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 $\Longrightarrow$ *set* (*tl q*) = *set q* − {*hd q*}
⟨*proof*⟩

1.2 Type of abstract priority queues

typedef (**overloaded**) ('*a*, '*b*::*linorder*) *pq* =
{*xs* :: ('*a* × '*b*) *list*. *distinct* (*map fst xs*) ∧ *sorted* (*map snd xs*)}
morphisms *alist-of Abs-pq*
⟨*proof*⟩

lemma *alist-of-Abs-pq*:
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* ⇒ '*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* ≠ []

⟨*proof*⟩

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* (‹‖(-)‖›)

2 Functional Binomial Queues

2.1 Type definition and projections

datatype $('a, 'b)$ *bintree* = *Node* $'a$ $'b$ $('a, 'b)$ *bintree list*

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

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

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

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

lemma *binqueue-induct* [*case-names Empty None Some*, *induct type*: *binqueue*]:
assumes *P* []
and $\bigwedge$*xs*. *P xs* $\Longrightarrow$ *P* (*None* # *xs*)
and $\bigwedge$*x xs*. *P xs* $\Longrightarrow$ *P* (*Some x* # *xs*)
shows *P xs*
⟨*proof*⟩

Terminology:

- values v, w or $v1$, $v2$
- priorities a, b or $a1$, $a2$
- bintrees t, r or $t1$, $t2$
- bintree lists ts, rs or $ts1$, $ts2$
- binqueue element x, y or $x1$, $x2$
- binqueues = binqueue element lists xs, ys or $xs1$, $xs2$
- abstract priority queues q, p or $q1$, $q2$

2.2 Binomial queue properties

Binomial tree property

inductive *is-bintree-list* :: *nat* $\Rightarrow$ $('a, 'b)$ *bintree list* $\Rightarrow$ *bool* **where**
is-bintree-list-Nil [*simp*]: *is-bintree-list* 0 []
| *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 $\Longrightarrow$ *is-heap-list l* (*rev ts*)
⟨*proof*⟩

lemma *is-heap-children-larger*:
is-heap t $\Longrightarrow \forall\ x \in$ *set* (*children t*). *priority x* $\geq$ *priority t*
⟨*proof*⟩

lemma *is-heap-Min-children-larger*:
is-heap t $\Longrightarrow$ *children t* $\neq$ [] $\Longrightarrow$
priority t $\leq$ *Min* (*priority* ‘ *set* (*children t*))
⟨*proof*⟩

Combination of both: binqueue property

inductive *is-binqueue* :: *nat* $\Rightarrow$ (*'a*::*linorder*, *'b*) *binqueue* $\Rightarrow$ *bool* **where**
Empty: *is-binqueue l* []
| *None*: *is-binqueue* (*Suc l*) *xs* $\Longrightarrow$ *is-binqueue l* (*None* # *xs*)
| *Some*: *is-binqueue* (*Suc l*) *xs* $\Longrightarrow$ *is-bintree l t*
$\Longrightarrow$ *is-heap t* $\Longrightarrow$ *is-binqueue l* (*Some t* # *xs*)

lemma *is-binqueue-simp* [*simp*]:
is-binqueue l [] $\longleftrightarrow$ *True*
is-binqueue l (*Some t* # *xs*) $\longleftrightarrow$
is-bintree l t $\wedge$ *is-heap t* $\wedge$ *is-binqueue* (*Suc l*) *xs*
is-binqueue l (*None* # *xs*) $\longleftrightarrow$ *is-binqueue* (*Suc l*) *xs*
⟨*proof*⟩

lemma *is-binqueue-trans*:
is-binqueue l (*x*#*xs*) $\Longrightarrow$ *is-binqueue* (*Suc l*) *xs*
⟨*proof*⟩

lemma *is-binqueue-head*:
is-binqueue l (*x*#*xs*) $\Longrightarrow$ *is-binqueue l* [*x*]
⟨*proof*⟩

lemma *is-binqueue-append*:
is-binqueue l xs $\Longrightarrow$ *is-binqueue* (*length xs* + *l*) *ys* $\Longrightarrow$ *is-binqueue l* (*xs* @ *ys*)
⟨*proof*⟩

lemma *is-binqueue-append-dest* [*dest*]:
is-binqueue l (*xs* @ *ys*) $\Longrightarrow$ *is-binqueue l xs*
⟨*proof*⟩

