Amortized Complexity Verified

Tobias Nipkow

June 24, 2019

Abstract

A framework for the analysis of the amortized complexity of (functional) data structures is formalized in Isabelle/HOL and applied to a number of standard examples and to the following non-trivial ones: skew heaps, splay trees, splay heaps and pairing heaps. This work is described in [4] (except for pairing heaps). An extended version (including pairing heaps) is available online [5].

Contents

1 Amortized Complexity (Unary Operations)

theory *Amortized_Framework0*
imports *Complex_Main*
begin

This theory provides a simple amortized analysis framework where all operations act on a single data type, i.e. no union-like operations. This is the basis of the ITP 2015 paper by Nipkow. Although it is superseded by the model in *Amortized_Framework* that allows arbitrarily many parameters, it is still of interest because of its simplicity.

locale *Amortized* =
fixes *init* :: $'s$
fixes *nxt* :: $'o \Rightarrow 's \Rightarrow 's$
fixes *inv* :: $'s \Rightarrow bool$
fixes *t* :: $'o \Rightarrow 's \Rightarrow real$
fixes Φ :: $'s \Rightarrow real$
fixes *U* :: $'o \Rightarrow 's \Rightarrow real$
assumes *inv_init*: *inv init*
assumes *inv_nxt*: $inv\ s \Longrightarrow inv(nxt\ f\ s)$
assumes *ppos*: $inv\ s \Longrightarrow \Phi\ s \geq 0$
assumes *p0*: $\Phi\ init = 0$
assumes *U*: $inv\ s \Longrightarrow t\ f\ s + \Phi(nxt\ f\ s) - \Phi\ s \leq U\ f\ s$
begin

fun *state* :: $(nat \Rightarrow 'o) \Rightarrow nat \Rightarrow 's$ **where**
$state\ f\ 0 = init\ |$
$state\ f\ (Suc\ n) = nxt\ (f\ n)\ (state\ f\ n)$

lemma *inv_state*: $inv(state\ f\ n)$
by(*induction n*)(*simp_all add*: *inv_init inv_nxt*)

definition *a* :: $(nat \Rightarrow 'o) \Rightarrow nat \Rightarrow real$ **where**
$a\ f\ i = t\ (f\ i)\ (state\ f\ i) + \Phi(state\ f\ (i+1)) - \Phi(state\ f\ i)$

lemma *aeq*: $(\sum i{<}n.\ t\ (f\ i)\ (state\ f\ i)) = (\sum i{<}n.\ a\ f\ i) - \Phi(state\ f\ n)$
apply(*induction n*)
apply (*simp add*: *p0*)
apply (*simp add*: *a_def*)
done

corollary *ta*: $(\sum i{<}n.\ t\ (f\ i)\ (state\ f\ i)) \leq (\sum i{<}n.\ a\ f\ i)$
by (*metis add.commute aeq diff_add_cancel le_add_same_cancel2 ppos*[*OF inv_state*])

lemma *aa1*: *a f i* $\leq$ *U* (*f i*) (*state f i*)
by(*simp add*: *a_def U inv_state*)

lemma *ub*: ($\sum$ *i*<*n*. *t* (*f i*) (*state f i*)) $\leq$ ($\sum$ *i*<*n*. *U* (*f i*) (*state f i*))
by (*metis* (*mono_tags*) *aa1 order.trans sum_mono ta*)

end

1.1 Binary Counter

fun *incr* **where**
incr [] = [*True*] |
incr (*False*#*bs*) = *True* # *bs* |
incr (*True*#*bs*) = *False* # *incr bs*

fun t_{incr} :: *bool list* $\Rightarrow$ *real* **where**
t_{incr} [] = *1* |
t_{incr} (*False*#*bs*) = *1* |
t_{incr} (*True*#*bs*) = t_{incr} *bs* + *1*

definition *p_incr* :: *bool list* $\Rightarrow$ *real* (Φ_{incr}) **where**
Φ_{incr} *bs* = *length*(*filter id bs*)

