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 \times 'b) list. distinct (map fst xs) \&\& 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 ⇒ p = q
  ⟨proof⟩

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

definition priorities :: ('a, 'b::linorder) pq ⇒ 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)
s shows priority q (fst xp) = Some (snd xp)
⟨proof⟩

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

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

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

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

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

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

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

lemma push-values [simp]:
\( \text{set } | \text{push } k \ pq | = \text{set } | q | \cup \{k\} \)
\( \langle \text{proof} \rangle \)

lemma push-priorities [simp]:
\( k \notin \text{set } | q | \implies \text{set } | \text{push } k \ pq | = \text{set } | q | \cup \{p\} \)
\( k \in \text{set } | q | \implies \text{set } | \text{push } k \ pq | = \text{set } | q | \)
\( \langle \text{proof} \rangle \)

lemma not-is-empty-push [simp]:
\( \neg \text{is-empty } (\text{push } k \ pq) \)
\( \langle \text{proof} \rangle \)
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::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 'b ('a, 'b) bintree list

primrec priority :: ('a, 'b) bintree ⇒ 'a where
  priority (Node a - _) = a

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

primrec children :: ('a, 'b) bintree ⇒ ('a, 'b) bintree list where
  children (Node - - ts) = ts

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

lemma binqueue-induct [case-names Empty None Some, induct type: binqueue]:
  assumes P []
  and ⋀xs. P xs =⇒ P (None # xs)
  and ⋀x xs. P xs =⇒ P (Some x # xs)
  shows P xs
⟨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 =⇒ is-bintree-list l (children t)
  =⇒ is-bintree-list (Suc l) (t # ts)

abbreviation (input) is-bintree k t ≡ is-bintree-list k (children t)
lemma is-bintree-list-triv [simp]:
is-bintree-list 0 ts \iff ts = []
is-bintree-list l [] \iff l = 0
⟨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 ≠ []
s-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 \implies is-heap-list (priority t) (children t)
\implies (priority t) ≥ h \implies 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 [] \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 ≥ 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-heap-list l ts \implies is-heap-list l (\text{rev} ts)

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

\textbf{lemma} \textit{is-heap-Min-children-larger}:
\text{is-heap} t \implies \text{children} t \neq [] \implies \text{priority} t \leq \text{Min} \ (\text{priority} \ ' \ \text{set} \ (\text{children} t))

Combination of both: binqueue property

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

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

\textbf{lemma} \textit{is-binqueue-trans}:
is-binqueue l (x\#xs) \implies is-binqueue (Suc l) xs

\textbf{lemma} \textit{is-binqueue-head}:
is-binqueue l (x\#xs) \implies is-binqueue l [x]

\textbf{lemma} \textit{is-binqueue-append}:
is-binqueue l xs \implies is-binqueue (length xs + l) ys \implies is-binqueue l (xs \oplus ys)

\textbf{lemma} \textit{is-binqueue-append-dest} [dest]:
is-binqueue l (xs \oplus ys) \implies is-binqueue l xs

\textbf{lemma} \textit{is-binqueue-children}:
\textbf{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 ⇒ 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 if x = [] ∨ last xs ≠ None
(proof)

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

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

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

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

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

primrec normalize′ :: ('a, 'b) binqueue ⇒ ('a, 'b) binqueue where
normalize′ [] = []
| normalize′ (x # xs) =
(case x of None ⇒ normalize' xs | Some t ⇒ (x ≠ xs))

definition normalize :: ('a, 'b) binqueue ⇒ ('a, 'b) binqueue where
    normalize xs = rev (normalize' (rev xs))

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

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

2.3 Operations

Adding data

definition merge :: ('a::linorder, 'b) bintree ⇒ ('a, 'b) bintree ⇒ ('a, 'b) bintree
    where
    merge t1 t2 = (if priority t1 < priority t2
        then Node (priority t1) (val t1) (t2 ≠ children t1)
        else Node (priority t2) (val t2) (t1 ≠ children t2))

lemma is-bintree-list-merge:
    assumes is-bintree l t1 is-bintree l t2
    shows is-bintree (Suc l) (merge t1 t2)
    ⟨proof⟩

