Binomial Heaps and Skew Binomial Heaps

Rene Meis Finn Nielsen Peter Lammich

February 6, 2026

Abstract

We implement and prove correct binomial heaps and skew binomial heaps. Both are data-structures for priority queues. While binomial heaps have logarithmic *findMin*, *deleteMin*, *insert*, and *meld* operations, skew binomial heaps have constant time *findMin*, *insert*, and *meld* operations, and only the *deleteMin*-operation is logarithmic. This is achieved by using *skew links* to avoid cascading linking on *insert*-operations, and *data-structural bootstrapping* to get constant-time *findMin* and *meld* operations. Our implementation follows the paper of Brodal and Okasaki [1].

Contents

1 Binomial Heaps **3**
- 1.1 Datatype Definition . . . 3
  - 1.1.1 Abstraction to Multiset . . . 3
  - 1.1.2 Invariant . . . 3
  - 1.1.3 Heap Ordering . . . 5
  - 1.1.4 Height and Length . . . 6
- 1.2 Operations . . . 7
  - 1.2.1 Empty . . . 7
  - 1.2.2 Insert . . . 7
  - 1.2.3 Meld . . . 8
  - 1.2.4 Find Minimal Element . . . 9
  - 1.2.5 Delete Minimal Element . . . 10
- 1.3 Hiding the Invariant . . . 12
  - 1.3.1 Datatype . . . 12
  - 1.3.2 Operations . . . 12
  - 1.3.3 Correctness . . . 13
- 1.4 Documentation . . . 14

2 Skew Binomial Heaps **15**
- 2.1 Datatype . . . 16
  - 2.1.1 Abstraction to Multisets . . . 16
  - 2.1.2 Invariant . . . 16
  - 2.1.3 Heap Order . . . 20
  - 2.1.4 Height and Length . . . 20
- 2.2 Operations . . . 21
  - 2.2.1 Empty Tree . . . 21
  - 2.2.2 Insert . . . 22
  - 2.2.3 meld . . . 23
  - 2.2.4 Find Minimal Element . . . 25
  - 2.2.5 Delete Minimal Element . . . 26
- 2.3 Bootstrapping . . . 29
  - 2.3.1 Auxiliary . . . 29
  - 2.3.2 Datatype . . . 30
  - 2.3.3 Specialization Boilerplate . . . 30
  - 2.3.4 Bootstrapping: Phase 1 . . . 35
  - 2.3.5 Bootstrapping: Phase 2 . . . 37
- 2.4 Hiding the Invariant . . . 39
  - 2.4.1 Datatype . . . 39
  - 2.4.2 Operations . . . 40
  - 2.4.3 Correctness . . . 41
- 2.5 Documentation . . . 42

1 Binomial Heaps

theory *BinomialHeap*
imports *Main HOL−Library.Multiset*
begin

locale *BinomialHeapStruc-loc*
begin

1.1 Datatype Definition

Binomial heaps are lists of binomial trees.

datatype ('*e*, '*a*) *BinomialTree* =
Node (*val*: '*e*) (*prio*: '*a*::*linorder*) (*rank*: *nat*) (*children*: ('*e* , '*a*) *BinomialTree list*)
type-synonym ('*e*, '*a*) *BinomialQueue-inv* = ('*e*, '*a*::*linorder*) *BinomialTree list*

Combine two binomial trees (of rank r) to one (of rank $r + 1$).

fun *link* :: ('*e*, '*a*::*linorder*) *BinomialTree* ⇒ ('*e*, '*a*) *BinomialTree* ⇒
('*e*, '*a*) *BinomialTree* **where**
link (*Node e1 a1 r1 ts1*) (*Node e2 a2 r2 ts2*) =
(*if* *a1*≤*a2*
then (*Node e1 a1* (*Suc r1*) ((*Node e2 a2 r2 ts2*)#*ts1*))
else (*Node e2 a2* (*Suc r2*) ((*Node e1 a1 r1 ts1*)#*ts2*)))

1.1.1 Abstraction to Multiset

Return a multiset with all (element, priority) pairs from a queue.

fun *tree-to-multiset*
:: ('*e*, '*a*::*linorder*) *BinomialTree* ⇒ ('*e* × '*a*) *multiset*
and *queue-to-multiset*
:: ('*e*, '*a*::*linorder*) *BinomialQueue-inv* ⇒ ('*e* × '*a*) *multiset* **where**
tree-to-multiset (*Node e a r ts*) = {#(*e*,*a*)#} + *queue-to-multiset ts* |
queue-to-multiset [] = {#} |
queue-to-multiset (*t*#*q*) = *tree-to-multiset t* + *queue-to-multiset q*

lemma *qtmset-append-union*[*simp*]: *queue-to-multiset* (*q* @ *q*′) =
queue-to-multiset q + *queue-to-multiset q*′
⟨*proof*⟩

lemma *qtmset-rev*[*simp*]: *queue-to-multiset* (*rev q*) = *queue-to-multiset q*
⟨*proof*⟩

1.1.2 Invariant

We first formulate the invariant for single binomial trees, and then extend the invariant to binomial heaps (lists of binomial trees). The invariant for

trees claims that a tree labeled rank 0 has no children, and a tree labeled rank $r + 1$ is the result of a link operation of two rank r trees.

function *tree-invar* :: (*'e*, *'a*::*linorder*) *BinomialTree* ⇒ *bool* **where**
tree-invar (*Node e a 0 ts*) = (*ts* = []) |
tree-invar (*Node e a* (*Suc r*) *ts*) =
(∃ *e1 a1 ts1 e2 a2 ts2*.
tree-invar (*Node e1 a1 r ts1*) ∧
tree-invar (*Node e2 a2 r ts2*) ∧
(*Node e a* (*Suc r*) *ts*) = *link* (*Node e1 a1 r ts1*) (*Node e2 a2 r ts2*))
⟨*proof*⟩
termination
⟨*proof*⟩

A queue satisfies the invariant, iff all trees inside the queue satisfy the invariant, and the queue contains only trees of distinct rank and is ordered by rank

First part: All trees of the queue satisfy the tree invariant:

definition *queue-invar* :: (*'e*, *'a*::*linorder*) *BinomialQueue-inv* ⇒ *bool* **where**
queue-invar q ≡ (∀ *t* ∈ *set q*. *tree-invar t*)

Second part: Trees have distinct rank, and are ordered by ascending rank:

fun *rank-invar* :: (*'e*, *'a*::*linorder*) *BinomialQueue-inv* ⇒ *bool* **where**
rank-invar [] = *True* |
rank-invar [*t*] = *True* |
rank-invar (*t* # *t'* # *bq*) = (*rank t* < *rank t'* ∧ *rank-invar* (*t'* # *bq*))

lemma *queue-invar-simps*[*simp*]:
queue-invar []
queue-invar (*t*#*q*) ⟷ *tree-invar t* ∧ *queue-invar q*
queue-invar (*q*@*q'*) ⟷ *queue-invar q* ∧ *queue-invar q'*
⟨*proof*⟩

Invariant for binomial queues:

definition *invar q* == *queue-invar q* ∧ *rank-invar q*

lemma *mset-link*[*simp*]: (*tree-to-multiset* (*link t1 t2*))
= (*tree-to-multiset t1*) + (*tree-to-multiset t2*)
⟨*proof*⟩

lemma *link-tree-invar*:
⟦*tree-invar t1*; *tree-invar t2*; *rank t1* = *rank t2*⟧ ⟹ *tree-invar* (*link t1 t2*)
⟨*proof*⟩

lemma *invar-children*:
assumes *tree-invar* ((*Node e a r ts*)::((*'e*, *'a*::*linorder*) *BinomialTree*))
shows *queue-invar ts* ⟨*proof*⟩

lemma *invar-children'*: *tree-invar t* $\Longrightarrow$ *queue-invar* (*children t*)
⟨*proof*⟩

lemma *rank-link*: *rank t* = *rank t'* $\Longrightarrow$ *rank* (*link t t'*) = *rank t* + *1*
⟨*proof*⟩

lemma *rank-invar-not-empty-hd*: ⟦*rank-invar* (*t* # *bq*); *bq* ≠ []⟧ $\Longrightarrow$
rank t < *rank* (*hd bq*)
⟨*proof*⟩

lemma *rank-invar-to-set*: *rank-invar* (*t* # *bq*) $\Longrightarrow$
∀ *t'* ∈ *set bq*. *rank t* < *rank t'*
⟨*proof*⟩

lemma *set-to-rank-invar*: ⟦∀ *t'* ∈ *set bq*. *rank t* < *rank t'*; *rank-invar bq*⟧
$\Longrightarrow$ *rank-invar* (*t* # *bq*)
⟨*proof*⟩

lemma *rank-invar-hd-cons*:
⟦*rank-invar bq*; *rank t* < *rank* (*hd bq*)⟧ $\Longrightarrow$ *rank-invar* (*t* # *bq*)
⟨*proof*⟩

lemma *rank-invar-cons*: *rank-invar* (*t* # *bq*) $\Longrightarrow$ *rank-invar bq*
⟨*proof*⟩

lemma *invar-cons-up*:
⟦*invar* (*t* # *bq*); *rank t'* < *rank t*; *tree-invar t'*⟧ $\Longrightarrow$ *invar* (*t'* # *t* # *bq*)
⟨*proof*⟩

lemma *invar-cons-down*: *invar* (*t* # *bq*) $\Longrightarrow$ *invar bq*
⟨*proof*⟩

lemma *invar-app-single*:
⟦*invar bq*; ∀ *t* ∈ *set bq*. *rank t* < *rank t'*; *tree-invar t'*⟧
$\Longrightarrow$ *invar* (*bq* @ [*t'*])
⟨*proof*⟩

1.1.3 Heap Ordering

fun *heap-ordered* :: (*'e*, *'a*::*linorder*) *BinomialTree* ⇒ *bool* **where**
heap-ordered (*Node e a r ts*) = (∀ *x* ∈ *set-mset*(*queue-to-multiset ts*). *a* ≤ *snd x*)

The invariant for trees implies heap order.

lemma *tree-invar-heap-ordered*:
assumes *tree-invar t*
shows *heap-ordered t*
⟨*proof*⟩

1.1.4 Height and Length

Although complexity of HOL-functions cannot be expressed within HOL, we can express the height and length of a binomial heap. By showing that both, height and length, are logarithmic in the number of contained elements, we give strong evidence that our functions have logarithmic complexity in the number of elements.