lemma *a_incr*: t_{incr} *bs* + Φ_{incr}(*incr bs*) − Φ_{incr} *bs* = *2*
apply(*induction bs rule*: *incr.induct*)
apply (*simp_all add*: *p_incr_def*)
done

interpretation *incr*: *Amortized*
where *init* = [] **and** *nxt* = %_. *incr* **and** *inv* = λ_. *True*
and *t* = λ_. t_{incr} **and** Φ = Φ_{incr} **and** *U* = λ_ _. *2*
proof (*standard*, *goal_cases*)
  case *1* **show** *?case* **by** *simp*
next
  case *2* **show** *?case* **by** *simp*
next
  case *3* **show** *?case* **by**(*simp add*: *p_incr_def*)
next
  case *4* **show** *?case* **by**(*simp add*: *p_incr_def*)
next
  case *5* **show** *?case* **by**(*simp add*: *a_incr*)
qed

thm *incr.ub*

1.2 Dynamic tables: insert only

fun t_{ins} :: *nat* × *nat* ⇒ *real* **where**
t_{ins} (*n*,*l*) = (*if n*<*l then 1 else n*+*1*)

interpretation *ins*: *Amortized*
where *init* = (*0*::*nat*,*0*::*nat*)
and *nxt* = λ_ (*n*,*l*). (*n*+*1*, *if n*<*l then l else if l*=*0 then 1 else 2*∗*l*)
and *inv* = λ(*n*,*l*). *if l*=*0 then n*=*0 else n* ≤ *l* ∧ *l* < *2*∗*n*
and *t* = λ_. t_{ins} **and** Φ = λ(*n*,*l*). *2*∗*n* − *l* **and** *U* = λ_ _. *3*
proof (*standard*, *goal_cases*)
case *1* **show** *?case* **by** *auto*
next
case (*2 s*) **thus** *?case* **by**(*cases s*) *auto*
next
case (*3 s*) **thus** *?case* **by**(*cases s*)(*simp split*: *if_splits*)
next
case *4* **show** *?case* **by**(*simp*)
next
case (*5 s*) **thus** *?case* **by**(*cases s*) *auto*
qed

locale *table_insert* =
fixes *a* :: *real*
fixes *c* :: *real*
assumes *c1*[*arith*]: *c* > *1*
assumes *ac2*: *a* ≥ *c*/(*c* − *1*)
begin

lemma *ac*: *a* ≥ *1*/(*c* − *1*)
using *ac2* **by**(*simp add*: *field_simps*)

lemma *a0*[*arith*]: *a*>*0*
proof−
have *1*/(*c* − *1*) > *0* **using** *ac* **by** *simp*
thus *?thesis* **by** (*metis ac dual_order.strict_trans1*)
qed

definition *b* = *1*/(*c* − *1*)

