Script **generated by TTT**

Title: Lammich: FDS Tutorial (07.07.2017)

Date: Fri Jul 07 08:36:18 CEST 2017

Duration: 89:50 min

Pages: 55

`Thys/BinHeap.thy`

Binomial Heaps — Correctness and Complexity

Numerical Method

See *Chris Okasaki. Purely Functional Data Structures. Cambridge University Press, 1999*

Only use trees t_i of size 2^i

Numerical Method

See *Chris Okasaki. Purely Functional Data Structures. Cambridge University Press, 1999*

Only use trees t_i of size 2^i

E.g., to store $0b11001$ elements: $[t_0,0,0,t_3,t_4]$

Numerical Method

See *Chris Okasaki. Purely Functional Data Structures. Cambridge University Press, 1999*

Only use trees t_i of size 2^i

E.g., to store $0b11001$ elements: $[t_0,0,0,t_3,t_4]$

Meld: Addition with carry.

Numerical Method

See *Chris Okasaki. Purely Functional Data Structures. Cambridge University Press, 1999*

Only use trees t_i of size 2^i

E.g., to store $0b11001$ elements: $[t_0,0,0,t_3,t_4]$

Meld: Addition with carry.

Linking two trees of size 2^i: Yields size 2^{i+1}

Organization of Trees

datatype $'a\ tree = Node\ (rank\colon nat)\ (root\colon 'a)\ (children\colon 'a\ tree\ list)$

Organization of Trees

datatype $'a\ tree = Node\ (rank{:}\ nat)\ (root{:}\ 'a)\ (children{:}\ 'a\ tree\ list)$

Node with rank i has successors $[t_{i-1}, \ldots t_0]$ with ranks $[i{-}1, \ldots, 0]$

$btree\_invar\ (Node\ r\ uu\ c) =$
$(Ball\ (set\ c)\ btree\_invar \wedge map\ rank\ c = rev\ [0..<r])$

Organization of Trees

datatype $'a\ tree = Node\ (rank{:}\ nat)\ (root{:}\ 'a)\ (children{:}\ 'a\ tree\ list)$

Node with rank i has successors $[t_{i-1}, \ldots t_0]$ with ranks $[i{-}1, \ldots, 0]$

$btree\_invar\ (Node\ r\ uu\ c) =$
$(Ball\ (set\ c)\ btree\_invar \wedge map\ rank\ c = rev\ [0..<r])$

Tree has exactly $2^{rank\ t}$ nodes

$btree\_invar\ t \Longrightarrow |t| = 2^{rank\ t}$

Linking two Trees

Given two trees of rank i, join them to tree of rank $i{+}1$.

Linking two Trees

Given two trees of rank i, join them to tree of rank $i{+}1$.

Idea: Insert one tree under root of other tree

Heap Datatype

Using sparse representation for binary numbers:
$[t_0, 0, 0, t_3, t_4]$ represented as $[\ (0, t_0),\ (3, t_3), (4, t_4)\ ]$

Heap Datatype

Using sparse representation for binary numbers:
$[t_0, 0, 0, t_3, t_4]$ represented as $[\ (0, t_0),\ (3, t_3), (4, t_4)\ ]$

type_synonym *'a heap = 'a tree list*

Heap Datatype

Using sparse representation for binary numbers:
$[t_0, 0, 0, t_3, t_4]$ represented as $[\ (0, t_0),\ (3, t_3), (4, t_4)\ ]$

type_synonym *'a heap = 'a tree list*

bheap_invar c = ((∀ *t*∈*set c*. *btree_invar t*) ∧
strictly_ascending (*map rank c*))
Ranks in ascending order

Inserting a Tree

ins_tree t [] = [*t*]
ins_tree t_1 (t_2 # *rest*) =
(if *rank* t_1 < *rank* t_2 then t_1 # t_2 # *rest*
 else *ins_tree* (*link* t_1 t_2) *rest*)

Inserting a Tree

$ins\_tree\ t\ []\ =\ [t]$
$ins\_tree\ t_1\ (t_2\ \#\ rest)\ =$
$(\textsf{if}\ rank\ t_1\ <\ rank\ t_2\ \textsf{then}\ t_1\ \#\ t_2\ \#\ rest$
$\ \textsf{else}\ ins\_tree\ (link\ t_1\ t_2)\ rest)$

Intuition: Handle a carry

Merge

$merge\ ts_1\ []\ =\ ts_1$
$merge\ []\ ts_2\ =\ ts_2$
$merge\ (t_1\ \#\ ts_1)\ (t_2\ \#\ ts_2)\ =$
$(\textsf{if}\ rank\ t_1\ <\ rank\ t_2\ \textsf{then}\ t_1\ \#\ merge\ ts_1\ (t_2\ \#\ ts_2)$
$\ \textsf{else if}\ rank\ t_2\ <\ rank\ t_1\ \textsf{then}\ t_2\ \#\ merge\ (t_1\ \#\ ts_1)\ ts_2$
$\quad\quad \textsf{else}\ ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2))$

Merge

$merge\ ts_1\ []\ =\ ts_1$
$merge\ []\ ts_2\ =\ ts_2$
$merge\ (t_1\ \#\ ts_1)\ (t_2\ \#\ ts_2)\ =$
$(\textsf{if}\ rank\ t_1\ <\ rank\ t_2\ \textsf{then}\ t_1\ \#\ merge\ ts_1\ (t_2\ \#\ ts_2)$
$\ \textsf{else if}\ rank\ t_2\ <\ rank\ t_1\ \textsf{then}\ t_2\ \#\ merge\ (t_1\ \#\ ts_1)\ ts_2$
$\quad\quad \textsf{else}\ ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2))$

Intuition: Addition of binary numbers

Find/Delete Minimum Element

All trees are min-heaps

Smallest element may be any root node

$ts \neq []\ \Longrightarrow\ find\_min\ ts\ =\ Min\ (set\ (map\ root\ ts))$

Find/Delete Minimum Element

All trees are min-heaps

Smallest element may be any root node

$ts \neq [] \Longrightarrow \mathit{find\_min}\ ts = \mathit{Min}\ (\mathit{set}\ (\mathit{map}\ \mathit{root}\ ts))$

Similar: $\mathit{get\_min} :: 'a\ \mathit{tree}\ \mathit{list} \Rightarrow 'a\ \mathit{tree} \times 'a\ \mathit{tree}\ \mathit{list}$

Returns tree with minimal root, and other trees

Find/Delete Minimum Element

All trees are min-heaps

Smallest element may be any root node

$ts \neq [] \Longrightarrow \mathit{find\_min}\ ts = \mathit{Min}\ (\mathit{set}\ (\mathit{map}\ \mathit{root}\ ts))$

Similar: $\mathit{get\_min} :: 'a\ \mathit{tree}\ \mathit{list} \Rightarrow 'a\ \mathit{tree} \times 'a\ \mathit{tree}\ \mathit{list}$

Returns tree with minimal root, and other trees

Delete via merge

$\mathit{delete\_min}\ ts =$
$(\text{case } \mathit{get\_min}\ ts \text{ of}$
$\quad (\mathit{Node}\ \mathit{xa}\ x\ ts_1,\ ts_2) \Rightarrow \mathit{merge}\ (\mathit{rev}\ ts_1)\ ts_2)$

Note the rev!

Complexity

Recall: $|t| = 2^{\mathit{rank}\ t}$

Complexity

Recall: $|t| = 2^{\mathit{rank}\ t}$

Complexity

Recall: $|t| = 2^{rank\ t}$

Similarly for heap: $2^{length\ h} \le |h|$

Complexity

Recall: $|t| = 2^{rank\ t}$

Similarly for heap: $2^{length\ h} \le |h|$

Complexity of operations: linear in length of heap

Complexity

Recall: $|t| = 2^{rank\ t}$

Similarly for heap: $2^{length\ h} \le |h|$

Complexity of operations: linear in length of heap

i.e., logarithmic in number of elements

Find/Delete Minimum Element

All trees are min-heaps

Smallest element may be any root node

$ts \neq [] \Longrightarrow find\_min\ ts = Min\ (set\ (map\ root\ ts))$

Similar: $get\_min :: 'a\ tree\ list \Rightarrow 'a\ tree \times 'a\ tree\ list$

Returns tree with minimal root, and other trees

Find/Delete Minimum Element

All trees are min-heaps

Smallest element may be any root node

$ts \neq [] \Longrightarrow find\_min\ ts = Min\ (set\ (map\ root\ ts))$

Similar: $get\_min :: 'a\ tree\ list \Rightarrow 'a\ tree \times 'a\ tree\ list$

Returns tree with minimal root, and other trees

Delete via merge

$delete\_min\ ts =$
$($case $get\_min\ ts$ of
$\quad (Node\ xa\ x\ ts_1,\ ts_2) \Rightarrow merge\ (rev\ ts_1)\ ts_2)$

Note the *rev*!

