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 ⟹ 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)}
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]:
\[ \text{distinct \ (alist-of } q) \]
⟨proof⟩

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

lemma alist-of-eqI:
\[ \text{alist-of } p = \text{alist-of } q \implies p = q \]
⟨proof⟩

definition values :: (′a, ′b::linorder) pq \Rightarrow \text{′a list} \ (\|(-)\|) \ where
values q = map \text{fst} \ (alist-of q)

definition priorities :: (′a, ′b::linorder) pq \Rightarrow ′b list \ (\text{∥(-)∥}) \ where
priorities q = map \text{snd} \ (alist-of q)

lemma values-set:
\[ \text{set } |q| = \text{fst } \cdot \text{set } \text{(alist-of } q) \]
⟨proof⟩

lemma priorities-set:
\[ \text{set } ∥q∥ = \text{snd } \cdot \text{set } \text{(alist-of } q) \]
⟨proof⟩

definition is-empty :: (′a, ′b::linorder) pq \Rightarrow \text{bool} \ where
is-empty \ q \iff \text{alist-of } q = []

definition priority :: (′a, ′b::linorder) pq \Rightarrow ′a \Rightarrow ′b option \ where
priority q = \text{map-of } \ (alist-of q)

definition min :: (′a, ′b::linorder) pq \Rightarrow ′a \ where
min q = \text{fst } \ (\text{hd } \text{(alist-of } q))

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

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

lemma not-is-empty-alist-of [dest]:
\[ \neg \text{is-empty } q \implies \text{alist-of } q \neq [] \]
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 \notin set |empty|
⟨proof⟩

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

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

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

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

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

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

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

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

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

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

4
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 \( \forall xs. P xs \implies P (\text{None} \# xs) \)
  and \( \forall x xs. P xs \implies P (\text{Some} x \# xs) \)
  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 \( ts1, ts2 \)
- 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)

6
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 \ (\text{rev} \ ts)
(proof)

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

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

Combination of both: binqueue property

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

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

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

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

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

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

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

\textbf{lemma} is-binqueue-select:
\[\text{is-binqueue } (\text{rev } ts) \implies \exists k. \text{is-bintree } k t \land \text{is-heap } t\]
(\text{proof})

\textbf{Normalized representation}

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

\textbf{lemma} normalized\text{-last-not-None}:
\begin{itemize}
  \item sometimes the inductive definition might work better
\end{itemize}
\[\text{normalized } \text{xs} \leftrightarrow \text{xs} = [] \lor \text{last } \text{xs} \neq \text{None}\]
(\text{proof})

\textbf{lemma} normalized\text{-simps} [\text{simp}]:
\begin{align*}
\text{normalized } [] & \leftrightarrow \text{True} \\
\text{normalized } (\text{Some } t \# \text{xs}) & \leftrightarrow \text{normalized } \text{xs} \\
\text{normalized } (\text{None } \# \text{xs}) & \leftrightarrow \text{xs} \neq [] \land \text{normalized } \text{xs}
\end{align*}
(\text{proof})

\textbf{lemma} normalized\text{-map-\text{Some}} [\text{simp}]:
\[\text{normalized } (\text{map } \text{Some } \text{xs})\]
(\text{proof})

\textbf{lemma} normalized\text{-Cons}:
\[\text{normalized } (x \# \text{xs}) \implies \text{normalized } \text{xs}\]
(\text{proof})

\textbf{lemma} normalized\text{-append}:
\[\text{normalized } \text{xs} \implies \text{normalized } \text{ys} \implies \text{normalized } (\text{xs} @ \text{ys})\]
(\text{proof})

\textbf{lemma} normalized\text{-not-None}:
\[\text{normalized } \text{xs} \implies \text{set } \text{xs} \neq \{\text{None}\}\]
(\text{proof})

\textbf{primrec} normalize' :: ('a, 'b) \text{binqueue} \Rightarrow ('a, 'b) \text{binqueue where}
\begin{align*}
normalize'[] & = [] \\
normalize'(x \# \text{xs}) & =
\end{align*}
(case x of None ⇒ normalize′ xs | Some t ⇒ (x # xs))

