

Binomial Heaps and Skew Binomial Heaps

Rene Meis Finn Nielsen Peter Lammich

April 19, 2020

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  $\Rightarrow$  ('e, 'a) BinomialTree  $\Rightarrow$ 
  ('e, 'a) BinomialTree where
  link (Node e1 a1 r1 ts1) (Node e2 a2 r2 ts2) =
    (if a1  $\leq$  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  $\Rightarrow$  ('e  $\times$  'a) multiset
and queue-to-multiset
  :: ('e, 'a::linorder) BinomialQueue-inv  $\Rightarrow$  ('e  $\times$  '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 \Rightarrow bool **where**
tree-invar (Node e a 0 ts) = (ts = []) |
tree-invar (Node e a (Suc r) ts) =
 (\exists e1 a1 ts1 e2 a2 ts2.
tree-invar (Node e1 a1 r ts1) \wedge
tree-invar (Node e2 a2 r ts2) \wedge
 (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 \Rightarrow bool **where**
queue-invar q \equiv ($\forall t \in$ 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 \Rightarrow bool **where**
rank-invar [] = True |
rank-invar [t] = True |
rank-invar (t # t' # bq) = (rank t < rank t' \wedge *rank-invar* (t' # bq))

lemma *queue-invar-simps*[simp]:
queue-invar []
queue-invar (t#q) \longleftrightarrow *tree-invar* t \wedge *queue-invar* q
queue-invar (q@q') \longleftrightarrow *queue-invar* q \wedge *queue-invar* q'
 <proof>

Invariant for binomial queues:

definition *invar* q == *queue-invar* q \wedge *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]] \implies *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'*: $tree\text{-invar } t \implies queue\text{-invar } (children\ t)$
 ⟨proof⟩

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

lemma *rank-invar-not-empty-hd*: $\llbracket rank\text{-invar } (t \# bq); bq \neq [] \rrbracket \implies$
 $rank\ t < rank\ (hd\ bq)$
 ⟨proof⟩

lemma *rank-invar-to-set*: $rank\text{-invar } (t \# bq) \implies$
 $\forall t' \in set\ bq. rank\ t < rank\ t'$
 ⟨proof⟩

lemma *set-to-rank-invar*: $\llbracket \forall t' \in set\ bq. rank\ t < rank\ t'; rank\text{-invar } bq \rrbracket$
 $\implies rank\text{-invar } (t \# bq)$
 ⟨proof⟩

lemma *rank-invar-hd-cons*:
 $\llbracket rank\text{-invar } bq; rank\ t < rank\ (hd\ bq) \rrbracket \implies rank\text{-invar } (t \# bq)$
 ⟨proof⟩

lemma *rank-invar-cons*: $rank\text{-invar } (t \# bq) \implies rank\text{-invar } bq$
 ⟨proof⟩

lemma *invar-cons-up*:
 $\llbracket invar\ (t \# bq); rank\ t' < rank\ t; tree\text{-invar } t' \rrbracket \implies invar\ (t' \# t \# bq)$
 ⟨proof⟩

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

lemma *invar-app-single*:
 $\llbracket invar\ bq; \forall t \in set\ bq. rank\ t < rank\ t'; tree\text{-invar } t' \rrbracket$
 $\implies invar\ (bq\ @\ [t'])$
 ⟨proof⟩

1.1.3 Heap Ordering

fun *heap-ordered* :: $(\prime e, \prime a::linorder)\ BinomialTree \Rightarrow bool$ **where**
 $heap\text{-ordered } (Node\ e\ a\ r\ ts) = (\forall x \in set\text{-mset}(queue\text{-to-multiset } ts). a \leq snd\ x)$

The invariant for trees implies heap order.

lemma *tree-invar-heap-ordered*:
assumes $tree\text{-invar } t$
shows $heap\text{-ordered } t$
 ⟨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  $\Rightarrow$  nat and
    height-queue :: ('e, ('a::linorder)) BinomialQueue-inv  $\Rightarrow$  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) \implies
 size (tree-to-multiset (Node e a r ts)) = (2::nat)^r
 <proof>

lemma tree-rank-height:
 tree-invar (Node e a r ts) \implies 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 \implies size (tree-to-multiset t) = (2::nat)^(height-tree t)
 <proof>

lemma size-mset-tree: tree-invar t \implies
 size (tree-to-multiset t) = (2::nat)^(rank t)
 <proof>

lemma invar-butlast: invar (bq @ [t]) \implies invar bq
 <proof>

lemma invar-last-max: invar (bq @ [m]) $\implies \forall t \in \text{set } bq. \text{rank } t < \text{rank } m$
 <proof>

lemma invar-length: invar bq \implies length bq \leq Suc (rank (last bq))
 <proof>

lemma *size-queue-sum-list*:

$size (queue\text{-}to\text{-}multiset\ bq) = sum\text{-}list (map (size \circ tree\text{-}to\text{-}multiset) bq)$
(*proof*)

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

theorem *queue-length-estimate-lower*:

$invar\ bq \implies (size (queue\text{-}to\text{-}multiset\ bq)) \geq 2^{(length\ bq)} - 1$
(*proof*)

1.2 Operations

1.2.1 Empty

lemma *empty-correct[simp]*:

$invar\ Nil$
 $queue\text{-}to\text{-}multiset\ Nil = \{\#\}$
(*proof*)

The empty multiset is represented by exactly the empty queue

lemma *empty-iff*: $t = Nil \iff queue\text{-}to\text{-}multiset\ t = \{\#\}$

(*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* :: $(e, 'a::linorder) BinomialTree \Rightarrow (e, 'a) BinomialQueue\text{-}inv \Rightarrow$

$(e, 'a) BinomialQueue\text{-}inv$ **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* :: $e \Rightarrow 'a::linorder \Rightarrow (e, 'a) BinomialQueue\text{-}inv \Rightarrow$

$(e, 'a) BinomialQueue\text{-}inv$ **where**

$insert\ e\ a\ bq = ins\ (Node\ e\ a\ 0\ [])\ bq$

lemma *ins-mset*:

$[tree\text{-}invar\ t; queue\text{-}invar\ q] \implies queue\text{-}to\text{-}multiset\ (ins\ t\ q)$
 $= tree\text{-}to\text{-}multiset\ t + queue\text{-}to\text{-}multiset\ q$

(*proof*)

lemma *insert-mset*: $queue\text{-}invar\ q \implies$

$queue\text{-}to\text{-}multiset\ (insert\ e\ a\ q) = queue\text{-}to\text{-}multiset\ q + \{\#\ (e, a)\ \#\}$

<proof>

lemma *ins-queue-invar*: $\llbracket \text{tree-invar } t; \text{queue-invar } q \rrbracket \implies \text{queue-invar } (\text{ins } t \ q)$
<proof>

lemma *insert-queue-invar*:
 assumes *queue-invar* q
 shows *queue-invar* $(\text{insert } e \ a \ q)$
<proof>

lemma *rank-ins*: $(\text{rank-invar } (t \ \# \ bq) \implies$
 $(\text{rank } (\text{hd } (\text{ins } t' \ (t \ \# \ bq))) \geq \text{rank } t) \vee$
 $(\text{rank } (\text{hd } (\text{ins } t' \ (t \ \# \ bq))) \geq \text{rank } t'))$
<proof>

lemma *rank-ins2*: *rank-invar* $bq \implies$
 $\text{rank } t \leq \text{rank } (\text{hd } (\text{ins } t \ bq)) \vee$
 $(\text{rank } (\text{hd } (\text{ins } t \ bq)) = \text{rank } (\text{hd } bq) \wedge bq \neq [])$
<proof>

lemma *rank-invar-ins*: *rank-invar* $bq \implies \text{rank-invar } (\text{ins } t \ bq)$
<proof>

lemma *rank-invar-insert*: *rank-invar* $bq \implies \text{rank-invar } (\text{insert } e \ a \ bq)$
<proof>

lemma *insert-correct*:
 assumes $I: \text{invar } q$
 shows
 $\text{invar } (\text{insert } e \ a \ q)$
 $\text{queue-to-multiset } (\text{insert } e \ a \ q) = \text{queue-to-multiset } q + \{\# (e, a) \# \}$
<proof>

1.2.3 Meld

Melds two queues.