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

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

lemma *is-binqueue-select*:
is-binqueue l xs ⟹ *Some t* ∈ *set xs* ⟹ ∃ *k*. *is-bintree k t* ∧ *is-heap t*
⟨*proof*⟩

Normalized representation

inductive *normalized* :: ('*a*, '*b*) *binqueue* ⇒ *bool* **where**
normalized-Nil: *normalized* []
| *normalized-single*: *normalized* [*Some t*]
| *normalized-append*: *xs* ≠ [] ⟹ *normalized xs* ⟹ *normalized* (*ys* @ *xs*)

lemma *normalized-last-not-None*:
— sometimes the inductive definition might work better
normalized xs ⟷ *xs* = [] ∨ *last xs* ≠ *None*
⟨*proof*⟩

lemma *normalized-simps* [*simp*]:
normalized [] ⟷ *True*
normalized (*Some t* # *xs*) ⟷ *normalized xs*
normalized (*None* # *xs*) ⟷ *xs* ≠ [] ∧ *normalized xs*
⟨*proof*⟩

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

lemma *normalized-Cons*:
normalized (*x*#*xs*) ⟹ *normalized xs*
⟨*proof*⟩

lemma *normalized-append*:
normalized xs ⟹ *normalized ys* ⟹ *normalized* (*xs*@*ys*)
⟨*proof*⟩

lemma *normalized-not-None*:
normalized xs ⟹ *set xs* ≠ {*None*}
⟨*proof*⟩

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

(*case x of None ⇒ normalize′ xs | Some t ⇒ (x # xs)*)

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

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

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

2.3 Operations

Adding data

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

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

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

fun
add :: (*′a*::*linorder*, *′b*) *bintree option* ⇒ (*′a*, *′b*) *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 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*:
min [*x*] = *Some a* ⟹ *priority* (*the x*) = *a*
min [*x*] = *None* ⟹ *x* = *None*
⟨*proof*⟩

lemma *min-Some-not-None*:
min (*Some t* # *xs*) ≠ *None*
⟨*proof*⟩

lemma *min-None-trans*:
assumes *min* (*x*#*xs*) = *None*
shows *min xs* = *None*
⟨*proof*⟩

lemma *min-None-None*:
min xs = *None* ⟷ *xs* = [] ∨ *set xs* = {*None*}
⟨*proof*⟩

lemma *normalized-min-not-None*:
normalized xs ⟹ *xs* ≠ [] ⟹ *min xs* ≠ *None*
⟨*proof*⟩

lemma *min-is-min*:
assumes *normalized xs*
and *xs* ≠ []
and *min xs* = *Some a*
shows ∀ *x* ∈ *set xs*. *x* = *None* ∨ *a* ≤ *priority* (*the x*)
⟨*proof*⟩

lemma *min-exists*:
assumes *min xs* = *Some a*
shows *Some a* ∈ *map-option priority* ‘ *set xs*
⟨*proof*⟩

primrec *find* :: *'a*::*linorder* ⇒ (*'a*, *'b*) *binqueue* ⇒ (*'a*, *'b*) *bintree option* **where**
find a [] = *None*
| *find a* (*x*#*xs*) = (*case x of None* ⇒ *find a xs*
| *Some t* ⇒ *if priority t* = *a then Some t else find a xs*)