Complexity of Merge

$merge\ (t_1\ \#\ ts_1)\ (t_2\ \#\ ts_2) =$
$($if $rank\ t_1 < rank\ t_2$ then $t_1\ \#\ merge\ ts_1\ (t_2\ \#\ ts_2)$
$\ $else if $rank\ t_2 < rank\ t_1$ then $t_2\ \#\ merge\ (t_1\ \#\ ts_1)\ ts_2$
$\quad\quad$else $ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2))$

Complexity of ins_tree call depends on length of result of recursive call.

Naive: $length\ (merge\ ts_1\ ts_2) \leq length\ ts_1 + length\ ts_2$

Complexity of Merge

$merge\ (t_1\ \#\ ts_1)\ (t_2\ \#\ ts_2) =$
$($if $rank\ t_1 < rank\ t_2$ then $t_1\ \#\ merge\ ts_1\ (t_2\ \#\ ts_2)$
$\ $else if $rank\ t_2 < rank\ t_1$ then $t_2\ \#\ merge\ (t_1\ \#\ ts_1)\ ts_2$
$\quad\quad$else $ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2))$

Complexity of ins_tree call depends on length of result of recursive call.

Naive: $length\ (merge\ ts_1\ ts_2) \leq length\ ts_1 + length\ ts_2$

Yields (roughly) $m\ n = m\ (n{-}2) + n$

Complexity of Merge

$merge\ (t_1\ \#\ ts_1)\ (t_2\ \#\ ts_2) =$
$($if $rank\ t_1 < rank\ t_2$ then $t_1\ \#\ merge\ ts_1\ (t_2\ \#\ ts_2)$
$\ $else if $rank\ t_2 < rank\ t_1$ then $t_2\ \#\ merge\ (t_1\ \#\ ts_1)\ ts_2$
$\quad\quad$else $ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2))$

Complexity of Merge

$merge\ (t_1\ \#\ ts_1)\ (t_2\ \#\ ts_2) =$
(if $rank\ t_1 < rank\ t_2$ then $t_1\ \#\ merge\ ts_1\ (t_2\ \#\ ts_2)$
else if $rank\ t_2 < rank\ t_1$ then $t_2\ \#\ merge\ (t_1\ \#\ ts_1)\ ts_2$
else $ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2))$

Idea: Estimate cost and length of result:

$t\_ins\_tree\ t\ ts + length\ (ins\_tree\ t\ ts) = 2 + length\ ts$
$length\ (merge\ ts_1\ ts_2) + t\_merge\ ts_1\ ts_2$
$\leq 2 * (length\ ts_1 + length\ ts_2) + 1$

Yields desired linear bound!

Complexity of Merge

$merge\ (t_1\ \#\ ts_1)\ (t_2\ \#\ ts_2) =$
(if $rank\ t_1 < rank\ t_2$ then $t_1\ \#\ merge\ ts_1\ (t_2\ \#\ ts_2)$
else if $rank\ t_2 < rank\ t_1$ then $t_2\ \#\ merge\ (t_1\ \#\ ts_1)\ ts_2$
else $ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2))$

Idea: Estimate cost and length of result:

$t\_ins\_tree\ t\ ts + length\ (ins\_tree\ t\ ts) = 2 + length\ ts$
$length\ (merge\ ts_1\ ts_2) + t\_merge\ ts_1\ ts_2$
$\leq 2 * (length\ ts_1 + length\ ts_2) + 1$

Yields desired linear bound!

171

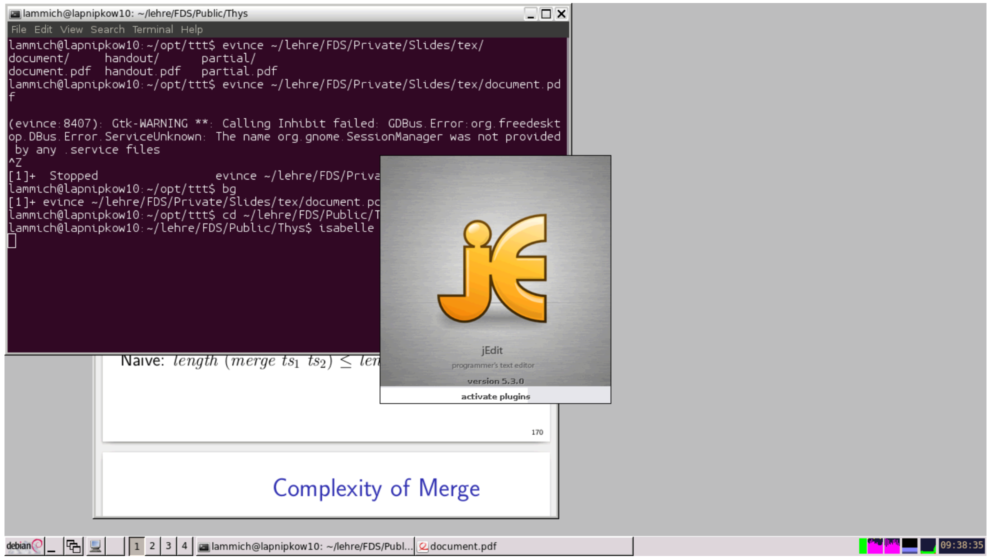
Complexity of Merge

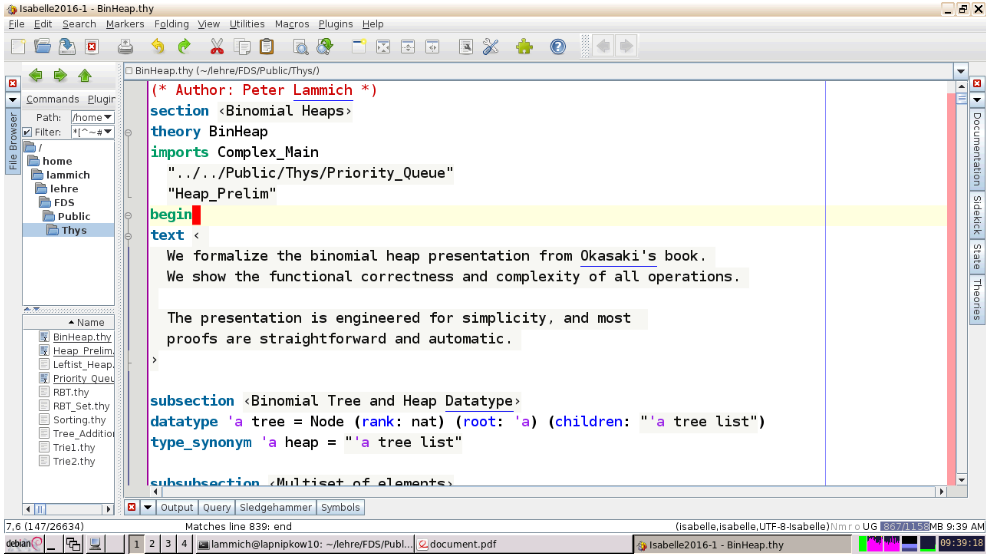
$merge\ (t_1\ \#\ ts_1)\ (t_2\ \#\ ts_2) =$
(if $rank\ t_1 < rank\ t_2$ then $t_1\ \#\ merge\ ts_1\ (t_2\ \#\ ts_2)$
else if $rank\ t_2 < rank\ t_1$ then $t_2\ \#\ merge\ (t_1\ \#\ ts_1)\ ts_2$
else $ins\_tree\ (link\ t_1\ t_2)\ (merge\ ts_1\ ts_2))$