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

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

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

2.3 Operations
Adding data

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

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

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

fun add :: ('a::linorder, 'b) bintree option ⇒ ('a, 'b) binqueue ⇒ ('a, 'b) binqueue
where
  add None xs = xs
| add (Some t) [] = [Some t]
| add (Some t) (None # xs) = Some t # xs
| add (Some t) (Some r # xs) = None # add (Some (merge t r)) xs

lemma add-Some-not-Nil [simp]:
  add (Some t) xs ≠ []
⟨proof⟩

lemma normalized-add:
assumes normalized xs
shows normalized (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 Some x′ else y′))

lemma least-simps [simp, code]:
  least None x = x
  least x None = x
  least (Some x′) (Some y′) = (if x′ ≤ y′ then Some x′ else Some y′)
⟨proof⟩

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

interpretation least!: semilattice least ⟨proof⟩

definition min :: ('a::linorder, 'b) binqueue ⇒ 'a option
  where
min xs = fold least (map (map-option priority) xs) None

lemma min-simps [simp]:
  min [] = None
  min (None # xs) = min xs
  min (Some t # xs) = least (Some (priority t)) (min xs)
⟨proof⟩

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

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

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

lemma min-None-None:
\[
\begin{align*}
\text{min } xs &= \text{None } \iff xs = [] \land \text{set } xs = \{\text{None}\}
\end{align*}
\]
⟨proof⟩

lemma normalized-min-not-None:
\[
\begin{align*}
\text{normalized } xs &\implies xs \neq [] \implies \text{min } xs \neq \text{None}
\end{align*}
\]
⟨proof⟩

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

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

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

declare find.simps [simp del]

lemma find-simps [simp, code]:
\[
\begin{align*}
\text{find } a [] &= \text{None} \\
\text{find } a (\text{None } \# xs) &= \text{find } a xs
\end{align*}
\]
\begin{quote}
find a (Some t \# xs) = (if priority t = a then Some t else find a xs)
\end{quote}

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

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

\textbf{lemma} find-None:
find a xs = None \implies Some a \notin set (map (map-option priority) xs)
\textbf{⟨proof⟩}

\textbf{lemma} find-exist:
find a xs = Some t \implies Some t \in set xs
\textbf{⟨proof⟩}

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

\textbf{lemma} find-min-simps [simp]:
find-min [] = None
find-min (None \# xs) = find-min xs
\textbf{⟨proof⟩}

\textbf{lemma} find-min-single:
find-min [x] = x
\textbf{⟨proof⟩}

\textbf{lemma} min-eq-find-min-None:
min xs = None \iff find-min xs = None
\textbf{⟨proof⟩}

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

\textbf{lemma} find-min-exist:
\textbf{assumes} find-min xs = Some t
\textbf{shows} Some t \in 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-binqueue-delete-min:
assumes is-binqueue 0 xs
shows is-binqueue 0 (delete-min xs)
⟨proof⟩

lemma normalized-delete-min:
normalized (delete-min xs)
⟨proof⟩

Dedicated grand unified operation for generated program
meld :: (a, b) bintree option \Rightarrow (a::linorder, b) binqueue \Rightarrow (a, b) binqueue \Rightarrow (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 (Some t # xs) (Some r # ys) =
  meld' None xs 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 \equiv []

definition
insert :: a::linorder \Rightarrow b \Rightarrow (a, b) binqueue \Rightarrow (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 \Rightarrow is-binqueue 0 (insert a v xs)
⟨proof⟩
lemma normalized-insert:
  \[\text{normalized } xs \implies \text{normalized } (\text{insert } a \; v \; xs)\]
  ⟨proof⟩

definition pop :: \('a::linorder, 'b\) \text{binqueue} \Rightarrow (\('b \times 'a\) \text{option} \times (\('a, 'b\) \text{binqueue})
where
  pop \; xs = (\text{case find-min } xs \text{ of}
  \quad \text{None} \Rightarrow (\text{None}, xs)
  \quad \text{Some } t \Rightarrow (\text{Some } (\text{val } t, \text{priority } t), \text{delete-min } xs))