fun *meld* :: $(\text{'e}, \text{'a}::\text{linorder}) \text{ BinomialQueue-inv} \Rightarrow (\text{'e}, \text{'a}) \text{ BinomialQueue-inv}$
 $\Rightarrow (\text{'e}, \text{'a}) \text{ BinomialQueue-inv}$
 where
 $\text{meld } [] \ bq = bq \mid$
 $\text{meld } bq \ [] = bq \mid$
 $\text{meld } (t1 \ \# \ bq1) \ (t2 \ \# \ bq2) =$
 $(\text{if } (\text{rank } t1) < (\text{rank } t2)$
 $\text{then } t1 \ \# \ (\text{meld } bq1 \ (t2 \ \# \ bq2))$
 $\text{else } ($
 $\text{if } (\text{rank } t2 < \text{rank } t1)$
 $\text{then } t2 \ \# \ (\text{meld } (t1 \ \# \ bq1) \ bq2)$
 $\text{else } \text{ins } (\text{link } t1 \ t2) \ (\text{meld } bq1 \ bq2)$
 $)$
 $)$

)

lemma *meld-queue-invar*:

$\llbracket \text{queue-invar } q; \text{queue-invar } q' \rrbracket \implies \text{queue-invar } (\text{meld } q \ q')$
<proof>

lemma *rank-ins-min*: *rank-invar* *bq* \implies

$\text{rank } (\text{hd } (\text{ins } t \ bq)) \geq \min (\text{rank } t) (\text{rank } (\text{hd } bq))$
<proof>

lemma *rank-invar-meld-strong*:

$\llbracket \text{rank-invar } bq1; \text{rank-invar } bq2 \rrbracket \implies \text{rank-invar } (\text{meld } bq1 \ bq2) \wedge$
 $\text{rank } (\text{hd } (\text{meld } bq1 \ bq2)) \geq \min (\text{rank } (\text{hd } bq1)) (\text{rank } (\text{hd } bq2))$
<proof>

lemma *rank-invar-meld*:

$\llbracket \text{rank-invar } bq1; \text{rank-invar } bq2 \rrbracket \implies \text{rank-invar } (\text{meld } bq1 \ bq2)$
<proof>

lemma *meld-mset*: $\llbracket \text{queue-invar } q; \text{queue-invar } q' \rrbracket \implies$

$\text{queue-to-multiset } (\text{meld } q \ q') =$
 $\text{queue-to-multiset } q + \text{queue-to-multiset } q'$
<proof>

lemma *meld-correct*:

assumes *invar* *q* *invar* *q'*

shows

invar $(\text{meld } q \ q')$

$\text{queue-to-multiset } (\text{meld } q \ q') = \text{queue-to-multiset } q + \text{queue-to-multiset } q'$

<proof>

1.2.4 Find Minimal Element

Finds the tree containing the minimal element.

fun *getMinTree* :: $(\text{'e}, \text{'a}::\text{linorder}) \text{ BinomialQueue-inv} \implies$

$(\text{'e}, \text{'a}) \text{ BinomialTree}$ **where**

getMinTree $[t] = t \mid$

getMinTree $(t\#bq) = (\text{if } \text{prio } t \leq \text{prio } (\text{getMinTree } bq)$

$\text{then } t \text{ else } (\text{getMinTree } bq))$

lemma *mintree-exists*: $(bq \neq []) = (\text{getMinTree } bq \in \text{set } bq)$

<proof>

lemma *treehead-in-multiset*:

$t \in \text{set } bq \implies (\text{val } t, \text{prio } t) \in\# \text{queue-to-multiset } bq$

<proof>

lemma *heap-ordered-single*:

heap-ordered $t = (\forall x \in \text{set-mset } (\text{tree-to-multiset } t). \text{prio } t \leq \text{snd } x)$

<proof>

lemma *getMinTree-cons*:

$\text{prio } (\text{getMinTree } (y \# x \# xs)) \leq \text{prio } (\text{getMinTree } (x \# xs))$
<proof>

lemma *getMinTree-min-tree*:

$t \in \text{set } bq \implies \text{prio } (\text{getMinTree } bq) \leq \text{prio } t$
<proof>

lemma *getMinTree-min-prio*:

assumes *queue-invar* *bq*
and $y \in \text{set-mset } (\text{queue-to-multiset } bq)$
shows $\text{prio } (\text{getMinTree } bq) \leq \text{snd } y$
<proof>

Finds the minimal Element in the queue.

definition *findMin* :: $('e, 'a::\text{linorder}) \text{ BinomialQueue-inv} \implies ('e \times 'a)$ **where**
 $\text{findMin } bq = (\text{let } \text{min} = \text{getMinTree } bq \text{ in } (\text{val } \text{min}, \text{prio } \text{min}))$

lemma *findMin-correct*:

assumes *I*: *invar* *q*
assumes *NE*: $q \neq \text{Nil}$
shows
 $\text{findMin } q \in \# \text{queue-to-multiset } q$
 $\forall y \in \text{set-mset } (\text{queue-to-multiset } q). \text{snd } (\text{findMin } q) \leq \text{snd } y$
<proof>

1.2.5 Delete Minimal Element

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

fun *remove1Prio* :: $'a \implies ('e, 'a::\text{linorder}) \text{ BinomialQueue-inv} \implies$
 $('e, 'a) \text{ BinomialQueue-inv}$ **where**
 $\text{remove1Prio } a \ [] = [] \mid$
 $\text{remove1Prio } a (t \# bq) =$
 $(\text{if } (\text{prio } t) = a \text{ then } bq \text{ else } t \# (\text{remove1Prio } a \ bq))$

Returns the queue without the minimal element.

definition *deleteMin* :: $('e, 'a::\text{linorder}) \text{ BinomialQueue-inv} \implies$
 $('e, 'a) \text{ BinomialQueue-inv}$ **where**
 $\text{deleteMin } bq \equiv (\text{let } \text{min} = \text{getMinTree } bq \text{ in}$
 $\text{meld } (\text{rev } (\text{children } \text{min}))$
 $(\text{remove1Prio } (\text{prio } \text{min}) \ bq))$

lemma *queue-invar-rev*: $\text{queue-invar } q \implies \text{queue-invar } (\text{rev } q)$
<proof>

lemma *queue-invar-remove1*: $\text{queue-invar } q \implies \text{queue-invar } (\text{remove1 } t \ q)$

<proof>

lemma *qtm-in-set-subset*: $t \in \text{set } q \implies$
 $\text{tree-to-multiset } t \subseteq\# \text{ queue-to-multiset } q$
<proof>

lemma *remove1-mset*: $t \in \text{set } q \implies$
 $\text{queue-to-multiset } (\text{remove1 } t \ q) =$
 $\text{queue-to-multiset } q - \text{tree-to-multiset } t$
<proof>

lemma *remove1Prio-remove1[simp]*:
 $\text{remove1Prio } (\text{prio } (\text{getMinTree } bq)) \ bq = \text{remove1 } (\text{getMinTree } bq) \ bq$
<proof>

lemma *deleteMin-queue-invar*:
assumes *INV*: $\text{queue-invar } q$
assumes *NE*: $q \neq \text{Nil}$
shows $\text{queue-invar } (\text{deleteMin } q)$
<proof>

lemma *children-rank-less*:
assumes *tree-invar* t
shows $\forall t' \in \text{set } (\text{children } t). \text{rank } t' < \text{rank } t$
<proof>

lemma *strong-rev-children*:
assumes *tree-invar* t
shows $\text{invar } (\text{rev } (\text{children } t))$
<proof>

lemma *first-less*: $\text{rank-invar } (t \ \# \ bq) \implies \forall t' \in \text{set } bq. \text{rank } t < \text{rank } t'$
<proof>

lemma *strong-remove1*: $\text{invar } bq \implies \text{invar } (\text{remove1 } t \ bq)$
<proof>

theorem *deleteMin-invar*:
assumes $\text{invar } bq$
and $bq \neq []$
shows $\text{invar } (\text{deleteMin } bq)$
<proof>

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

lemma *deleteMin-mset*:
assumes $\text{queue-invar } q$

```

  and  $q \neq Nil$ 
  shows  $queue\text{-}to\text{-}multiset (deleteMin\ q) = queue\text{-}to\text{-}multiset\ q - \{\# (findMin\ q)\}$ 
  #}
  <proof>