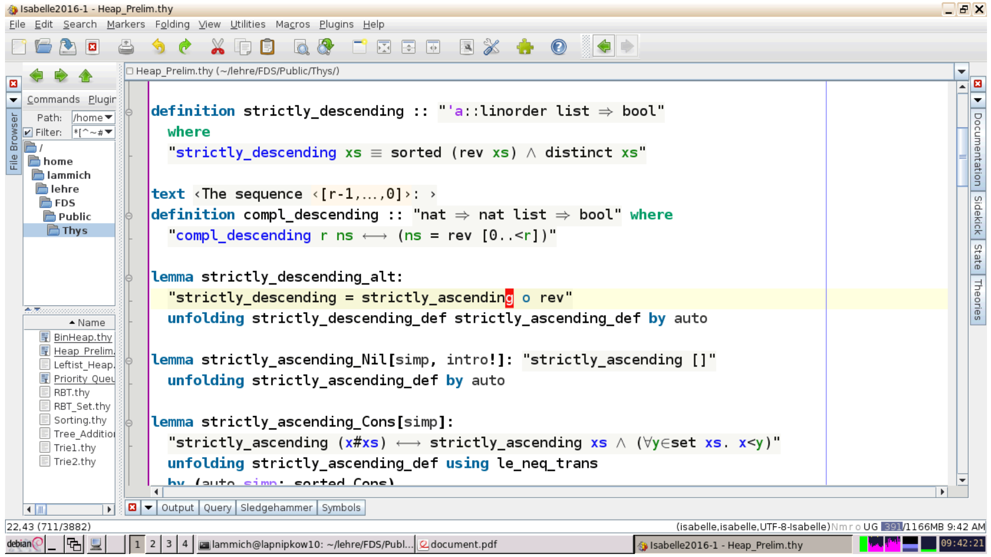
Complexity of ins_tree call depends on length of result of recursive call.

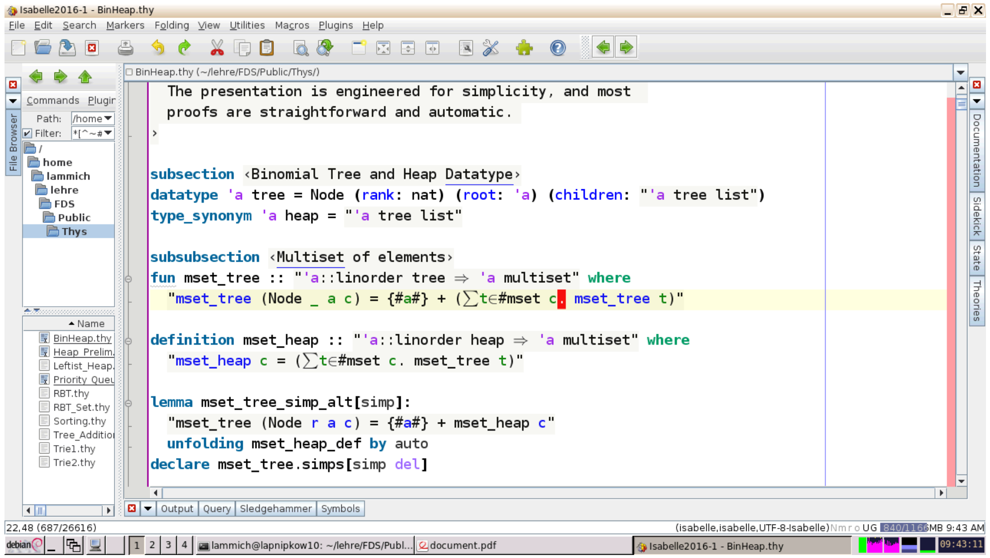
Naive: $length\ (merge\ ts_1\ ts_2) \leq length\ ts_1 + length\ ts_2$

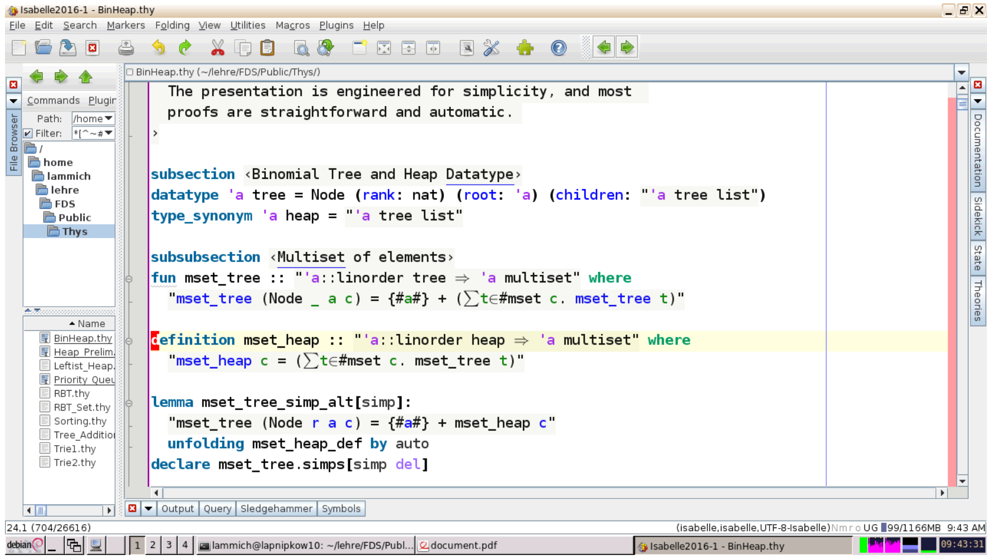
Yields (roughly) $m\ n = m\ (n{-}2) + n$ quadratic!

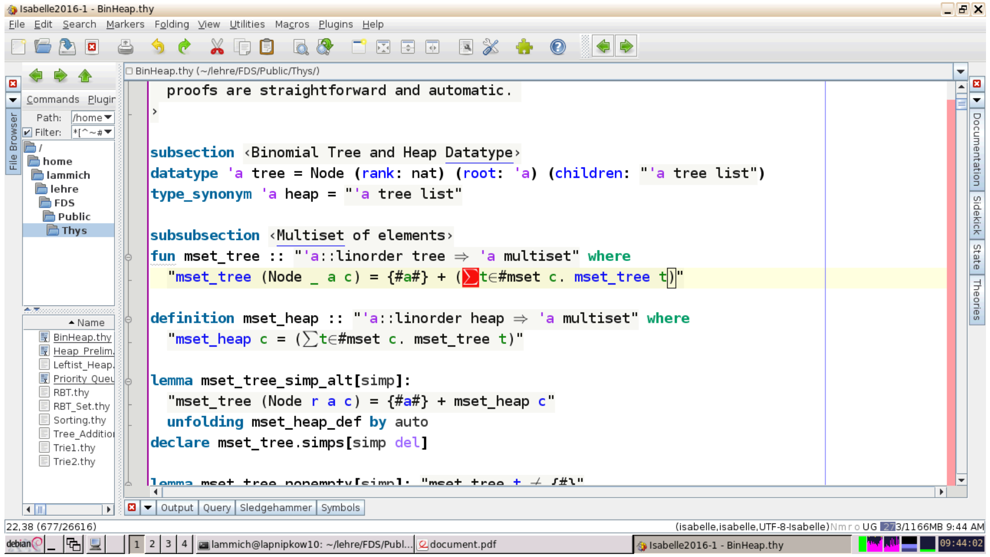


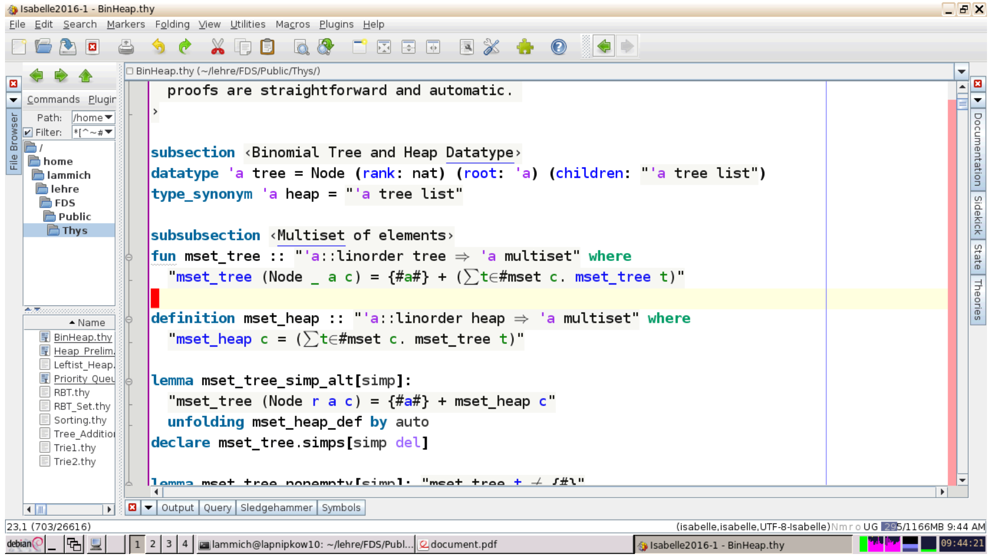

Isabelle2016-1 - BinHeap.thy
BinHeap.thy (~/lehre/FDS/Public/Thys/)
(* Author: Peter Lammich *)
section ‹Binomial Heaps›
theory BinHeap
imports Complex_Main
"../../Public/Thys/Priority_Queue"
"Heap_Prelim"
begin
text ‹
We formalize the binomial heap presentation from Okasaki's book.
We show the functional correctness and complexity of all operations.
The presentation is engineered for simplicity, and most
proofs are straightforward and automatic.
›
subsection ‹Binomial Tree and Heap Datatype›
datatype 'a tree = Node (rank: nat) (root: 'a) (children: "'a tree list")
type_synonym 'a heap = "'a tree list"
Output Query Sledgehammer Symbols
Matches line 839: end