Height of a tree and queue

fun *height-tree* :: ('*e*, ('*a*::*linorder*)) *BinomialTree* ⇒ *nat* **and**
  height-queue :: ('*e*, ('*a*::*linorder*)) *BinomialQueue-inv* ⇒ *nat*
 where
 height-tree (*Node e a r ts*) = *height-queue ts* |
 height-queue [] = *0* |
 height-queue (*t* # *ts*) = *max* (*Suc* (*height-tree t*)) (*height-queue ts*)

lemma *link-length*: *size* (*tree-to-multiset* (*link t1 t2*)) =
 size (*tree-to-multiset t1*) + *size* (*tree-to-multiset t2*)
 ⟨*proof*⟩

lemma *tree-rank-estimate*:
 tree-invar (*Node e a r ts*) ⟹
  size (*tree-to-multiset* (*Node e a r ts*)) = (*2*::*nat*)^*r*
⟨*proof*⟩

lemma *tree-rank-height*:
 tree-invar (*Node e a r ts*) ⟹ *height-tree* (*Node e a r ts*) = *r*
⟨*proof*⟩

A binomial tree of height h contains exactly 2^h elements

theorem *tree-height-estimate*:
 tree-invar t ⟹ *size* (*tree-to-multiset t*) = (*2*::*nat*)^(*height-tree t*)
 ⟨*proof*⟩

lemma *size-mset-tree*: *tree-invar t* ⟹
 size (*tree-to-multiset t*) = (*2*::*nat*)^(*rank t*)
 ⟨*proof*⟩

lemma *invar-butlast*: *invar* (*bq* @ [*t*]) ⟹ *invar bq*
 ⟨*proof*⟩

lemma *invar-last-max*: *invar* (*bq* @ [*m*]) ⟹ ∀ *t* ∈ *set bq*. *rank t* < *rank m*
 ⟨*proof*⟩

lemma *invar-length*: *invar bq* ⟹ *length bq* ≤ *Suc* (*rank* (*last bq*))
⟨*proof*⟩

lemma *size-queue-sum-list*:
size (*queue-to-multiset bq*) = *sum-list* (*map* (*size* ∘ *tree-to-multiset*) *bq*)
⟨*proof*⟩

A binomial heap of length l contains at least $2^l - 1$ elements.

theorem *queue-length-estimate-lower*:
invar bq ⟹ (*size* (*queue-to-multiset bq*)) ≥ *2^(length bq) − 1*
⟨*proof*⟩

1.2 Operations

1.2.1 Empty

lemma *empty-correct*[*simp*]:
invar Nil
queue-to-multiset Nil = {#}
⟨*proof*⟩

The empty multiset is represented by exactly the empty queue

lemma *empty-iff*: *t=Nil* ⟷ *queue-to-multiset t* = {#}
⟨*proof*⟩

1.2.2 Insert

Inserts a binomial tree into a binomial queue, such that the queue does not contain two trees of same rank.

fun *ins* :: ('*e*, '*a*::*linorder*) *BinomialTree* ⇒ ('*e*, '*a*) *BinomialQueue-inv* ⇒
('*e*, '*a*) *BinomialQueue-inv* **where**
ins t [] = [*t*] |
ins t' (*t* # *bq*) = (*if* (*rank t'*) < (*rank t*)
then t' # *t* # *bq*
else (*if* (*rank t*) < (*rank t'*)
then t # (*ins t' bq*)
else ins (*link t' t*) *bq*))

Inserts an element with priority into the queue.

definition *insert* :: '*e* ⇒ '*a*::*linorder* ⇒ ('*e*, '*a*) *BinomialQueue-inv* ⇒
('*e*, '*a*) *BinomialQueue-inv* **where**
insert e a bq = *ins* (*Node e a 0* []) *bq*

lemma *ins-mset*:
⟦*tree-invar t*; *queue-invar q*⟧ ⟹ *queue-to-multiset* (*ins t q*)
= *tree-to-multiset t* + *queue-to-multiset q*
⟨*proof*⟩

lemma *insert-mset*: *queue-invar q* ⟹
queue-to-multiset (*insert e a q*) = *queue-to-multiset q* + {# (*e*,*a*) #}

⟨*proof*⟩

lemma *ins-queue-invar*: ⟦*tree-invar t*; *queue-invar q*⟧ ⟹ *queue-invar* (*ins t q*)
⟨*proof*⟩

lemma *insert-queue-invar*:
assumes *queue-invar q*
shows *queue-invar* (*insert e a q*)
⟨*proof*⟩

lemma *rank-ins*: (*rank-invar* (*t* # *bq*) ⟹
(*rank* (*hd* (*ins t′* (*t* # *bq*))) ≥ *rank t*) ∨
(*rank* (*hd* (*ins t′* (*t* # *bq*))) ≥ *rank t′*))
⟨*proof*⟩

lemma *rank-ins2*: *rank-invar bq* ⟹
rank t ≤ *rank* (*hd* (*ins t bq*)) ∨
(*rank* (*hd* (*ins t bq*)) = *rank* (*hd bq*) ∧ *bq* ≠ [])
⟨*proof*⟩

lemma *rank-invar-ins*: *rank-invar bq* ⟹ *rank-invar* (*ins t bq*)
⟨*proof*⟩

lemma *rank-invar-insert*: *rank-invar bq* ⟹ *rank-invar* (*insert e a bq*)
⟨*proof*⟩

lemma *insert-correct*:
assumes *I*: *invar q*
shows
invar (*insert e a q*)
queue-to-multiset (*insert e a q*) = *queue-to-multiset q* + {# (*e*,*a*) #}
⟨*proof*⟩

1.2.3 Meld

Melds two queues.

```
fun meld :: ('e, 'a::linorder) BinomialQueue-inv ⇒ ('e, 'a) BinomialQueue-inv
  ⇒ ('e, 'a) BinomialQueue-inv
  where
  meld [] bq = bq |
  meld bq [] = bq |
  meld (t1#bq1) (t2#bq2) =
   (if (rank t1) < (rank t2)
       then t1 # (meld bq1 (t2 # bq2))
       else (
         if (rank t2 < rank t1)
            then t2 # (meld (t1 # bq1) bq2)
            else ins (link t1 t2) (meld bq1 bq2)
       )
```

)

lemma *meld-queue-invar*:
⟦*queue-invar q*; *queue-invar q'*⟧ ⟹ *queue-invar* (*meld q q'*)
⟨*proof*⟩

lemma *rank-ins-min*: *rank-invar bq* ⟹
rank (*hd* (*ins t bq*)) ≥ *min* (*rank t*) (*rank* (*hd bq*))
⟨*proof*⟩

lemma *rank-invar-meld-strong*:
⟦*rank-invar bq1*; *rank-invar bq2*⟧ ⟹ *rank-invar* (*meld bq1 bq2*) ∧
rank (*hd* (*meld bq1 bq2*)) ≥ *min* (*rank* (*hd bq1*)) (*rank* (*hd bq2*))
⟨*proof*⟩

lemma *rank-invar-meld*:
⟦*rank-invar bq1*; *rank-invar bq2*⟧ ⟹ *rank-invar* (*meld bq1 bq2*)
⟨*proof*⟩

lemma *meld-mset*: ⟦*queue-invar q*; *queue-invar q'*⟧ ⟹
queue-to-multiset (*meld q q'*) =
queue-to-multiset q + *queue-to-multiset q'*
⟨*proof*⟩

lemma *meld-correct*:
assumes *invar q invar q'*
shows
invar (*meld q q'*)
queue-to-multiset (*meld q q'*) = *queue-to-multiset q* + *queue-to-multiset q'*
⟨*proof*⟩

1.2.4 Find Minimal Element

Finds the tree containing the minimal element.

fun *getMinTree* :: ('*e*, '*a*::*linorder*) *BinomialQueue-inv* ⇒
('*e*, '*a*) *BinomialTree* **where**
getMinTree [*t*] = *t* |
getMinTree (*t*#*bq*) = (*if prio t* ≤ *prio* (*getMinTree bq*)
then t else (*getMinTree bq*))

lemma *mintree-exists*: (*bq* ≠ []) = (*getMinTree bq* ∈ *set bq*)
⟨*proof*⟩

lemma *treehead-in-multiset*:
t ∈ *set bq* ⟹ (*val t*, *prio t*) ∈# *queue-to-multiset bq*
⟨*proof*⟩

lemma *heap-ordered-single*:
heap-ordered t = (∀ *x* ∈ *set-mset* (*tree-to-multiset t*). *prio t* ≤ *snd x*)

⟨*proof*⟩

lemma *getMinTree-cons*:
prio (*getMinTree* (*y* # *x* # *xs*)) ≤ *prio* (*getMinTree* (*x* # *xs*))
⟨*proof*⟩

lemma *getMinTree-min-tree*:
t ∈ *set bq* ⟹ *prio* (*getMinTree bq*) ≤ *prio t*
⟨*proof*⟩

lemma *getMinTree-min-prio*:
assumes *queue-invar bq*
and *y* ∈ *set-mset* (*queue-to-multiset bq*)
shows *prio* (*getMinTree bq*) ≤ *snd y*
⟨*proof*⟩

Finds the minimal Element in the queue.

definition *findMin* :: ('*e*, '*a*::*linorder*) *BinomialQueue-inv* ⇒ ('*e* × '*a*) **where**
findMin bq = (*let min* = *getMinTree bq in* (*val min*, *prio min*))

lemma *findMin-correct*:
assumes *I*: *invar q*
assumes *NE*: *q* ≠ *Nil*
shows
findMin q ∈# *queue-to-multiset q*
∀ *y*∈*set-mset* (*queue-to-multiset q*). *snd* (*findMin q*) ≤ *snd y*
⟨*proof*⟩

1.2.5 Delete Minimal Element

Removes the first tree, which has the priority *a* within his root.

fun *remove1Prio* :: '*a* ⇒ ('*e*, '*a*::*linorder*) *BinomialQueue-inv* ⇒
('*e*, '*a*) *BinomialQueue-inv* **where**
remove1Prio a [] = [] |
remove1Prio a (*t*#*bq*) =
(*if* (*prio t*) = *a then bq else t* # (*remove1Prio a bq*))

Returns the queue without the minimal element.

definition *deleteMin* :: ('*e*, '*a*::*linorder*) *BinomialQueue-inv* ⇒
('*e*, '*a*) *BinomialQueue-inv* **where**
deleteMin bq ≡ (*let min* = *getMinTree bq in*
meld (*rev* (*children min*))
(*remove1Prio* (*prio min*) *bq*))

lemma *queue-invar-rev*: *queue-invar q* ⟹ *queue-invar* (*rev q*)
⟨*proof*⟩

lemma *queue-invar-remove1*: *queue-invar q* ⟹ *queue-invar* (*remove1 t q*)

⟨*proof*⟩

lemma *qtm-in-set-subset*: $t \in set\ q \Longrightarrow$
tree-to-multiset $t \subseteq\#$ *queue-to-multiset* q
⟨*proof*⟩

lemma *remove1-mset*: $t \in set\ q \Longrightarrow$
queue-to-multiset (*remove1* $t\ q$) =
queue-to-multiset $q -$ *tree-to-multiset* t
⟨*proof*⟩

lemma *remove1Prio-remove1*[*simp*]:
remove1Prio (*prio* (*getMinTree* bq)) bq = *remove1* (*getMinTree* bq) bq
⟨*proof*⟩

lemma *deleteMin-queue-invar*:
assumes *INV*: *queue-invar* q
assumes *NE*: $q \neq Nil$
shows *queue-invar* (*deleteMin* q)
⟨*proof*⟩

lemma *children-rank-less*:
assumes *tree-invar* t
shows $\forall\, t' \in set$ (*children* t). *rank* $t' <$ *rank* t
⟨*proof*⟩

lemma *strong-rev-children*:
assumes *tree-invar* t
shows *invar* (*rev* (*children* t))
⟨*proof*⟩

lemma *first-less*: *rank-invar* (t # bq) $\Longrightarrow \forall\, t' \in set\ bq.$ *rank* $t <$ *rank* t'
⟨*proof*⟩

lemma *strong-remove1*: *invar* $bq \Longrightarrow$ *invar* (*remove1* $t\ bq$)
⟨*proof*⟩

theorem *deleteMin-invar*:
assumes *invar* bq
and $bq \neq []$
shows *invar* (*deleteMin* bq)
⟨*proof*⟩

lemma *children-mset*: *queue-to-multiset* (*children* t) =
tree-to-multiset $t - \{\#$ (*val* t, *prio* t) $\#\}$
⟨*proof*⟩

