) - \{(\text{min } q, \text{the } (\text{priority } q \ (\text{min } q)))\}$

<proof>

definition *pop* :: $('a, 'b::\text{linorder}) \text{pq} \Rightarrow ('a \times ('a, 'b) \text{pq}) \text{option}$ **where**

$\text{pop } q = (\text{if is-empty } q \text{ then None else Some } (\text{min } q, \text{remove-min } q))$

lemma *pop-simps* [*simp*]:

$\text{is-empty } q \Longrightarrow \text{pop } q = \text{None}$

$\neg \text{is-empty } q \Longrightarrow \text{pop } q = \text{Some } (\text{min } q, \text{remove-min } q)$

<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 'b ('a, 'b) *bintree list*

primrec *priority* :: ('a, 'b) *bintree* \Rightarrow 'a **where**
priority (Node a -) = a

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

primrec *children* :: ('a, 'b) *bintree* \Rightarrow ('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 $P \square$
and $\bigwedge xs. P xs \Longrightarrow P (None \# xs)$
and $\bigwedge x xs. P xs \Longrightarrow P (Some x \# xs)$
shows $P xs$
(*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 \Rightarrow ('a, 'b) *bintree list* \Rightarrow bool **where**
is-bintree-list-Nil [simp]: *is-bintree-list* 0 \square
| *is-bintree-list-Cons*: *is-bintree-list* l ts \Longrightarrow *is-bintree-list* l (*children* t)
 \Longrightarrow *is-bintree-list* (Suc l) (t # ts)

abbreviation (*input*) *is-bintree* k t \equiv *is-bintree-list* k (*children* t)

lemma *is-bintree-list-triv* [simp]:

is-bintree-list 0 *ts* \longleftrightarrow *ts* = []

is-bintree-list *l* [] \longleftrightarrow *l* = 0

<proof>

lemma *is-bintree-list-simp* [simp]:

is-bintree-list (Suc *l*) (*t* # *ts*) \longleftrightarrow

is-bintree-list *l* (children *t*) \wedge *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* [] \longleftrightarrow True

is-heap-list *h* (*t* # *ts*) \longleftrightarrow

is-heap-list *h* *ts* \wedge *is-heap-list* (priority *t*) (children *t*) \wedge 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*:

$is\text{-heap-list } l \ ts \implies is\text{-heap-list } l \ (rev \ ts)$
(proof)

lemma *is-heap-children-larger*:

$is\text{-heap } t \implies \forall x \in set \ (children \ t). \ priority \ x \geq \ priority \ t$
(proof)

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

$is\text{-heap } t \implies children \ t \neq [] \implies$
 $priority \ t \leq Min \ (priority \ ` \ set \ (children \ t))$
(proof)

Combination of both: binqueue property

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

Empty: $is\text{-binqueue } l \ []$

| *None*: $is\text{-binqueue } (Suc \ l) \ xs \implies is\text{-binqueue } l \ (None \ \# \ xs)$

| *Some*: $is\text{-binqueue } (Suc \ l) \ xs \implies is\text{-bintree } l \ t$
 $\implies is\text{-heap } t \implies is\text{-binqueue } l \ (Some \ t \ \# \ xs)$

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

$is\text{-binqueue } l \ [] \longleftrightarrow True$

$is\text{-binqueue } l \ (Some \ t \ \# \ xs) \longleftrightarrow$

$is\text{-bintree } l \ t \wedge is\text{-heap } t \wedge is\text{-binqueue } (Suc \ l) \ xs$

$is\text{-binqueue } l \ (None \ \# \ xs) \longleftrightarrow is\text{-binqueue } (Suc \ l) \ xs$

(proof)

lemma *is-binqueue-trans*:

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

(proof)

lemma *is-binqueue-head*:

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

(proof)

lemma *is-binqueue-append*:

$is\text{-binqueue } l \ xs \implies is\text{-binqueue } (length \ xs + l) \ ys \implies is\text{-binqueue } l \ (xs \ @ \ ys)$

(proof)

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

$is\text{-binqueue } l \ (xs \ @ \ ys) \implies is\text{-binqueue } l \ xs$

(proof)

lemma *is-binqueue-children*:

assumes *is-bintree-list* $l \ ts$

and *is-heap-list* $t\ ts$
shows *is-binqueue* 0 (*map Some (rev ts)*)
 ⟨*proof*⟩

lemma *is-binqueue-select*:
is-binqueue $l\ xs \implies \text{Some } t \in \text{set } xs \implies \exists k. \text{is-bintree } k\ t \wedge \text{is-heap } t$
 ⟨*proof*⟩