lemma *b0*[*arith*]: *b* > *0*
using *ac* **by** (*simp add*: *b_def*)

```
fun ins :: nat * nat ⇒ nat * nat where
ins(n,l) = (n+1, if n<l then l else if l=0 then 1 else nat(ceiling(c*l)))

fun pins :: nat * nat => real where
pins(n,l) = a*n − b*l

interpretation ins: Amortized
where init = (0,0) and nxt = %_. ins
and inv = λ(n,l). if l=0 then n=0 else n ≤ l ∧ (b/a)*l ≤ n
and t = λ_. t_ins and Φ = pins and U = λ_ _. a + 1
proof (standard, goal_cases)
  case 1 show ?case by auto
next
  case (2 s)
  show ?case
  proof (cases s)
    case [simp]: (Pair n l)
    show ?thesis
    proof cases
      assume l=0 thus ?thesis using 2 ac
        by (simp add: b_def field_simps)
    next
      assume l≠0
      show ?thesis
      proof cases
        assume n<l
        thus ?thesis using 2 by(simp add: algebra_simps)
      next
        assume ¬ n<l
        hence [simp]: n=l using 2 ‹l≠0› by simp
        have 1: (b/a) * ceiling(c * l) ≤ real l + 1
        proof−
          have (b/a) * ceiling(c * l) = ceiling(c * l)/(a*(c − 1))
            by(simp add: b_def)
          also have ceiling(c * l) ≤ c*l + 1 by simp
          also have ... ≤ c*(real l+1) by (simp add: algebra_simps)
          also have ... / (a*(c − 1)) = (c/(a*(c − 1))) * (real l + 1) by
simp
          also have c/(a*(c − 1)) ≤ 1 using ac2 by (simp add: field_simps)
          finally show ?thesis by (simp add: divide_right_mono)
        qed
        have 2: real l + 1 ≤ ceiling(c * real l)
        proof−
```

```
        have real l + 1 = of_int(int(l)) + 1 by simp
        also have ... ≤ ceiling(c * real l) using ‹l ≠ 0›
          by(simp only: int_less_real_le[symmetric] less_ceiling_iff)
            (simp add: mult_less_cancel_right1)
        finally show ?thesis .
      qed
    from ‹l≠0› 1 2 show ?thesis by simp (simp add: not_le zero_less_mult_iff)
    qed
  qed
 qed
next
 case (3 s) thus ?case by(cases s)(simp add: field_simps split: if_splits)
next
 case 4 show ?case by(simp)
next
 case (5 s)
 show ?case
 proof (cases s)
  case [simp]: (Pair n l)
  show ?thesis
  proof cases
   assume l=0 thus ?thesis using 5 by (simp)
  next
   assume [arith]: l≠0
   show ?thesis
   proof cases
    assume n<l
    thus ?thesis using 5 ac by(simp add: algebra_simps b_def)
   next
    assume ¬ n<l
    hence [simp]: n=l using 5 by simp
    have t_ins s + pins (ins s) - pins s = l + a + 1 + (- b*ceiling(c*l))
+ b*l
     using ‹l≠0›
     by(simp add: algebra_simps less_trans[of -1::real 0])
     also have - b * ceiling(c*l) ≤ - b * (c*l) by (simp add: ceil-
ing_correct)
    also have l + a + 1 + - b*(c*l) + b*l = a + 1 + l*(1 - b*(c -
1))
     by (simp add: algebra_simps)
    also have b*(c - 1) = 1 by(simp add: b_def)
    also have a + 1 + (real l)*(1 - 1) = a+1 by simp
    finally show ?thesis by simp
   qed
```

qed
qed
qed

thm *ins.ub*

end

1.3 Stack with multipop

datatype $'a\ op_{stk} = Push\ 'a \mid Pop\ nat$

fun $nxt_{stk} :: 'a\ op_{stk} \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
$nxt_{stk}\ (Push\ x)\ xs = x\ \#\ xs \mid$
$nxt_{stk}\ (Pop\ n)\ xs = drop\ n\ xs$

fun $t_{stk}:: 'a\ op_{stk} \Rightarrow 'a\ list \Rightarrow real$ **where**
$t_{stk}\ (Push\ x)\ xs = 1 \mid$
$t_{stk}\ (Pop\ n)\ xs = min\ n\ (length\ xs)$

interpretation *stack*: *Amortized*
where $init = []$ **and** $nxt = nxt_{stk}$ **and** $inv = \lambda\_.\ True$
and $t = t_{stk}$ **and** $\Phi = length$ **and** $U = \lambda f\ \_.\ case\ f\ of\ Push\ \_ \Rightarrow 2 \mid Pop\ \_ \Rightarrow 0$
proof (*standard*, *goal_cases*)
case *1* **show** *?case* **by** *auto*
next
case (*2 s*) **thus** *?case* **by**(*cases s*) *auto*
next
case *3* **thus** *?case* **by** *simp*
next
case *4* **show** *?case* **by**(*simp*)
next
case (*5 _ f*) **thus** *?case* **by** (*cases f*) *auto*
qed

1.4 Queue

See, for example, the book by Okasaki [6].