lemma *deleteMin-mset*:
assumes *queue-invar* q

and $q \neq Nil$
shows *queue-to-multiset* (*deleteMin* q) = *queue-to-multiset* q − {# (*findMin* q) #}
⟨*proof*⟩

lemma *deleteMin-correct*:
assumes *INV*: *invar* q
assumes *NE*: $q \neq Nil$
shows
invar (*deleteMin* q)
queue-to-multiset (*deleteMin* q) = *queue-to-multiset* q − {# (*findMin* q) #}
⟨*proof*⟩

end

interpretation *BinomialHeapStruc*: *BinomialHeapStruc-loc* ⟨*proof*⟩

1.3 Hiding the Invariant

1.3.1 Datatype

typedef (**overloaded**) ('*e*, '*a*) *BinomialHeap* =
{q :: ('*e*,'*a*::*linorder*) *BinomialHeapStruc.BinomialQueue-inv*. *BinomialHeapStruc.invar* q }
⟨*proof*⟩

lemma *Rep-BinomialHeap-invar*[*simp*]:
BinomialHeapStruc.invar (*Rep-BinomialHeap* x)
⟨*proof*⟩

lemma [*simp*]:
BinomialHeapStruc.invar $q \Longrightarrow$ *Rep-BinomialHeap* (*Abs-BinomialHeap* q) = q
⟨*proof*⟩

lemma [*simp*, *code abstype*]: *Abs-BinomialHeap* (*Rep-BinomialHeap* q) = q
⟨*proof*⟩

locale *BinomialHeap-loc*
begin

1.3.2 Operations

definition [*code*]:
to-mset t == *BinomialHeapStruc.queue-to-multiset* (*Rep-BinomialHeap* t)

definition *empty* **where** *empty* == *Abs-BinomialHeap Nil*
lemma [*code abstract*, *simp*]: *Rep-BinomialHeap empty* = []
⟨*proof*⟩

definition [*code*]: *isEmpty q == Rep-BinomialHeap q = Nil*
lemma *empty-rep*: *q=empty ⟷ Rep-BinomialHeap q = Nil*
⟨*proof*⟩

lemma *isEmpty-correct*: *isEmpty q ⟷ q=empty*
⟨*proof*⟩

definition
insert
:: *'e* ⇒ (*'a*::*linorder*) ⇒ (*'e*,*'a*) *BinomialHeap* ⇒ (*'e*,*'a*) *BinomialHeap*
where *insert e a q ==*
Abs-BinomialHeap (*BinomialHeapStruc.insert e a* (*Rep-BinomialHeap q*))
lemma [*code abstract*]:
Rep-BinomialHeap (*insert e a q*)
= *BinomialHeapStruc.insert e a* (*Rep-BinomialHeap q*)
⟨*proof*⟩

definition [*code*]: *findMin q == BinomialHeapStruc.findMin* (*Rep-BinomialHeap q*)

definition *deleteMin q ==*
if q=empty then empty
else Abs-BinomialHeap (*BinomialHeapStruc.deleteMin* (*Rep-BinomialHeap q*))

In this lemma, we do not use equality, but case-distinction for checking non-emptyness. That prevents the code generator from introducing an equality-class parameter for the entry type *'a*.

lemma [*code abstract*]: *Rep-BinomialHeap* (*deleteMin q*) =
(*case* (*Rep-BinomialHeap q*) *of* [] ⇒ [] |
- ⇒ *BinomialHeapStruc.deleteMin* (*Rep-BinomialHeap q*))
⟨*proof*⟩

definition *meld q1 q2 ==*
Abs-BinomialHeap (*BinomialHeapStruc.meld* (*Rep-BinomialHeap q1*)
(*Rep-BinomialHeap q2*))
lemma [*code abstract*]:
Rep-BinomialHeap (*meld q1 q2*)
= *BinomialHeapStruc.meld* (*Rep-BinomialHeap q1*) (*Rep-BinomialHeap q2*)
⟨*proof*⟩

1.3.3 Correctness

lemma *empty-correct*: *to-mset q* = {#} ⟷ *q=empty*
⟨*proof*⟩

lemma *to-mset-of-empty*[*simp*]: *to-mset empty* = {#}
⟨*proof*⟩

lemma *insert-correct*: *to-mset* (*insert e a q*) = *to-mset q* + {#(*e*,*a*)#}
⟨*proof*⟩

lemma *findMin-correct*:
assumes *q*≠*empty*
shows
findMin q ∈# *to-mset q*
∀ *y*∈*set-mset* (*to-mset q*). *snd* (*findMin q*) ≤ *snd y*
⟨*proof*⟩

lemma *deleteMin-correct*:
assumes *q*≠*empty*
shows *to-mset* (*deleteMin q*) = *to-mset q* − {# *findMin q* #}
⟨*proof*⟩

lemma *meld-correct*:
shows *to-mset* (*meld q q*′) = *to-mset q* + *to-mset q*′
⟨*proof*⟩

Correctness lemmas to be used with simplifier

lemmas *correct* = *empty-correct deleteMin-correct meld-correct*

end
interpretation *BinomialHeap*: *BinomialHeap-loc* ⟨*proof*⟩

1.4 Documentation

BinomialHeap.to-mset
Abstraction to multiset.

BinomialHeap.empty
The empty heap. ($O(1)$)
Spec *BinomialHeap.empty-correct*:

(*BinomialHeap.to-mset q* = {#}) = (*q* = *BinomialHeap.empty*)

BinomialHeap.isEmpty
Checks whether heap is empty. Mainly used to work around code-generation issues. ($O(1)$)
Spec *BinomialHeap.isEmpty-correct*:

BinomialHeap.isEmpty q = (*q* = *BinomialHeap.empty*)

BinomialHeap.insert
Inserts element ($O(\log(n))$)
Spec *BinomialHeap.insert-correct*:

BinomialHeap.to-mset (*BinomialHeap.insert e a q*) =
BinomialHeap.to-mset q + {#(*e*, *a*)#}

BinomialHeap.findMin
Returns a minimal element ($O(\log(n))$)
Spec *BinomialHeap.findMin-correct*:

q ≠ *BinomialHeap.empty* ⟹ *BinomialHeap.findMin q* ∈# *BinomialHeap.to-mset q*
q ≠ *BinomialHeap.empty* ⟹
∀ *y*∈#*BinomialHeap.to-mset q*. *snd* (*BinomialHeap.findMin q*) ≤ *snd y*

BinomialHeap.deleteMin
Deletes *the* element that is returned by *find_min*
Spec *BinomialHeap.deleteMin-correct*:

q ≠ *BinomialHeap.empty* ⟹
BinomialHeap.to-mset (*BinomialHeap.deleteMin q*) =
BinomialHeap.to-mset q − {#*BinomialHeap.findMin q*#}

BinomialHeap.meld

BinomialHeap.meld

Melds two heaps ($O(\log(n + m))$)
Spec *BinomialHeap.meld-correct*:

BinomialHeap.to-mset (*BinomialHeap.meld q q*′) =
BinomialHeap.to-mset q + *BinomialHeap.to-mset q*′

end

2 Skew Binomial Heaps

theory *SkewBinomialHeap*
imports *Main HOL−Library.Multiset*
begin

Skew Binomial Queues as specified by Brodal and Okasaki [1] are a data structure for priority queues with worst case O(1) *findMin*, *insert*, and *meld* operations, and worst-case logarithmic *deleteMin* operation. They are derived from priority queues in three steps:

1. Skew binomial trees are used to eliminate the possibility of cascading links during insert operations. This reduces the complexity of an insert operation to $O(1)$.

2. The current minimal element is cached. This approach, known as *global root*, reduces the cost of a *findMin*-operation to O(1).

3. By allowing skew binomial queues to contain skew binomial queues, the cost for meld-operations is reduced to $O(1)$. This approach is known as *data-structural bootstrapping*.

In this theory, we combine Steps 2 and 3, i.e. we first implement skew binomial queues, and then bootstrap them. The bootstrapping implicitly introduces a global root, such that we also get a constant time findMin operation.

locale *SkewBinomialHeapStruc-loc*
begin

2.1 Datatype

datatype ('*e*, '*a*) *SkewBinomialTree* =
Node (*val*: '*e*) (*prio*: '*a*::*linorder*) (*rank*: *nat*) (*children*: ('*e* , '*a*) *SkewBinomialTree list*)

type-synonym ('*e*, '*a*) *SkewBinomialQueue* = ('*e*, '*a*::*linorder*) *SkewBinomialTree list*

2.1.1 Abstraction to Multisets

Returns a multiset with all (element, priority) pairs from a queue

fun *tree-to-multiset*
:: ('*e*, '*a*::*linorder*) *SkewBinomialTree* ⇒ ('*e* × '*a*) *multiset*
and *queue-to-multiset*
:: ('*e*, '*a*::*linorder*) *SkewBinomialQueue* ⇒ ('*e* × '*a*) *multiset* **where**
tree-to-multiset (*Node e a r ts*) = {#(*e*,*a*)#} + *queue-to-multiset ts* |
queue-to-multiset [] = {#} |
queue-to-multiset (*t*#*q*) = *tree-to-multiset t* + *queue-to-multiset q*

lemma *ttm-children*: *tree-to-multiset t* =
{#(*val t*,*prio t*)#} + *queue-to-multiset* (*children t*)
⟨*proof*⟩

lemma *qtm-conc*[*simp*]: *queue-to-multiset* (*q*@*q*′)
= *queue-to-multiset q* + *queue-to-multiset q*′
⟨*proof*⟩

2.1.2 Invariant

Link two trees of rank r to a new tree of rank $r + 1$

fun *link* :: ('*e*, '*a*::*linorder*) *SkewBinomialTree* ⇒ ('*e*, '*a*) *SkewBinomialTree* ⇒
('*e*, '*a*) *SkewBinomialTree* **where**
link (*Node e1 a1 r1 ts1*) (*Node e2 a2 r2 ts2*) =

```
  (if  a1≤a2
    then (Node e1 a1 (Suc r1) ((Node e2 a2 r2 ts2)#ts1))
    else (Node e2 a2 (Suc r2) ((Node e1 a1 r1 ts1)#ts2)))
```

Link two trees of rank r and a new element to a new tree of rank $r + 1$

```
fun skewlink :: 'e ⇒ 'a::linorder ⇒ ('e, 'a) SkewBinomialTree ⇒
  ('e, 'a) SkewBinomialTree ⇒ ('e, 'a) SkewBinomialTree where
  skewlink e a t t' = (if a ≤ (prio t) ∧ a ≤ (prio t')
  then (Node e a (Suc (rank t)) [t,t'])
  else (if (prio t) ≤ (prio t')
   then
    Node (val t)  (prio t)  (Suc (rank t))  (Node e a 0 [] # t' # children t)
   else
    Node (val t') (prio t') (Suc (rank t')) (Node e a 0 [] # t # children t')))
```

The invariant for trees claims that a tree labeled rank 0 has no children, and a tree labeled rank $r + 1$ is the result of an ordinary link or a skew link of two trees with rank r.

```
function tree-invar :: ('e, 'a::linorder) SkewBinomialTree ⇒ bool where
  tree-invar (Node e a 0 ts) = (ts = []) |
  tree-invar (Node e a (Suc r) ts) = (∃ e1 a1 ts1 e2 a2 ts2 e' a'.
  tree-invar (Node e1 a1 r ts1) ∧ tree-invar (Node e2 a2 r ts2) ∧
  ((Node e a (Suc r) ts) = link (Node e1 a1 r ts1) (Node e2 a2 r ts2) ∨
   (Node e a (Suc r) ts) = skewlink e' a' (Node e1 a1 r ts1) (Node e2 a2 r ts2)))
⟨proof⟩
termination
  ⟨proof⟩
```

A heap satisfies the invariant, if all contained trees satisfy the invariant, the ranks of the trees in the heap are distinct, except that the first two trees may have same rank, and the ranks are ordered in ascending order.