```

lemma *deleteMin-correct*:

assumes *INV*: $invar\ q$

assumes *NE*: $q \neq Nil$

shows

$invar (deleteMin\ q)$

$queue\text{-}to\text{-}multiset (deleteMin\ q) = queue\text{-}to\text{-}multiset\ q - \{\# (findMin\ q)\}$

<proof>

end

interpretation *BinomialHeapStruc*: *BinomialHeapStruc-loc* <proof>

1.3 Hiding the Invariant

1.3.1 Datatype

```

typedef (overloaded) ('e, 'a) BinomialHeap =
  {  $q :: ('e, 'a :: linorder)\ BinomialHeapStruc.BinomialQueue\text{-}inv.\ BinomialHeapStruc.invar\ q$  }
  <proof>

```

lemma *Rep-BinomialHeap-invar[simp]*:

$BinomialHeapStruc.invar (Rep\text{-}BinomialHeap\ x)$

<proof>

lemma [*simp*]:

$BinomialHeapStruc.invar\ q \implies Rep\text{-}BinomialHeap (Abs\text{-}BinomialHeap\ q) = q$

<proof>

lemma [*simp, code abstype*]: $Abs\text{-}BinomialHeap (Rep\text{-}BinomialHeap\ q) = q$

<proof>

locale *BinomialHeap-loc*

begin

1.3.2 Operations

definition [*code*]:

$to\text{-}mset\ t == BinomialHeapStruc.queue\text{-}to\text{-}multiset (Rep\text{-}BinomialHeap\ t)$

definition *empty* **where** $empty == Abs\text{-}BinomialHeap\ Nil$

lemma [*code abstract, simp*]: $Rep\text{-}BinomialHeap\ empty = []$

<proof>

definition [code]: $isEmpty\ q == Rep\text{-}BinomialHeap\ q = Nil$
lemma *empty-rep*: $q=empty \longleftrightarrow Rep\text{-}BinomialHeap\ q = Nil$
 ⟨proof⟩

lemma *isEmpty-correct*: $isEmpty\ q \longleftrightarrow q=empty$
 ⟨proof⟩

definition

insert

$:: 'e \Rightarrow ('a::linorder) \Rightarrow ('e,'a)\ BinomialHeap \Rightarrow ('e,'a)\ BinomialHeap$

where *insert e a q ==*

Abs-BinomialHeap (BinomialHeapStruct.insert e a (Rep-BinomialHeap

q))

lemma [code abstract]:

Rep-BinomialHeap (insert e a q)

= BinomialHeapStruct.insert e a (Rep-BinomialHeap q)

⟨proof⟩

definition [code]: $findMin\ q == BinomialHeapStruct.findMin\ (Rep\text{-}BinomialHeap\ q)$

definition *deleteMin q ==*

if q=empty then empty

else Abs-BinomialHeap (BinomialHeapStruct.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 [] => [] |

- => BinomialHeapStruct.deleteMin (Rep-BinomialHeap q))

⟨proof⟩

definition *meld q1 q2 ==*

Abs-BinomialHeap (BinomialHeapStruct.meld (Rep-BinomialHeap q1)

(Rep-BinomialHeap q2))

lemma [code abstract]:

Rep-BinomialHeap (meld q1 q2)

= BinomialHeapStruct.meld (Rep-BinomialHeap q1) (Rep-BinomialHeap q2)

⟨proof⟩

1.3.3 Correctness

lemma *empty-correct*: $to\text{-}mset\ q = \{\#\} \longleftrightarrow q=empty$

⟨proof⟩

lemma *to-mset-of-empty[simp]*: $to\text{-}mset\ empty = \{\#\}$

$\langle proof \rangle$

lemma *insert-correct*: $to\text{-}mset (insert\ e\ a\ q) = to\text{-}mset\ q + \{\#(e,a)\# \}$
 $\langle proof \rangle$

lemma *findMin-correct*:

assumes $q \neq empty$

shows

$findMin\ q \in \# to\text{-}mset\ q$

$\forall y \in set\text{-}mset (to\text{-}mset\ q). snd (findMin\ q) \leq snd\ y$

$\langle proof \rangle$

lemma *deleteMin-correct*:

assumes $q \neq empty$

shows $to\text{-}mset (deleteMin\ q) = to\text{-}mset\ q - \{\# findMin\ q \# \}$

$\langle proof \rangle$

lemma *meld-correct*:

shows $to\text{-}mset (meld\ q\ q') = to\text{-}mset\ q + to\text{-}mset\ q'$

$\langle proof \rangle$

Correctness lemmas to be used with simplifier

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

end

interpretation *BinomialHeap: BinomialHeap-loc* $\langle proof \rangle$

1.4 Documentation

BinomialHeap.to_mset::('a, 'b) BinomialHeap \Rightarrow ('a \times 'b) multiset
Abstraction to multiset.

BinomialHeap.empty::('a, 'b) BinomialHeap

The empty heap. ($O(1)$)

Spec *BinomialHeap.empty-correct*:

$(BinomialHeap.to\text{-}mset\ q = \{\#\}) = (q = BinomialHeap.empty)$

BinomialHeap.isEmpty::('a, 'b) BinomialHeap \Rightarrow bool

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::'a \Rightarrow 'b \Rightarrow ('a, 'b) BinomialHeap \Rightarrow ('a, 'b) BinomialHeap

Inserts element ($O(\log(n))$)

Spec *BinomialHeap.insert-correct*:

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

$BinomialHeap.findMin::('a, 'b) BinomialHeap \Rightarrow 'a \times 'b$
Returns a minimal element ($O(\log(n))$)

Spec $BinomialHeap.findMin$ -correct:

$q \neq BinomialHeap.empty \implies BinomialHeap.findMin q \in \# BinomialHeap.to-mset$
 q
 $q \neq BinomialHeap.empty \implies$
 $\forall y \in \# BinomialHeap.to-mset q. snd (BinomialHeap.findMin q) \leq snd y$

$BinomialHeap.deleteMin::('a, 'b) BinomialHeap \Rightarrow ('a, 'b) BinomialHeap$
Deletes the element that is returned by $find_min$

Spec $BinomialHeap.deleteMin$ -correct:

$q \neq BinomialHeap.empty \implies$
 $BinomialHeap.to-mset (BinomialHeap.deleteMin q) =$
 $BinomialHeap.to-mset q - \{\# BinomialHeap.findMin q \# \}$

$BinomialHeap.meld$

$BinomialHeap.meld::('a, 'b) BinomialHeap$
 $\Rightarrow ('a, 'b) BinomialHeap \Rightarrow ('a, 'b) BinomialHeap$

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* \Rightarrow (*e* \times *a*) *multiset*

and *queue-to-multiset*
 $::$ (*e*, *a::linorder*) *SkewBinomialQueue* \Rightarrow (*e* \times *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 *tm-children*: *tree-to-multiset* *t* =
 $\{\#(val\ t, prio\ t)\#$ + *queue-to-multiset* (*children* *t*)
 $\langle proof \rangle$

lemma *qtm-conc[simp]*: *queue-to-multiset* (*q@q'*)
= *queue-to-multiset* *q* + *queue-to-multiset* *q'*
 $\langle proof \rangle$

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* \Rightarrow *bool* **where**
rank-invar [] = *True* |
rank-invar [t] = *True* |
rank-invar (t # t' # bq) = (rank t < rank t' \wedge *rank-invar* (t' # bq))

For whole queue: First two elements may have same rank

fun *rank-skew-invar* :: ('e, 'a::linorder) *SkewBinomialQueue* \Rightarrow *bool* **where**
rank-skew-invar [] = *True* |
rank-skew-invar [t] = *True* |
rank-skew-invar (t # t' # bq) = ((rank t \leq rank t') \wedge *rank-invar* (t' # bq))

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

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

lemma *invar-empty[simp]*:

invar []
tail-invar []
 \langle *proof* \rangle

lemma *invar-tail-invar*:

invar (t # bq) \Longrightarrow *tail-invar* bq
 \langle *proof* \rangle

lemma *link-mset[simp]*: *tree-to-multiset* (*link* t1 t2)

= *tree-to-multiset* t1 + *tree-to-multiset* t2
 \langle *proof* \rangle

lemma *link-tree-invar*: \llbracket *tree-invar* t1; *tree-invar* t2; rank t1 = rank t2 $\rrbracket \Longrightarrow$

tree-invar (*link* t1 t2)
 \langle *proof* \rangle

lemma *skewlink-mset[simp]*: *tree-to-multiset* (*skewlink* e a t1 t2)