datatype $'a\ op_q = Enq\ 'a \mid Deq$

type_synonym $'a\ queue = 'a\ list * 'a\ list$

fun $nxt_q :: 'a\ op_q \Rightarrow 'a\ queue \Rightarrow 'a\ queue$ **where**
$nxt_q\ (Enq\ x)\ (xs,ys) = (x\#xs,ys)$ |
$nxt_q\ Deq\ (xs,ys) = (if\ ys = []\ then\ ([],\ tl(rev\ xs))\ else\ (xs,tl\ ys))$

fun $t_q :: 'a\ op_q \Rightarrow 'a\ queue \Rightarrow real$ **where**
$t_q\ (Enq\ x)\ (xs,ys) = 1$ |
$t_q\ Deq\ (xs,ys) = (if\ ys = []\ then\ length\ xs\ else\ 0)$

interpretation *queue*: *Amortized*
where $init = ([],[])$ **and** $nxt = nxt_q$ **and** $inv = \lambda\_.\ True$
and $t = t_q$ **and** $\Phi = \lambda(xs,ys).\ length\ xs$ **and** $U = \lambda f\ \_.\ case\ f\ of\ Enq\ \_ \Rightarrow 2 \mid Deq \Rightarrow 0$
proof (*standard*, *goal_cases*)
  case *1* **show** *?case* **by** *auto*
next
  case (*2 s*) **thus** *?case* **by**(*cases s*) *auto*
next
  case (*3 s*) **thus** *?case* **by**(*cases s*) *auto*
next
  case *4* **show** *?case* **by**(*simp*)
next
  case (*5 s f*) **thus** *?case*
    apply(*cases s*)
    apply(*cases f*)
    by *auto*
qed

fun $balance :: 'a\ queue \Rightarrow 'a\ queue$ **where**
$balance(xs,ys) = (if\ size\ xs \leq size\ ys\ then\ (xs,ys)\ else\ ([],\ ys\ @\ rev\ xs))$

fun $nxt\_q2 :: 'a\ op_q \Rightarrow 'a\ queue \Rightarrow 'a\ queue$ **where**
$nxt\_q2\ (Enq\ a)\ (xs,ys) = balance\ (a\#xs,ys)$ |
$nxt\_q2\ Deq\ (xs,ys) = balance\ (xs,\ tl\ ys)$

fun $t\_q2 :: 'a\ op_q \Rightarrow 'a\ queue \Rightarrow real$ **where**
$t\_q2\ (Enq\ \_)\ (xs,ys) = 1 + (if\ size\ xs + 1 \leq size\ ys\ then\ 0\ else\ size\ xs + 1 + size\ ys)$ |
$t\_q2\ Deq\ (xs,ys) = (if\ size\ xs \leq size\ ys - 1\ then\ 0\ else\ size\ xs + (size\ ys - 1))$

interpretation *queue2*: *Amortized*

where *init* = ([],[]) **and** *nxt* = *nxt_q2*
and *inv* = λ(*xs*,*ys*). *size xs* $\leq$ *size ys*
and *t* = *t_q2* **and** Φ = λ(*xs*,*ys*). *2* $*$ *size xs*
and *U* = λ*f* _. *case f of Enq* _ $\Rightarrow$ *3* | *Deq* $\Rightarrow$ *0*
proof (*standard*, *goal_cases*)
  case *1* **show** *?case* **by** *auto*
next
  case (*2 s f*) **thus** *?case* **by**(*cases s*) (*cases f*, *auto*)
next
  case (*3 s*) **thus** *?case* **by**(*cases s*) *auto*
next
  case *4* **show** *?case* **by**(*simp*)
next
  case (*5 s f*) **thus** *?case*
    apply(*cases s*)
    apply(*cases f*)
    by (*auto simp*: *split*: *prod.splits*)
qed