First part: All trees inside the queue satisfy the invariant.

```
definition queue-invar :: ('e, 'a::linorder) SkewBinomialQueue ⇒ bool where
  queue-invar q ≡ (∀ t ∈ set q. tree-invar t)

lemma queue-invar-simps[simp]:
  queue-invar []
  queue-invar (t#q) ⟷ tree-invar t ∧ queue-invar q
  queue-invar (q@q') ⟷ queue-invar q ∧ queue-invar q'
  queue-invar q ⟹ t∈set q ⟹ tree-invar t
  ⟨proof⟩
```

Second part: The ranks of the trees in the heap are distinct, except that the first two trees may have same rank, and the ranks are ordered in ascending order.

For tail of queue

fun *rank-invar* :: (*'e*, *'a*::*linorder*) *SkewBinomialQueue* ⇒ *bool* **where**
rank-invar [] = *True* |
rank-invar [*t*] = *True* |
rank-invar (*t* # *t'* # *bq*) = (*rank t* < *rank t'* ∧ *rank-invar* (*t'* # *bq*))

For whole queue: First two elements may have same rank

fun *rank-skew-invar* :: (*'e*, *'a*::*linorder*) *SkewBinomialQueue* ⇒ *bool* **where**
rank-skew-invar [] = *True* |
rank-skew-invar [*t*] = *True* |
rank-skew-invar (*t* # *t'* # *bq*) = ((*rank t* ≤ *rank t'*) ∧ *rank-invar* (*t'* # *bq*))

definition *tail-invar* :: (*'e*, *'a*::*linorder*) *SkewBinomialQueue* ⇒ *bool* **where**
tail-invar bq = (*queue-invar bq* ∧ *rank-invar bq*)

definition *invar* :: (*'e*, *'a*::*linorder*) *SkewBinomialQueue* ⇒ *bool* **where**
invar bq = (*queue-invar bq* ∧ *rank-skew-invar bq*)

lemma *invar-empty*[*simp*]:
invar []
tail-invar []
⟨*proof*⟩

lemma *invar-tail-invar*:
invar (*t* # *bq*) ⟹ *tail-invar bq*
⟨*proof*⟩

lemma *link-mset*[*simp*]: *tree-to-multiset* (*link t1 t2*)
= *tree-to-multiset t1* +*tree-to-multiset t2*
⟨*proof*⟩

lemma *link-tree-invar*: ⟦*tree-invar t1*; *tree-invar t2*; *rank t1* = *rank t2*⟧ ⟹
tree-invar (*link t1 t2*)
⟨*proof*⟩

lemma *skewlink-mset*[*simp*]: *tree-to-multiset* (*skewlink e a t1 t2*)
= {# (*e*,*a*) #} + *tree-to-multiset t1* + *tree-to-multiset t2*
⟨*proof*⟩

lemma *skewlink-tree-invar*: ⟦*tree-invar t1*; *tree-invar t2*; *rank t1* = *rank t2*⟧ ⟹
tree-invar (*skewlink e a t1 t2*)
⟨*proof*⟩

lemma *rank-link*: *rank t* = *rank t'* ⟹ *rank* (*link t t'*) = *rank t* + *1*
⟨*proof*⟩

lemma *rank-skew-rank-invar*: *rank-skew-invar* (*t* # *bq*) ⟹ *rank-invar bq*
⟨*proof*⟩

lemma *rank-invar-rank-skew*:
assumes *rank-invar q*
shows *rank-skew-invar q*
⟨*proof*⟩

lemma *rank-invar-cons-up*:
⟦*rank-invar* (*t* # *bq*); *rank t′* < *rank t*⟧ ⟹ *rank-invar* (*t′* # *t* # *bq*)
⟨*proof*⟩

lemma *rank-skew-cons-up*:
⟦*rank-invar* (*t* # *bq*); *rank t′* ≤ *rank t*⟧ ⟹ *rank-skew-invar* (*t′* # *t* # *bq*)
⟨*proof*⟩

lemma *rank-invar-cons-down*: *rank-invar* (*t* # *bq*) ⟹ *rank-invar bq*
⟨*proof*⟩

lemma *rank-invar-hd-cons*:
⟦*rank-invar bq*; *rank t* < *rank* (*hd bq*)⟧ ⟹ *rank-invar* (*t* # *bq*)
⟨*proof*⟩

lemma *tail-invar-cons-up*:
⟦*tail-invar* (*t* # *bq*); *rank t′* < *rank t*; *tree-invar t′*⟧
⟹ *tail-invar* (*t′* # *t* # *bq*)
⟨*proof*⟩

lemma *tail-invar-cons-up-invar*:
⟦*tail-invar* (*t* # *bq*); *rank t′* ≤ *rank t*; *tree-invar t′*⟧ ⟹ *invar* (*t′* # *t* # *bq*)
⟨*proof*⟩

lemma *tail-invar-cons-down*:
tail-invar (*t* # *bq*) ⟹ *tail-invar bq*
⟨*proof*⟩

lemma *tail-invar-app-single*:
⟦*tail-invar bq*; ∀ *t* ∈ *set bq*. *rank t* < *rank t′*; *tree-invar t′*⟧
⟹ *tail-invar* (*bq* @ [*t′*])
⟨*proof*⟩

lemma *invar-app-single*:
⟦*invar bq*; ∀ *t* ∈ *set bq*. *rank t* < *rank t′*; *tree-invar t′*⟧
⟹ *invar* (*bq* @ [*t′*])
⟨*proof*⟩

lemma *invar-children*:
assumes *tree-invar* ((*Node e a r ts*)::(('*e*, '*a*::*linorder*) *SkewBinomialTree*))
shows *queue-invar ts* ⟨*proof*⟩

2.1.3 Heap Order

fun *heap-ordered* :: (*'e*, *'a*::*linorder*) *SkewBinomialTree* ⇒ *bool* **where**
  heap-ordered (*Node e a r ts*)
   = (∀ *x* ∈ *set-mset* (*queue-to-multiset ts*). *a* ≤ *snd x*)

The invariant for trees implies heap order.

lemma *tree-invar-heap-ordered*:
  fixes *t* :: (*'e*, *'a*::*linorder*) *SkewBinomialTree*
  assumes *tree-invar t*
  shows *heap-ordered t*
⟨*proof*⟩

2.1.4 Height and Length

Although complexity of HOL-functions cannot be expressed within HOL, we can express the height and length of a binomial heap. By showing that both, height and length, are logarithmic in the number of contained elements, we give strong evidence that our functions have logarithmic complexity in the number of elements.

Height of a tree and queue

fun *height-tree* :: (*'e*, (*'a*::*linorder*)) *SkewBinomialTree* ⇒ *nat* **and**
    height-queue :: (*'e*, (*'a*::*linorder*)) *SkewBinomialQueue* ⇒ *nat*
  where
  height-tree (*Node e a r ts*) = *height-queue ts* |
  height-queue [] = *0* |
  height-queue (*t* # *ts*) = *max* (*Suc* (*height-tree t*)) (*height-queue ts*)

lemma *link-length*: *size* (*tree-to-multiset* (*link t1 t2*)) =
  size (*tree-to-multiset t1*) + *size* (*tree-to-multiset t2*)
  ⟨*proof*⟩

lemma *tree-rank-estimate-upper*:
  tree-invar (*Node e a r ts*) ⟹
   size (*tree-to-multiset* (*Node e a r ts*)) ≤ (*2*::*nat*)^(*Suc r*) − *1*
⟨*proof*⟩

lemma *tree-rank-estimate-lower*:
  tree-invar (*Node e a r ts*) ⟹
   size (*tree-to-multiset* (*Node e a r ts*)) ≥ (*2*::*nat*)^*r*
⟨*proof*⟩

lemma *tree-rank-height*:
  tree-invar (*Node e a r ts*) ⟹ *height-tree* (*Node e a r ts*) = *r*
⟨*proof*⟩

A skew binomial tree of height h contains at most $2^{h+1} - 1$ elements

theorem *tree-height-estimate-upper*:
tree-invar t $\Longrightarrow$
size (*tree-to-multiset t*) $\leq$ (*2*::*nat*)^(*Suc* (*height-tree t*)) − *1*
⟨*proof*⟩

A skew binomial tree of height h contains at least 2^h elements

theorem *tree-height-estimate-lower*:
tree-invar t $\Longrightarrow$ *size* (*tree-to-multiset t*) $\geq$ (*2*::*nat*)^(*height-tree t*)
⟨*proof*⟩

lemma *size-mset-tree-upper*: *tree-invar t* $\Longrightarrow$
size (*tree-to-multiset t*) $\leq$ (*2*::*nat*)^(*Suc* (*rank t*)) − (*1*::*nat*)
⟨*proof*⟩

lemma *size-mset-tree-lower*: *tree-invar t* $\Longrightarrow$
size (*tree-to-multiset t*) $\geq$ (*2*::*nat*)^(*rank t*)
⟨*proof*⟩

lemma *invar-butlast*: *invar* (*bq* @ [*t*]) $\Longrightarrow$ *invar bq*
⟨*proof*⟩

lemma *invar-last-max*:
invar ((*b*#*b'*#*bq*) @ [*m*]) $\Longrightarrow$ ∀ *t* ∈ *set* (*b'*#*bq*). *rank t* < *rank m*
⟨*proof*⟩

lemma *invar-last-max'*: *invar* ((*b*#*b'*#*bq*) @ [*m*]) $\Longrightarrow$ *rank b* $\leq$ *rank b'*
⟨*proof*⟩

lemma *invar-length*: *invar bq* $\Longrightarrow$ *length bq* $\leq$ *Suc* (*Suc* (*rank* (*last bq*)))
⟨*proof*⟩

lemma *size-queue-sum-list*:
size (*queue-to-multiset bq*) = *sum-list* (*map* (*size* ∘ *tree-to-multiset*) *bq*)
⟨*proof*⟩

A skew binomial heap of length l contains at least $2^{l-1} - 1$ elements.

theorem *queue-length-estimate-lower*:
invar bq $\Longrightarrow$ (*size* (*queue-to-multiset bq*)) $\geq$ *2*^(*length bq* − *1*) − *1*
⟨*proof*⟩

2.2 Operations

2.2.1 Empty Tree

lemma *empty-correct*: *q*=*Nil* $\longleftrightarrow$ *queue-to-multiset q* = {#}
⟨*proof*⟩

2.2.2 Insert

Inserts a tree into the queue, such that two trees of same rank get linked and are recursively inserted. This is the same definition as for binomial queues and is used for melding.

fun *ins* :: ('*e*, '*a*::*linorder*) *SkewBinomialTree* ⇒ ('*e*, '*a*) *SkewBinomialQueue* ⇒
('*e*, '*a*) *SkewBinomialQueue* **where**
ins t [] = [*t*] |
ins t' (*t* # *bq*) =
(*if* (*rank t'*) < (*rank t*)
then t' # *t* # *bq*
else (*if* (*rank t*) < (*rank t'*)
then t # (*ins t' bq*)
else ins (*link t' t*) *bq*))

