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

morphism 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-eql:
  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 |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 ⇒ 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 ≠ []
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 ∈ set |empty|
(proof)

lemma is-empty-empty:
is-empty q ←→ q = empty
(proof)

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

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

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

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

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

lemma priority-Min:
assumes ¬ is-empty q
shows priority q (min q) = Some (Min (the ‘ priority q ‘ set (values q)))
⟨proof⟩

lemma priority-Min-priorities:
assumes ¬ 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 ≠ [] ⇒ sorted (map snd q) ⇒ Min (snd ∥∥set q∥∥) = snd (hd q)
⟨proof⟩

lemma hd-construct:
assumes ¬ is-empty q
shows hd (alist-of q) = (min q, the (priority q (min q)))
⟨proof⟩

lemma not-in-first-image:
x ∉ fst ∖ s =⇒ (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 ∉ set ∥∥q∥∥ ⇒ set ∥∥push k p q∥∥ = set ∥∥q∥∥ ∪ {p}
k ∈ set ∥∥q∥∥ ⇒ set ∥∥push k p q∥∥ = set ∥∥q∥∥
⟨proof⟩

lemma not-is-empty-push [simp]:
¬ is-empty (push k p q)
⟨proof⟩
lemma push-commute:
assumes $a \neq b$ and $v \neq w$
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 (∥|-∥)

5
2 Functional Binomial Queues

2.1 Type definition and projections

datatype \((\prime a, \prime b)\) bintree = Node \(\prime a\) \(\prime b\) \((\prime a, \prime b)\) bintree list

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

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

primrec children :: \((\prime a, \prime b)\) bintree \Rightarrow \((\prime a, \prime b)\) bintree list where
children (Node - - ts) = ts

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

lemma binqueue-induct [case-names Empty None Some, induct type: binqueue]:
assumes \(P\) []
and \(\land xs. \ P \ xs \Longrightarrow \ P \ (\text{None} \ # \ xs)\)
and \(\land x xs. \ P \ xs \Longrightarrow \ P \ (\text{Some} \ x \ # \ xs)\)
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 \(ts1, ts2\)
- biquenelement \(x, y\) or \(x1, x2\)
- binqueues = biquenelement 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 \((\prime a, \prime b)\) bintree list \Rightarrow bool where
is-bintree-list-Nil [simp]: is-bintree-list 0 []
| 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 ↔ ts = []
  is-bintree-list l [] ↔ l = 0
⟨proof⟩

lemma is-bintree-list-simp [simp]:
  is-bintree-list (Suc l) (t ≠ ts) ↔
  is-bintree-list l (children t) ∧ is-bintree-list l ts
⟨proof⟩

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

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

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

Heap property

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

lemma is-heap-list-simps [simp]:
  is-heap-list h [] ↔ True
  is-heap-list h (t ≠ ts) ↔
  is-heap-list h ts ∧ is-heap-list (priority t) (children t) ∧ priority t ≥ h
⟨proof⟩

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

lemma is-heap-list-rev:
is-heap-list l ts \implies is-heap-list l (rev ts) 
(proof)

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

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

Combination of both: binqueue property

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

\text{Empty}: \text{is-binqueue}\ l [] 
| \text{None}: \text{is-binqueue} (\text{Suc}\ l)\ xs \implies \text{is-binqueue}\ l (\text{None} \# \text{xs}) 
| \text{Some}: \text{is-binqueue} (\text{Suc}\ l)\ xs \implies \text{is-bintree}\ l\ t 
\implies \text{is-heap}\ t \implies \text{is-binqueue}\ l (\text{Some}\ t \# \text{xs})

**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 is-binqueue (\text{Suc}\ l)\ \text{xs} 
(proof)

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

**lemma** is-binqueue-append:
is-binqueue l\ \text{xs} \implies is-binqueue\ (\text{length}\ \text{xs} + 1)\ \text{ys} \implies is-binqueue\ l\ (\text{xs} \@ \text{ys}) 
(proof)

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

**lemma** is-binqueue-children:
assumes is-bintree-list l ts

8
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) bqueue ⇒ ('a, 'b) bqueue where
  normalize xs = rev (normalize' (rev xs))

lemma normalized-normalize:
  normalized (normalize xs)
⟨proof⟩

lemma is-bqueue-normalize:
  is-bqueue l xs ⇒ is-bqueue 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) bqueue ⇒ ('a, 'b) bqueue
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 $(\text{add } x \ xs)$
(proof)