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

fun
    add :: ('a::linorder, 'b) bintree option ⇒ ('a, 'b) binqueue ⇒ ('a, 'b) binqueue
    where
    add None xs = xs |
    add (Some t) [] = [Some t] |
    add (Some t) (Some r # 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 (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 ⇒ ('a, 'b) binqueue ⇒ ('a, 'b) binqueue
where
meld [] ys = ys
meld xs [] = xs
meld (None # xs) (y # ys) = y # meld xs ys
meld (x # xs) (None # ys) = x # meld xs ys
meld (Some t # xs) (Some r # ys) =
  None # add (Some (merge t r)) (meld xs ys)
⟨proof⟩ termination ⟨proof⟩

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

lemma nonempty-meld [simp]:
xs ≠ [] ⇒ meld xs ys ≠ []
ys ≠ [] ⇒ meld xs ys ≠ []
⟨proof⟩

lemma nonempty-meld-commute:
meld xs ys ≠ [] ⇒ meld xs ys ≠ []
⟨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)
⟨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 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:
\[ \min (\text{Some } t \neq xs) \neq \text{None} \]
⟨proof⟩

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

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

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

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

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

primrec find :: 'a::linorder ⇒ ('a, 'b) binqueue ⇒ ('a, 'b) bintree option where
\[ \text{find } a \hspace{1em} [] = \text{None} \]
\[ \text{find } a \hspace{1em} (x#xs) = \begin{cases} \text{case } x \hspace{1em} \text{of} \hspace{1em} \text{None } \Rightarrow \text{find } a \hspace{1em} \text{xs} \\ \text{Some } t \Rightarrow \text{if priority } t = a \hspace{1em} \text{then} \hspace{1em} \text{Some } t \hspace{1em} \text{else} \hspace{1em} \text{find } a \hspace{1em} \text{xs} \end{cases} \]

declare find.simps [simp del]

lemma find-simps [simp, code]:
\[ \text{find } a \hspace{1em} [] = \text{None} \]
\[ \text{find } a \hspace{1em} (\text{None } \neq xs) = \text{find } a \hspace{1em} xs \]
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) bintree ⇒ ('a, 'b) bintree option where
find-min xs = (case min xs of None ⇒ None | Some a ⇒ find a xs)

lemma find-min-simps [simpl]:
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⟩
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)
(\text{proof})

lemma normalized-find-min-exists:
normalized xs \implies xs ≠ [] \implies ∃ t. find-min xs = Some t
(\text{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 [] = []
(\text{proof})

lemma delete-min-nonempty [simp]:
normalized xs \implies xs ≠ [] \implies find-min xs = Some t
\implies delete-min xs = normalize
  (meld (map Some (rev (children t))) (map (match (priority t)) xs))
(\text{proof})

lemma is-binqueue-delete-min:
assumes is-binqueue 0 xs
shows is-binqueue 0 (delete-min xs)
(\text{proof})

lemma normalized-delete-min:
normalized (delete-min xs)
(\text{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

\[
pop :: (\text{'a::linorder}, \text{'b}) \text{binqueue} \Rightarrow ((\text{'b} \times \text{'a}) \text{option} \times (\text{'a}, \text{'b}) \text{binqueue})
\]

Where

\[
pop \, xs = \text{(case find-min xs of}
\begin{align*}
None & \Rightarrow (\text{None, xs)} \\
\text{Some } t & \Rightarrow (\text{Some (val t, priority t), delete-min xs})
\end{align*}
\]

Lemma pop-empty [simp]:

\[
pop \, empty = (\text{None, empty})
\]
(proof)

Lemma pop-nonempty [simp]:

\[
normalized \, xs \Rightarrow xs \neq [] \Rightarrow \text{find-min xs = Some } t
\Rightarrow pop \, xs = (\text{Some (val t, priority t), normalize}
\begin{align*}
\text{(meld (map Some (rev (children t))) (map (match (priority t)) xs)))}
\end{align*}
\]
(proof)