Insert an element with priority into a queue using skewlinks.

fun *insert* :: '*e* ⇒ '*a*::*linorder* ⇒ ('*e*, '*a*) *SkewBinomialQueue* ⇒
('*e*, '*a*) *SkewBinomialQueue* **where**
insert e a [] = [*Node e a 0* []] |
insert e a [*t*] = [*Node e a 0* [],*t*] |
insert e a (*t* # *t'* # *bq*) =
(*if rank t* ≠ *rank t'*
then (*Node e a 0* []) # *t* # *t'* # *bq*
else (*skewlink e a t t'*) # *bq*)

lemma *ins-mset*:
⟦*tree-invar t*; *queue-invar q*⟧ ⟹
queue-to-multiset (*ins t q*) = *tree-to-multiset t* + *queue-to-multiset q*
⟨*proof*⟩

lemma *insert-mset*: *queue-invar q* ⟹
queue-to-multiset (*insert e a q*) =
queue-to-multiset q + {# (*e*,*a*) #}
⟨*proof*⟩

lemma *ins-queue-invar*: ⟦*tree-invar t*; *queue-invar q*⟧ ⟹ *queue-invar* (*ins t q*)
⟨*proof*⟩

lemma *insert-queue-invar*: *queue-invar q* ⟹ *queue-invar* (*insert e a q*)
⟨*proof*⟩

lemma *rank-ins2*:
rank-invar bq ⟹
rank t ≤ *rank* (*hd* (*ins t bq*))
∨ (*rank* (*hd* (*ins t bq*)) = *rank* (*hd bq*) ∧ *bq* ≠ [])
⟨*proof*⟩

lemma *insert-rank-invar*: *rank-skew-invar q* ⟹ *rank-skew-invar* (*insert e a q*)
⟨*proof*⟩

lemma *insert-invar*: *invar q* $\Longrightarrow$ *invar* (*insert e a q*)
⟨*proof*⟩

theorem *insert-correct*:
assumes *I*: *invar q*
shows
invar (*insert e a q*)
queue-to-multiset (*insert e a q*) = *queue-to-multiset q* + {# (*e*,*a*) #}
⟨*proof*⟩

2.2.3 meld

Remove duplicate tree ranks by inserting the first tree of the queue into the rest of the queue.

fun *uniqify*
:: (′*e*, ′*a*::*linorder*) *SkewBinomialQueue* ⇒ (′*e*, ′*a*) *SkewBinomialQueue*
where
uniqify [] = [] |
uniqify (*t*#*bq*) = *ins t bq*

Meld two uniquified queues using the same definition as for binomial queues.

fun *meldUniq*
:: (′*e*, ′*a*::*linorder*) *SkewBinomialQueue* ⇒ (′*e*,′*a*) *SkewBinomialQueue* ⇒
(′*e*, ′*a*) *SkewBinomialQueue* **where**
meldUniq [] *bq* = *bq* |
meldUniq bq [] = *bq* |
meldUniq (*t1*#*bq1*) (*t2*#*bq2*) = (*if rank t1* < *rank t2*
then t1 # (*meldUniq bq1* (*t2*#*bq2*))
else (*if rank t2* < *rank t1*
then t2 # (*meldUniq* (*t1*#*bq1*) *bq2*)
else ins (*link t1 t2*) (*meldUniq bq1 bq2*)))

Meld two queues using above functions.

definition *meld*
:: (′*e*, ′*a*::*linorder*) *SkewBinomialQueue* ⇒ (′*e*, ′*a*) *SkewBinomialQueue* ⇒
(′*e*, ′*a*) *SkewBinomialQueue* **where**
meld bq1 bq2 = *meldUniq* (*uniqify bq1*) (*uniqify bq2*)

lemma *invar-uniqify*: *queue-invar q* $\Longrightarrow$ *queue-invar* (*uniqify q*)
⟨*proof*⟩

lemma *invar-meldUniq*: ⟦*queue-invar q*; *queue-invar q*′⟧ $\Longrightarrow$ *queue-invar* (*meldUniq q q*′)
⟨*proof*⟩

lemma *meld-queue-invar*:

assumes *queue-invar q*
and *queue-invar q′*
shows *queue-invar* (*meld q q′*)
⟨*proof*⟩

lemma *uniqify-mset*: *queue-invar q* ⟹ *queue-to-multiset q* = *queue-to-multiset* (*uniqify q*)
⟨*proof*⟩

lemma *meldUniq-mset*: ⟦*queue-invar q*; *queue-invar q′*⟧ ⟹
queue-to-multiset (*meldUniq q q′*) =
queue-to-multiset q + *queue-to-multiset q′*
⟨*proof*⟩

lemma *meld-mset*:
⟦ *queue-invar q*; *queue-invar q′* ⟧ ⟹
queue-to-multiset (*meld q q′*) = *queue-to-multiset q* + *queue-to-multiset q′*
⟨*proof*⟩

Ins operation satisfies rank invariant, see binomial queues

lemma *rank-ins*: *rank-invar bq* ⟹ *rank-invar* (*ins t bq*)
⟨*proof*⟩

lemma *rank-uniqify*:
assumes *rank-skew-invar q*
shows *rank-invar* (*uniqify q*)
⟨*proof*⟩

lemma *rank-ins-min*: *rank-invar bq* ⟹ *rank* (*hd* (*ins t bq*)) ≥ *min* (*rank t*) (*rank* (*hd bq*))
⟨*proof*⟩

lemma *rank-invar-not-empty-hd*: ⟦*rank-invar* (*t* # *bq*); *bq* ≠ []⟧ ⟹ *rank t* < *rank* (*hd bq*)
⟨*proof*⟩

lemma *rank-invar-meldUniq-strong*:
⟦*rank-invar bq1*; *rank-invar bq2*⟧ ⟹
rank-invar (*meldUniq bq1 bq2*)
∧ *rank* (*hd* (*meldUniq bq1 bq2*)) ≥ *min* (*rank* (*hd bq1*)) (*rank* (*hd bq2*))
⟨*proof*⟩

lemma *rank-meldUniq*:
⟦*rank-invar bq1*; *rank-invar bq2*⟧ ⟹ *rank-invar* (*meldUniq bq1 bq2*)
⟨*proof*⟩

lemma *rank-meld*:
⟦*rank-skew-invar q1*; *rank-skew-invar q2*⟧ ⟹ *rank-skew-invar* (*meld q1 q2*)

⟨*proof*⟩

theorem *meld-invar*:
⟦*invar bq1*; *invar bq2*⟧
⟹ *invar* (*meld bq1 bq2*)
⟨*proof*⟩

theorem *meld-correct*:
assumes *I*: *invar q invar q′*
shows
invar (*meld q q′*)
queue-to-multiset (*meld q q′*) = *queue-to-multiset q* + *queue-to-multiset q′*
⟨*proof*⟩

2.2.4 Find Minimal Element

Find the tree containing the minimal element.

fun *getMinTree* :: ('*e*, '*a*::*linorder*) *SkewBinomialQueue* ⇒
('*e*, '*a*) *SkewBinomialTree* **where**
getMinTree [*t*] = *t* |
getMinTree (*t*#*bq*) =
(*if prio t* ≤ *prio* (*getMinTree bq*)
then t
else (*getMinTree bq*))

Find the minimal Element in the queue.

definition *findMin* :: ('*e*, '*a*::*linorder*) *SkewBinomialQueue* ⇒ ('*e* × '*a*) **where**
findMin bq = (*let min* = *getMinTree bq in* (*val min*, *prio min*))

lemma *mintree-exists*: (*bq* ≠ []) = (*getMinTree bq* ∈ *set bq*)
⟨*proof*⟩

lemma *treehead-in-multiset*:
t ∈ *set bq* ⟹ (*val t*, *prio t*) ∈# (*queue-to-multiset bq*)
⟨*proof*⟩

lemma *heap-ordered-single*:
heap-ordered t = (∀ *x* ∈ *set-mset* (*tree-to-multiset t*). *prio t* ≤ *snd x*)
⟨*proof*⟩

lemma *getMinTree-cons*:
prio (*getMinTree* (*y* # *x* # *xs*)) ≤ *prio* (*getMinTree* (*x* # *xs*))
⟨*proof*⟩

lemma *getMinTree-min-tree*: *t* ∈ *set bq* ⟹ *prio* (*getMinTree bq*) ≤ *prio t*
⟨*proof*⟩

lemma *getMinTree-min-prio*:

assumes *queue-invar bq*
and $y \in$ *set-mset* (*queue-to-multiset bq*)
shows *prio* (*getMinTree bq*) $\leq$ *snd y*
⟨*proof*⟩

lemma *findMin-mset*:
assumes *I*: *queue-invar q*
assumes *NE*: $q \neq$ *Nil*
shows *findMin q* $\in\#$ *queue-to-multiset q*
$\forall\, y \in$ *set-mset* (*queue-to-multiset q*). *snd* (*findMin q*) $\leq$ *snd y*
⟨*proof*⟩

theorem *findMin-correct*:
assumes *I*: *invar q*
assumes *NE*: $q \neq$ *Nil*
shows *findMin q* $\in\#$ *queue-to-multiset q*
$\forall\, y \in$ *set-mset* (*queue-to-multiset q*). *snd* (*findMin q*) $\leq$ *snd y*
⟨*proof*⟩

2.2.5 Delete Minimal Element

Insert the roots of a given queue into an other queue.

fun *insertList* ::
('*e*, '*a*::*linorder*) *SkewBinomialQueue* ⇒ ('*e*, '*a*) *SkewBinomialQueue* ⇒
('*e*, '*a*) *SkewBinomialQueue* **where**
insertList [] *tbq* = *tbq* |
insertList (*t*#*bq*) *tbq* = *insertList bq* (*insert* (*val t*) (*prio t*) *tbq*)

Remove the first tree, which has the priority *a* within his root.

fun *remove1Prio* :: '*a* ⇒ ('*e*, '*a*::*linorder*) *SkewBinomialQueue* ⇒
('*e*, '*a*) *SkewBinomialQueue* **where**
remove1Prio a [] = [] |
remove1Prio a (*t*#*bq*) =
(*if* (*prio t*) = *a then bq else t* # (*remove1Prio a bq*))

lemma *remove1Prio-remove1*[*simp*]:
remove1Prio (*prio* (*getMinTree bq*)) *bq* = *remove1* (*getMinTree bq*) *bq*
⟨*proof*⟩

