Title: Lammich: FDS Tutorial (22.06.2018)
Date: Fri Jun 22 12:18:58 CEST 2018
Duration: 92:56 min
Pages: 106
fun lh_insert :: "'a ord => 'a heap => 'a heap" where
  "lh_insert _ _ = undefined"

lemma most_lh_insert: "met_heap (lh_insert x t) = met_heap t + (# x #)"
  sorry

lemma "heap t ⇒ heap (lh_insert x t)"
  sorry

fun t_lh_insert :: "'a ord => 'a heap => nat"
- Show that your function is correct

```haskell
fun lh_insert :: 'a ord => 'a heap => 'a heap
where
  "lh_insert x o = node x o" = undefined

lemma meet_lh_insert: "meet_tree (lh_insert x t) = meet_tree t + (x \#)"
  sorry

lemma "heap t => heap (lh_insert x t)"
  sorry

lemma "tree t => tree (lh_insert x t)"
  sorry
```

- Show that your function is correct

```haskell
fun lh_insert :: 'a ord => 'a heap => 'a heap
where
  "lh_insert x o = node x o" = undefined

lemma meet_lh_insert: "meet_tree (lh_insert x t) = meet_tree t + (x \#)"
  sorry

lemma "heap t => heap (lh_insert x t)"
  sorry

lemma "tree t => tree (lh_insert x t)"
  sorry
```
fun lh_insert :: "'a::ord ⇒ 'aheap ⇒ 'aheap"
where
  "lh_insert x (l, h) = (if x < l then (x, h) else l, h)"
lemma lh_insert: "set_tree (lh_insert x t) = set_tree t + {x}
apply (induction t arbitrary: x)
apply (auto)"

proof (prove)
goal (2 subgoals):
1. \x. set_tree (lh_insert x t1) = add_set x (set_tree t1);
   \x. set_tree (lh_insert x t2) = add_set x (set_tree t2); x < x
   → set_tree (node t1 x (lh_insert x t2)) = add_set x (add_set x (set_tree t1 + set_tree t2))
2. \x. x < t1 ∧ t2 x

fun lh_insert :: "'a::ord ⇒ 'aheap ⇒ 'aheap"
where
  "lh_insert x (l, h) = (if x < l then (x, h) else l, h)"
lemma lh_insert: "set_tree (lh_insert x t) = set_tree t + {x}
apply (induction t arbitrary: x)
apply (auto)"

proof (prove)
goal (2 subgoals):
1. \x. set_tree (lh_insert x t) = add_set x (set_tree t);
   \x. set_tree (lh_insert x t) = add_set x (set_tree t); x < x
   → set_tree (node t1 x (lh_insert x t)) = add_set x (add_set x (set_tree t1 + set_tree t2))
2. \x. x < t1 ∧ x < t2

fun lh_insert :: "'a::ord ⇒ 'aheap ⇒ 'aheap"
where
  "lh_insert x (l, h) = (if x < l then (x, h) else l, h)"
lemma lh_insert: "set_tree (lh_insert x t) = set_tree t + {x}
apply (induction t arbitrary: x)
apply (auto)"

proof (prove)
goal (1 subgoal):
1. heap t = heap (lh_insert x t)
Lemma meet_LH_insert: "meet_tree (LH_insert x t) = meet_tree t + (x x #)"
  by (induction t arbitrary: x) (auto)

Lemma "heap t == heap (LH_insert x t)"

find_theories heap node

sorry

proof (prove)
goal 1 subgoal:
1. heap t == heap (LH_insert x t)

proof (prove)
goal 1 subgoal:
1. heap t == heap (LH_insert x t)

proof (prove)
goal 2 subgoal:
1. heap t == heap (LH_insert x t)

proof (prove)
goal 3 subgoal:
1. a x t x x a
   \{x. heap (LH_insert x t1); \forall x. heap (LH_insert x t2); heap t1; heap t2; meet_set (LH_insert x t1) + meet_set (LH_insert x t2); x1 < x; x2 < x; x x x; x1 x2 x \}
proof (prove)
goal (1 subgoal):
1. \( \forall x \exists t x t x t \exists t x \) 

\[ \exists t \quad \exists t \quad \exists t \quad \exists t \] 

proof (prove)
goal (2 subgoal):
1. \( \forall x \exists t x t x t \exists t x \) 

\[ \exists t \quad \exists t \quad \exists t \quad \exists t \] 

theorem \( \forall t \quad \exists t \) 

\[ \forall t \quad \exists t \quad \exists t \quad \exists t \]
Theorem \( \text{t} \rightarrow \text{t} \) \((\text{th\_insert \ x \ t})\)
fun t_lh_insert :: 'a list ⇒ 'a heap ⇒ nat
where
  | t_lh_insert x (_l,a,r) = (if a = x then l a (t_lh_insert x r) else node l x (t_lh_insert a r))

lemma "t_lh_insert x t ≤ rank t + 1"
  sorry

Exercise (Bootstrapping a Priority Queue)

proof (prove)
goal (1 subgoal)
1. t_lh_insert x t ≤ rank t + 1

apply (induction t arbitrary: x)
apply auto
done
Exercise (Bootstrapping a Priority Queue)

Given a generic priority queue implementation with
- \( O(1) \): empty, is_empty: operations, \( O(f(n)) \): insert, and \( O(\min(f(n)) \): get_min and del_min: operations.