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

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

function $\text{meld} :: (\prime a :: \text{linorder}, \prime b) \text{binqueue} \Rightarrow (\prime a, \prime b) \text{binqueue} \Rightarrow (\prime a, \prime b) \text{binqueue}$

where
meld $[] \ ys = ys$
meld $xs \ [] = xs$
meld $(\text{None } \# \ xs) \ (y \# \ ys) = y \# \ meld \ xs \ ys$
meld $(x \# \ xs) \ (\text{None } \# \ ys) = x \# \ meld \ xs \ ys$
meld $(\text{Some } t \# \ xs) \ (\text{Some } r \# \ ys) =$
None \# \ add \ (\text{Some } (\text{merge } t \ r)) \ (\text{meld } xs \ ys)$
(proof) termination (proof)

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

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

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

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

12
lemma min-single:
\[ \min [x] = \text{Some } a \implies \text{priority (the } x) = a \]
\[ \min [x] = \text{None } \implies x = \text{None} \]
⟨proof⟩

lemma min-Some-not-None:
\[ \min (\text{Some } t \# xs) \neq \text{None} \]
⟨proof⟩

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

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

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

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

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

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

declare find.simps [simp del]

lemma find-simps [simp, code]:
\[ \text{find } a [] = \text{None} \]
\[ \text{find } a (\text{None } \# xs) = \text{find } a xs \]
\[ \text{find } a \text{ (Some } t \neq x) = (\text{if priority } t = a \text{ then Some } t \text{ else find } a \text{ xs)} \]

\textbf{lemma find-works:}
\textbf{assumes} Some \( a \in \text{ set (map (map-option priority) xs)} \)
\textbf{shows} \( \exists t. \text{ find } a \text{ xs } = \text{ Some } t \land \text{ priority } t = a \)
\textbf{proof}\]

\textbf{lemma find-works-not-None:}
\( \text{ Some } a \in \text{ set (map (map-option priority) xs)} \implies \text{ find } a \text{ xs } \neq \text{ None} \)
\textbf{proof}\]

\textbf{lemma find-None:}
\( \text{ find } a \text{ xs } = \text{ None } \implies \text{ Some } a \not\in \text{ set (map (map-option priority) xs)} \)
\textbf{proof}\]

\textbf{lemma find-exist:}
\( \text{ find } a \text{ xs } = \text{ Some } t \implies \text{ Some } t \in \text{ set xs} \)
\textbf{proof}\]

\textbf{definition find-min :: (\'a::linorder, \'b) binqueue \Rightarrow (\'a, \'b) bintree option where}
\( \text{ find-min xs } = (\text{case min xs of None } \Rightarrow \text{ None } \mid \text{ Some } a \Rightarrow \text{ find } a \text{ xs}) \)

\textbf{lemma find-min-simps [simp]:}
\( \text{ find-min [] } = \text{ None } \)
\( \text{ find-min (None } \neq x) = \text{ find-min } x \)
\textbf{proof}\]

\textbf{lemma find-min-single:}
\( \text{ find-min } [x] = x \)
\textbf{proof}\]

\textbf{lemma min-eq-find-min-None:}
\( \text{ min } xs = \text{ None } \iff \text{ find-min } xs = \text{ None } \)
\textbf{proof}\]

\textbf{lemma min-eq-find-min-Some:}
\( \text{ min } xs = \text{ Some } a \iff (\exists t. \text{ find-min } xs = \text{ Some } t \land \text{ priority } t = a) \)
\textbf{proof}\]

\textbf{lemma find-min-exist:}
\textbf{assumes} find-min \( xs = \text{ Some } t \)
\textbf{shows} Some \( t \in \text{ set xs} \)
\textbf{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-binqueuee-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) 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:
  \[ \text{normalized } xs \Rightarrow \text{normalized } (\text{insert } a \; v \; xs) \]
  