Normalized representation

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

lemma *normalized-last-not-None*:
 — sometimes the inductive definition might work better
normalized $xs \longleftrightarrow xs = [] \vee \text{last } xs \neq \text{None}$
 ⟨*proof*⟩

lemma *normalized-simps* [*simp*]:
normalized [] $\longleftrightarrow \text{True}$
normalized (Some $t \# xs$) $\longleftrightarrow \text{normalized } xs$
normalized (None $\# xs$) $\longleftrightarrow xs \neq [] \wedge \text{normalized } xs$
 ⟨*proof*⟩

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

lemma *normalized-Cons*:
normalized ($x \# xs$) $\implies \text{normalized } xs$
 ⟨*proof*⟩

lemma *normalized-append*:
normalized $xs \implies \text{normalized } ys \implies \text{normalized } (xs @ ys)$
 ⟨*proof*⟩

lemma *normalized-not-None*:
normalized $xs \implies \text{set } xs \neq \{\text{None}\}$
 ⟨*proof*⟩

primrec *normalize'* :: ('a, 'b) *binqueue* \Rightarrow ('a, 'b) *binqueue* **where**
normalize' [] = []
 | *normalize'* ($x \# xs$) =

(case x of $None \Rightarrow \text{normalize}' xs \mid Some\ t \Rightarrow (x \# xs)$)

definition $\text{normalize} :: ('a, 'b) \text{binqueue} \Rightarrow ('a, 'b) \text{binqueue}$ **where**
 $\text{normalize}\ xs = \text{rev} (\text{normalize}' (\text{rev}\ xs))$

lemma $\text{normalized-normalize}$:
 $\text{normalized} (\text{normalize}\ xs)$
<proof>

lemma $\text{is-binqueue-normalize}$:
 $\text{is-binqueue}\ l\ xs \Longrightarrow \text{is-binqueue}\ l (\text{normalize}\ xs)$
<proof>

2.3 Operations

Adding data

definition $\text{merge} :: ('a::\text{linorder}, 'b) \text{bintree} \Rightarrow ('a, 'b) \text{bintree} \Rightarrow ('a, 'b) \text{bintree}$
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 $\text{is-bintree-list-merge}$:
assumes $\text{is-bintree}\ l\ t1\ \text{is-bintree}\ l\ t2$
shows $\text{is-bintree} (\text{Suc}\ l) (\text{merge}\ t1\ t2)$
<proof>

lemma is-heap-merge :
assumes $\text{is-heap}\ t1\ \text{is-heap}\ t2$
shows $\text{is-heap} (\text{merge}\ t1\ t2)$
<proof>

fun
 $\text{add} :: ('a::\text{linorder}, 'b) \text{bintree}\ \text{option} \Rightarrow ('a, 'b) \text{binqueue} \Rightarrow ('a, 'b) \text{binqueue}$
where
 $\text{add}\ None\ xs = xs$
 $\mid \text{add}\ (Some\ t)\ [] = [Some\ t]$
 $\mid \text{add}\ (Some\ t)\ (None \# xs) = Some\ t \# xs$
 $\mid \text{add}\ (Some\ t)\ (Some\ r \# xs) = None \# \text{add}\ (Some\ (\text{merge}\ t\ r))\ xs$

lemma add-Some-not-Nil [simp]:
 $\text{add}\ (Some\ t)\ xs \neq []$
<proof>

lemma normalized-add :

assumes *normalized xs*
shows *normalized (add x xs)*
<proof>

lemma *is-binqueue-add-None*:
assumes *is-binqueue l xs*
shows *is-binqueue l (add 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 (add (Some t) xs)*
<proof>

function
meld :: ('a::linorder, 'b) binqueue \Rightarrow ('a, 'b) binqueue \Rightarrow ('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)
<proof> **termination** *<proof>*

lemma *meld-singleton-add [simp]*:
meld [Some t] xs = add (Some t) xs
<proof>

lemma *nonempty-meld [simp]*:
xs \neq [] \implies meld xs ys \neq []
ys \neq [] \implies meld xs ys \neq []
<proof>

lemma *nonempty-meld-commute*:
meld xs ys \neq [] \implies meld xs ys \neq []
<proof>

lemma *is-binqueue-meld*:
assumes *is-binqueue l xs*
and *is-binqueue l ys*
shows *is-binqueue l (meld xs ys)*
<proof>

lemma *normalized-meld*:
assumes *normalized xs*
and *normalized ys*
shows *normalized (meld xs ys)*
 \langle *proof* \rangle

lemma *normalized-meld-weak*:
assumes *normalized xs*
and *length ys \leq length xs*
shows *normalized (meld xs ys)*
 \langle *proof* \rangle