Lemma pop-code [code]:

\[
pop \, xs = \text{(case find-min xs of}
\begin{align*}
None & \Rightarrow (\text{None, xs)} \\
\text{Some } t & \Rightarrow (\text{Some (val t, priority t), normalize}
\begin{align*}
\text{(meld (map Some (rev (children t))) (map (match (priority t)) xs)))}
\end{align*}
\]
(proof)

3 Relating Functional Binomial Queues To The Abstract Priority Queues

Notation

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

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

Primrec

\[
\begin{align*}
bt-dfs :: ((\text{'a::linorder, 'b}) \text{bintree} \Rightarrow 'c) & \Rightarrow ('a, 'b) \text{bintree} \Rightarrow 'c \text{ list} \\
bts-dfs :: ((\text{'a::linorder, 'b}) \text{bintree} \Rightarrow 'c) & \Rightarrow ('a, 'b) \text{bintree list} \Rightarrow 'c \text{ list}
\end{align*}
\]

Where

\[
\begin{align*}
bt-dfs \, f \, (\text{Node } a \, v \, ts) & = f \, (\text{Node } a \, v \, ts) \# \, \text{bts-dfs} \, f \, ts \\
\text{bts-dfs} \, f \, [] & = []
\end{align*}
\]
lemmas

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

lemma bts-dfs-append [simp]:
\[ \text{bts-dfs}\ f\ (ts\ @\ rs) = \text{bts-dfs}\ f\ ts\ @\ \text{bts-dfs}\ f\ rs \]
(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:
\[ \text{distinct}\ (\text{bt-dfs}\ f\ x) \Rightarrow \text{distinct}\ (\text{bts-dfs}\ f\ (\text{children}\ x)) \]
(proof)

fun dfs : (:: ('a::linorder, 'b) bintree ⇒ 'c) ⇒ ('a, 'b) binqueue ⇒ 'c list where
dfs f [] = []
| dfs f (None # xs) = dfs f xs
| dfs f (Some t # xs) = bt-dfs f t # dfs f xs

lemma dfs-append:
dfs f (xs @ ys) = (dfs f xs) @ (dfs f ys)
(proof)

lemma set-dfs-rev:
\[ \text{set}\ (\text{dfs}\ f\ (\text{rev}\ xs)) = \text{set}\ (\text{dfs}\ f\ xs) \]
(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 \cdot set (dfs alist xs)
set (dfs priority xs) = snd \cdot 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 ≡ dfs val
abbreviation prios where prios ≡ dfs priority
abbreviation elements where elements ≡ dfs alist

primrec
  bt-augment :: ('a::linorder, 'b) bintree ⇒ ('b, 'a) PQ.pq ⇒ ('b, 'a) PQ.pq
and
  bts-augment :: ('a::linorder, 'b) bintree list ⇒ ('b, 'a) PQ.pq ⇒ ('b, 'a) PQ.pq
where
  bt-augment (Node a v ts) q = PQ.push v a (bts-augment ts q)
  | bts-augment [] q = q
  | bts-augment (t # ts) q = bts-augment ts (bt-augment t q)

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

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

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

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

fun pqueue :: ('a::linorder, 'b) binqueue ⇒ ('b, 'a) PQ.pq where
  Empty: pqueue [] = PQ.empty
  | None: pqueue (None # xs) = pqueue xs
  | Some: pqueue (Some t # xs) = bt-augment t (pqueue xs)

lemma bt-augment-v-subset:
  set | q | ⊆ set | bt-augment t q|
  set | q | ⊆ set | bts-augment ts q|
⟨proof⟩

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

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

\textbf{lemma} \ bt-val-augment:
\begin{itemize}
\item \textbf{shows} \set (\bt-dfs \val t) \cup \set q = \set \mid \bt-augment t q
\item \textbf{and} \set (\bts-dfs \val ts) \cup \set q = \set \mid \bt-augment ts q
\end{itemize}
\langle \text{proof} \rangle