  ⟨proof⟩

definition
  \[ \text{pop} :: (\text{bt-linorder, } \text{b}) \; \text{bqueue} \Rightarrow ((\text{b} \times \text{a}) \; \text{option} \times (\text{a}, \text{b}) \; \text{bqueue}) \]
  
  where
  \[ \text{pop } xs = \text{case find-min } xs \text{ of} \]
  \[ \text{None} \Rightarrow (\text{None}, \; xs) \]
  \[ | \; \text{Some } t \Rightarrow (\text{Some } (\text{val } t, \text{priority } t), \; \text{delete-min } xs) \]

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

lemma pop-nonempty [simp]:
  \[ \text{normalized } xs \Rightarrow \text{normxs } \Rightarrow \text{find-min } xs = \text{Some } t \]
  \[ \Rightarrow \text{pop } xs = (\text{Some } (\text{val } t, \text{priority } t), \; \text{normalize} \]
  \[ (\text{meld } (\text{map Some } (\text{rev } (\text{children } t)))) (\text{map } (\text{match } (\text{priority } t)) \; xs)) \]
  
  ⟨proof⟩

lemma pop-code [code]:
  \[ \text{pop } xs = \text{case find-min } xs \text{ of} \]
  \[ \text{None} \Rightarrow (\text{None, empty}) \]
  \[ | \; \text{Some } t \Rightarrow (\text{Some } (\text{val } t, \text{priority } t), \; \text{normalize} \]
  \[ (\text{meld } (\text{map Some } (\text{rev } (\text{children } t)))) (\text{map } (\text{match } (\text{priority } t)) \; xs)) \]
  
  ⟨proof⟩

3 Relating Functional Binomial Queues To The Abstract Priority Queues

notation
  \[ \text{PQ.values } (\varepsilon) \]
  \[ \text{and PQ.priorities } (\varepsilon) \]

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

primrec \text{bt-dfs} :: ((\text{bt-linorder, } \text{b}) \; \text{bintree} \Rightarrow \text{c}) \Rightarrow (\text{a, b}) \; \text{bintree} \Rightarrow \text{c list}

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

where
  \[ \text{bt-dfs } f \; (\text{Node } a \; v \; ts) = f \; (\text{Node } a \; v \; ts) \# \text{bts-dfs } f \; ts \]


| bts-dfs f [] = []
| bts-dfs f (t # ts) = bt-dfs f t @ bts-dfs f ts

**lemma** bt-dfs-simp:

\[\text{bt-dfs } f \ t = f \ t \ # \ bts-dfs \ f \ (\text{children } t)\]

⟨proof⟩

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

\[\text{bts-dfs } f \ (ts \ @ \ rs) = bts-dfs f \ ts \ @ bts-dfs f \ rs\]

⟨proof⟩

**lemma** set-bts-dfs-rev:

\[\text{set } (\text{bts-dfs } f \ (\text{rev } ts)) = \text{set } (\text{bts-dfs } f \ ts)\]

⟨proof⟩

**lemma** bts-dfs-rev-distinct:

\[\text{distinct } (\text{bts-dfs } f \ ts) \Rightarrow \text{distinct } (\text{bts-dfs } f \ (\text{rev } ts))\]

⟨proof⟩

**lemma** bt-dfs-comp:

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

⟨proof⟩

**lemma** bt-dfs-comp-distinct:

\[\text{distinct } (\text{bt-dfs } (f \circ g) \ t) \Rightarrow \text{distinct } (\text{bt-dfs } g \ t)\]
\[\text{distinct } (\text{bts-dfs } (f \circ g) \ ts) \Rightarrow \text{distinct } (\text{bts-dfs } g \ ts)\]

⟨proof⟩

**lemma** bt-dfs-distinct-children:

\[\text{distinct } (\text{bt-dfs } f \ x) \Rightarrow \text{distinct } (\text{bt-dfs } f \ (\text{children } x))\]

⟨proof⟩

**fun** dfs :: (′a::linorder, ′b) bintree ⇒ ′c list 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:

\[\text{dfs } f \ (xs \ @ \ ys) = (\text{dfs } f \ xs) \ @ \ (\text{dfs } f \ ys)\]

⟨proof⟩

**lemma** set-dfs-rev:

\[\text{set } (\text{dfs } f \ (\text{rev } xs)) = \text{set } (\text{dfs } f \ xs)\]