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')
 \langle *proof* \rangle

lemma *least-split*:
assumes *least x y = Some z*
shows *x = Some z \vee y = Some z*
 \langle *proof* \rangle

interpretation *least*: *semilattice least* \langle *proof* \rangle

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)
 \langle *proof* \rangle

lemma [*code*]:
min xs = fold (λ x. least (map-option priority x)) xs None
 \langle *proof* \rangle

lemma *min-single*:

$\text{min } [x] = \text{Some } a \implies \text{priority } (\text{the } x) = a$

$\text{min } [x] = \text{None} \implies x = \text{None}$

$\langle \text{proof} \rangle$

lemma *min-Some-not-None*:

$\text{min } (\text{Some } t \# xs) \neq \text{None}$

$\langle \text{proof} \rangle$

lemma *min-None-trans*:

assumes $\text{min } (x \# xs) = \text{None}$

shows $\text{min } xs = \text{None}$

$\langle \text{proof} \rangle$

lemma *min-None-None*:

$\text{min } xs = \text{None} \longleftrightarrow xs = [] \vee \text{set } xs = \{\text{None}\}$

$\langle \text{proof} \rangle$

lemma *normalized-min-not-None*:

$\text{normalized } xs \implies xs \neq [] \implies \text{min } xs \neq \text{None}$

$\langle \text{proof} \rangle$

lemma *min-is-min*:

assumes *normalized xs*

and $xs \neq []$

and $\text{min } xs = \text{Some } a$

shows $\forall x \in \text{set } xs. x = \text{None} \vee a \leq \text{priority } (\text{the } x)$

$\langle \text{proof} \rangle$

lemma *min-exists*:

assumes $\text{min } xs = \text{Some } a$

shows $\text{Some } a \in \text{map-option priority ' set } xs$

$\langle \text{proof} \rangle$

primrec *find* :: $'a::\text{linorder} \Rightarrow ('a, 'b) \text{ binqueue} \Rightarrow ('a, 'b) \text{ bintree option}$ **where**

$\text{find } a [] = \text{None}$

| $\text{find } a (x \# xs) = (\text{case } x \text{ of } \text{None} \Rightarrow \text{find } a \ xs$

| $\text{Some } t \Rightarrow \text{if priority } t = a \text{ then } \text{Some } t \text{ else } \text{find } a \ xs)$

declare *find.simps* [*simp del*]

lemma *find-simps* [*simp, code*]:

$\text{find } a [] = \text{None}$

$\text{find } a (\text{None} \# xs) = \text{find } a \ xs$

$find\ a\ (Some\ t\ \#\ xs) = (if\ priority\ t = a\ then\ Some\ t\ else\ find\ a\ xs)$
 $\langle proof \rangle$

lemma *find-works*:

assumes $Some\ a \in set\ (map\ (map\ option\ priority)\ xs)$

shows $\exists t. find\ a\ xs = Some\ t \wedge priority\ t = a$

$\langle proof \rangle$

lemma *find-works-not-None*:

$Some\ a \in set\ (map\ (map\ option\ priority)\ xs) \implies find\ a\ xs \neq None$

$\langle proof \rangle$

lemma *find-None*:

$find\ a\ xs = None \implies Some\ a \notin set\ (map\ (map\ option\ priority)\ xs)$

$\langle proof \rangle$

lemma *find-exist*:

$find\ a\ xs = Some\ t \implies Some\ t \in set\ xs$

$\langle proof \rangle$

definition *find-min* :: $(a::linorder, 'b)\ binqueue \Rightarrow (a, 'b)\ bintree\ option$ **where**

$find\ min\ xs = (case\ min\ xs\ of\ None \Rightarrow None \mid Some\ a \Rightarrow find\ a\ xs)$

lemma *find-min-simps* [*simp*]:

$find\ min\ [] = None$

$find\ min\ (None\ \#\ xs) = find\ min\ xs$

$\langle proof \rangle$

lemma *find-min-single*:

$find\ min\ [x] = x$

$\langle proof \rangle$

lemma *min-eq-find-min-None*:

$min\ xs = None \longleftrightarrow find\ min\ xs = None$

$\langle proof \rangle$

lemma *min-eq-find-min-Some*:

$min\ xs = Some\ a \longleftrightarrow (\exists\ t. find\ min\ xs = Some\ t \wedge priority\ t = a)$