\textbf{lemma} \ vals-pqueue:
\begin{align*}
\set (\vals xs) & = \set \mid \pqueue xs
\end{align*}
\langle \text{proof} \rangle

\textbf{lemma} \ bt-augment-v-push:
\begin{align*}
\set \mid \bt-augment t (PQ.\push v a q) & = \set \mid \bt-augment t q \cup \{v\} \\
\set \mid \bts-augment ts (PQ.\push v a q) & = \set \mid \bts-augment ts q \cup \{v\}
\end{align*}
\langle \text{proof} \rangle

\textbf{lemma} \ bt-augment-v-push-commute:
\begin{align*}
\set \mid \bt-augment t (PQ.\push v a q) & = \set \mid PQ.\push v a (\bt-augment t q) \\
\set \mid \bts-augment ts (PQ.\push v a q) & = \set \mid PQ.\push v a (\bts-augment ts q)
\end{align*}
\langle \text{proof} \rangle

\textbf{lemma} \ bts-augment-v-union:
\begin{align*}
\set \mid \bt-augment t (\bts-augment rs q) & = \\
\set \mid \bt-augment t q \cup \set \mid \bts-augment rs q \\
\set \mid \bts-augment ts (\bts-augment rs q) & = \\
\set \mid \bts-augment ts q \cup \set \mid \bts-augment rs q
\end{align*}
\langle \text{proof} \rangle

\textbf{lemma} \ bt-augment-v-commute:
\begin{align*}
\set \mid \bt-augment t (\bt-augment r q) & = \set \mid \bt-augment r (\bt-augment t q) \\
\set \mid \bt-augment t (\bts-augment rs q) & = \set \mid \bts-augment rs (\bt-augment t q) \\
\set \mid \bts-augment ts (\bts-augment rs q) & = \\
\set \mid \bts-augment rs (\bts-augment ts q)
\end{align*}
\langle \text{proof} \rangle

\textbf{lemma} \ bt-augment-v-merge:
\begin{align*}
\set \mid \bt-augment (\merge t r q) & = \set \mid \bt-augment t (\bt-augment r q)
\end{align*}
\langle \text{proof} \rangle

\textbf{lemma} \ vals-merge \ [simp]:
\begin{align*}
\set (\bt-dfs \val (\merge t r)) & = \set (\bt-dfs \val t) \cup \set (\bt-dfs \val r)
\end{align*}
\langle \text{proof} \rangle

\[21\]
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⟩
lemma bt-augment-alist-union:
\[
\begin{aligned}
distinct \ (bt\text{-dfs} \ val \ (r \ # \ [t])) &\implies \\
set \ (bt\text{-dfs} \ val \ (r \ # \ [t])) \cap \ set \ |q| &\implies \\
set \ (PQ.\text{alist-of} \ (bt\text{-augment} \ t \ (bt\text{-augment} \ r \ q))) &\implies \\
set \ (PQ.\text{alist-of} \ (bt\text{-augment} \ t \ q)) \cup \ set \ (PQ.\text{alist-of} \ (bt\text{-augment} \ r \ q)) \\
\end{aligned}
\]

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

\langle proof \rangle

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

\langle proof \rangle

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

\langle proof \rangle

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

\langle proof \rangle

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

\langle proof \rangle

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

\langle proof \rangle

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

lemma insert-push:
\[\text{distinct} \ (\text{vals} \ xs) \implies v \notin \text{set} \ (\text{vals} \ xs) \implies \text{set} \ (\text{dfs} \ \text{alist} \ (\text{insert} \ a \ v \ xs)) = \text{set} \ \text{dfs} \ \text{alist} \ xs \cup \{(v,a)\}\]

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

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

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

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

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

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

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

\textbf{no-notation} 
PQ.values (\{(\_\_\_)\}) 
\textbf{and} PQ.priorities (\{(\_\_\_\_)\})