⟨proof⟩

18
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 \text{proof} \rangle

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

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

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

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

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

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

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

lemma alist-split-set:
  \[ \text{set} \ (\text{dfs} \ \text{val} \ xs) = \text{fst} \circ \text{set} \ (\text{dfs} \ \text{alist} \ xs) \]
  \[ \text{set} \ (\text{dfs} \ \text{priority} \ xs) = \text{snd} \circ \text{set} \ (\text{dfs} \ \text{alist} \ xs) \]
  \langle \text{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 \text{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]:
bt-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)| =
lemma bt-val-augment:
  shows set (bt-dfs val t) ∪ set |q| = set |bt-augment t q|
  and set (bts-dfs val ts) ∪ 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| ∪ {v}
  set |bts-augment ts (PQ.push v a q)| = set |bts-augment ts q| ∪ {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| ∪ set |bts-augment rs q|
  set |bts-augment ts (bts-augment rs q)| =
    set |bts-augment ts q| ∪ 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) ∪ set (bt-dfs val r)
⟨proof⟩
lemma vals-merge-distinct:
\[
\text{distinct } (\text{bt-dfs val t}) \implies \text{distinct } (\text{bt-dfs val r}) \implies \\
\text{set } (\text{bt-dfs val t}) \cap \text{set } (\text{bt-dfs val r}) = \{\} \implies \\
\text{distinct } (\text{bt-dfs val (merge t r)})
\]
⟨proof⟩

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

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

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

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

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

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

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

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

\{\text{proof}\}

lemma bt-alist-augment:
\begin{align*}
distinct (bt-dfs val t) & \implies 
set (bt-dfs val t) \cap set |q| = \{\} \implies 
set (bt-dfs alist t) \cup set (PQ.alist-of q) = set (PQ.alist-of (bt-augment t q)) \\
distinct (bt-dfs val ts) & \implies 
set (bt-dfs val ts) \cap set |q| = \{\} \implies 
set (bt-dfs alist ts) \cup set (PQ.alist-of q) = set (PQ.alist-of (bt-augment ts q))
\end{align*}
\{\text{proof}\}

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

lemma alist-pqueue-priority:
distinct (vals xs) \implies (v, a) \in set (dfs alist xs)
\implies PQ.priority (pqueue xs) v = Some a
\{\text{proof}\}

lemma prios-pqueue:
distinct (vals xs) \implies set (prios xs) = set \parallel pqueue xs \parallel
\{\text{proof}\}

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

lemma alist-add-Cons:
assumes distinct (vals (x#xs))
shows $\text{set (dfs alist (add x xs))} = \text{set (dfs alist (x # xs))}$

⟨proof⟩

lemma alist-insert [simp]:

distinct (vals xs) $\implies$
$v \not\in \text{set (vals xs)}$ $\implies$
$\text{set (dfs alist (insert a v xs))} = \text{set (dfs alist xs)} \cup \{(v,a)\}$

⟨proof⟩

lemma insert-push:

distinct (vals xs) $\implies$
$v \not\in \text{set (vals xs)}$ $\implies$
$\text{set (dfs alist (insert a v xs))} = \text{set (PQ.alist-of (PQ.push v a (pqueue xs)))}$

⟨proof⟩

lemma insert-p-push:

assumes distinct (vals xs)
and $v \not\in \text{set (vals xs)}$
shows $\text{set (prios (insert a v xs))} = \text{set (PQ.push v a (pqueue xs))}$

⟨proof⟩

lemma empty-empty:

normalized xs $\implies$ xs = empty $\iff$ PQ.is-empty (pqueue xs)

⟨proof⟩

lemma bt-dfs-Min-priority:

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

⟨proof⟩

lemma is-binqueue-min-Min-prios:

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

⟨proof⟩

lemma min-p-min:

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

⟨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) ⟹ distinct (dfs val ys) ⟹
  set (dfs val xs) ∩ set (dfs val ys) = {} ⟹
  set (dfs alist (meld xs ys)) = set (dfs alist xs) ∪ 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 ≠ []
  shows set (dfs alist (delete-min xs)) =
  set (PQ.alist-of (PQ.remove-min (pqueue xs)))
⟨proof⟩

no-notation
  PQ.values (\(|-\))
  and PQ.priorities (\(||(\cdot)||\))