1.5 Dynamic tables: insert and delete

datatype op_{tb} = *Ins* | *Del*

fun nxt_{tb} :: op_{tb} $\Rightarrow$ *nat*$*$*nat* $\Rightarrow$ *nat*$*$*nat* **where**
nxt_{tb} *Ins* (*n*,*l*) = (*n*+*1*, *if n*<*l then l else if l*=*0 then 1 else 2*$*$*l*) |
nxt_{tb} *Del* (*n*,*l*) = (*n* $-$ *1*, *if n*=*1 then 0 else if 4*$*$(*n* $-$ *1*)<*l then l div 2 else l*)

fun t_{tb} :: op_{tb} $\Rightarrow$ *nat*$*$*nat* $\Rightarrow$ *real* **where**
t_{tb} *Ins* (*n*,*l*) = (*if n*<*l then 1 else n*+*1*) |
t_{tb} *Del* (*n*,*l*) = (*if n*=*1 then 1 else if 4*$*$(*n* $-$ *1*)<*l then n else 1*)

interpretation *tb*: *Amortized*
where *init* = (*0*,*0*) **and** *nxt* = nxt_{tb}
and *inv* = λ(*n*,*l*). *if l*=*0 then n*=*0 else n* $\leq$ *l* $\wedge$ *l* $\leq$ *4*$*$*n*
and *t* = t_{tb} **and** Φ = (λ(*n*,*l*). *if 2*$*$*n* < *l then l*/*2* $-$ *n else 2*$*$*n* $-$ *l*)
and *U* = λ*f* _. *case f of Ins* $\Rightarrow$ *3* | *Del* $\Rightarrow$ *2*
proof (*standard*, *goal_cases*)
  case *1* **show** *?case* **by** *auto*
next
  case (*2 s f*) **thus** *?case* **by**(*cases s*, *cases f*) (*auto split*: *if_splits*)
next
  case (*3 s*) **thus** *?case* **by**(*cases s*)(*simp split*: *if_splits*)
next

case *4* **show** *?case* **by**(*simp*)
next
case (*5 s f*) **thus** *?case* **apply**(*cases s*) **apply**(*cases f*)
by (*auto simp*: *field_simps*)
qed

end

2 Amortized Complexity Framework

theory *Amortized_Framework*
imports *Complex_Main*
begin

This theory provides a framework for amortized analysis.

datatype *'a rose_tree* = *T 'a 'a rose_tree list*

declare *length_Suc_conv* [*simp*]

locale *Amortized* =
fixes *arity* :: *'op* ⇒ *nat*
fixes *exec* :: *'op* ⇒ *'s list* ⇒ *'s*
fixes *inv* :: *'s* ⇒ *bool*
fixes *cost* :: *'op* ⇒ *'s list* ⇒ *nat*
fixes Φ :: *'s* ⇒ *real*
fixes *U* :: *'op* ⇒ *'s list* ⇒ *real*
assumes *inv_exec*: ⟦∀ *s* ∈ *set ss*. *inv s*; *length ss* = *arity f* ⟧ ⟹ *inv*(*exec f ss*)
assumes *ppos*: *inv s* ⟹ Φ *s* ≥ *0*
assumes *U*: ⟦ ∀ *s* ∈ *set ss*. *inv s*; *length ss* = *arity f* ⟧
⟹ *cost f ss* + Φ(*exec f ss*) − *sum_list* (*map* Φ *ss*) ≤ *U f ss*
begin

fun *wf* :: *'op rose_tree* ⇒ *bool* **where**
wf (*T f ts*) = (*length ts* = *arity f* ∧ (∀ *t* ∈ *set ts*. *wf t*))

fun *state* :: *'op rose_tree* ⇒ *'s* **where**
state (*T f ts*) = *exec f* (*map state ts*)

lemma *inv_state*: *wf ot* ⟹ *inv*(*state ot*)
by(*induction ot*)(*simp_all add*: *inv_exec*)

definition *acost* :: *'op* ⇒ *'s list* ⇒ *real* **where**
acost f ss = *cost f ss* + Φ (*exec f ss*) − *sum_list* (*map* Φ *ss*)

fun *acost_sum* :: *'op rose_tree* ⇒ *real* **where**
acost_sum (*T f ts*) = *acost f* (*map state ts*) + *sum_list* (*map acost_sum ts*)

fun *cost_sum* :: *'op rose_tree* ⇒ *real* **where**
cost_sum (*T f ts*) = *cost f* (*map state ts*) + *sum_list* (*map cost_sum ts*)