= {# (e,a) #} + *tree-to-multiset* t1 + *tree-to-multiset* t2
 \langle *proof* \rangle

lemma *skewlink-tree-invar*: \llbracket *tree-invar* t1; *tree-invar* t2; rank t1 = rank t2 $\rrbracket \Longrightarrow$

tree-invar (*skewlink* e a t1 t2)
 \langle *proof* \rangle

lemma *rank-link*: rank t = rank t' \Longrightarrow rank (*link* t t') = rank t + 1

\langle *proof* \rangle

lemma *rank-skew-rank-invar*: *rank-skew-invar* (t # bq) \Longrightarrow *rank-invar* bq

\langle *proof* \rangle

lemma *rank-invar-rank-skew*:

assumes *rank-invar* q

shows *rank-skew-invar* q

\langle *proof* \rangle

lemma *rank-invar-cons-up*:

$\llbracket \text{rank-invar } (t \# bq); \text{rank } t' < \text{rank } t \rrbracket \implies \text{rank-invar } (t' \# t \# bq)$

\langle *proof* \rangle

lemma *rank-skew-cons-up*:

$\llbracket \text{rank-invar } (t \# bq); \text{rank } t' \leq \text{rank } t \rrbracket \implies \text{rank-skew-invar } (t' \# t \# bq)$

\langle *proof* \rangle

lemma *rank-invar-cons-down*: $\text{rank-invar } (t \# bq) \implies \text{rank-invar } bq$

\langle *proof* \rangle

lemma *rank-invar-hd-cons*:

$\llbracket \text{rank-invar } bq; \text{rank } t < \text{rank } (\text{hd } bq) \rrbracket \implies \text{rank-invar } (t \# bq)$

\langle *proof* \rangle

lemma *tail-invar-cons-up*:

$\llbracket \text{tail-invar } (t \# bq); \text{rank } t' < \text{rank } t; \text{tree-invar } t \rrbracket$

$\implies \text{tail-invar } (t' \# t \# bq)$

\langle *proof* \rangle

lemma *tail-invar-cons-up-invar*:

$\llbracket \text{tail-invar } (t \# bq); \text{rank } t' \leq \text{rank } t; \text{tree-invar } t \rrbracket \implies \text{invar } (t' \# t \# bq)$

\langle *proof* \rangle

lemma *tail-invar-cons-down*:

$\text{tail-invar } (t \# bq) \implies \text{tail-invar } bq$

\langle *proof* \rangle

lemma *tail-invar-app-single*:

$\llbracket \text{tail-invar } bq; \forall t \in \text{set } bq. \text{rank } t < \text{rank } t'; \text{tree-invar } t \rrbracket$

$\implies \text{tail-invar } (bq @ [t'])$

\langle *proof* \rangle

lemma *invar-app-single*:

$\llbracket \text{invar } bq; \forall t \in \text{set } bq. \text{rank } t < \text{rank } t'; \text{tree-invar } t \rrbracket$

$\implies \text{invar } (bq @ [t'])$

\langle *proof* \rangle

lemma *invar-children*:

assumes *tree-invar* $((\text{Node } e \ a \ r \ ts)::((e, 'a)::\text{linorder}) \text{SkewBinomialTree})$

shows *queue-invar* ts \langle *proof* \rangle

2.1.3 Heap Order

fun *heap-ordered* :: ('e, 'a::linorder) *SkewBinomialTree* \Rightarrow *bool* **where**
heap-ordered (*Node e a r ts*)
 $= (\forall x \in \text{set-mset } (\text{queue-to-multiset } ts). a \leq \text{snd } x)$

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*
 $\langle \text{proof} \rangle$

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* \Rightarrow *nat* **and**
height-queue :: ('e, ('a::linorder)) *SkewBinomialQueue* \Rightarrow *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*)
 $\langle \text{proof} \rangle$

lemma *tree-rank-estimate-upper*:
tree-invar (*Node e a r ts*) \implies
size (*tree-to-multiset* (*Node e a r ts*)) $\leq (2::\text{nat})^{(\text{Suc } r)} - 1$
 $\langle \text{proof} \rangle$

lemma *tree-rank-estimate-lower*:
tree-invar (*Node e a r ts*) \implies
size (*tree-to-multiset* (*Node e a r ts*)) $\geq (2::\text{nat})^r$
 $\langle \text{proof} \rangle$

lemma *tree-rank-height*:
tree-invar (*Node e a r ts*) \implies *height-tree* (*Node e a r ts*) = *r*
 $\langle \text{proof} \rangle$

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

theorem *tree-height-estimate-upper:*

$tree\text{-}invar\ t \implies$
 $size\ (tree\text{-}to\text{-}multiset\ t) \leq (2::nat)^\wedge(Suc\ (height\text{-}tree\ t)) - 1$
 $\langle proof \rangle$

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

theorem *tree-height-estimate-lower:*

$tree\text{-}invar\ t \implies size\ (tree\text{-}to\text{-}multiset\ t) \geq (2::nat)^\wedge(height\text{-}tree\ t)$
 $\langle proof \rangle$

lemma *size-mset-tree-upper: tree-invar t \implies*

$size\ (tree\text{-}to\text{-}multiset\ t) \leq (2::nat)^\wedge(Suc\ (rank\ t)) - (1::nat)$
 $\langle proof \rangle$

lemma *size-mset-tree-lower: tree-invar t \implies*

$size\ (tree\text{-}to\text{-}multiset\ t) \geq (2::nat)^\wedge(rank\ t)$
 $\langle proof \rangle$

lemma *invar-butlast: invar (bq @ [t]) \implies invar bq*

$\langle proof \rangle$

lemma *invar-last-max:*

$invar\ ((b\#\#b'\#\#bq)\ @\ [m]) \implies \forall\ t \in\ set\ (b'\#\#bq). rank\ t < rank\ m$
 $\langle proof \rangle$

lemma *invar-last-max': invar ((b#b'#bq) @ [m]) \implies rank b \leq rank b'*

$\langle proof \rangle$

lemma *invar-length: invar bq \implies length bq \leq Suc (Suc (rank (last bq)))*

$\langle proof \rangle$

lemma *size-queue-sum-list:*

$size\ (queue\text{-}to\text{-}multiset\ bq) = sum\text{-}list\ (map\ (size \circ tree\text{-}to\text{-}multiset)\ bq)$
 $\langle proof \rangle$

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

theorem *queue-length-estimate-lower:*

$invar\ bq \implies (size\ (queue\text{-}to\text{-}multiset\ bq)) \geq 2^\wedge(length\ bq - 1) - 1$
 $\langle proof \rangle$

2.2 Operations

2.2.1 Empty Tree

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

$\langle proof \rangle$

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  $\Rightarrow$  ('e, 'a) SkewBinomialQueue  $\Rightarrow$ 
('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  $\Rightarrow$  'a::linorder  $\Rightarrow$  ('e, 'a) SkewBinomialQueue  $\Rightarrow$ 
('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  $\neq$  rank t'
      then (Node e a 0 []) # t # t' # bq
      else (skewlink e a t') # bq)
```

lemma ins-mset:

```
[[tree-invar t; queue-invar q]]  $\Longrightarrow$ 
  queue-to-multiset (ins t q) = tree-to-multiset t + queue-to-multiset q
<proof>
```

lemma insert-mset: queue-invar q \Longrightarrow

```
  queue-to-multiset (insert e a q) =
  queue-to-multiset q + {# (e,a) #}
<proof>
```

lemma ins-queue-invar: [[tree-invar t; queue-invar q]] \Longrightarrow queue-invar (ins t q)

<proof>

lemma insert-queue-invar: queue-invar q \Longrightarrow queue-invar (insert e a q)

<proof>

lemma rank-ins2:

```
rank-invar bq  $\Longrightarrow$ 
  rank t  $\leq$  rank (hd (ins t bq))
   $\vee$  (rank (hd (ins t bq)) = rank (hd bq)  $\wedge$  bq  $\neq$  [])
<proof>
```

lemma insert-rank-invar: rank-skew-invar q \Longrightarrow rank-skew-invar (insert e a q)

<proof>

lemma *insert-invar*: $\text{invar } q \implies \text{invar } (\text{insert } e \ a \ q)$
 ⟨proof⟩

theorem *insert-correct*:

assumes I : $\text{invar } q$

shows

$\text{invar } (\text{insert } e \ a \ q)$

$\text{queue-to-multiset } (\text{insert } e \ a \ q) = \text{queue-to-multiset } q + \{\# (e, a) \#\}$

⟨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*

$:: ('e, 'a::\text{linorder}) \text{SkewBinomialQueue} \Rightarrow ('e, 'a) \text{SkewBinomialQueue}$

where

$\text{uniqify } [] = [] \mid$

$\text{uniqify } (t\#bq) = \text{ins } t \ bq$

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

fun *meldUniq*

$:: ('e, 'a::\text{linorder}) \text{SkewBinomialQueue} \Rightarrow ('e, 'a) \text{SkewBinomialQueue} \Rightarrow$

$(('e, 'a) \text{SkewBinomialQueue})$ **where**

$\text{meldUniq } [] \ bq = bq \mid$

$\text{meldUniq } bq \ [] = bq \mid$

$\text{meldUniq } (t1\#bq1) \ (t2\#bq2) = (\text{if } \text{rank } t1 < \text{rank } t2$

$\text{then } t1 \# (\text{meldUniq } bq1 \ (t2\#bq2))$

$\text{else } (\text{if } \text{rank } t2 < \text{rank } t1$

$\text{then } t2 \# (\text{meldUniq } (t1\#bq1) \ bq2)$

$\text{else } \text{ins } (\text{link } t1 \ t2) \ (\text{meldUniq } bq1 \ bq2)))$

Meld two queues using above functions.

definition *meld*

$:: ('e, 'a::\text{linorder}) \text{SkewBinomialQueue} \Rightarrow ('e, 'a) \text{SkewBinomialQueue} \Rightarrow$

$(('e, 'a) \text{SkewBinomialQueue})$ **where**

$\text{meld } bq1 \ bq2 = \text{meldUniq } (\text{uniqify } bq1) \ (\text{uniqify } bq2)$

lemma *invar-uniqify*: $\text{queue-invar } q \implies \text{queue-invar } (\text{uniqify } q)$

⟨proof⟩

lemma *invar-meldUniq*: $\llbracket \text{queue-invar } q; \text{queue-invar } q' \rrbracket \implies \text{queue-invar } (\text{meldUniq } q \ q')$

⟨proof⟩

lemma *meld-queue-invar*:

assumes *queue-invar* q
and *queue-invar* q'
shows *queue-invar* ($\text{meld } q \ q'$)
 $\langle \text{proof} \rangle$

lemma *uniqify-mset*: *queue-invar* $q \implies \text{queue-to-multiset } q = \text{queue-to-multiset } (\text{uniqify } q)$
 $\langle \text{proof} \rangle$

lemma *meldUniq-mset*: $\llbracket \text{queue-invar } q; \text{queue-invar } q' \rrbracket \implies$
 $\text{queue-to-multiset } (\text{meldUniq } q \ q') =$
 $\text{queue-to-multiset } q + \text{queue-to-multiset } q'$
 $\langle \text{proof} \rangle$

lemma *meld-mset*:
 $\llbracket \text{queue-invar } q; \text{queue-invar } q' \rrbracket \implies$
 $\text{queue-to-multiset } (\text{meld } q \ q') = \text{queue-to-multiset } q + \text{queue-to-multiset } q'$
 $\langle \text{proof} \rangle$

Ins operation satisfies rank invariant, see binomial queues

lemma *rank-ins*: *rank-invar* $bq \implies \text{rank-invar } (\text{ins } t \ bq)$
 $\langle \text{proof} \rangle$

lemma *rank-uniqify*:
assumes *rank-skew-invar* q
shows *rank-invar* ($\text{uniqify } q$)
 $\langle \text{proof} \rangle$

lemma *rank-ins-min*: *rank-invar* $bq \implies \text{rank } (\text{hd } (\text{ins } t \ bq)) \geq \min (\text{rank } t) (\text{rank } (\text{hd } bq))$
 $\langle \text{proof} \rangle$

lemma *rank-invar-not-empty-hd*: $\llbracket \text{rank-invar } (t \ \# \ bq); bq \neq [] \rrbracket \implies \text{rank } t < \text{rank } (\text{hd } bq)$
 $\langle \text{proof} \rangle$

lemma *rank-invar-meldUniq-strong*:
 $\llbracket \text{rank-invar } bq1; \text{rank-invar } bq2 \rrbracket \implies$
 $\text{rank-invar } (\text{meldUniq } bq1 \ bq2)$
 $\wedge \text{rank } (\text{hd } (\text{meldUniq } bq1 \ bq2)) \geq \min (\text{rank } (\text{hd } bq1)) (\text{rank } (\text{hd } bq2))$
 $\langle \text{proof} \rangle$

lemma *rank-meldUniq*:
 $\llbracket \text{rank-invar } bq1; \text{rank-invar } bq2 \rrbracket \implies \text{rank-invar } (\text{meldUniq } bq1 \ bq2)$
 $\langle \text{proof} \rangle$

lemma *rank-meld*:
 $\llbracket \text{rank-skew-invar } q1; \text{rank-skew-invar } q2 \rrbracket \implies \text{rank-skew-invar } (\text{meld } q1 \ q2)$

<proof>

theorem *meld-invar*:

$\llbracket \text{invar } bq1; \text{invar } bq2 \rrbracket$
 $\implies \text{invar } (\text{meld } bq1 \ bq2)$
<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* \Rightarrow

('e, 'a) *SkewBinomialTree* **where**

getMinTree [t] = t |

getMinTree (t#bq) =

(if *prio* t \leq *prio* (*getMinTree* bq)

then t

else (*getMinTree* bq))

Find the minimal Element in the queue.

definition *findMin* :: ('e, 'a::linorder) *SkewBinomialQueue* \Rightarrow ('e \times 'a) **where**

findMin bq = (let *min* = *getMinTree* bq in (val *min*, *prio* *min*))

lemma *mintree-exists*: (bq \neq []) = (*getMinTree* bq \in set bq)

<proof>

lemma *treehead-in-multiset*:

t \in set bq \implies (val t, *prio* t) \in # (*queue-to-multiset* bq)

<proof>

lemma *heap-ordered-single*:

heap-ordered t = ($\forall x \in$ set-mset (*tree-to-multiset* t). *prio* t \leq *snd* x)

<proof>

lemma *getMinTree-cons*:

prio (*getMinTree* (y # x # xs)) \leq *prio* (*getMinTree* (x # xs))

<proof>

lemma *getMinTree-min-tree*: t \in set bq \implies *prio* (*getMinTree* bq) \leq *prio* t

<proof>

lemma *getMinTree-min-prio*:

assumes *queue-invar* *bq*
and $y \in \text{set-mset } (\text{queue-to-multiset } bq)$
shows $\text{prio } (\text{getMinTree } bq) \leq \text{snd } y$
 $\langle \text{proof} \rangle$

lemma *findMin-mset*:
assumes *I*: *queue-invar* *q*
assumes *NE*: $q \neq \text{Nil}$
shows $\text{findMin } q \in \# \text{ queue-to-multiset } q$
 $\forall y \in \text{set-mset } (\text{queue-to-multiset } q). \text{snd } (\text{findMin } q) \leq \text{snd } y$
 $\langle \text{proof} \rangle$

theorem *findMin-correct*:
assumes *I*: *invar* *q*
assumes *NE*: $q \neq \text{Nil}$
shows $\text{findMin } q \in \# \text{ queue-to-multiset } q$
 $\forall y \in \text{set-mset } (\text{queue-to-multiset } q). \text{snd } (\text{findMin } q) \leq \text{snd } y$
 $\langle \text{proof} \rangle$

2.2.5 Delete Minimal Element

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

fun *insertList* ::
 $(\text{'e}, \text{'a}::\text{linorder}) \text{ SkewBinomialQueue} \Rightarrow (\text{'e}, \text{'a}) \text{ SkewBinomialQueue} \Rightarrow$
 $(\text{'e}, \text{'a}) \text{ SkewBinomialQueue}$ **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* :: $\text{'a} \Rightarrow (\text{'e}, \text{'a}::\text{linorder}) \text{ SkewBinomialQueue} \Rightarrow$
 $(\text{'e}, \text{'a}) \text{ SkewBinomialQueue}$ **where**
remove1Prio *a* [] = [] |
remove1Prio *a* (*t*#*bq*) =
 $(\text{if } (\text{prio } t) = a \text{ then } bq \text{ else } t \# (\text{remove1Prio } a \text{ } bq))$

lemma *remove1Prio-remove1[simp]*:
 $\text{remove1Prio } (\text{prio } (\text{getMinTree } bq)) \text{ } bq = \text{remove1 } (\text{getMinTree } bq) \text{ } bq$
 $\langle \text{proof} \rangle$

Return the queue without the minimal element found by *findMin*

definition *deleteMin* :: $(\text{'e}, \text{'a}::\text{linorder}) \text{ SkewBinomialQueue} \Rightarrow$
 $(\text{'e}, \text{'a}) \text{ SkewBinomialQueue}$ **where**
deleteMin *bq* = $(\text{let } \text{min} = \text{getMinTree } bq \text{ in } \text{insertList}$
 $(\text{filter } (\lambda t. \text{rank } t = 0) (\text{children } \text{min}))$
 $(\text{meld } (\text{rev } (\text{filter } (\lambda t. \text{rank } t > 0) (\text{children } \text{min})))$
 $(\text{remove1Prio } (\text{prio } \text{min}) \text{ } bq))$

lemma *invar-rev[simp]*: $\text{queue-invar } (\text{rev } q) \longleftrightarrow \text{queue-invar } q$

$\langle \text{proof} \rangle$

lemma *invar-remove1*: $\text{queue-invar } q \implies \text{queue-invar } (\text{remove1 } t \ q)$
 $\langle \text{proof} \rangle$

lemma *mset-rev*: $\text{queue-to-multiset } (\text{rev } q) = \text{queue-to-multiset } q$
 $\langle \text{proof} \rangle$

lemma *in-set-subset*: $t \in \text{set } q \implies \text{tree-to-multiset } t \subseteq\# \text{queue-to-multiset } q$
 $\langle \text{proof} \rangle$

lemma *mset-remove1*: $t \in \text{set } q \implies$
 $\text{queue-to-multiset } (\text{remove1 } t \ q) =$