$\langle proof \rangle$

lemma *find-min-exist*:

assumes $find\ min\ xs = Some\ t$

shows $Some\ t \in set\ xs$

$\langle proof \rangle$

lemma *find-min-is-min*:
assumes *normalized xs*
and $xs \neq []$
and $find_min\ xs = Some\ t$
shows $\forall x \in set\ xs. x = None \vee (priority\ t) \leq priority\ (the\ x)$
 $\langle proof \rangle$

lemma *normalized-find-min-exists*:
 $normalized\ xs \implies xs \neq [] \implies \exists t. find_min\ xs = Some\ t$
 $\langle proof \rangle$

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

definition *delete-min* :: $('a::linorder, 'b)\ binqueue \Rightarrow ('a, 'b)\ binqueue$ **where**
 $delete_min\ xs = (case\ find_min\ xs$
 $of\ Some\ (Node\ a\ v\ ts) \Rightarrow$
 $normalize\ (meld\ (map\ Some\ (rev\ ts))\ (map\ (match\ a)\ xs))$
 $| None \Rightarrow [])$

lemma *delete-min-empty [simp]*:
 $delete_min\ [] = []$
 $\langle proof \rangle$

lemma *delete-min-nonempty [simp]*:
 $normalized\ xs \implies xs \neq [] \implies find_min\ xs = Some\ t$
 $\implies delete_min\ xs = normalize$
 $(meld\ (map\ Some\ (rev\ (children\ t)))\ (map\ (match\ (priority\ t))\ xs))$
 $\langle proof \rangle$

lemma *is-binqueue-delete-min*:
assumes $is_binqueue\ 0\ xs$
shows $is_binqueue\ 0\ (delete_min\ xs)$
 $\langle proof \rangle$

lemma *normalized-delete-min*:
 $normalized\ (delete_min\ xs)$
 $\langle proof \rangle$

Dedicated grand unified operation for generated program

definition

$\text{meld}' :: ('a, 'b) \text{ bintree option} \Rightarrow ('a::\text{linorder}, 'b) \text{ binqueue}$
 $\Rightarrow ('a, 'b) \text{ binqueue} \Rightarrow ('a, 'b) \text{ binqueue}$

where

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

lemma [code]:

$\text{add } z xs = \text{meld}' z [] xs$
 $\text{meld } xs ys = \text{meld}' \text{None } xs ys$
 ⟨proof⟩

lemma [code]:

$\text{meld}' z (\text{Some } t \# xs) (\text{Some } r \# ys) =$
 $z \# (\text{meld}' (\text{Some } (\text{merge } t r)) xs ys)$
 $\text{meld}' (\text{Some } t) (\text{Some } r \# xs) (\text{None } \# ys) =$
 $\text{None } \# (\text{meld}' (\text{Some } (\text{merge } t r)) xs ys)$
 $\text{meld}' (\text{Some } t) (\text{None } \# xs) (\text{Some } r \# ys) =$
 $\text{None } \# (\text{meld}' (\text{Some } (\text{merge } t r)) xs ys)$
 $\text{meld}' \text{None } (x \# xs) (\text{None } \# ys) = x \# (\text{meld}' \text{None } xs ys)$
 $\text{meld}' \text{None } (\text{None } \# xs) (y \# ys) = y \# (\text{meld}' \text{None } xs ys)$
 $\text{meld}' z (\text{None } \# xs) (\text{None } \# ys) = z \# (\text{meld}' \text{None } xs ys)$
 $\text{meld}' z xs [] = \text{meld}' z [] xs$
 $\text{meld}' z [] (y \# ys) = \text{meld}' \text{None } [z] (y \# ys)$
 $\text{meld}' (\text{Some } t) [] ys = \text{meld}' \text{None } [\text{Some } t] ys$
 $\text{meld}' \text{None } [] ys = ys$
 ⟨proof⟩

Interface operations

abbreviation (input) $\text{empty} :: ('a, 'b) \text{ binqueue}$ **where**

$\text{empty} \equiv []$

definition

$\text{insert} :: 'a::\text{linorder} \Rightarrow 'b \Rightarrow ('a, 'b) \text{ binqueue} \Rightarrow ('a, 'b) \text{ binqueue}$

where

$\text{insert } a v xs = \text{add } (\text{Some } (\text{Node } a v [])) xs$

lemma insert-simps [simp]:

$\text{insert } a v [] = [\text{Some } (\text{Node } a v [])]$
 $\text{insert } a v (\text{None } \# xs) = \text{Some } (\text{Node } a v []) \# xs$
 $\text{insert } a v (\text{Some } t \# xs) = \text{None } \# \text{add } (\text{Some } (\text{merge } (\text{Node } a v []) t)) xs$
 ⟨proof⟩

lemma $\text{is-binqueue-insert}$:

$\text{is-binqueue } 0 xs \Longrightarrow \text{is-binqueue } 0 (\text{insert } a v xs)$
 ⟨proof⟩

lemma *normalized-insert*:

normalized xs \implies *normalized (insert a v xs)*
(*proof*)

definition

pop :: ('a::linorder, 'b) binqueue \Rightarrow (('b \times 'a) option \times ('a, 'b) binqueue)

where

pop xs = (case *find-min xs* of
 None \Rightarrow (None, *xs*)
 | Some *t* \Rightarrow (Some (val *t*, priority *t*), delete-min *xs*))

lemma *pop-empty [simp]*:

pop empty = (None, empty)
(*proof*)

lemma *pop-nonempty [simp]*:

normalized xs \implies *xs* \neq [] \implies *find-min xs* = Some *t*
 \implies *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 \Rightarrow (None, *xs*)
 | Some *t* \Rightarrow (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 \Rightarrow 'c) \Rightarrow ('a, 'b) bintree \Rightarrow 'c list
and *bts-dfs* :: (('a::linorder, 'b) bintree \Rightarrow 'c) \Rightarrow ('a, 'b) bintree list \Rightarrow 'c list

where

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

| $bt\text{-}dfs\ f\ (t\ \#\ ts) = bt\text{-}dfs\ f\ t\ \@\ bt\text{-}dfs\ f\ ts$

lemma *bt-dfs-simp*:

$bt\text{-}dfs\ f\ t = f\ t\ \#\ bt\text{-}dfs\ f\ (children\ t)$
(proof)

lemma *bts-dfs-append [simp]*:

$bts\text{-}dfs\ f\ (ts\ \@\ rs) = bts\text{-}dfs\ f\ ts\ \@\ bt\text{-}dfs\ f\ rs$
(proof)

lemma *set-bts-dfs-rev*:

$set\ (bts\text{-}dfs\ f\ (rev\ ts)) = set\ (bts\text{-}dfs\ f\ ts)$
(proof)

lemma *bts-dfs-rev-distinct*:

$distinct\ (bts\text{-}dfs\ f\ ts) \implies distinct\ (bts\text{-}dfs\ f\ (rev\ ts))$
(proof)

lemma *bt-dfs-comp*:

$bt\text{-}dfs\ (f\ \circ\ g)\ t = map\ f\ (bt\text{-}dfs\ g\ t)$
 $bts\text{-}dfs\ (f\ \circ\ g)\ ts = map\ f\ (bts\text{-}dfs\ g\ ts)$
(proof)

lemma *bt-dfs-comp-distinct*:

$distinct\ (bt\text{-}dfs\ (f\ \circ\ g)\ t) \implies distinct\ (bt\text{-}dfs\ g\ t)$
 $distinct\ (bts\text{-}dfs\ (f\ \circ\ g)\ ts) \implies distinct\ (bts\text{-}dfs\ g\ ts)$
(proof)

lemma *bt-dfs-distinct-children*:

$distinct\ (bt\text{-}dfs\ f\ x) \implies distinct\ (bts\text{-}dfs\ f\ (children\ x))$
(proof)

fun *dfs* :: (*'a*::*linorder*, *'b*) *bintree* \Rightarrow *'c* \Rightarrow (*'a*, *'b*) *binqueue* \Rightarrow *'c list* **where**

$dfs\ f\ [] = []$
| $dfs\ f\ (None\ \#\ xs) = dfs\ f\ xs$
| $dfs\ f\ (Some\ t\ \#\ xs) = bt\text{-}dfs\ f\ t\ \@\ dfs\ f\ xs$

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*:

$set (dfs f (x \# xs)) = set (dfs f xs) \cup set (dfs f [x])$
<proof>

lemma *dfs-comp*:

$dfs (f \circ g) xs = map f (dfs g xs)$
<proof>

lemma *dfs-comp-distinct*:

$distinct (dfs (f \circ g) xs) \implies distinct (dfs g xs)$
<proof>

lemma *dfs-distinct-member*:

$distinct (dfs f xs) \implies$
 $Some x \in set xs \implies$
 $distinct (bt-dfs f x)$
<proof>

lemma *dfs-map-Some-idem*:

$dfs f (map Some xs) = bts-dfs f xs$
<proof>

primrec *alist* :: ('a, 'b) bintree \Rightarrow ('b \times 'a) **where**

$alist (Node a v _) = (v, a)$

lemma *alist-split-pre*:

$val t = (fst \circ alist) t$
 $priority t = (snd \circ alist) t$
<proof>

lemma *alist-split*:

$val = fst \circ alist$
 $priority = snd \circ alist$
<proof>

lemma *alist-split-set*:

$set (dfs val xs) = fst ' set (dfs alist xs)$
 $set (dfs priority xs) = snd ' set (dfs alist xs)$
<proof>

lemma *in-set-in-alist*:

assumes $Some t \in set xs$
shows $(val t, priority t) \in set (dfs alist xs)$
<proof>

abbreviation *vals* **where** *vals* \equiv *dfs val*
abbreviation *prios* **where** *prios* \equiv *dfs priority*
abbreviation *elements* **where** *elements* \equiv *dfs alist*

primrec

bt-augment :: ('a::linorder, 'b) bintree \Rightarrow ('b, 'a) PQ.pq \Rightarrow ('b, 'a) PQ.pq

and

bts-augment :: ('a::linorder, 'b) bintree list \Rightarrow ('b, 'a) PQ.pq \Rightarrow ('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*

\langle proof \rangle

lemma *bt-augment-Node* [simp]:

bt-augment (Node a v ts) q = PQ.push v a (fold *bt-augment* ts q)

\langle proof \rangle

lemma *bt-augment-simp*:

bt-augment t q = PQ.push (val t) (priority t) (fold *bt-augment* (children t) q)

\langle proof \rangle

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

fun *pqueue* :: ('a::linorder, 'b) binqueue \Rightarrow ('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| \subseteq set |*bt-augment* t q|

set |q| \subseteq set |*bts-augment* ts q|

\langle proof \rangle

lemma *bt-augment-v-in*:

$v \in$ set |q| $\implies v \in$ set |*bt-augment* t q|

$v \in$ set |q| $\implies v \in$ set |*bts-augment* ts q|

\langle proof \rangle

lemma *bt-augment-v-union*:

set |*bt-augment* t (*bt-augment* r q)| =

set |*bt-augment* t q| \cup set |*bt-augment* r q|

$set \ |bts\text{-}augment\ ts\ (bt\text{-}augment\ r\ q)| =$
 $set \ |bts\text{-}augment\ ts\ q| \cup set \ |bt\text{-}augment\ r\ q|$
 <proof>

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

lemma *vals-pqueue*:
 $set \ (vals\ xs) = set \ |pqueue\ xs|$
 <proof>

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

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

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

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

lemma *bt-augment-v-merge*:
 $set \ |bt\text{-}augment\ (merge\ t\ r)\ q| = set \ |bt\text{-}augment\ t\ (bt\text{-}augment\ r\ q)|$
 <proof>

lemma *vals-merge [simp]*:
 $set \ (bt\text{-}dfs\ val\ (merge\ t\ r)) = set \ (bt\text{-}dfs\ val\ t) \cup set \ (bt\text{-}dfs\ val\ r)$
 <proof>

lemma *vals-merge-distinct*:

$distinct (bt\text{-}dfs\ val\ t) \implies distinct (bt\text{-}dfs\ val\ r) \implies$
 $set (bt\text{-}dfs\ val\ t) \cap set (bt\text{-}dfs\ val\ r) = \{\} \implies$
 $distinct (bt\text{-}dfs\ val\ (merge\ t\ r))$
 $\langle proof \rangle$

lemma *vals-add-Cons*:

$set (vals (add\ x\ xs)) = set (vals (x\ \#\ xs))$
 $\langle proof \rangle$

lemma *vals-add-distinct*:

assumes $distinct (vals\ xs)$
and $distinct (dfs\ val\ [x])$
and $set (vals\ xs) \cap set (dfs\ val\ [x]) = \{\}$
shows $distinct (vals (add\ x\ xs))$
 $\langle proof \rangle$

lemma *vals-insert [simp]*:

$set (vals (insert\ a\ v\ xs)) = set (vals\ xs) \cup \{v\}$
 $\langle proof \rangle$

lemma *insert-v-push*:

$set (vals (insert\ a\ v\ xs)) = set |PQ.push\ v\ a\ (pqueue\ xs)|$
 $\langle proof \rangle$

lemma *vals-meld*:

$set (dfs\ val\ (meld\ xs\ ys)) = set (dfs\ val\ xs) \cup set (dfs\ val\ ys)$
 $\langle proof \rangle$

lemma *vals-meld-distinct*:

$distinct (dfs\ val\ xs) \implies distinct (dfs\ val\ ys) \implies$
 $set (dfs\ val\ xs) \cap set (dfs\ val\ ys) = \{\} \implies$
 $distinct (dfs\ val\ (meld\ xs\ ys))$
 $\langle proof \rangle$

lemma *bt-augment-alist-subset*:

$set (PQ.alist\ of\ q) \subseteq set (PQ.alist\ of\ (bt\text{-}augment\ t\ q))$
 $set (PQ.alist\ of\ q) \subseteq set (PQ.alist\ of\ (bts\text{-}augment\ ts\ q))$
 $\langle proof \rangle$

lemma *bt-augment-alist-in*:

$(v, a) \in set (PQ.alist\ of\ q) \implies (v, a) \in set (PQ.alist\ of\ (bt\text{-}augment\ t\ q))$
 $(v, a) \in set (PQ.alist\ of\ q) \implies (v, a) \in set (PQ.alist\ of\ (bts\text{-}augment\ ts\ q))$
 $\langle proof \rangle$

lemma *bt-augment-alist-union*:

$$\begin{aligned} & \text{distinct (bts-dfs val (r \# [t]))} \implies \\ & \text{set (bts-dfs val (r \# [t]))} \cap \text{set |q|} = \{\} \implies \\ & \text{set (PQ.alist-of (bt-augment t (bt-augment r q)))} = \\ & \text{set (PQ.alist-of (bt-augment t q))} \cup \text{set (PQ.alist-of (bt-augment r q))} \end{aligned}$$

$$\begin{aligned} & \text{distinct (bts-dfs val (r \# ts))} \implies \\ & \text{set (bts-dfs val (r \# ts))} \cap \text{set |q|} = \{\} \implies \\ & \text{set (PQ.alist-of (bts-augment ts (bt-augment r q)))} = \\ & \text{set (PQ.alist-of (bts-augment ts q))} \cup \text{set (PQ.alist-of (bt-augment r q))} \end{aligned}$$

<proof>

lemma *bt-alist-augment*:

$$\begin{aligned} & \text{distinct (bt-dfs val t)} \implies \\ & \text{set (bt-dfs val t)} \cap \text{set |q|} = \{\} \implies \\ & \text{set (bt-dfs alist t)} \cup \text{set (PQ.alist-of q)} = \text{set (PQ.alist-of (bt-augment t q))} \end{aligned}$$

$$\begin{aligned} & \text{distinct (bts-dfs val ts)} \implies \\ & \text{set (bts-dfs val ts)} \cap \text{set |q|} = \{\} \implies \\ & \text{set (bts-dfs alist ts)} \cup \text{set (PQ.alist-of q)} = \\ & \text{set (PQ.alist-of (bts-augment ts q))} \end{aligned}$$

<proof>

lemma *alist-pqueue*:

$$\text{distinct (vals xs)} \implies \text{set (dfs alist xs)} = \text{set (PQ.alist-of (pqueue xs))}$$

<proof>

lemma *alist-pqueue-priority*:

$$\begin{aligned} & \text{distinct (vals xs)} \implies (v, a) \in \text{set (dfs alist xs)} \\ & \implies \text{PQ.priority (pqueue xs)} v = \text{Some } a \end{aligned}$$

<proof>

lemma *prios-pqueue*:

$$\text{distinct (vals xs)} \implies \text{set (prios xs)} = \text{set ||pqueue xs||}$$

<proof>

lemma *alist-merge [simp]*:

$$\begin{aligned} & \text{distinct (bt-dfs val t)} \implies \text{distinct (bt-dfs val r)} \implies \\ & \text{set (bt-dfs val t)} \cap \text{set (bt-dfs val r)} = \{\} \implies \\ & \text{set (bt-dfs alist (merge t r))} = \text{set (bt-dfs alist t)} \cup \text{set (bt-dfs alist r)} \end{aligned}$$

<proof>

lemma *alist-add-Cons*:

$$\begin{aligned} & \text{assumes } \text{distinct (vals (x\#xs))} \\ & \text{shows } \text{set (dfs alist (add x xs))} = \text{set (dfs alist (x \# xs))} \end{aligned}$$

$\langle \text{proof} \rangle$

lemma *alist-insert* [*simp*]:

$\text{distinct } (\text{vals } xs) \implies$
 $v \notin \text{set } (\text{vals } xs) \implies$
 $\text{set } (\text{dfs alist } (\text{insert } a \ v \ xs)) = \text{set } (\text{dfs alist } xs) \cup \{(v,a)\}$
 $\langle \text{proof} \rangle$

lemma *insert-push*:

$\text{distinct } (\text{vals } xs) \implies$
 $v \notin \text{set } (\text{vals } xs) \implies$
 $\text{set } (\text{dfs alist } (\text{insert } a \ v \ xs)) = \text{set } (PQ.\text{alist-of } (PQ.\text{push } v \ a \ (\text{pqueue } xs)))$
 $\langle \text{proof} \rangle$

lemma *insert-p-push*:

assumes $\text{distinct } (\text{vals } xs)$
and $v \notin \text{set } (\text{vals } xs)$
shows $\text{set } (\text{prios } (\text{insert } a \ v \ xs)) = \text{set } \parallel PQ.\text{push } v \ a \ (\text{pqueue } xs) \parallel$
 $\langle \text{proof} \rangle$

lemma *empty-empty*:

$\text{normalized } xs \implies xs = \text{empty} \longleftrightarrow PQ.\text{is-empty } (\text{pqueue } xs)$
 $\langle \text{proof} \rangle$

lemma *bt-dfs-Min-priority*:

assumes $\text{is-heap } t$
shows $\text{priority } t = \text{Min } (\text{set } (\text{bt-dfs priority } t))$
 $\langle \text{proof} \rangle$

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

assumes $\text{is-binqueue } l \ xs$
and $\text{normalized } xs$
and $xs \neq []$
shows $\text{min } xs = \text{Some } (\text{Min } (\text{set } (\text{prios } xs)))$
 $\langle \text{proof} \rangle$

lemma *min-p-min*:

assumes $\text{is-binqueue } l \ xs$
and $xs \neq []$
and $\text{normalized } xs$
and $\text{distinct } (\text{vals } xs)$
and $\text{distinct } (\text{prios } xs)$
shows $\text{min } xs = PQ.\text{priority } (\text{pqueue } xs) \ (PQ.\text{min } (\text{pqueue } xs))$
 $\langle \text{proof} \rangle$

lemma *find-min-p-min*:
assumes *is-binqueue l xs*
and $xs \neq []$
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)))
 \langle *proof* \rangle

lemma *find-min-v-min*:
assumes *is-binqueue l xs*
and $xs \neq []$
and *normalized xs*
and *distinct (vals xs)*
and *distinct (prios xs)*
shows *val (the (find-min xs)) = PQ.min (pqueue xs)*
 \langle *proof* \rangle

lemma *alist-normalize-idem*:
dfs alist (normalize xs) = dfs alist xs
 \langle *proof* \rangle

lemma *dfs-match-not-in*:
 $(\forall t. \text{Some } t \in \text{set } xs \longrightarrow \text{priority } t \neq a) \implies$
 $\text{set } (dfs\ f\ (\text{map } (\text{match } a)\ xs)) = \text{set } (dfs\ f\ xs)$
 \langle *proof* \rangle

lemma *dfs-match-subset*:
 $\text{set } (dfs\ f\ (\text{map } (\text{match } a)\ xs)) \subseteq \text{set } (dfs\ f\ xs)$
 \langle *proof* \rangle

lemma *dfs-match-distinct*:
 $\text{distinct } (dfs\ f\ xs) \implies \text{distinct } (dfs\ f\ (\text{map } (\text{match } a)\ xs))$
 \langle *proof* \rangle

lemma *dfs-match*:
 $\text{distinct } (\text{prios } xs) \implies$
 $\text{distinct } (dfs\ f\ xs) \implies$
 $\text{Some } t \in \text{set } xs \implies$
 $\text{priority } t = a \implies$
 $\text{set } (dfs\ f\ (\text{map } (\text{match } a)\ xs)) = \text{set } (dfs\ f\ xs) - \text{set } (bt\text{-dfs } f\ t)$
 \langle *proof* \rangle

lemma *alist-meld*:

$distinct (dfs\ val\ xs) \implies distinct (dfs\ val\ ys) \implies$
 $set (dfs\ val\ xs) \cap set (dfs\ val\ ys) = \{\} \implies$
 $set (dfs\ alist\ (meld\ xs\ ys)) = set (dfs\ alist\ xs) \cup set (dfs\ alist\ ys)$
 <proof>

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>

lemma *alist-remove-min*:
assumes *is-binqueue l xs*
and *distinct (vals xs)*
and *distinct (prios xs)*
and *normalized xs*
and $xs \neq []$
shows $set (dfs\ alist\ (delete-min\ xs)) =$
 $set (PQ.alist-of (PQ.remove-min (pqueue\ xs)))$
 <proof>

no-notation
PQ.values ($|(- |)$)
and *PQ.priorities* ($|| (-) ||$)