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

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

lemma pop-code [code]:
  pop \; xs = (\text{case find-min } xs \text{ of}
  \quad \text{None} \Rightarrow (\text{None}, xs)
  \quad \text{Some } t \Rightarrow (\text{Some } (\text{val } t, \text{priority } t), \text{normalize}
  \quad \quad (\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
  \(\text{PQ.values } ([\text{-}])\)
  \text{and } \(\text{PQ.priorities } [[[\text{-}]]]\)

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

primrec bt-dfs :: ((\'a::linorder, \'b) \text{bintree} \Rightarrow \text{'c}) \Rightarrow (\'a, \'b) \text{bintree} \Rightarrow \text{'c list}
  \text{and } bts-dfs :: ((\'a::linorder, \'b) \text{bintree} \Rightarrow \text{'c}) \Rightarrow (\'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
lemma \textit{bt-dfs-simp}: 
\textit{bt-dfs f t} = f t \# \textit{bt-dfs f} \text{ (children t)}
\langle \text{proof} \rangle

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

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

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

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

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

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

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

lemma \textit{dfs-append}: 
\text{dfs f} \; (xs @ ys) = (\text{dfs f} \; x) @ (\text{dfs f} \; y)
\langle \text{proof} \rangle

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

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

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

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

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

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

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

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

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

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

lemma bt-alist-augment:
  distinct (bt-dfs val t) \implies
  set (bt-dfs val t) \cap set |q| = \{\} \implies
  set (bt-alist t) \cup set (PQ-alist-of q) = set (PQ-alist-of (bt-augment t q))
\langle proof \rangle

lemma alist-pqueue:
  distinct (vals xs) \implies set (dfs alist xs) = set (PQ-alist-of (pqueue xs))
\langle proof \rangle

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

lemma prios-pqueue:
  distinct (vals xs) \implies set (prios xs) = set ||pqueue xs||
\langle proof \rangle

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-alist (merge t r)) = set (bt-alist t) \cup set (bt-alist r)
\langle proof \rangle

lemma alist-add-Cons:
  assumes distinct (vals (x#xs))
shows set (dfs alist (add x xs)) = set (dfs alist (x ≠ xs))
⟨proof⟩

lemma alist-insert [simp]:
distinct (vals xs) ⟹
v ∉ set (vals xs) ⟹
set (dfs alist (insert a v xs)) = set (dfs alist xs) ∪ {(v,a)}
⟨proof⟩

lemma insert-push:
distinct (vals xs) ⟹
v ∉ set (vals xs) ⟹
set (dfs alist (insert a v xs)) = set (PQ.alist-of (PQ.push v a (pqueue xs)))
⟨proof⟩

lemma insert-p-push:
assumes distinct (vals xs)
and v ∉ set (vals xs)
shows set (prios (insert a v xs)) = set ∥PQ.push v a (pqueue xs)∥
⟨proof⟩

lemma empty-empty:
normalized xs ⟹ xs = empty ⟷ PQ.is-empty (pqueue xs)
⟨proof⟩

lemma bt-dfs-Min-priority:
assumes is-heap t
shows priority t = Min (set (bt-dfs priority t))
⟨proof⟩

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

lemma min-p-min:
assumes is-binqueue l xs
and xs ≠ []
and normalized xs
and distinct (vals xs)
and distinct (prios xs)
shows min xs = 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⟩

25
lemma alist-meld:
\[
\text{distinct (dfs val xs) } \implies \text{distinct (dfs val ys) } \implies \\
\text{set (dfs val xs) } \cap \text{set (dfs val ys) } = \{\} \implies \\
\text{set (dfs alist (meld xs ys)) } = \text{set (dfs alist xs) } \cup \text{set (dfs alist ys)}
\]
⟨proof⟩

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

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

no-notation
\[
PQ.values (\{|-\}|) \\
\text{and PQ.priorities (\{||-\}||)}
\]