Return the queue without the minimal element found by findMin

definition *deleteMin* :: ('*e*, '*a*::*linorder*) *SkewBinomialQueue* ⇒
('*e*, '*a*) *SkewBinomialQueue* **where**
deleteMin bq = (*let min* = *getMinTree bq in insertList*
(*filter* (λ *t*. *rank t* = *0*) (*children min*))
(*meld* (*rev* (*filter* (λ *t*. *rank t* > *0*) (*children min*)))
(*remove1Prio* (*prio min*) *bq*)))

lemma *invar-rev*[*simp*]: *queue-invar* (*rev q*) ⟷ *queue-invar q*

⟨*proof*⟩

lemma *invar-remove1*: *queue-invar q* ⟹ *queue-invar* (*remove1 t q*)
⟨*proof*⟩

lemma *mset-rev*: *queue-to-multiset* (*rev q*) = *queue-to-multiset q*
⟨*proof*⟩

lemma *in-set-subset*: *t* ∈ *set q* ⟹ *tree-to-multiset t* ⊆# *queue-to-multiset q*
⟨*proof*⟩

lemma *mset-remove1*: *t* ∈ *set q* ⟹
queue-to-multiset (*remove1 t q*) =
queue-to-multiset q − *tree-to-multiset t*
⟨*proof*⟩

lemma *invar-children′*:
assumes *tree-invar t*
shows *queue-invar* (*children t*)
⟨*proof*⟩

lemma *invar-filter*: *queue-invar q* ⟹ *queue-invar* (*filter f q*)
⟨*proof*⟩

lemma *insertList-queue-invar*: *queue-invar q* ⟹ *queue-invar* (*insertList ts q*)
⟨*proof*⟩

lemma *deleteMin-queue-invar*:
⟦*queue-invar q*; *queue-to-multiset q* ≠ {#}⟧ ⟹
queue-invar (*deleteMin q*)
⟨*proof*⟩

lemma *mset-children*: *queue-to-multiset* (*children t*) =
tree-to-multiset t − {# (*val t*, *prio t*) #}
⟨*proof*⟩

lemma *mset-insertList*:
⟦∀ *t* ∈ *set ts*. *rank t* = *0* ∧ *children t* = [] ; *queue-invar q*⟧ ⟹
queue-to-multiset (*insertList ts q*) =
queue-to-multiset ts + *queue-to-multiset q*
⟨*proof*⟩

lemma *mset-filter*: (*queue-to-multiset* [*t*←*q* . *rank t* = *0*]) +
queue-to-multiset [*t*←*q* . *0* < *rank t*] =
queue-to-multiset q
⟨*proof*⟩

lemma *deleteMin-mset*:
assumes *queue-invar q*

and *queue-to-multiset q* $\neq$ {#}
shows *queue-to-multiset* (*deleteMin q*) = *queue-to-multiset q* − {# (*findMin q*) #}
⟨*proof*⟩

lemma *rank-insertList*: *rank-skew-invar q* $\Longrightarrow$ *rank-skew-invar* (*insertList ts q*)
⟨*proof*⟩

lemma *insertList-invar*: *invar q* $\Longrightarrow$ *invar* (*insertList ts q*)
⟨*proof*⟩

lemma *children-rank-less*:
assumes *tree-invar t*
shows $\forall\, t' \in$ *set* (*children t*). *rank t'* < *rank t*
⟨*proof*⟩

lemma *strong-rev-children*:
assumes *tree-invar t*
shows *invar* (*rev* [*t* ← *children t*. *0* < *rank t*])
⟨*proof*⟩

lemma *first-less*: *rank-invar* (*t* # *bq*) $\Longrightarrow$ $\forall\, t' \in$ *set bq*. *rank t* < *rank t'*
⟨*proof*⟩

lemma *first-less-eq*:
rank-skew-invar (*t* # *bq*) $\Longrightarrow$ $\forall\, t' \in$ *set bq*. *rank t* ≤ *rank t'*
⟨*proof*⟩

lemma *remove1-tail-invar*: *tail-invar bq* $\Longrightarrow$ *tail-invar* (*remove1 t bq*)
⟨*proof*⟩

lemma *invar-cons-down*: *invar* (*t* # *bq*) $\Longrightarrow$ *invar bq*
⟨*proof*⟩

lemma *remove1-invar*: *invar bq* $\Longrightarrow$ *invar* (*remove1 t bq*)
⟨*proof*⟩

lemma *deleteMin-invar*:
assumes *invar bq*
and *bq* $\neq$ []
shows *invar* (*deleteMin bq*)
⟨*proof*⟩

theorem *deleteMin-correct*:
assumes *I*: *invar q*
and *NE*: *q* $\neq$ *Nil*
shows *invar* (*deleteMin q*)
and *queue-to-multiset* (*deleteMin q*) = *queue-to-multiset q* − {#*findMin q*#}
⟨*proof*⟩

lemmas [*simp del*] = *insert.simps*

end

interpretation *SkewBinomialHeapStruc*: *SkewBinomialHeapStruc-loc* ⟨*proof*⟩

2.3 Bootstrapping

In this section, we implement datastructural bootstrapping, to reduce the complexity of meld-operations to $O(1)$. The bootstrapping also contains a *global root*, caching the minimal element of the queue, and thus also reducing the complexity of findMin-operations to $O(1)$.

Bootstrapping adds one more level of recursion: An *element* is an entry and a priority queues of elements.

In the original paper on skew binomial queues [1], higher order functors and recursive structures are used to elegantly implement bootstrapped heaps on top of ordinary heaps. However, such concepts are not supported in Isabelle/HOL, nor in Standard ML. Hence we have to use the „much less clean" [1] alternative: We manually specialize the heap datastructure, and re-implement the functions on the specialized data structure.

The correctness proofs are done by defining a mapping from teh specialized to the original data structure, and reusing the correctness statements of the original data structure.

2.3.1 Auxiliary

We first have to state some auxiliary lemmas and functions, mainly about multisets.

Finding the preimage of an element

lemma *in-image-msetE*:
assumes $x \in \# \textit{image-mset}\ f\ M$
obtains y **where** $y \in \# M\ x = f\ y$
⟨*proof*⟩

Very special lemma for images multisets of pairs, where the second component is a function of the first component

lemma *mset-image-fst-dep-pair-diff-split*:
$(\forall\ e\ a.\ (e,a) \in \# M \longrightarrow a = f\ e) \Longrightarrow$
$\textit{image-mset fst}\ (M - \{\#(e, f\ e)\#\}) = \textit{image-mset fst}\ M - \{\#e\#\}$
⟨*proof*⟩

locale *Bootstrapped*
begin

2.3.2 Datatype

We manually specialize the binomial tree to contain elements, that, in, turn, may contain trees. Note that we specify nodes without explicit priority, as the priority is contained in the elements stored in the nodes.

datatype ('*e*, '*a*) *BsSkewBinomialTree* =
  BsNode (*val*: ('*e*, '*a*::*linorder*) *BsSkewElem*)
      (*rank*: *nat*) (*children*: ('*e* , '*a*) *BsSkewBinomialTree list*)
and
('*e*,'*a*) *BsSkewElem* =
  Element '*e* (*eprio*: '*a*) ('*e*,'*a*) *BsSkewBinomialTree list*

type-synonym ('*e*,'*a*) *BsSkewHeap* = *unit* + ('*e*,'*a*) *BsSkewElem*
type-synonym ('*e*,'*a*) *BsSkewBinomialQueue* = ('*e*,'*a*) *BsSkewBinomialTree list*

2.3.3 Specialization Boilerplate

In this section, we re-define the functions on the specialized priority queues, and show there correctness. This is done by defining a mapping to original priority queues, and re-using the correctness lemmas proven there.

Mapping to original binomial trees and queues

fun *bsmapt* **where**
  bsmapt (*BsNode e r q*) = *SkewBinomialHeapStruc.Node e* (*eprio e*) *r* (*map bsmapt q*)

abbreviation *bsmap* **where**
  bsmap q == *map bsmapt q*

Invariant and mapping to multiset are defined via the mapping

abbreviation *invar q* == *SkewBinomialHeapStruc.invar* (*bsmap q*)
abbreviation *queue-to-multiset q*
  == *image-mset fst* (*SkewBinomialHeapStruc.queue-to-multiset* (*bsmap q*))
abbreviation *tree-to-multiset t*
  == *image-mset fst* (*SkewBinomialHeapStruc.tree-to-multiset* (*bsmapt t*))

abbreviation *queue-to-multiset-aux q*
  == (*SkewBinomialHeapStruc.queue-to-multiset* (*bsmap q*))

Now starts the re-implementation of the functions

primrec *prio* :: (*'e*, *'a*::*linorder*) *BsSkewBinomialTree* ⇒ *'a* **where**
prio (*BsNode e r ts*) = *eprio e*

lemma *proj-xlate*:
val t = *SkewBinomialHeapStruc.val* (*bsmapt t*)
prio t = *SkewBinomialHeapStruc.prio* (*bsmapt t*)
rank t = *SkewBinomialHeapStruc.rank* (*bsmapt t*)
bsmap (*children t*) = *SkewBinomialHeapStruc.children* (*bsmapt t*)
eprio (*SkewBinomialHeapStruc.val* (*bsmapt t*))
= *SkewBinomialHeapStruc.prio* (*bsmapt t*)
⟨*proof*⟩

fun *link* :: (*'e*, *'a*::*linorder*) *BsSkewBinomialTree*
⇒ (*'e*, *'a*) *BsSkewBinomialTree* ⇒
(*'e*, *'a*) *BsSkewBinomialTree* **where**
link (*BsNode e1 r1 ts1*) (*BsNode e2 r2 ts2*) =
(*if* *eprio e1*≤*eprio e2*
then (*BsNode e1* (*Suc r1*) ((*BsNode e2 r2 ts2*)#*ts1*))
else (*BsNode e2* (*Suc r2*) ((*BsNode e1 r1 ts1*)#*ts2*)))

Link two trees of rank r and a new element to a new tree of rank $r + 1$

fun *skewlink* :: (*'e*,*'a*::*linorder*) *BsSkewElem* ⇒ (*'e*, *'a*) *BsSkewBinomialTree* ⇒
(*'e*, *'a*) *BsSkewBinomialTree* ⇒ (*'e*, *'a*) *BsSkewBinomialTree* **where**
skewlink e t t' = (*if eprio e* ≤ (*prio t*) ∧ *eprio e* ≤ (*prio t'*)
then (*BsNode e* (*Suc* (*rank t*)) [*t*,*t'*])
else (*if* (*prio t*) ≤ (*prio t'*)
then
BsNode (*val t*) (*Suc* (*rank t*)) (*BsNode e 0* [] # *t'* # *children t*)
else
BsNode (*val t'*) (*Suc* (*rank t'*)) (*BsNode e 0* [] # *t* # *children t'*)))

lemma *link-xlate*:
bsmapt (*link t t'*) = *SkewBinomialHeapStruc.link* (*bsmapt t*) (*bsmapt t'*)
bsmapt (*skewlink e t t'*) =
SkewBinomialHeapStruc.skewlink e (*eprio e*) (*bsmapt t*) (*bsmapt t'*)
⟨*proof*⟩

fun *ins* :: (*'e*, *'a*::*linorder*) *BsSkewBinomialTree* ⇒
(*'e*, *'a*) *BsSkewBinomialQueue* ⇒
(*'e*, *'a*) *BsSkewBinomialQueue* **where**
ins t [] = [*t*] |
ins t' (*t* # *bq*) =
(*if* (*rank t'*) < (*rank t*)
then t' # *t* # *bq*
else (*if* (*rank t*) < (*rank t'*)
then t # (*ins t'* *bq*)
else ins (*link t' t*) *bq*))

lemma *ins-xlate*:
bsmap (*ins t q*) = *SkewBinomialHeapStruc.ins* (*bsmapt t*) (*bsmap q*)
⟨*proof*⟩

Insert an element with priority into a queue using skewlinks.