 $\text{queue-to-multiset } q - \text{tree-to-multiset } t$
 $\langle \text{proof} \rangle$

lemma *invar-children'*:
assumes *tree-invar* t
shows $\text{queue-invar } (\text{children } t)$
 $\langle \text{proof} \rangle$

lemma *invar-filter*: $\text{queue-invar } q \implies \text{queue-invar } (\text{filter } f \ q)$
 $\langle \text{proof} \rangle$

lemma *insertList-queue-invar*: $\text{queue-invar } q \implies \text{queue-invar } (\text{insertList } ts \ q)$
 $\langle \text{proof} \rangle$

lemma *deleteMin-queue-invar*:
 $\llbracket \text{queue-invar } q; \text{queue-to-multiset } q \neq \{\#\} \rrbracket \implies$
 $\text{queue-invar } (\text{deleteMin } q)$
 $\langle \text{proof} \rangle$

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

lemma *mset-insertList*:
 $\llbracket \forall t \in \text{set } ts. \text{rank } t = 0 \wedge \text{children } t = [] ; \text{queue-invar } q \rrbracket \implies$
 $\text{queue-to-multiset } (\text{insertList } ts \ q) =$
 $\text{queue-to-multiset } ts + \text{queue-to-multiset } q$
 $\langle \text{proof} \rangle$

lemma *mset-filter*: $(\text{queue-to-multiset } [t \leftarrow q . \text{rank } t = 0]) +$
 $\text{queue-to-multiset } [t \leftarrow q . 0 < \text{rank } t] =$
 $\text{queue-to-multiset } q$
 $\langle \text{proof} \rangle$

lemma *deleteMin-mset*:
assumes $\text{queue-invar } q$

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* $q \implies$ *rank-skew-invar* (*insertList* ts q)
 <proof>

lemma *insertList-invar*: *invar* $q \implies$ *invar* (*insertList* ts q)
 <proof>

lemma *children-rank-less*:
assumes *tree-invar* t
shows $\forall t' \in \text{set } (\text{children } t). \text{rank } t' < \text{rank } t$
 <proof>

lemma *strong-rev-children*:
assumes *tree-invar* t
shows *invar* (*rev* [$t \leftarrow \text{children } t$]. $0 < \text{rank } t$)
 <proof>

lemma *first-less*: *rank-invar* ($t \# bq$) $\implies \forall t' \in \text{set } bq. \text{rank } t < \text{rank } t'$
 <proof>

lemma *first-less-eq*:
rank-skew-invar ($t \# bq$) $\implies \forall t' \in \text{set } bq. \text{rank } t \leq \text{rank } t'$
 <proof>

lemma *remove1-tail-invar*: *tail-invar* $bq \implies$ *tail-invar* (*remove1* t bq)
 <proof>

lemma *invar-cons-down*: *invar* ($t \# bq$) \implies *invar* bq
 <proof>

lemma *remove1-invar*: *invar* $bq \implies$ *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 : $q \neq \text{Nil}$
shows *invar* (*deleteMin* q)
and *queue-to-multiset* (*deleteMin* q) = *queue-to-multiset* $q - \{\# \text{findMin } q \#$
 $\#\}$
 <proof>

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

end

interpretation *SkewBinomialHeapStruc*: *SkewBinomialHeapStruc-loc* $\langle proof \rangle$

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 $x \in \# \text{image-mset } f \ M$

obtains y **where** $y \in \# M$ $x = f \ y$

$\langle proof \rangle$

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*:

$(\forall e \ a. (e, a) \in \# M \longrightarrow a = f \ e) \implies$

$\text{image-mset } \text{fst} \ (M - \{\#(e, f \ e)\}) = \text{image-mset } \text{fst} \ M - \{\#e\}$

$\langle proof \rangle$

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 their 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 \Rightarrow '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
 \Rightarrow ('e, 'a) BsSkewBinomialTree \Rightarrow
('e, 'a) BsSkewBinomialTree **where**
link (BsNode e1 r1 ts1) (BsNode e2 r2 ts2) =
(if *eprio* e1 \leq *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 \Rightarrow ('e, 'a) BsSkewBinomialTree \Rightarrow
('e, 'a) BsSkewBinomialTree \Rightarrow ('e, 'a) BsSkewBinomialTree **where**
skewlink e t t' = (if *eprio* e \leq (*prio* t) \wedge *eprio* e \leq (*prio* t')
then (BsNode e (Suc (rank t)) [t, t'])
else (if (*prio* t) \leq (*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 \Rightarrow
('e, 'a) BsSkewBinomialQueue \Rightarrow
('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* \Rightarrow

(*'e, 'a*) *BsSkewBinomialQueue* \Rightarrow
(*'e, 'a*) *BsSkewBinomialQueue* **where**
insert *e* [] = [*BsNode e 0* []] |
insert *e* [*t*] = [*BsNode e 0* [],*t*] |
insert *e* (*t* # *t'* # *bq*) =
 (*if* *rank t* \neq *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* \Rightarrow (*'e, 'a*) *BsSkewBinomialQueue*
where
uniqify [] = [] |
uniqify (*t*#*bq*) = *ins t bq*

fun *meldUniq*

:: (*'e, 'a::linorder*) *BsSkewBinomialQueue* \Rightarrow (*'e, 'a*) *BsSkewBinomialQueue* \Rightarrow
(*'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* \Rightarrow (*'e, 'a*) *BsSkewBinomialQueue* \Rightarrow
(*'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* ::

$(e, 'a::linorder)\ BsSkewBinomialQueue \Rightarrow (e, 'a)\ BsSkewBinomialQueue \Rightarrow$

$(e, 'a)\ BsSkewBinomialQueue$ **where**

$insertList []\ tbq = tbq \mid$

$insertList (t\#\!bq)\ tbq = insertList\ bq (insert (val\ t)\ tbq)$

fun *remove1Prio* :: $'a \Rightarrow (e, 'a::linorder)\ BsSkewBinomialQueue \Rightarrow$

$(e, 'a)\ BsSkewBinomialQueue$ **where**

$remove1Prio\ a [] = [] \mid$

$remove1Prio\ a (t\#\!bq) =$

$(if (prio\ t) = a\ then\ bq\ else\ t\ \#\ (remove1Prio\ a\ bq))$

fun *getMinTree* :: $(e, 'a::linorder)\ BsSkewBinomialQueue \Rightarrow$

$(e, 'a)\ BsSkewBinomialTree$ **where**

$getMinTree [t] = t \mid$

$getMinTree (t\#\!bq) =$

$(if\ prio\ t \leq\ prio (getMinTree\ bq)$

$\ then\ t$

$\ else\ (getMinTree\ bq))$

definition *findMin*

$:: (e, 'a::linorder)\ BsSkewBinomialQueue \Rightarrow (e, 'a)\ BsSkewElem$ **where**

$findMin\ bq = val (getMinTree\ bq)$

definition *deleteMin* :: $(e, 'a::linorder)\ BsSkewBinomialQueue \Rightarrow$

$(e, 'a)\ BsSkewBinomialQueue$ **where**

$deleteMin\ bq = (let\ min = getMinTree\ bq\ in\ insertList$

$(filter (\lambda\ t.\ rank\ t = 0) (children\ min))$

$(meld (rev (filter (\lambda\ t.\ rank\ t > 0) (children\ min))))$

$(remove1Prio (prio\ min)\ bq))$

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*:

$q \neq [] \implies bsmap (getMinTree q) = SkewBinomialHeapStruc.getMinTree (bsmap q)$
<proof>

lemma *findMin-xlate*:

$q \neq [] \implies findMin q = fst (SkewBinomialHeapStruc.findMin (bsmap q))$
<proof>

lemma *findMin-xlate-aux*:

$q \neq [] \implies (findMin q, eprio (findMin q)) =$
 $(SkewBinomialHeapStruc.findMin (bsmap q))$
<proof>

lemma *bsmap-filter-xlate*:

$bsmap [x \leftarrow l . P (bsmap x)] = [x \leftarrow bsmap l . P x]$
<proof>

lemma *bsmap-rev-xlate*:

$bsmap (rev q) = rev (bsmap q)$
<proof>

lemma *deleteMin-xlate*:

$q \neq [] \implies bsmap (deleteMin q) = SkewBinomialHeapStruc.deleteMin (bsmap q)$
<proof>

lemma *deleteMin-correct-aux*:

assumes *I*: *invar q*

assumes *NE*: $q \neq []$

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*:

$(e,a) \in \#SkewBinomialHeapStruct.tree\text{-}to\text{-}multiset (bsmapt\ t) \implies a = eprio\ e$
 $(e,a) \in \#SkewBinomialHeapStruct.queue\text{-}to\text{-}multiset (bimap\ q) \implies a = eprio\ e$
thm *SkewBinomialHeapStruct.tree-to-multiset-queue-to-multiset.induct*
 <proof>

lemma *bsmap-fs-depD*:
 $(e,a) \in \#SkewBinomialHeapStruct.tree\text{-}to\text{-}multiset (bsmapt\ t)$
 $\implies e \in \#tree\text{-}to\text{-}multiset\ t \wedge a = eprio\ e$
 $(e,a) \in \#SkewBinomialHeapStruct.queue\text{-}to\text{-}multiset (bimap\ q)$
 $\implies e \in \#queue\text{-}to\text{-}multiset\ q \wedge a = eprio\ e$
 <proof>

lemma *findMin-correct-aux*:
assumes *I*: *invar* *q*
assumes *NE*: $q \neq []$
shows $(findMin\ q, eprio (findMin\ q)) \in \#queue\text{-}to\text{-}multiset\text{-}aux\ q$
 $\forall y \in set\text{-}mset (queue\text{-}to\text{-}multiset\text{-}aux\ q). snd (findMin\ q, eprio (findMin\ q)) \leq snd\ y$
 <proof>

lemma *findMin-correct*:
assumes *I*: *invar* *q*
and *NE*: $q \neq []$
shows $findMin\ q \in \#queue\text{-}to\text{-}multiset\ q$
and $\forall y \in set\text{-}mset (queue\text{-}to\text{-}multiset\ q). eprio (findMin\ q) \leq eprio\ y$
 <proof>

lemma *deleteMin-correct*:
assumes *I*: *invar* *q*
assumes *NE*: $q \neq []$
shows
 $invar (deleteMin\ q)$
 $queue\text{-}to\text{-}multiset (deleteMin\ q) = queue\text{-}to\text{-}multiset\ q - \{\# findMin\ q \#\}$
 <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'*:: $(e,'a::linorder)\ BsSkewElem \Rightarrow$