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

lemma *find-simps* [*simp*, *code*]:
find a [] = *None*
find a (*None* # *xs*) = *find a xs*

find a (Some t # xs) = (if priority t = a then Some t else find a xs)
⟨*proof*⟩

lemma *find-works*:
assumes *Some a ∈ set (map (map-option priority) xs)*
shows *∃ t. find a xs = Some t ∧ priority t = a*
⟨*proof*⟩

lemma *find-works-not-None*:
Some a ∈ set (map (map-option priority) xs) ⟹ find a xs ≠ None
⟨*proof*⟩

lemma *find-None*:
find a xs = None ⟹ Some a ∉ set (map (map-option priority) xs)
⟨*proof*⟩

lemma *find-exist*:
find a xs = Some t ⟹ Some t ∈ set xs
⟨*proof*⟩

definition *find-min* :: (*'a*::*linorder*, *'b*) *binqueue* ⇒ (*'a*, *'b*) *bintree option* **where**
find-min xs = (case min xs of None ⇒ None | Some a ⇒ find a xs)

lemma *find-min-simps* [*simp*]:
find-min [] = None
find-min (None # xs) = find-min xs
⟨*proof*⟩

lemma *find-min-single*:
find-min [x] = x
⟨*proof*⟩

lemma *min-eq-find-min-None*:
min xs = None ⟷ find-min xs = None
⟨*proof*⟩

lemma *min-eq-find-min-Some*:
min xs = Some a ⟷ (∃ t. find-min xs = Some t ∧ priority t = a)
⟨*proof*⟩

lemma *find-min-exist*:
assumes *find-min xs = Some t*
shows *Some t ∈ set xs*
⟨*proof*⟩

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

lemma *normalized-find-min-exists*:
normalized xs $\Longrightarrow$ $xs \neq []$ $\Longrightarrow$ $\exists t.$ *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 $\Longrightarrow$ $xs \neq []$ $\Longrightarrow$ *find-min xs* = *Some t*
$\Longrightarrow$ *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

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 $\Longrightarrow$ *normalized* (*insert a v xs*)
⟨*proof*⟩

definition
pop :: (′*a*::*linorder*, ′*b*) *binqueue* ⇒ ((′*b* × ′*a*) *option* × (′*a*, ′*b*) *binqueue*)
where
pop xs = (*case find-min xs of*
None ⇒ (*None*, *xs*)
| *Some t* ⇒ (*Some* (*val t*, *priority t*), *delete-min xs*))

lemma *pop-empty* [*simp*]:
pop empty = (*None*, *empty*)
⟨*proof*⟩

lemma *pop-nonempty* [*simp*]:
normalized xs $\Longrightarrow$ *xs* ≠ [] $\Longrightarrow$ *find-min xs* = *Some t*
$\Longrightarrow$ *pop xs* = (*Some* (*val t*, *priority t*), *normalize*
(*meld* (*map Some* (*rev* (*children t*))) (*map* (*match* (*priority t*)) *xs*)))
⟨*proof*⟩

lemma *pop-code* [*code*]:
pop xs = (*case find-min xs of*
None ⇒ (*None*, *xs*)
| *Some t* ⇒ (*Some* (*val t*, *priority t*), *normalize*
(*meld* (*map Some* (*rev* (*children t*))) (*map* (*match* (*priority t*)) *xs*))))
⟨*proof*⟩

3 Relating Functional Binomial Queues To The Abstract Priority Queues

notation
PQ.values (‹|(-)|›)
and *PQ.priorities* (‹‖(-)‖›)

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

primrec *bt-dfs* :: ((′*a*::*linorder*, ′*b*) *bintree* ⇒ ′*c*) ⇒ (′*a*, ′*b*) *bintree* ⇒ ′*c list*
and *bts-dfs* :: ((′*a*::*linorder*, ′*b*) *bintree* ⇒ ′*c*) ⇒ (′*a*, ′*b*) *bintree list* ⇒ ′*c list*
where
bt-dfs f (*Node a v ts*) = *f* (*Node a v ts*) # *bts-dfs f ts*
| *bts-dfs f* [] = []