fun *insert* :: ('*e*,'*a*::*linorder*) *BsSkewElem* ⇒
('*e*, '*a*) *BsSkewBinomialQueue* ⇒
('*e*, '*a*) *BsSkewBinomialQueue* **where**
insert e [] = [*BsNode e 0* []] |
insert e [*t*] = [*BsNode e 0* [],*t*] |
insert e (*t* # *t'* # *bq*) =
(*if rank t* ≠ *rank t'*
then (*BsNode e 0* []) # *t* # *t'* # *bq*
else (*skewlink e t t'*) # *bq*)

lemma *insert-xlate*:
bsmap (*insert e q*) = *SkewBinomialHeapStruc.insert e* (*eprio e*) (*bsmap q*)
⟨*proof*⟩

lemma *insert-correct*:
assumes *I*: *invar q*
shows
invar (*insert e q*)
queue-to-multiset (*insert e q*) = *queue-to-multiset q* + {#(*e*)#}
⟨*proof*⟩

fun *uniqify*
:: ('*e*, '*a*::*linorder*) *BsSkewBinomialQueue* ⇒ ('*e*, '*a*) *BsSkewBinomialQueue*
where
uniqify [] = [] |
uniqify (*t*#*bq*) = *ins t bq*

fun *meldUniq*
:: ('*e*, '*a*::*linorder*) *BsSkewBinomialQueue* ⇒ ('*e*,'*a*) *BsSkewBinomialQueue* ⇒
('*e*, '*a*) *BsSkewBinomialQueue* **where**
meldUniq [] *bq* = *bq* |
meldUniq bq [] = *bq* |
meldUniq (*t1*#*bq1*) (*t2*#*bq2*) = (*if rank t1* < *rank t2*
then t1 # (*meldUniq bq1* (*t2*#*bq2*))
else (*if rank t2* < *rank t1*
then t2 # (*meldUniq* (*t1*#*bq1*) *bq2*)
else ins (*link t1 t2*) (*meldUniq bq1 bq2*)))

definition *meld*
:: ('*e*, '*a*::*linorder*) *BsSkewBinomialQueue* ⇒ ('*e*, '*a*) *BsSkewBinomialQueue* ⇒
('*e*, '*a*) *BsSkewBinomialQueue* **where**
meld bq1 bq2 = *meldUniq* (*uniqify bq1*) (*uniqify bq2*)

lemma *uniqify-xlate*:

bsmap (*uniqify q*) = *SkewBinomialHeapStruc.uniqify* (*bsmap q*)
⟨*proof*⟩

lemma *meldUniq-xlate*:
bsmap (*meldUniq q q'*) = *SkewBinomialHeapStruc.meldUniq* (*bsmap q*) (*bsmap q'*)
⟨*proof*⟩

lemma *meld-xlate*:
bsmap (*meld q q'*) = *SkewBinomialHeapStruc.meld* (*bsmap q*) (*bsmap q'*)
⟨*proof*⟩

lemma *meld-correct*:
assumes *I*: *invar q invar q'*
shows
invar (*meld q q'*)
queue-to-multiset (*meld q q'*) = *queue-to-multiset q* + *queue-to-multiset q'*
⟨*proof*⟩

fun *insertList* ::
('*e*, '*a*::*linorder*) *BsSkewBinomialQueue* ⇒ ('*e*, '*a*) *BsSkewBinomialQueue* ⇒
('*e*, '*a*) *BsSkewBinomialQueue* **where**
insertList [] *tbq* = *tbq* |
insertList (*t*#*bq*) *tbq* = *insertList bq* (*insert* (*val t*) *tbq*)

fun *remove1Prio* :: '*a* ⇒ ('*e*, '*a*::*linorder*) *BsSkewBinomialQueue* ⇒
('*e*, '*a*) *BsSkewBinomialQueue* **where**
remove1Prio a [] = [] |
remove1Prio a (*t*#*bq*) =
(*if* (*prio t*) = *a then bq else t* # (*remove1Prio a bq*))

fun *getMinTree* :: ('*e*, '*a*::*linorder*) *BsSkewBinomialQueue* ⇒
('*e*, '*a*) *BsSkewBinomialTree* **where**
getMinTree [*t*] = *t* |
getMinTree (*t*#*bq*) =
(*if prio t* ≤ *prio* (*getMinTree bq*)
then t
else (*getMinTree bq*))

definition *findMin*
:: ('*e*, '*a*::*linorder*) *BsSkewBinomialQueue* ⇒ ('*e*,'*a*) *BsSkewElem* **where**
findMin bq = *val* (*getMinTree bq*)

definition *deleteMin* :: ('*e*, '*a*::*linorder*) *BsSkewBinomialQueue* ⇒
('*e*, '*a*) *BsSkewBinomialQueue* **where**
deleteMin bq = (*let min* = *getMinTree bq in insertList*
(*filter* (λ *t*. *rank t* = *0*) (*children min*))
(*meld* (*rev* (*filter* (λ *t*. *rank t* > *0*) (*children min*)))
(*remove1Prio* (*prio min*) *bq*)))

lemma *insertList-xlate*:
bsmap (*insertList q q'*)
= *SkewBinomialHeapStruc.insertList* (*bsmap q*) (*bsmap q'*)
⟨*proof*⟩

lemma *remove1Prio-xlate*:
bsmap (*remove1Prio a q*) = *SkewBinomialHeapStruc.remove1Prio a* (*bsmap q*)
⟨*proof*⟩

lemma *getMinTree-xlate*:
q≠[] ⟹ *bsmapt* (*getMinTree q*) = *SkewBinomialHeapStruc.getMinTree* (*bsmap q*)
⟨*proof*⟩

lemma *findMin-xlate*:
q≠[] ⟹ *findMin q* = *fst* (*SkewBinomialHeapStruc.findMin* (*bsmap q*))
⟨*proof*⟩

lemma *findMin-xlate-aux*:
q≠[] ⟹ (*findMin q*, *eprio* (*findMin q*)) =
(*SkewBinomialHeapStruc.findMin* (*bsmap q*))
⟨*proof*⟩

lemma *bsmap-filter-xlate*:
bsmap [*x*←*l* . *P* (*bsmapt x*)] = [*x* ← *bsmap l*. *P x*]
⟨*proof*⟩

lemma *bsmap-rev-xlate*:
bsmap (*rev q*) = *rev* (*bsmap q*)
⟨*proof*⟩

lemma *deleteMin-xlate*:
q≠[] ⟹ *bsmap* (*deleteMin q*) = *SkewBinomialHeapStruc.deleteMin* (*bsmap q*)
⟨*proof*⟩

lemma *deleteMin-correct-aux*:
assumes *I*: *invar q*
assumes *NE*: *q*≠[]
shows
invar (*deleteMin q*)
queue-to-multiset-aux (*deleteMin q*) = *queue-to-multiset-aux q* −
{# (*findMin q*, *eprio* (*findMin q*)) #}
⟨*proof*⟩

lemma *bsmap-fs-dep*:

(e,a)∈#SkewBinomialHeapStruc.tree-to-multiset (bsmapt t) ⟹ a=eprio e
(e,a)∈#SkewBinomialHeapStruc.queue-to-multiset (bsmap q) ⟹ a=eprio e
thm SkewBinomialHeapStruc.tree-to-multiset-queue-to-multiset.induct
⟨proof⟩

lemma bsmap-fs-depD:
(e,a)∈#SkewBinomialHeapStruc.tree-to-multiset (bsmapt t)
⟹ e ∈# tree-to-multiset t ∧ a=eprio e
(e,a)∈#SkewBinomialHeapStruc.queue-to-multiset (bsmap q)
⟹ e ∈# queue-to-multiset q ∧ a=eprio e
⟨proof⟩

lemma findMin-correct-aux:
assumes I: invar q
assumes NE: q≠[]
shows (findMin q, eprio (findMin q)) ∈# queue-to-multiset-aux q
∀ y∈set-mset (queue-to-multiset-aux q). snd (findMin q,eprio (findMin q)) ≤ snd y
⟨proof⟩

lemma findMin-correct:
assumes I: invar q
and NE: q≠[]
shows findMin q ∈# queue-to-multiset q
and ∀ y∈set-mset (queue-to-multiset q). eprio (findMin q) ≤ eprio y
⟨proof⟩

lemma deleteMin-correct:
assumes I: invar q
assumes NE: q≠[]
shows
invar (deleteMin q)
queue-to-multiset (deleteMin q) = queue-to-multiset q −
{# findMin q #}
⟨proof⟩

declare insert.simps[simp del]

2.3.4 Bootstrapping: Phase 1

In this section, we define the ticked versions of the functions, as defined in [1]. These functions work on elements, i.e. only on heaps that contain at least one entry. Additionally, we define an invariant for elements, and a mapping to multisets of entries, and prove correct the ticked functions.

primrec findMin′ **where** findMin′ (Element e a q) = (e,a)
fun meld′:: ('e,'a::linorder) BsSkewElem ⇒

```
  ('e,'a) BsSkewElem ⇒ ('e,'a) BsSkewElem
  where meld' (Element e1 a1 q1) (Element e2 a2 q2) =
  (if a1≤a2 then
    Element e1 a1 (insert (Element e2 a2 q2) q1)
   else
    Element e2 a2 (insert (Element e1 a1 q1) q2)
  )
fun insert' where
  insert' e a q = meld' (Element e a []) q
fun deleteMin' where
  deleteMin' (Element e a q) = (
    case (findMin q) of
      Element ey ay q1 ⇒
        Element ey ay (meld q1 (deleteMin q))
  )
```

Size-function for termination proofs

```
fun tree-level and queue-level where
  tree-level (BsNode (Element - - qd) - q) =
  max (Suc (queue-level qd)) (queue-level q) |
  queue-level [] = (0::nat) |
  queue-level (t#q) = max (tree-level t) (queue-level q)

fun level where
  level (Element - - q) = Suc (queue-level q)

lemma level-m:
  x∈#tree-to-multiset t ⟹ level x < Suc (tree-level t)
  x∈#queue-to-multiset q ⟹ level x < Suc (queue-level q)
  ⟨proof⟩

lemma level-measure:
  x ∈ set-mset (queue-to-multiset q) ⟹ (x,(Element e a q))∈measure level
  x ∈# (queue-to-multiset q) ⟹ (x,(Element e a q))∈measure level
  ⟨proof⟩
```

Invariant for elements

```
function elem-invar where
  elem-invar (Element e a q) ⟷
  (∀ x. x∈# (queue-to-multiset q) ⟶ a ≤ eprio x ∧ elem-invar x) ∧
  invar q
  ⟨proof⟩
termination
⟨proof⟩
```

Abstraction to multisets

```
function elem-to-mset where
  elem-to-mset (Element e a q) = {# (e,a) #}
  + sum-mset (image-mset elem-to-mset (queue-to-multiset q))
```

⟨*proof*⟩
termination
⟨*proof*⟩

lemma *insert-correct′*:
assumes *I*: *elem-invar x*
shows
elem-invar (*insert′ e a x*)
elem-to-mset (*insert′ e a x*) = *elem-to-mset x* + {#(*e*,*a*)#}
⟨*proof*⟩

lemma *meld-correct′*:
assumes *I*: *elem-invar x elem-invar x′*
shows
elem-invar (*meld′ x x′*)
elem-to-mset (*meld′ x x′*) = *elem-to-mset x* + *elem-to-mset x′*
⟨*proof*⟩

lemma *findMin′-min*:
⟦*elem-invar x*; *y*∈#*elem-to-mset x*⟧ ⟹ *snd* (*findMin′ x*) ≤ *snd y*
⟨*proof*⟩

lemma *findMin-correct′*:
assumes *I*: *elem-invar x*
shows
findMin′ x ∈# *elem-to-mset x*
∀ *y*∈*set-mset* (*elem-to-mset x*). *snd* (*findMin′ x*) ≤ *snd y*
⟨*proof*⟩

lemma *deleteMin-correct′*:
assumes *I*: *elem-invar* (*Element e a q*)
assumes *NE*[*simp*]: *q*≠[]
shows
elem-invar (*deleteMin′* (*Element e a q*))
elem-to-mset (*deleteMin′* (*Element e a q*)) =
elem-to-mset (*Element e a q*) − {# *findMin′* (*Element e a q*) #}