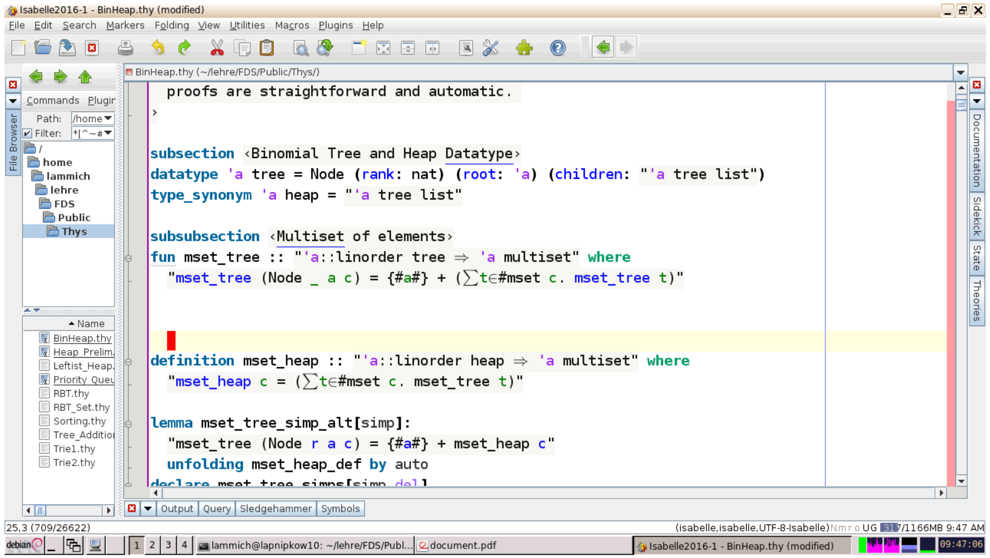

Isabelle2016-1 - Heap_Prelim.thy
Heap_Prelim.thy (~/lehre/FDS/Public/Thys/)
definition strictly_descending :: "'a::linorder list ⇒ bool"
where
"strictly_descending xs ≡ sorted (rev xs) ∧ distinct xs"
text ‹The sequence ‹[r-1,…,0]›.›
definition compl_descending :: "nat ⇒ nat list ⇒ bool" where
"compl_descending r ns ⟷ (ns = rev [0..<r])"
lemma strictly_descending_alt:
"strictly_descending = strictly_ascending o rev"
unfolding strictly_descending_def strictly_ascending_def by auto
lemma strictly_ascending_Nil[simp, intro!]: "strictly_ascending []"
unfolding strictly_ascending_def by auto
lemma strictly_ascending_Cons[simp]:
"strictly_ascending (x#xs) ⟷ strictly_ascending xs ∧ (∀y∈set xs. x<y)"
unfolding strictly_ascending_def using le_neq_trans
Output Query Sledgehammer Symbols

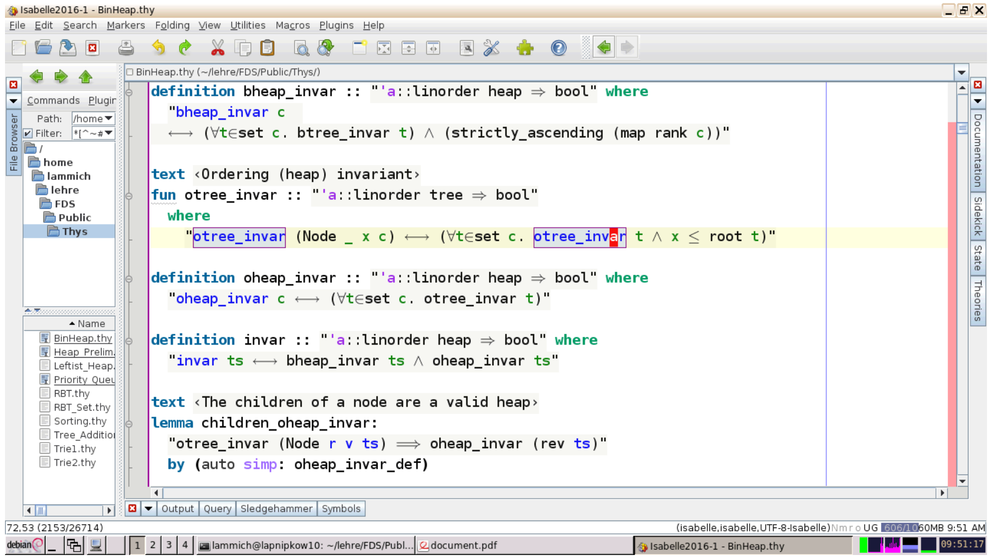

Isabelle2016-1 - BinHeap.thy
BinHeap.thy (~/lehre/FDS/Public/Thys/)
The presentation is engineered for simplicity, and most
proofs are straightforward and automatic.
›
subsection ‹Binomial Tree and Heap Datatype›
datatype 'a tree = Node (rank: nat) (root: 'a) (children: "'a tree list")
type_synonym 'a heap = "'a tree list"
subsubsection ‹Multiset of elements›
fun mset_tree :: "'a::linorder tree ⇒ 'a multiset" where
"mset_tree (Node _ a c) = {#a#} + (∑t∈#mset c. mset_tree t)"
definition mset_heap :: "'a::linorder heap ⇒ 'a multiset" where
"mset_heap c = (∑t∈#mset c. mset_tree t)"
lemma mset_tree_simp_alt[simp]:
"mset_tree (Node r a c) = {#a#} + mset_heap c"
unfolding mset_heap_def by auto
declare mset_tree.simps[simp del]
Output Query Sledgehammer Symbols