Derive an implementation with \( O(1) \): get_min, and the asymptotic complexities of the other operations unchanged!

Hint: Store the current minimal element! As you know nothing about \( f_1 \) and \( f_2 \), you must not use \( \text{get\_min}/\text{del\_min} \)
in your new \( \text{insert} \) operation, and vice versa!

Note: For technical reasons, you have to define the new implementations type
outside the locale!

datatype \( \{\cdot, \cdot\} \text{ be\_pq = PUT\_CONSTRUCTORS\_HERE} \)
asymptotic complexities of the other operations unchanged!

Hint: Store the current minimal element! As you know nothing
about f₁ and f₂, you must not use get_min/del_min;
in your new insert operation, and vice versa!

text: For technical reasons, you have to define the new implementations type
outside the locale!

datatype f (s:3) bs_pq = PUT_CONSTRUCTORS HERE

locale Bs_Priority_Queue =
  orig: Priority_Queue where
  empty = orig_empty and
  is_empty = orig_is_empty and

locale

begin
In here, the original implementation is available with the prefix orig, e.g.

locale

begin
In here, the original implementation is available with the prefix orig, e.g.

locale

begin
In here, the original implementation is available with the prefix orig, e.g.

locale

begin
In here, the original implementation is available with the prefix orig, e.g.
begin

   text : In here, the original implementation is available with the prefix :orig
   term orig_empty term orig_invar
   the orig_empty_empty

   fun meet : ((a, q), bs, pq) => "a multiset"
   where
     "meet Empty = (\)"
     | "meet (Map a q) = orig_meet q"

   fun invar : ((a, q), bs, pq) => bool
   where
     "invar" = undefined

consts

   meet : ((a, q), bs, pq) => "a multiset"
   Found termination order: "()"
locale Priority_Queue_Merge = Priority_Queue where empty = empty for empty :: 

   locale merge_merge :: [invar q1, invar q2] == (merge q1 q2) = (meet q1 q2)

proof cases

case 1 then show TCase

next

proof cases

proof (state)
goal (8 subgoals):
1. local meet local.empty = ()
2. q. invar q == is_empty q == (local.meet q == ())
3. Q x. invar q == local.meet (local.insert x q) == local.meet q + (x#)
4. Q x. invar q == local.meet q + (x#) == local.meet (local.del_min q) + local.meet q - (local.get_min q)
5. Q x. invar q == local.meet q + (x#) == local.get_min q + Min_meet (local.meet q)
6. n. invar n

proof (prove)
goal (2 subgoals):
1. [invar q, is_empty q] == (local.meet q == ()
2. local.meet q == (is_empty q)
proof (prove)
goal (2 subgoals!):  
1. [invar q: is_empty q] → local.mut q = (q)  
2. [invar q: local.mut q = (q)] → is_empty q

proof (prove)
goal (5 subgoals!):  
1. [invar q: is_empty q] → local.mut q = (q)  
2. [invar q: local.mut q = (q)] → is_empty q  
3. [invar q: is_empty q] → local.mut q = (q)  
4. [invar q: local.mut q = (q)] → is_empty q  
5. [invar q: local.mut q = (q)] → is_empty q

proof (prove)
using this:  
invar q

proof (prove)
goal (1 subgoal):  
1. invar (local.insert x q)
proof (prove)
goal (2 subgoals):
  1. \( a \leq 22 \). \( \begin{align*}
    \varphi &= \text{Heap} (\text{Min} \cdot \text{meet} \cdot (\text{orig} \cdot \text{meet} \cdot x_22)) \cdot x_22; \cdot \text{orig} \cdot \text{invar} \cdot x_22; \\
    \text{Min} \cdot \text{meet} \cdot (\text{orig} \cdot \text{meet} \cdot x_22) &\neq (\text{orig} \cdot \text{meet} \cdot x_22); \quad \neg \chi \leq \text{Min} \cdot (\text{set} \cdot \text{insert} \cdot x \cdot (\text{set} \cdot \text{meet} \cdot (\text{orig} \cdot \text{meet} \cdot x_22)))
  \end{align*} \)