⟨*proof*⟩

2.3.5 Bootstrapping: Phase 2

In this phase, we extend the ticked versions to also work with empty priority queues.

definition *bs-empty* **where** *bs-empty* ≡ *Inl* ()

primrec *bs-findMin* **where**
bs-findMin (*Inr x*) = *findMin′ x*

fun *bs-meld*

:: ('e,'a::linorder) BsSkewHeap ⇒ ('e,'a) BsSkewHeap ⇒ ('e,'a) BsSkewHeap
where
bs-meld (*Inl* -) *x* = *x* |
bs-meld x (*Inl* -) = *x* |
bs-meld (*Inr x*) (*Inr x'*) = *Inr* (*meld' x x'*)
lemma [*simp*]: *bs-meld x* (*Inl u*) = *x*
⟨*proof*⟩

primrec *bs-insert*
:: 'e ⇒ ('a::linorder) ⇒ ('e,'a) BsSkewHeap ⇒ ('e,'a) BsSkewHeap
where
bs-insert e a (*Inl* -) = *Inr* (*Element e a* []) |
bs-insert e a (*Inr x*) = *Inr* (*insert' e a x*)

fun *bs-deleteMin*
:: ('e,'a::linorder) BsSkewHeap ⇒ ('e,'a) BsSkewHeap
where
bs-deleteMin (*Inr* (*Element e a* [])) = *Inl* () |
bs-deleteMin (*Inr* (*Element e a q*)) = *Inr* (*deleteMin'* (*Element e a q*))

primrec *bs-invar* :: ('e,'a::linorder) BsSkewHeap ⇒ bool
where
bs-invar (*Inl* -) ⟷ *True* |
bs-invar (*Inr x*) ⟷ *elem-invar x*

lemma [*simp*]: *bs-invar bs-empty* ⟨*proof*⟩

primrec *bs-to-mset* :: ('e,'a::linorder) BsSkewHeap ⇒ ('e×'a) multiset
where
bs-to-mset (*Inl* -) = {#} |
bs-to-mset (*Inr x*) = *elem-to-mset x*

theorem *bs-empty-correct*: *h*=*bs-empty* ⟷ *bs-to-mset h* = {#}
⟨*proof*⟩

lemma *bs-mset-of-empty*[*simp*]:
bs-to-mset bs-empty = {#}
⟨*proof*⟩

theorem *bs-findMin-correct*:
assumes *I*: *bs-invar h*
assumes *NE*: *h*≠*bs-empty*
shows *bs-findMin h* ∈# *bs-to-mset h*
∀ *y*∈*set-mset* (*bs-to-mset h*). *snd* (*bs-findMin h*) ≤ *snd y*
⟨*proof*⟩

theorem *bs-insert-correct*:
assumes *I*: *bs-invar h*
shows

bs-invar (*bs-insert e a h*)
bs-to-mset (*bs-insert e a h*) = {#(*e*,*a*)#} + *bs-to-mset h*
⟨*proof*⟩

theorem *bs-meld-correct*:
assumes *I*: *bs-invar h bs-invar h′*
shows
bs-invar (*bs-meld h h′*)
bs-to-mset (*bs-meld h h′*) = *bs-to-mset h* + *bs-to-mset h′*
⟨*proof*⟩

theorem *bs-deleteMin-correct*:
assumes *I*: *bs-invar h*
assumes *NE*: *h* ≠ *bs-empty*
shows
bs-invar (*bs-deleteMin h*)
bs-to-mset (*bs-deleteMin h*) = *bs-to-mset h* − {#*bs-findMin h*#}
⟨*proof*⟩

end

interpretation *BsSkewBinomialHeapStruc*: *Bootstrapped* ⟨*proof*⟩

2.4 Hiding the Invariant

2.4.1 Datatype

typedef (**overloaded**) (*′e*, *′a*) *SkewBinomialHeap* =
{*q* :: (*′e*,*′a*::*linorder*) *BsSkewBinomialHeapStruc.BsSkewHeap*. *BsSkewBinomialHeapStruc.bs-invar q* }
⟨*proof*⟩

lemma *Rep-SkewBinomialHeap-invar*[*simp*]:
BsSkewBinomialHeapStruc.bs-invar (*Rep-SkewBinomialHeap x*)
⟨*proof*⟩

lemma [*simp*]:
BsSkewBinomialHeapStruc.bs-invar q
⟹ *Rep-SkewBinomialHeap* (*Abs-SkewBinomialHeap q*) = *q*
⟨*proof*⟩

lemma [*simp*, *code abstype*]: *Abs-SkewBinomialHeap* (*Rep-SkewBinomialHeap q*) = *q*
⟨*proof*⟩

locale *SkewBinomialHeap-loc*
begin

2.4.2 Operations

definition [*code*]:
to-mset t
== *BsSkewBinomialHeapStruc.bs-to-mset* (*Rep-SkewBinomialHeap t*)

definition *empty* **where**
empty == *Abs-SkewBinomialHeap BsSkewBinomialHeapStruc.bs-empty*
lemma [*code abstract*, *simp*]:
Rep-SkewBinomialHeap empty = *BsSkewBinomialHeapStruc.bs-empty*
⟨*proof*⟩

definition [*code*]:
isEmpty q == *Rep-SkewBinomialHeap q* = *BsSkewBinomialHeapStruc.bs-empty*
lemma *empty-rep*:
q=*empty* ⟷ *Rep-SkewBinomialHeap q* = *BsSkewBinomialHeapStruc.bs-empty*
⟨*proof*⟩

lemma *isEmpty-correct*: *isEmpty q* ⟷ *q*=*empty*
⟨*proof*⟩

definition
insert
:: *'e* ⇒ (*'a*::*linorder*) ⇒ (*'e*,*'a*) *SkewBinomialHeap*
⇒ (*'e*,*'a*) *SkewBinomialHeap*
where *insert e a q* ==
Abs-SkewBinomialHeap (
BsSkewBinomialHeapStruc.bs-insert e a (*Rep-SkewBinomialHeap q*))
lemma [*code abstract*]:
Rep-SkewBinomialHeap (*insert e a q*)
= *BsSkewBinomialHeapStruc.bs-insert e a* (*Rep-SkewBinomialHeap q*)
⟨*proof*⟩

definition [*code*]: *findMin q*
== *BsSkewBinomialHeapStruc.bs-findMin* (*Rep-SkewBinomialHeap q*)

definition *deleteMin q* ==
if q=*empty then empty*
else Abs-SkewBinomialHeap (
BsSkewBinomialHeapStruc.bs-deleteMin (*Rep-SkewBinomialHeap q*))

We don't use equality here, to prevent the code-generator from introducing equality-class parameter for type *'a*. Instead we use a case-distinction to check for emptiness.

lemma [*code abstract*]: *Rep-SkewBinomialHeap* (*deleteMin q*) =
(*case* (*Rep-SkewBinomialHeap q*) *of Inl* - ⇒ *BsSkewBinomialHeapStruc.bs-empty*
|
- ⇒ *BsSkewBinomialHeapStruc.bs-deleteMin* (*Rep-SkewBinomialHeap q*))
⟨*proof*⟩

definition *meld q1 q2* ==
Abs-SkewBinomialHeap (*BsSkewBinomialHeapStruc.bs-meld*
(*Rep-SkewBinomialHeap q1*) (*Rep-SkewBinomialHeap q2*))
lemma [*code abstract*]:
Rep-SkewBinomialHeap (*meld q1 q2*)
= *BsSkewBinomialHeapStruc.bs-meld* (*Rep-SkewBinomialHeap q1*)
(*Rep-SkewBinomialHeap q2*)
⟨*proof*⟩

2.4.3 Correctness

lemma *empty-correct*: *to-mset q* = {#} ⟷ *q*=*empty*
⟨*proof*⟩

lemma *to-mset-of-empty*[*simp*]: *to-mset empty* = {#}
⟨*proof*⟩

lemma *insert-correct*: *to-mset* (*insert e a q*) = *to-mset q* + {#(*e*,*a*)#}
⟨*proof*⟩

lemma *findMin-correct*:
assumes *q*≠*empty*
shows
findMin q ∈# *to-mset q*
∀ *y*∈*set-mset* (*to-mset q*). *snd* (*findMin q*) ≤ *snd y*
⟨*proof*⟩

lemma *deleteMin-correct*:
assumes *q*≠*empty*
shows *to-mset* (*deleteMin q*) = *to-mset q* − {# *findMin q* #}
⟨*proof*⟩

lemma *meld-correct*:
shows *to-mset* (*meld q q*′) = *to-mset q* + *to-mset q*′
⟨*proof*⟩

Correctness lemmas to be used with simplifier

lemmas *correct* = *empty-correct deleteMin-correct meld-correct*

end

interpretation *SkewBinomialHeap*: *SkewBinomialHeap-loc* ⟨*proof*⟩

2.5 Documentation

SkewBinomialHeap.to-mset
Abstraction to multiset.

SkewBinomialHeap.empty
The empty heap. ($O(1)$)
Spec *SkewBinomialHeap.empty-correct*:

(*SkewBinomialHeap.to-mset* $q = \{\#\}$) = (q = *SkewBinomialHeap.empty*)

SkewBinomialHeap.isEmpty
Checks whether heap is empty. Mainly used to work around code-generation issues. ($O(1)$)
Spec *SkewBinomialHeap.isEmpty-correct*:

SkewBinomialHeap.isEmpty q = (q = *SkewBinomialHeap.empty*)

SkewBinomialHeap.insert

SkewBinomialHeap.insert

Inserts element ($O(1)$)
Spec *SkewBinomialHeap.insert-correct*:

SkewBinomialHeap.to-mset (*SkewBinomialHeap.insert* e a q) =
SkewBinomialHeap.to-mset q + $\{\#(e, a)\#\}$

SkewBinomialHeap.findMin
Returns a minimal element ($O(1)$)
Spec *SkewBinomialHeap.findMin-correct*:

$q \neq$ *SkewBinomialHeap.empty* $\Longrightarrow$
SkewBinomialHeap.findMin $q \in\#$ *SkewBinomialHeap.to-mset* q
$q \neq$ *SkewBinomialHeap.empty* $\Longrightarrow$
$\forall\, y \in\#$ *SkewBinomialHeap.to-mset* q. *snd* (*SkewBinomialHeap.findMin* q) $\leq$ *snd* y

SkewBinomialHeap.deleteMin

SkewBinomialHeap.deleteMin

Deletes *the* element that is returned by *find_min*. $O(\log(n))$
Spec *SkewBinomialHeap.deleteMin-correct*:

$q \neq$ *SkewBinomialHeap.empty* $\Longrightarrow$
SkewBinomialHeap.to-mset (*SkewBinomialHeap.deleteMin* q) =
SkewBinomialHeap.to-mset q − $\{\#$*SkewBinomialHeap.findMin* $q\#\}$

SkewBinomialHeap.meld

SkewBinomialHeap.meld

Melds two heaps ($O(1)$)
Spec *SkewBinomialHeap.meld-correct*:

SkewBinomialHeap.to-mset (*SkewBinomialHeap.meld* q q') =
SkewBinomialHeap.to-mset q + *SkewBinomialHeap.to-mset* q'

end

References

[1] G. S. Brodal and C. Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, 6:839–857, 1996.