fun *U_sum* :: *'op rose_tree* ⇒ *real* **where**
U_sum (*T f ts*) = *U f* (*map state ts*) + *sum_list* (*map U_sum ts*)

lemma *t_sum_a_sum*: *wf ot* ⟹ *cost_sum ot* = *acost_sum ot* − Φ(*state ot*)
by (*induction ot*) (*auto simp*: *acost_def Let_def sum_list_subtractf cong*: *map_cong*)

corollary *t_sum_le_a_sum*: *wf ot* ⟹ *cost_sum ot* ≤ *acost_sum ot*
by (*metis add.commute t_sum_a_sum diff_add_cancel le_add_same_cancel2 ppos*[*OF inv_state*])

lemma *a_le_U*: ⟦ ∀ *s* ∈ *set ss*. *inv s*; *length ss* = *arity f* ⟧ ⟹ *acost f ss* ≤ *U f ss*
by(*simp add*: *acost_def U*)

lemma *u_sum_le_U_sum*: *wf ot* ⟹ *acost_sum ot* ≤ *U_sum ot*
proof(*induction ot*)
case (*T f ts*)
with *a_le_U*[*of map state ts f*] *sum_list_mono* **show** *?case*
by (*force simp*: *inv_state*)
qed

corollary *t_sum_le_U_sum*: *wf ot* ⟹ *cost_sum ot* ≤ *U_sum ot*
by (*blast intro*: *t_sum_le_a_sum u_sum_le_U_sum order.trans*)

end

hide_const *T*

Amortized2 supports the transfer of amortized analysis of one datatype (*Amortized arity exec inv cost* Φ *U* on type *'s*) to an implementation (primed identifiers on type *'t*). Function *hom* is assumed to be a homomorphism from *'t* to *'s*, not just w.r.t. *exec* but also *cost* and *U*. The assumptions about *inv'* are weaker than the obvious *inv'* = *inv* ∘ *hom*: the latter does not allow *inv* to be weaker than *inv'* (which we need in one application).

locale *Amortized2* = *Amortized arity exec inv cost* Φ *U*
for *arity* :: *'op* ⇒ *nat* **and** *exec* **and** *inv* :: *'s* ⇒ *bool* **and** *cost* Φ *U* +

fixes *exec′* :: *′op* ⇒ *′t list* ⇒ *′t*
fixes *inv′* :: *′t* ⇒ *bool*
fixes *cost′* :: *′op* ⇒ *′t list* ⇒ *nat*
fixes *U′* :: *′op* ⇒ *′t list* ⇒ *real*
fixes *hom* :: *′t* ⇒ *′s*
assumes *exec′*: ⟦∀ *s* ∈ *set ts*. *inv′ s*; *length ts* = *arity f* ⟧
⟹ *hom*(*exec′ f ts*) = *exec f* (*map hom ts*)
assumes *inv_exec′*: ⟦∀ *s* ∈ *set ss*. *inv′ s*; *length ss* = *arity f* ⟧
⟹ *inv′*(*exec′ f ss*)
assumes *inv_hom*: *inv′ t* ⟹ *inv* (*hom t*)
assumes *cost′*: ⟦∀ *s* ∈ *set ts*. *inv′ s*; *length ts* = *arity f* ⟧
⟹ *cost′ f ts* = *cost f* (*map hom ts*)
assumes *U′*: ⟦∀ *s* ∈ *set ts*. *inv′ s*; *length ts* = *arity f* ⟧
⟹ *U′ f ts* = *U f* (*map hom ts*)
begin

sublocale *A′*: *Amortized arity exec′ inv′ cost′* Φ *o hom U′*
proof (*standard*, *goal_cases*)
case *1* **thus** *?case* **by**(*simp add*: *exec′ inv_exec′ inv_exec*)
next
case *2* **thus** *?case* **by**(*simp add*: *inv_hom ppos*)
next
case *3* **thus** *?case*
by(*simp add*: *U exec′ U′ map_map*[*symmetric*] *cost′ inv_exec inv_hom del*: *map_map*)
qed

end

end

3 Simple Examples

theory *Amortized_Examples*
imports *Amortized_Framework*
begin

This theory applies the amortized analysis framework to a number of simple classical examples.