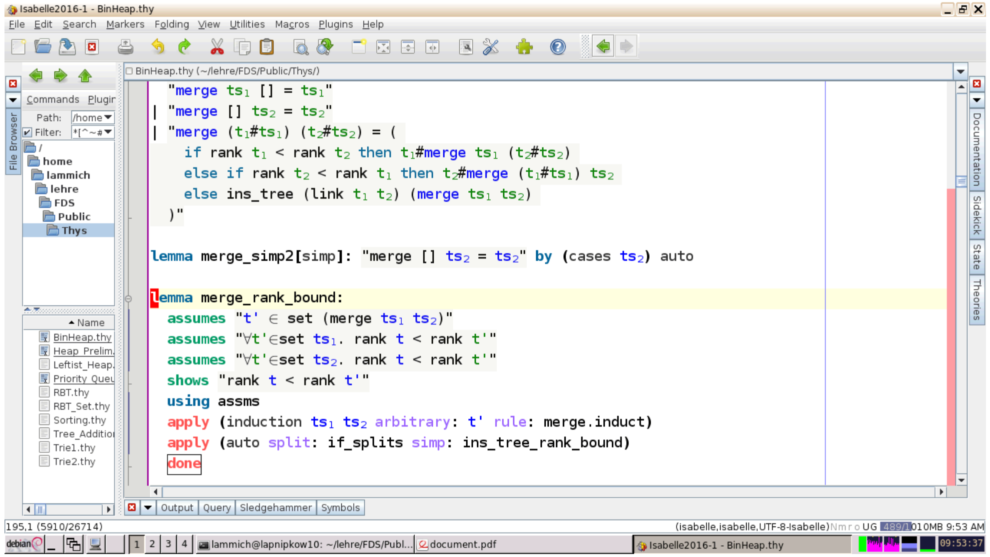
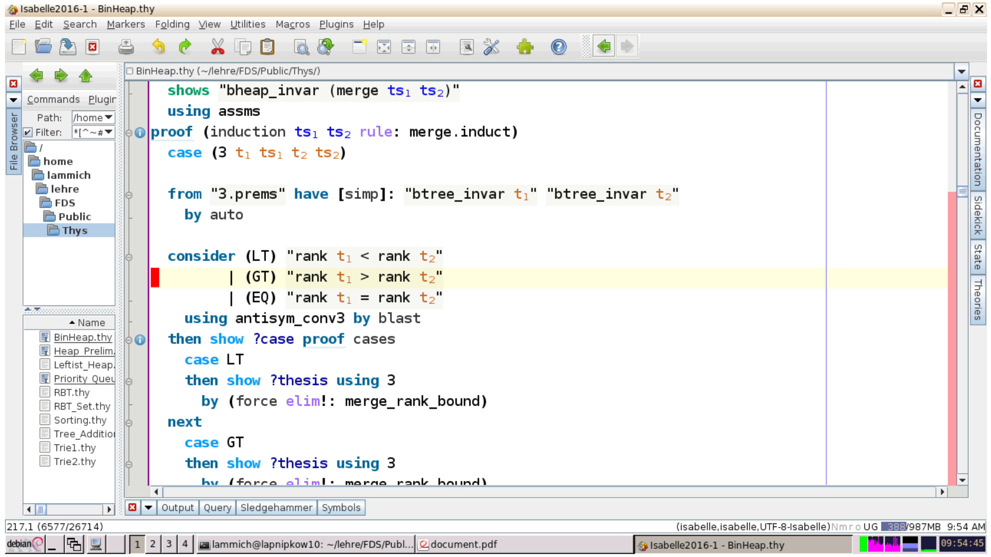
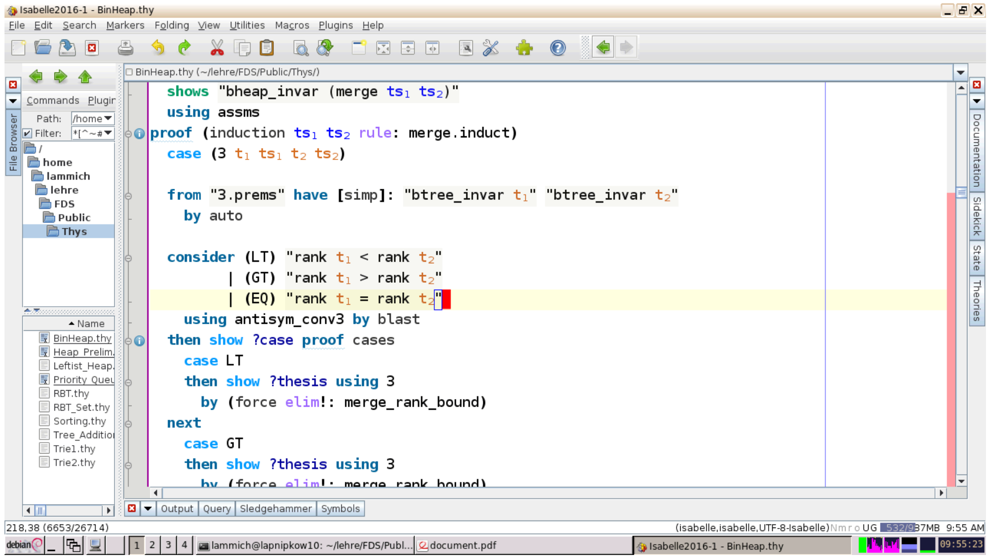
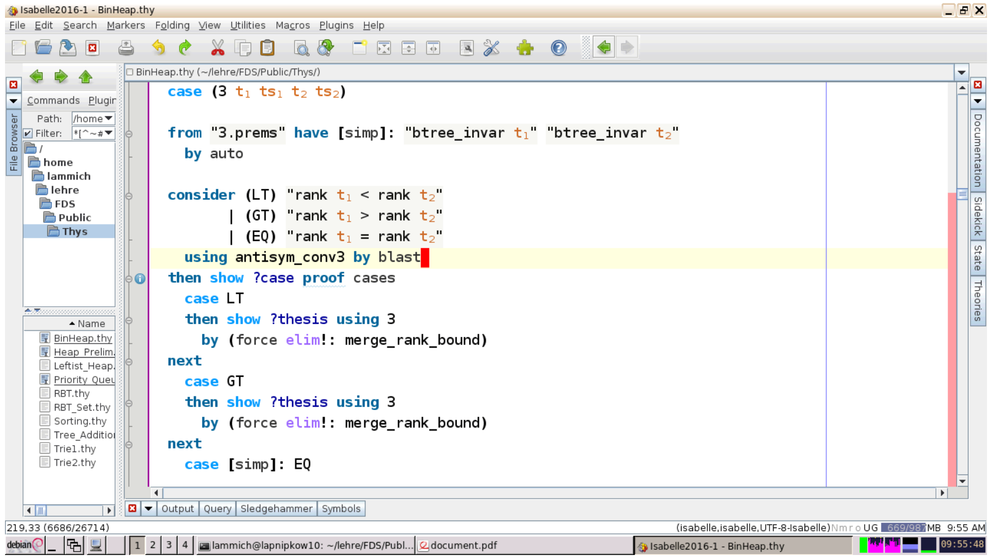

Isabelle2016-1 - BinHeap.thy
BinHeap.thy (~/lehre/FDS/Public/Thys/)
The presentation is engineered for simplicity, and most
proofs are straightforward and automatic.
›
subsection ‹Binomial Tree and Heap Datatype›
datatype 'a tree = Node (rank: nat) (root: 'a) (children: "'a tree list")
type_synonym 'a heap = "'a tree list"
subsubsection ‹Multiset of elements›
fun mset_tree :: "'a::linorder tree ⇒ 'a multiset" where
"mset_tree (Node _ a c) = {#a#} + (∑t∈#mset c. mset_tree t)"
definition mset_heap :: "'a::linorder heap ⇒ 'a multiset" where
"mset_heap c = (∑t∈#mset c. mset_tree t)"
lemma mset_tree_simp_alt[simp]:
"mset_tree (Node r a c) = {#a#} + mset_heap c"
unfolding mset_heap_def by auto
declare mset_tree.simps[simp del]
Output Query Sledgehammer Symbols

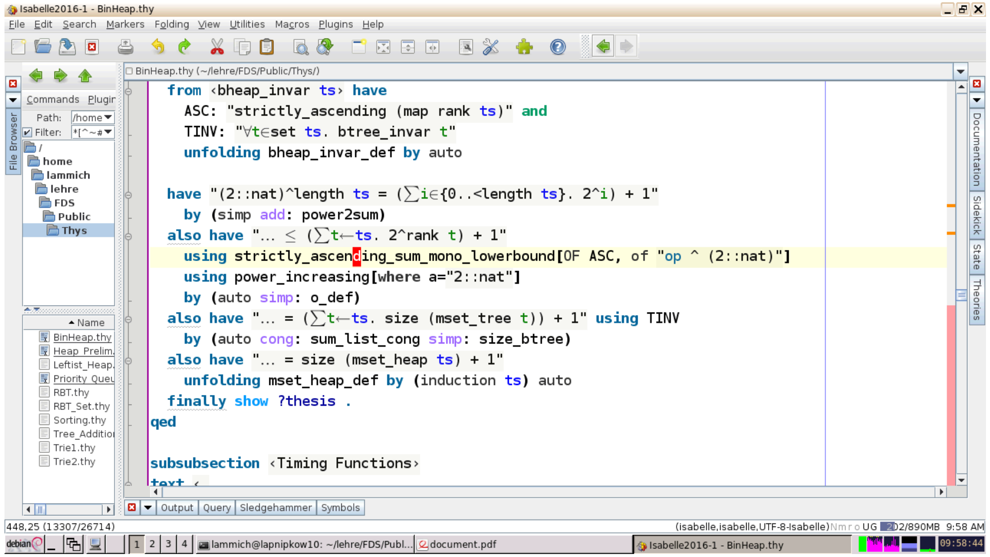

Isabelle2016-1 - BinHeap.thy
proofs are straightforward and automatic.
subsection ‹Binomial Tree and Heap Datatype›
datatype 'a tree = Node (rank: nat) (root: 'a) (children: "'a tree list")
type_synonym 'a heap = "'a tree list"
subsubsection ‹Multiset of elements›
fun mset_tree :: "'a::linorder tree ⇒ 'a multiset" where
"mset_tree (Node _ a c) = {#a#} + (∑t∈#mset c. mset_tree t)"
definition mset_heap :: "'a::linorder heap ⇒ 'a multiset" where
"mset_heap c = (∑t∈#mset c. mset_tree t)"
lemma mset_tree_simp_alt[simp]:
"mset_tree (Node r a c) = {#a#} + mset_heap c"
unfolding mset_heap_def by auto
declare mset_tree.simps[simp del]
Output Query Sledgehammer Symbols