Sledgehammering...
Proof found.

proof (prove)
goal (2 subgoals):
  1. \( a \leq 22 \). \( \begin{align*}
    \varphi &= \text{Heap} (\text{Min} \cdot \text{meet} \cdot (\text{orig} \cdot \text{meet} \cdot x_22)) \cdot x_22; \cdot \text{orig} \cdot \text{invar} \cdot x_22; \\
    \text{Min} \cdot \text{meet} \cdot (\text{orig} \cdot \text{meet} \cdot x_22) &\neq (\text{orig} \cdot \text{meet} \cdot x_22); \quad \neg \chi \leq \text{Min} \cdot (\text{set} \cdot \text{insert} \cdot x \cdot (\text{set} \cdot \text{meet} \cdot (\text{orig} \cdot \text{meet} \cdot x_22)))
  \end{align*} \)

Sledgehammering...
Proof found.
proof (state)

end

text

A binary tree can be encoded as an array \([a_1, \ldots, a_n]\) such that
the parent of node \(a_i\) is node \(a_{i+1} \div 2\).

Thus, for a heap, each node is greater than or equal to its parent:

\[
\text{definition parent :: } \text{nat} \rightarrow \text{nat} \text{ where } \text{parent } p \downarrow = (\text{idiv } p \downarrow 1)
\]
A binary tree can be encoded as an array \([a_1, \ldots, a_n]\), such that the parent of node \(a_i\) is node \(a_{i \div 2}\).

Thus, for a heap, each node is greater than or equal to its parent:

\[
definition parent :: \text{nat} \Rightarrow \text{nat} \quad \text{where} \quad \text{parent} \ i = (i + 1) \div 2 - 1
\]

\[
definition is_heap :: a : \text{list} \Rightarrow \text{bool} \quad \text{where} \quad \text{is_heap} \ h = \forall \text{length} \ h. \ h \geq \text{length} \ h . \ (\forall \text{parent} \ i. \ h \geq \text{parent} \ i)
\]

A heap with a single defect at index \(j\) is characterized as follows:

- The heap property holds for all elements except \(j\).
- The heap property holds for all elements except \(j\), and the children of \(j\) must also be greater than their grand-parent.

\[
definition is_heap_except :: \text{nat} \Rightarrow \text{a : list} \Rightarrow \text{bool} \quad \text{where} \quad \text{is_heap_except} \ j h = \forall \text{length} \ h. \ j + 1 \geq \text{length} \ h . \ (\forall \text{parent} \ i. \ h \geq \text{parent} \ i)
\]

\[
\text{test} \quad \text{A heap with a single defect at index } j \text{ is characterized as follows:}
\]

- The heap property holds for all elements except \(j\).
- The heap property holds for all elements except \(j\), and the children of \(j\) must also be greater than their grand-parent.

\[
\text{consts}
\]

\[
\text{proof (prove)}
\]
**Lemma 1:** Let $h$ be a heap. The function $\text{is_heap}(h) \Rightarrow \text{is_heap}(\text{sift_up}(h))$.

**Proof:**
- Base case: If $h$ is empty, then $\text{is_heap}(h)$ is true and $\text{is_heap}(\text{sift_up}(h))$ is also true because $\text{sift_up}$ preserves the heap property.
- Inductive step: Assume $\text{is_heap}(h)$ holds. We need to show that $\text{is_heap}(\text{sift_up}(h))$ holds. Consider two cases:
  - If $h$ has a single element, then $\text{is_heap}(h)$ and $\text{is_heap}(\text{sift_up}(h))$ are both true.
  - If $h$ has more than one element, then by the definition of $\text{sift_up}$, $\text{sift_up}(h)$ maintains the heap property, so $\text{is_heap}(\text{sift_up}(h))$ holds.

**Theorem 2:** Let $h$ be a heap and $i$ be an index. The function $\text{is_heap}(h) \Rightarrow \text{is_heap}(\text{sift_up_i}(h, i))$.

**Proof:**
- Base case: If $i$ is the first element of $h$, then $\text{is_heap}(h)$ and $\text{is_heap}(\text{sift_up_i}(h, i))$ are both true.
- Inductive step: Assume $\text{is_heap}(h)$ holds. We need to show that $\text{is_heap}(\text{sift_up_i}(h, i))$ holds. Consider two cases:
  - If $i$ is the first element of $h$, then $\text{is_heap}(h)$ and $\text{is_heap}(\text{sift_up_i}(h, i))$ are both true.
  - If $i$ is not the first element of $h$, then by the definition of $\text{sift_up_i}$, $\text{sift_up_i}(h, i)$ maintains the heap property, so $\text{is_heap}(\text{sift_up_i}(h, i))$ holds.