```

('e,'a) BsSkewElem  $\Rightarrow$  ('e,'a) BsSkewElem
where meld' (Element e1 a1 q1) (Element e2 a2 q2) =
  (if a1  $\leq$  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  $\Rightarrow$ 
        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  $\in$  #tree-to-multiset t  $\implies$  level x < Suc (tree-level t)
  x  $\in$  #queue-to-multiset q  $\implies$  level x < Suc (queue-level q)
  <proof>

```

```

lemma level-measure:
  x  $\in$  set-mset (queue-to-multiset q)  $\implies$  (x,(Element e a q))  $\in$  measure level
  x  $\in$  # (queue-to-multiset q)  $\implies$  (x,(Element e a q))  $\in$  measure level
  <proof>

```

Invariant for elements

```

function elem-invar where
  elem-invar (Element e a q)  $\longleftrightarrow$ 
  ( $\forall$  x. x  $\in$  # (queue-to-multiset q)  $\longrightarrow$  a  $\leq$  eprio x  $\wedge$  elem-invar x)  $\wedge$ 
  invar q
  <proof>
termination
  <proof>

```

Abstraction to multisets

```

function elem-to-mset where
  elem-to-mset (Element e a q) = {# (e,a) #}
  + Union-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*:

$\llbracket \text{elem-invar } x; y \in \# \text{elem-to-mset } x \rrbracket \implies \text{snd } (\text{findMin}' x) \leq \text{snd } y$

<proof>

lemma *findMin-correct'*:

assumes *I*: *elem-invar x*

shows

findMin' x ∈# elem-to-mset x

$\forall y \in \text{set-mset } (\text{elem-to-mset } x). \text{snd } (\text{findMin}' x) \leq \text{snd } y$

<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* \equiv *Inl ()*

primrec *bs-findMin* **where**

bs-findMin (Inr x) = findMin' x

fun *bs-meld*

$:: ('e, 'a::\text{linorder}) \text{BsSkewHeap} \Rightarrow ('e, 'a) \text{BsSkewHeap} \Rightarrow ('e, 'a) \text{BsSkewHeap}$
where
 $\text{bs-meld } (\text{Inl } -) x = x \mid$
 $\text{bs-meld } x (\text{Inl } -) = x \mid$
 $\text{bs-meld } (\text{Inr } x) (\text{Inr } x') = \text{Inr } (\text{meld}' x x')$
lemma [simp]: $\text{bs-meld } x (\text{Inl } u) = x$
 ⟨proof⟩

primrec *bs-insert*
 $:: 'e \Rightarrow ('a::\text{linorder}) \Rightarrow ('e, 'a) \text{BsSkewHeap} \Rightarrow ('e, 'a) \text{BsSkewHeap}$
where
 $\text{bs-insert } e a (\text{Inl } -) = \text{Inr } (\text{Element } e a []) \mid$
 $\text{bs-insert } e a (\text{Inr } x) = \text{Inr } (\text{insert}' e a x)$

fun *bs-deleteMin*
 $:: ('e, 'a::\text{linorder}) \text{BsSkewHeap} \Rightarrow ('e, 'a) \text{BsSkewHeap}$
where
 $\text{bs-deleteMin } (\text{Inr } (\text{Element } e a [])) = \text{Inl } () \mid$
 $\text{bs-deleteMin } (\text{Inr } (\text{Element } e a q)) = \text{Inr } (\text{deleteMin}' (\text{Element } e a q))$

primrec *bs-invar* $:: ('e, 'a::\text{linorder}) \text{BsSkewHeap} \Rightarrow \text{bool}$
where
 $\text{bs-invar } (\text{Inl } -) \longleftrightarrow \text{True} \mid$
 $\text{bs-invar } (\text{Inr } x) \longleftrightarrow \text{elem-invar } x$

lemma [simp]: $\text{bs-invar } \text{bs-empty} \langle \text{proof} \rangle$

primrec *bs-to-mset* $:: ('e, 'a::\text{linorder}) \text{BsSkewHeap} \Rightarrow ('e \times 'a) \text{multiset}$
where
 $\text{bs-to-mset } (\text{Inl } -) = \{\#\} \mid$
 $\text{bs-to-mset } (\text{Inr } x) = \text{elem-to-mset } x$

theorem *bs-empty-correct*: $h = \text{bs-empty} \longleftrightarrow \text{bs-to-mset } h = \{\#\}$
 ⟨proof⟩