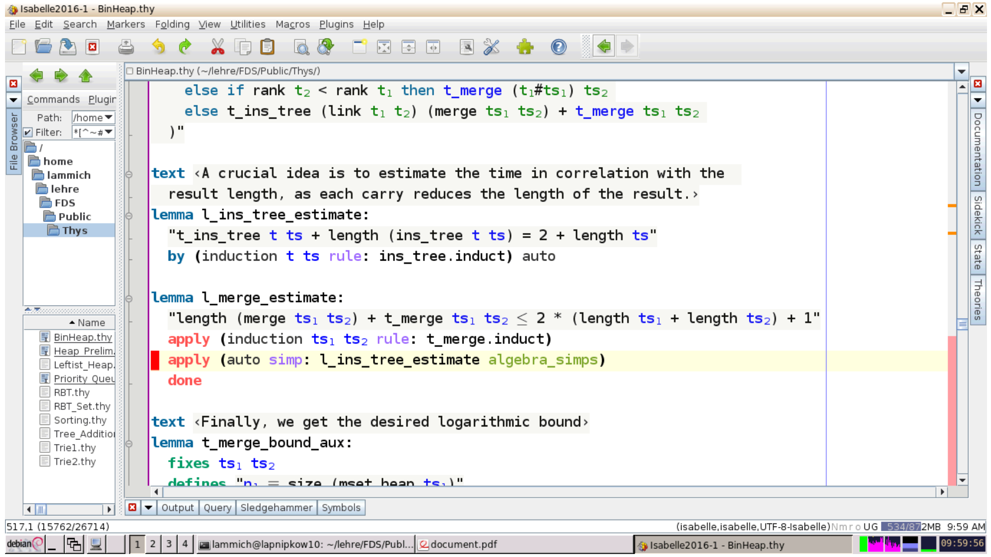

Isabelle2016-1 - BinHeap.thy
proofs are straightforward and automatic.
subsection ‹Binomial Tree and Heap Datatype›
datatype 'a tree = Node (rank: nat) (root: 'a) (children: "'a tree list")
type_synonym 'a heap = "'a tree list"
subsubsection ‹Multiset of elements›
fun mset_tree :: "'a::linorder tree ⇒ 'a multiset" where
"mset_tree (Node _ a c) = {#a#} + (∑t∈#mset c. mset_tree t)"
definition mset_heap :: "'a::linorder heap ⇒ 'a multiset" where
"mset_heap c = (∑t∈#mset c. mset_tree t)"
lemma mset_tree_simp_alt[simp]:
"mset_tree (Node r a c) = {#a#} + mset_heap c"
unfolding mset_heap_def by auto
declare mset_tree.simps[simp del]
Output Query Sledgehammer Symbols

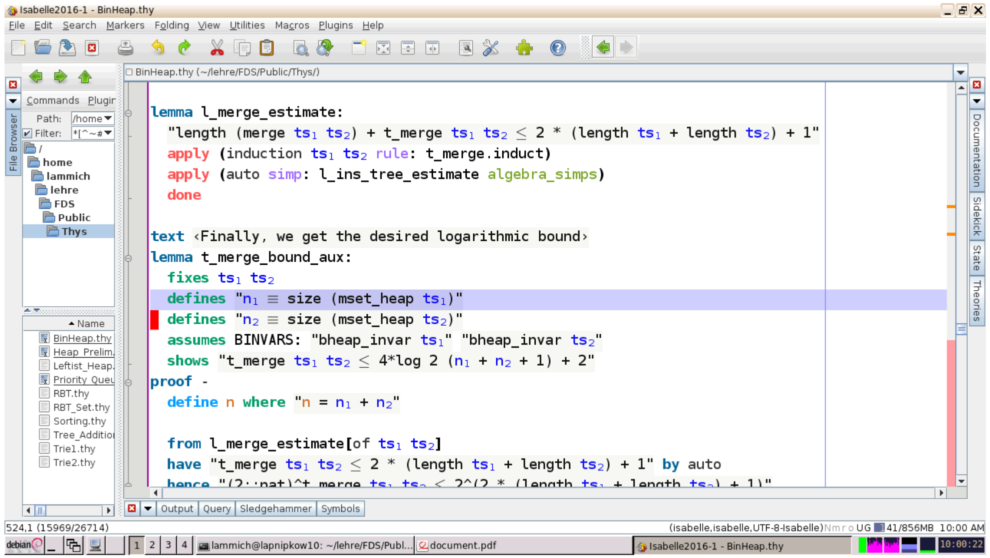

Isabelle2016-1 - BinHeap.thy (modified)
proofs are straightforward and automatic.
subsection ‹Binomial Tree and Heap Datatype›
datatype 'a tree = Node (rank: nat) (root: 'a) (children: "'a tree list")
type_synonym 'a heap = "'a tree list"
subsubsection ‹Multiset of elements›
fun mset_tree :: "'a::linorder tree ⇒ 'a multiset" where
"mset_tree (Node _ a c) = {#a#} + (∑t∈#mset c. mset_tree t)"
definition mset_heap :: "'a::linorder heap ⇒ 'a multiset" where
"mset_heap c = (∑t∈#mset c. mset_tree t)"
lemma mset_tree_simp_alt[simp]:
"mset_tree (Node r a c) = {#a#} + mset_heap c"
unfolding mset_heap_def by auto
Output Query Sledgehammer Symbols

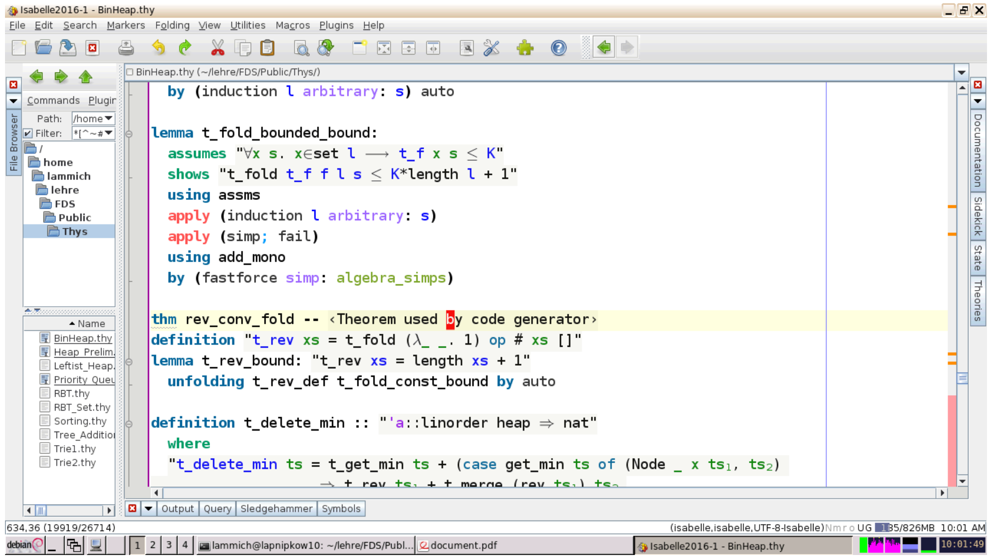

Isabelle2016-1 - BinHeap.thy
definition bheap_invar :: "'a::linorder heap ⇒ bool" where
"bheap_invar c
⟷ (∀t∈set c. btree_invar t) ∧ (strictly_ascending (map rank c))"
text ‹Ordering (heap) invariant›
fun otree_invar :: "'a::linorder tree ⇒ bool"
where
"otree_invar (Node _ x c) ⟷ (∀t∈set c. otree_invar t ∧ x ≤ root t)"
definition oheap_invar :: "'a::linorder heap ⇒ bool" where
"oheap_invar c ⟷ (∀t∈set c. otree_invar t)"
definition invar :: "'a::linorder heap ⇒ bool" where
"invar ts ⟷ bheap_invar ts ∧ oheap_invar ts"
text ‹The children of a node are a valid heap›
lemma children_oheap_invar:
"otree_invar (Node r v ts) ⟹ oheap_invar (rev ts)"
by (auto simp: oheap_invar_def)
Output Query Sledgehammer Symbols