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

lemma *bt-dfs-simp*:
bt-dfs f t = f t # bts-dfs f (children t)
⟨*proof*⟩

lemma *bts-dfs-append* [*simp*]:
bts-dfs f (ts @ rs) = bts-dfs f ts @ bts-dfs f rs
⟨*proof*⟩

lemma *set-bts-dfs-rev*:
set (bts-dfs f (rev ts)) = set (bts-dfs f ts)
⟨*proof*⟩

lemma *bts-dfs-rev-distinct*:
distinct (bts-dfs f ts) ⟹ distinct (bts-dfs f (rev ts))
⟨*proof*⟩

lemma *bt-dfs-comp*:
bt-dfs (f ∘ g) t = map f (bt-dfs g t)
bts-dfs (f ∘ g) ts = map f (bts-dfs g ts)
⟨*proof*⟩

lemma *bt-dfs-comp-distinct*:
distinct (bt-dfs (f ∘ g) t) ⟹ distinct (bt-dfs g t)
distinct (bts-dfs (f ∘ g) ts) ⟹ distinct (bts-dfs g ts)
⟨*proof*⟩

lemma *bt-dfs-distinct-children*:
distinct (bt-dfs f x) ⟹ distinct (bts-dfs f (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*:
set (dfs f (rev xs)) = set (dfs f xs)
⟨*proof*⟩

lemma *set-dfs-Cons*:
set (*dfs f* (*x* # *xs*)) = *set* (*dfs f xs*) ∪ *set* (*dfs f* [*x*])
⟨*proof*⟩

lemma *dfs-comp*:
dfs (*f* ∘ *g*) *xs* = *map f* (*dfs g xs*)
⟨*proof*⟩

lemma *dfs-comp-distinct*:
distinct (*dfs* (*f* ∘ *g*) *xs*) ⟹ *distinct* (*dfs g xs*)
⟨*proof*⟩

lemma *dfs-distinct-member*:
distinct (*dfs f xs*) ⟹
Some x ∈ *set xs* ⟹
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* ⇒ ('*b* × '*a*) **where**
alist (*Node a v* -) = (*v*, *a*)

lemma *alist-split-pre*:
val t = (*fst* ∘ *alist*) *t*
priority t = (*snd* ∘ *alist*) *t*
⟨*proof*⟩

lemma *alist-split*:
val = *fst* ∘ *alist*
priority = *snd* ∘ *alist*
⟨*proof*⟩

lemma *alist-split-set*:
set (*dfs val xs*) = *fst* ‘ *set* (*dfs alist xs*)
set (*dfs priority xs*) = *snd* ‘ *set* (*dfs alist xs*)
⟨*proof*⟩

lemma *in-set-in-alist*:
assumes *Some t* ∈ *set xs*
shows (*val t*, *priority t*) ∈ *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*|

set |*bts-augment ts* (*bt-augment r q*)| =
set |*bts-augment ts q*| ∪ *set* |*bt-augment r q*|
⟨*proof*⟩

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*) ⟹ *distinct* (*bt-dfs val r*) ⟹
set (*bt-dfs val t*) ∩ *set* (*bt-dfs val r*) = {} ⟹
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*) ∩ *set* (*dfs val* [*x*]) = {}
shows *distinct* (*vals* (*add x xs*))
⟨*proof*⟩

lemma *vals-insert* [*simp*]:
set (*vals* (*insert a v xs*)) = *set* (*vals xs*) ∪ {*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*) ∪ *set* (*dfs val ys*)
⟨*proof*⟩

lemma *vals-meld-distinct*:
distinct (*dfs val xs*) ⟹ *distinct* (*dfs val ys*) ⟹
set (*dfs val xs*) ∩ *set* (*dfs val ys*) = {} ⟹
distinct (*dfs val* (*meld xs ys*))
⟨*proof*⟩