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

theorem *bs-findMin-correct*:
assumes *I*: $\text{bs-invar } h$
assumes *NE*: $h \neq \text{bs-empty}$
shows $\text{bs-findMin } h \in \# \text{bs-to-mset } h$
 $\forall y \in \text{set-mset } (\text{bs-to-mset } h). \text{snd } (\text{bs-findMin } h) \leq \text{snd } y$
 ⟨proof⟩

theorem *bs-insert-correct*:
assumes *I*: $\text{bs-invar } h$
shows

$bs\text{-invar } (bs\text{-insert } e \ a \ h)$
 $bs\text{-to-mset } (bs\text{-insert } e \ a \ h) = \{\#(e,a)\# \} + bs\text{-to-mset } h$
 $\langle proof \rangle$

theorem *bs-meld-correct*:

assumes *I*: $bs\text{-invar } h \ bs\text{-invar } h'$

shows

$bs\text{-invar } (bs\text{-meld } h \ h')$

$bs\text{-to-mset } (bs\text{-meld } h \ h') = bs\text{-to-mset } h + bs\text{-to-mset } h'$

$\langle proof \rangle$

theorem *bs-deleteMin-correct*:

assumes *I*: $bs\text{-invar } h$

assumes *NE*: $h \neq bs\text{-empty}$

shows

$bs\text{-invar } (bs\text{-deleteMin } h)$

$bs\text{-to-mset } (bs\text{-deleteMin } h) = bs\text{-to-mset } h - \{\#bs\text{-findMin } h\# \}$

$\langle proof \rangle$

end

interpretation *BsSkewBinomialHeapStruc*: *Bootstrapped* $\langle proof \rangle$

2.4 Hiding the Invariant

2.4.1 Datatype

typedef (**overloaded**) (e, a) *SkewBinomialHeap* =
 $\{q :: (e, a :: linorder) \ BsSkewBinomialHeapStruc.BsSkewHeap. \ BsSkewBinomialHeapStruc.bs\text{-invar } q \}$
 $\langle proof \rangle$

lemma *Rep-SkewBinomialHeap-invar[simp]*:

$BsSkewBinomialHeapStruc.bs\text{-invar } (Rep\text{-SkewBinomialHeap } x)$

$\langle proof \rangle$

lemma *[simp]*:

$BsSkewBinomialHeapStruc.bs\text{-invar } q$

$\implies Rep\text{-SkewBinomialHeap } (Abs\text{-SkewBinomialHeap } q) = q$

$\langle proof \rangle$

lemma *[simp, code abstype]*: $Abs\text{-SkewBinomialHeap } (Rep\text{-SkewBinomialHeap } q)$

$= q$

$\langle proof \rangle$

locale *SkewBinomialHeap-loc*

begin

2.4.2 Operations

definition [code]:

to-mset t

$==$ *BsSkewBinomialHeapStruct.bs-to-mset (Rep-SkewBinomialHeap t)*

definition *empty* **where**

empty == Abs-SkewBinomialHeap BsSkewBinomialHeapStruct.bs-empty

lemma [code abstract, simp]:

Rep-SkewBinomialHeap empty = BsSkewBinomialHeapStruct.bs-empty

<proof>

definition [code]:

isEmpty q == Rep-SkewBinomialHeap q = BsSkewBinomialHeapStruct.bs-empty

lemma *empty-rep*:

q=empty \longleftrightarrow Rep-SkewBinomialHeap q = BsSkewBinomialHeapStruct.bs-empty

<proof>

lemma *isEmpty-correct*: *isEmpty q \longleftrightarrow q=empty*

<proof>

definition

insert

$:: 'e \Rightarrow ('a::linorder) \Rightarrow ('e,'a) SkewBinomialHeap$

$\Rightarrow ('e,'a) SkewBinomialHeap$

where *insert e a q ==*

Abs-SkewBinomialHeap (

BsSkewBinomialHeapStruct.bs-insert e a (Rep-SkewBinomialHeap q))

lemma [code abstract]:

Rep-SkewBinomialHeap (insert e a q)

$=$ *BsSkewBinomialHeapStruct.bs-insert e a (Rep-SkewBinomialHeap q)*

<proof>

definition [code]: *findMin q*

$==$ *BsSkewBinomialHeapStruct.bs-findMin (Rep-SkewBinomialHeap q)*

definition *deleteMin q ==*

if q=empty then empty

else Abs-SkewBinomialHeap (

BsSkewBinomialHeapStruct.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 - \Rightarrow BsSkewBinomialHeapStruct.bs-empty

|

- \Rightarrow BsSkewBinomialHeapStruct.bs-deleteMin (Rep-SkewBinomialHeap q))

<proof>

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

2.4.3 Correctness

lemma *empty-correct*: *to-mset* $q = \{\#\}$ \longleftrightarrow $q = \text{empty}$
⟨*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 $q \neq \text{empty}$
shows
findMin $q \in \#\ \text{to-mset}\ q$
 $\forall y \in \text{set-mset}\ (\text{to-mset}\ q). \text{snd}\ (\text{findMin}\ q) \leq \text{snd}\ y$
⟨*proof*⟩

lemma *deleteMin-correct*:
assumes $q \neq \text{empty}$
shows *to-mset* (*deleteMin* q) = *to-mset* $q - \{\#\ \text{findMin}\ q\ \#\}$
⟨*proof*⟩

lemma *meld-correct*:
shows *to-mset* (*meld* $q\ q'$) = *to-mset* $q + \text{to-mset}\ q'$
⟨*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::('a, 'b) SkewBinomialHeap ⇒ ('a × 'b) multiset
Abstraction to multiset.

SkewBinomialHeap.empty::('a, 'b) SkewBinomialHeap
The empty heap. ($O(1)$)

Spec *SkewBinomialHeap.empty-correct*:

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

SkewBinomialHeap.isEmpty::('a, 'b) SkewBinomialHeap ⇒ bool
Checks whether heap is empty. Mainly used to work around code-generation issues. ($O(1)$)

Spec *SkewBinomialHeap.isEmpty-correct*:

$\text{SkewBinomialHeap.isEmpty } q = (q = \text{SkewBinomialHeap.empty})$

SkewBinomialHeap.insert

$\text{SkewBinomialHeap.insert}::'a \Rightarrow 'b \Rightarrow ('a, 'b) \text{ SkewBinomialHeap}$
 $\Rightarrow ('a, 'b) \text{ SkewBinomialHeap}$

Inserts element ($O(1)$)

Spec *SkewBinomialHeap.insert-correct*:

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

SkewBinomialHeap.findMin::('a, 'b) SkewBinomialHeap ⇒ 'a × 'b
Returns a minimal element ($O(1)$)

Spec *SkewBinomialHeap.findMin-correct*:

$q \neq \text{SkewBinomialHeap.empty} \implies$
 $\text{SkewBinomialHeap.findMin } q \in \# \text{ SkewBinomialHeap.to-mset } q$
 $q \neq \text{SkewBinomialHeap.empty} \implies$
 $\forall y \in \# \text{ SkewBinomialHeap.to-mset } q. \text{snd } (\text{SkewBinomialHeap.findMin } q) \leq \text{snd } y$

SkewBinomialHeap.deleteMin

$\text{SkewBinomialHeap.deleteMin}::('a, 'b) \text{ SkewBinomialHeap}$
 $\Rightarrow ('a, 'b) \text{ SkewBinomialHeap}$

Deletes *the* element that is returned by *find_min*. $O(\log(n))$

Spec *SkewBinomialHeap.deleteMin-correct*:

$q \neq \text{SkewBinomialHeap.empty} \implies$
 $\text{SkewBinomialHeap.to-mset } (\text{SkewBinomialHeap.deleteMin } q) =$
 $\text{SkewBinomialHeap.to-mset } q - \{\# \text{ SkewBinomialHeap.findMin } q \# \}$

SkewBinomialHeap.meld

SkewBinomialHeap.meld::('a, 'b) *SkewBinomialHeap*
 \Rightarrow ('a, 'b) *SkewBinomialHeap*
 \Rightarrow ('a, 'b) *SkewBinomialHeap*

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.