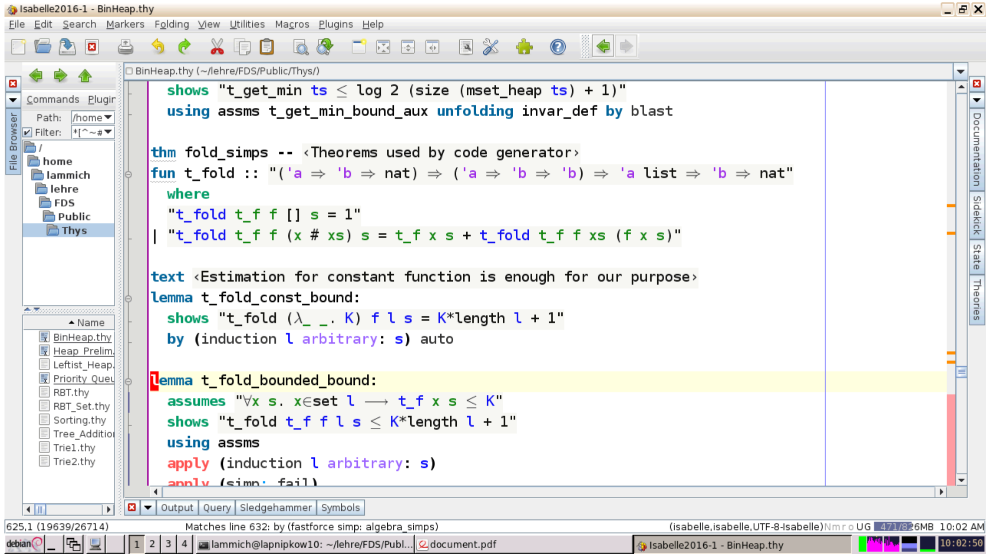

"merge ts1 [] = ts1"
| "merge [] ts2 = ts2"
| "merge (t1#ts1) (t2#ts2) = (
if rank t1 < rank t2 then t1#merge ts1 (t2#ts2)
else if rank t2 < rank t1 then t2#merge (t1#ts1) ts2
else ins_tree (link t1 t2) (merge ts1 ts2)
)"
lemma merge_simp2[simp]: "merge [] ts2 = ts2" by (cases ts2) auto
lemma merge_rank_bound:
assumes "t' ∈ set (merge ts1 ts2)"
assumes "∀t∈set ts1. rank t < rank t'"
assumes "∀t∈set ts2. rank t < rank t'"
shows "rank t < rank t'"
using assms
apply (induction ts1 ts2 arbitrary: t' rule: merge.induct)
apply (auto split: if_splits simp: ins_tree_rank_bound)
done

shows "bheap_invar (merge ts1 ts2)"
using assms
proof (induction ts1 ts2 rule: merge.induct)
case (3 t1 ts1 t2 ts2)
from "3.prems" have [simp]: "btree_invar t1" "btree_invar t2"
by auto
consider (LT) "rank t1 < rank t2"
| (GT) "rank t1 > rank t2"
| (EQ) "rank t1 = rank t2"
using antisym_conv3 by blast
then show ?case proof cases
case LT
then show ?thesis using 3
by (force elim!: merge_rank_bound)
next
case GT
then show ?thesis using 3

shows "bheap_invar (merge ts1 ts2)"
using assms
proof (induction ts1 ts2 rule: merge.induct)
case (3 t1 ts1 t2 ts2)
from "3.prems" have [simp]: "btree_invar t1" "btree_invar t2"
by auto
consider (LT) "rank t1 < rank t2"
| (GT) "rank t1 > rank t2"
| (EQ) "rank t1 = rank t2"
using antisym_conv3 by blast
then show ?case proof cases
case LT
then show ?thesis using 3
by (force elim!: merge_rank_bound)
next
case GT
then show ?thesis using 3

case (3 t1 ts1 t2 ts2)
from "3.prems" have [simp]: "btree_invar t1" "btree_invar t2"
by auto
consider (LT) "rank t1 < rank t2"
| (GT) "rank t1 > rank t2"
| (EQ) "rank t1 = rank t2"
using antisym_conv3 by blast
then show ?case proof cases
case LT
then show ?thesis using 3
by (force elim!: merge_rank_bound)
next
case GT
then show ?thesis using 3
by (force elim!: merge_rank_bound)
next
case [simp]: EQ

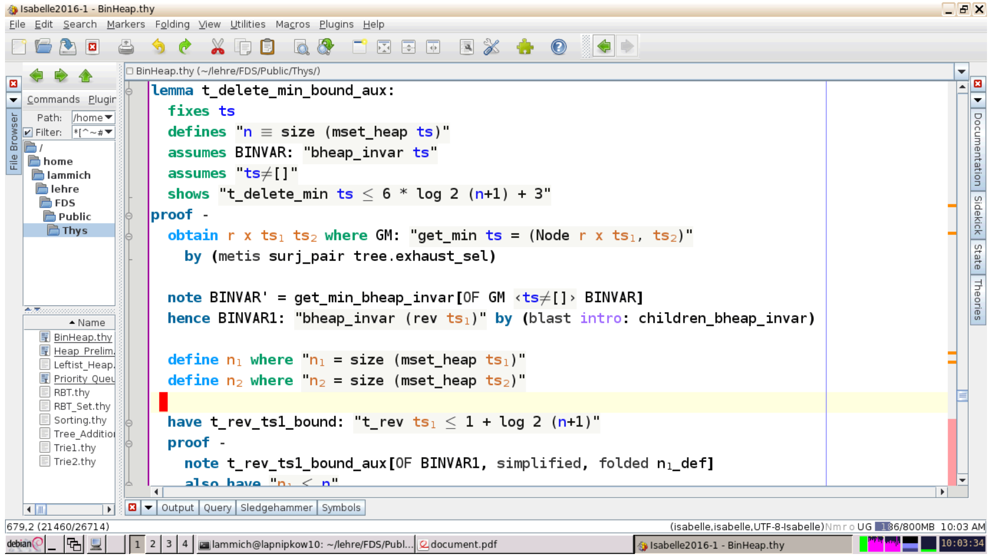

Isabelle2016-1 - BinHeap.thy
from ‹bheap_invar ts› have
ASC: "strictly_ascending (map rank ts)" and
TINV: "∀t∈set ts. btree_invar t"
unfolding bheap_invar_def by auto
have "(2::nat)^length ts = (∑i∈{0..<length ts}. 2^i) + 1"
by (simp add: power2sum)
also have "... ≤ (∑t←ts. 2^rank t) + 1"
using strictly_ascending_sum_mono_lowerbound[OF ASC, of "op ^ (2::nat)"]
using power_increasing[where a="2::nat"]
by (auto simp: o_def)
also have "... = (∑t←ts. size (mset_tree t)) + 1" using TINV
by (auto cong: sum_list_cong simp: size_btree)
also have "... = size (mset_heap ts) + 1"
unfolding mset_heap_def by (induction ts) auto
finally show ?thesis .
qed
subsubsection ‹Timing Functions›