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

lemma *bt-augment-alist-in*:
(*v*,*a*) ∈ *set* (*PQ.alist-of q*) ⟹ (*v*,*a*) ∈ *set* (*PQ.alist-of* (*bt-augment t q*))
(*v*,*a*) ∈ *set* (*PQ.alist-of q*) ⟹ (*v*,*a*) ∈ *set* (*PQ.alist-of* (*bts-augment ts q*))
⟨*proof*⟩

lemma *bt-augment-alist-union*:
distinct (*bts-dfs val* (*r* # [*t*])) ⟹
set (*bts-dfs val* (*r* # [*t*])) ∩ *set* |*q*| = {} ⟹
set (*PQ.alist-of* (*bt-augment t* (*bt-augment r q*))) =
set (*PQ.alist-of* (*bt-augment t q*)) ∪ *set* (*PQ.alist-of* (*bt-augment r q*))

distinct (*bts-dfs val* (*r* # *ts*)) ⟹
set (*bts-dfs val* (*r* # *ts*)) ∩ *set* |*q*| = {} ⟹
set (*PQ.alist-of* (*bts-augment ts* (*bt-augment r q*))) =
set (*PQ.alist-of* (*bts-augment ts q*)) ∪ *set* (*PQ.alist-of* (*bt-augment r q*))
⟨*proof*⟩

lemma *bt-alist-augment*:
distinct (*bt-dfs val t*) ⟹
set (*bt-dfs val t*) ∩ *set* |*q*| = {} ⟹
set (*bt-dfs alist t*) ∪ *set* (*PQ.alist-of q*) = *set* (*PQ.alist-of* (*bt-augment t q*))

distinct (*bts-dfs val ts*) ⟹
set (*bts-dfs val ts*) ∩ *set* |*q*| = {} ⟹
set (*bts-dfs alist ts*) ∪ *set* (*PQ.alist-of q*) =
set (*PQ.alist-of* (*bts-augment ts q*))
⟨*proof*⟩

lemma *alist-pqueue*:
distinct (*vals xs*) ⟹ *set* (*dfs alist xs*) = *set* (*PQ.alist-of* (*pqueue xs*))
⟨*proof*⟩

lemma *alist-pqueue-priority*:
distinct (*vals xs*) ⟹ (*v*, *a*) ∈ *set* (*dfs alist xs*)
⟹ *PQ.priority* (*pqueue xs*) *v* = *Some a*
⟨*proof*⟩

lemma *prios-pqueue*:
distinct (*vals xs*) ⟹ *set* (*prios xs*) = *set* ‖*pqueue xs*‖
⟨*proof*⟩

lemma *alist-merge* [*simp*]:
distinct (*bt-dfs val t*) ⟹ *distinct* (*bt-dfs val r*) ⟹
set (*bt-dfs val t*) ∩ *set* (*bt-dfs val r*) = {} ⟹
set (*bt-dfs alist* (*merge t r*)) = *set* (*bt-dfs alist t*) ∪ *set* (*bt-dfs alist r*)
⟨*proof*⟩

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 $xs \neq []$
and *normalized xs*
and *distinct* (*vals xs*)
and *distinct* (*prios xs*)
shows *priority* (*the* (*find-min xs*)) =
the (*PQ.priority* (*pqueue xs*) (*PQ.min* (*pqueue xs*)))
⟨*proof*⟩

lemma *find-min-v-min*:
assumes *is-binqueue l xs*
and $xs \neq []$
and *normalized xs*
and *distinct* (*vals xs*)
and *distinct* (*prios xs*)
shows *val* (*the* (*find-min xs*)) = *PQ.min* (*pqueue xs*)
⟨*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*) $\Longrightarrow$ *distinct* (*dfs val ys*) $\Longrightarrow$
set (*dfs val xs*) $\cap$ *set* (*dfs val ys*) = {} $\Longrightarrow$
set (*dfs alist* (*meld xs ys*)) = *set* (*dfs alist xs*) $\cup$ *set* (*dfs alist ys*)
⟨*proof*⟩

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

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

no-notation
PQ.values (‹|(-)|›)
and *PQ.priorities* (‹‖(-)‖›)