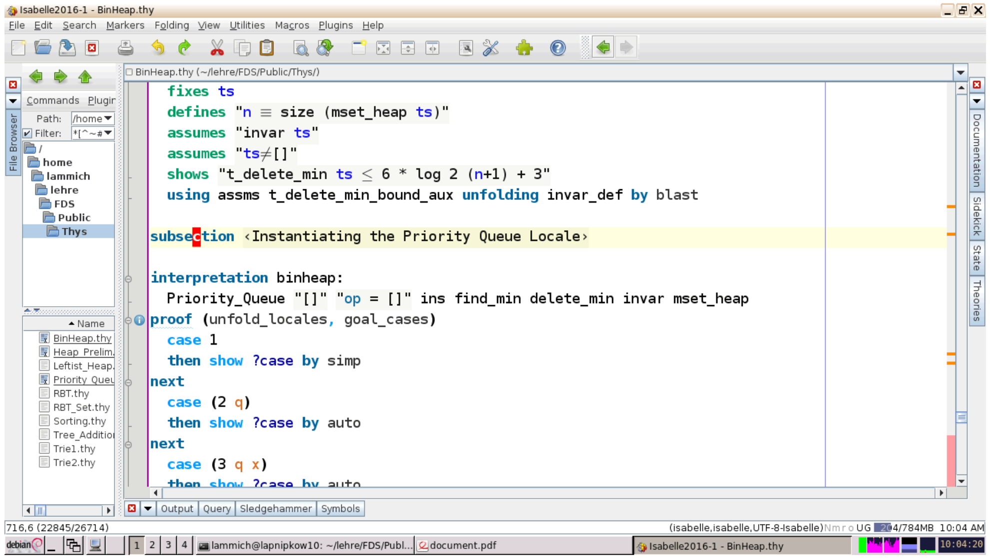

Isabelle2016-1 - BinHeap.thy
else if rank t₂ < rank t₁ then t_merge (t₁#ts₁) ts₂
else t_ins_tree (link t₁ t₂) (merge ts₁ ts₂) + t_merge ts₁ ts₂
)"
text ‹A crucial idea is to estimate the time in correlation with the
result length, as each carry reduces the length of the result.›
lemma l_ins_tree_estimate:
"t_ins_tree t ts + length (ins_tree t ts) = 2 + length ts"
by (induction t ts rule: ins_tree.induct) auto
lemma l_merge_estimate:
"length (merge ts₁ ts₂) + t_merge ts₁ ts₂ ≤ 2 * (length ts₁ + length ts₂) + 1"
apply (induction ts₁ ts₂ rule: t_merge.induct)
apply (auto simp: l_ins_tree_estimate algebra_simps)
done
text ‹Finally, we get the desired logarithmic bound›
lemma t_merge_bound_aux:
fixes ts₁ ts₂

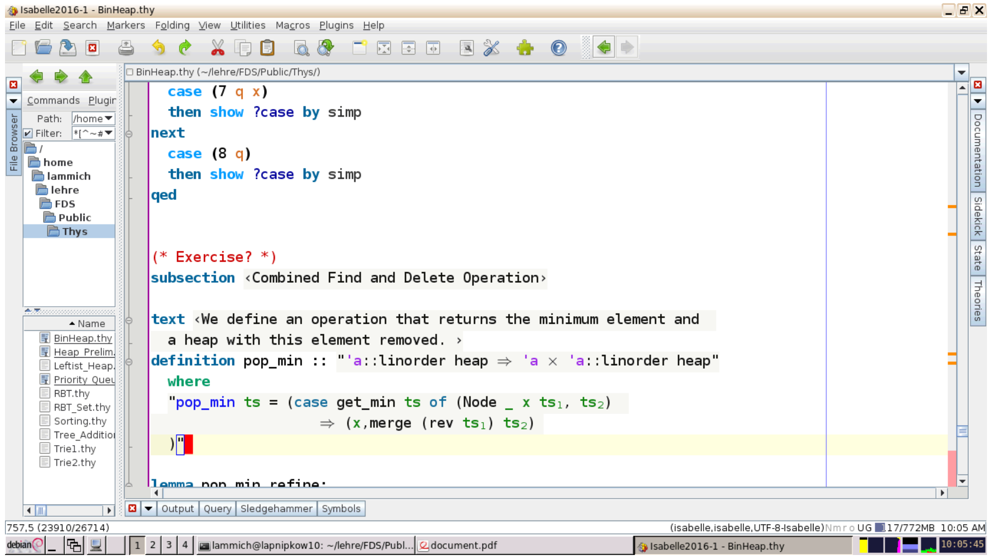

Isabelle2016-1 - BinHeap.thy
lemma l_merge_estimate:
"length (merge ts₁ ts₂) + t_merge ts₁ ts₂ ≤ 2 * (length ts₁ + length ts₂) + 1"
apply (induction ts₁ ts₂ rule: t_merge.induct)
apply (auto simp: l_ins_tree_estimate algebra_simps)
done
text ‹Finally, we get the desired logarithmic bound›
lemma t_merge_bound_aux:
fixes ts₁ ts₂
defines "n₁ ≡ size (mset_heap ts₁)"
defines "n₂ ≡ size (mset_heap ts₂)"
assumes BINVARS: "bheap_invar ts₁" "bheap_invar ts₂"
shows "t_merge ts₁ ts₂ ≤ 4*log 2 (n₁ + n₂ + 1) + 2"
proof -
define n where "n = n₁ + n₂"
from l_merge_estimate[of ts₁ ts₂]
have "t_merge ts₁ ts₂ ≤ 2 * (length ts₁ + length ts₂) + 1" by auto


Isabelle2016-1 - BinHeap.thy
by (induction l arbitrary: s) auto
lemma t_fold_bounded_bound:
assumes "∀x s. x∈set l ⟶ t_f x s ≤ K"
shows "t_fold t_f f l s ≤ K*length l + 1"
using assms
apply (induction l arbitrary: s)
apply (simp; fail)
using add_mono
by (fastforce simp: algebra_simps)
thm rev_conv_fold -- ‹Theorem used by code generator›
definition "t_rev xs = t_fold (λ_ _. 1) op # xs []"
lemma t_rev_bound: "t_rev xs = length xs + 1"
unfolding t_rev_def t_fold_const_bound by auto
definition t_delete_min :: "'a::linorder heap ⇒ nat"
where
"t_delete_min ts = t_get_min ts + (case get_min ts of (Node _ x ts₁, ts₂)


Isabelle2016-1 - BinHeap.thy
shows "t_get_min ts ≤ log 2 (size (mset_heap ts) + 1)"
using assms t_get_min_bound_aux unfolding invar_def by blast
thm fold_simps -- ‹Theorems used by code generator›
fun t_fold :: "('a ⇒ 'b ⇒ nat) ⇒ ('a ⇒ 'b ⇒ 'b) ⇒ 'a list ⇒ 'b ⇒ nat"
where
"t_fold t_f f [] s = 1"
| "t_fold t_f f (x # xs) s = t_f x s + t_fold t_f f xs (f x s)"
text ‹Estimation for constant function is enough for our purpose›
lemma t_fold_const_bound:
shows "t_fold (λ_ _. K) f l s = K*length l + 1"
by (induction l arbitrary: s) auto
lemma t_fold_bounded_bound:
assumes "∀x s. x∈set l ⟶ t_f x s ≤ K"
shows "t_fold t_f f l s ≤ K*length l + 1"
using assms
apply (induction l arbitrary: s)
Matches line 632: by (fastforce simp: algebra_simps)


Isabelle2016-1 - BinHeap.thy
lemma t_delete_min_bound_aux:
fixes ts
defines "n ≡ size (mset_heap ts)"
assumes BINVAR: "bheap_invar ts"
assumes "ts≠[]"
shows "t_delete_min ts ≤ 6 * log 2 (n+1) + 3"
proof -
obtain r x ts₁ ts₂ where GM: "get_min ts = (Node r x ts₁, ts₂)"
by (metis surj_pair tree.exhaust_sel)
note BINVAR' = get_min_bheap_invar[OF GM ‹ts≠[]› BINVAR]
hence BINVAR1: "bheap_invar (rev ts₁)" by (blast intro: children_bheap_invar)
define n₁ where "n₁ = size (mset_heap ts₁)"
define n₂ where "n₂ = size (mset_heap ts₂)"
have t_rev_ts1_bound: "t_rev ts₁ ≤ 1 + log 2 (n+1)"
proof -
note t_rev_ts1_bound_aux[OF BINVAR1, simplified, folded n₁_def]


Isabelle2016-1 - BinHeap.thy
fixes ts
defines "n ≡ size (mset_heap ts)"
assumes "invar ts"
assumes "ts≠[]"
shows "t_delete_min ts ≤ 6 * log 2 (n+1) + 3"
using assms t_delete_min_bound_aux unfolding invar_def by blast
subsection ‹Instantiating the Priority Queue Locale›
interpretation binheap:
Priority_Queue "[]" "op = []" ins find_min delete_min invar mset_heap
proof (unfold_locales, goal_cases)
case 1
then show ?case by simp
next
case (2 q)
then show ?case by auto
next
case (3 q x)


Isabelle2016-1 - BinHeap.thy
case (7 q x)
then show ?case by simp
next
case (8 q)
then show ?case by simp
qed
(* Exercise? *)
subsection ‹Combined Find and Delete Operation›
text ‹We define an operation that returns the minimum element and
a heap with this element removed. ›
definition pop_min :: "'a::linorder heap ⇒ 'a × 'a::linorder heap"
where
"pop_min ts = (case get_min ts of (Node _ x ts₁, ts₂)
⇒ (x,merge (rev ts₁) ts₂)
)"