3.1 Binary Counter

locale *Bin_Counter*
begin

```
datatype op = Empty | Incr

fun arity :: op ⇒ nat where
arity Empty = 0 |
arity Incr = 1

fun incr :: bool list ⇒ bool list where
incr [] = [True] |
incr (False#bs) = True # bs |
incr (True#bs) = False # incr bs

fun t_incr :: bool list ⇒ nat where
t_incr [] = 1 |
t_incr (False#bs) = 1 |
t_incr (True#bs) = t_incr bs + 1

definition Φ :: bool list ⇒ real where
Φ bs = length(filter id bs)

lemma a_incr: t_incr bs + Φ(incr bs) − Φ bs = 2
apply(induction bs rule: incr.induct)
apply (simp_all add: Φ_def)
done

fun exec :: op ⇒ bool list list ⇒ bool list where
exec Empty [] = [] |
exec Incr [bs] = incr bs

fun cost :: op ⇒ bool list list ⇒ nat where
cost Empty _ = 1 |
cost Incr [bs] = t_incr bs

interpretation Amortized
where exec = exec and arity = arity and inv = λ_. True
and cost = cost and Φ = Φ and U = λf _. case f of Empty ⇒ 1 | Incr ⇒
2
proof (standard, goal_cases)
  case 1 show ?case by simp
next
  case 2 show ?case by(simp add: Φ_def)
next
  case 3 thus ?case using a_incr by(auto simp: Φ_def split: op.split)
qed
```

end

3.2 Stack with multipop

locale *Multipop*
begin

datatype *'a op = Empty | Push 'a | Pop nat*

fun *arity* :: *'a op* ⇒ *nat* **where**
arity Empty = 0 |
arity (*Push* _) = *1* |
arity (*Pop* _) = *1*

fun *exec* :: *'a op* ⇒ *'a list list* ⇒ *'a list* **where**
exec Empty [] = [] |
exec (*Push x*) [*xs*] = *x* # *xs* |
exec (*Pop n*) [*xs*] = *drop n xs*

fun *cost* :: *'a op* ⇒ *'a list list* ⇒ *nat* **where**
cost Empty _ = *1* |
cost (*Push x*) _ = *1* |
cost (*Pop n*) [*xs*] = *min n* (*length xs*)

interpretation *Amortized*
where *arity* = *arity* **and** *exec* = *exec* **and** *inv* = λ_. *True*
and *cost* = *cost* **and** Φ = *length*
and *U* = λ*f* _. *case f of Empty* ⇒ *1* | *Push* _ ⇒ *2* | *Pop* _ ⇒ *0*
proof (*standard*, *goal_cases*)
  case *1* **show** *?case* **by** *simp*
next
  case *2* **thus** *?case* **by** *simp*
next
  case *3* **thus** *?case* **by** (*auto split*: *op.split*)
qed

end

3.3 Dynamic tables: insert only

locale *Dyn_Tab1*
begin

```
type_synonym tab = nat × nat

datatype op = Empty | Ins

fun arity :: op ⇒ nat where
arity Empty = 0 |
arity Ins = 1

fun exec :: op ⇒ tab list ⇒ tab where
exec Empty [] = (0::nat,0::nat) |
exec Ins [(n,l)] = (n+1, if n<l then l else if l=0 then 1 else 2∗l)

fun cost :: op ⇒ tab list ⇒ nat where
cost Empty _ = 1 |
cost Ins [(n,l)] = (if n<l then 1 else n+1)

interpretation Amortized
where exec = exec and arity = arity
and inv = λ(n,l). if l=0 then n=0 else n ≤ l ∧ l < 2∗n
and cost = cost and Φ = λ(n,l). 2∗n − l
and U = λf _. case f of Empty ⇒ 1 | Ins ⇒ 3
proof (standard, goal_cases)
  case (1 _ f) thus ?case by(cases f) (auto split: if_splits)
next
  case 2 thus ?case by(auto split: prod.splits)
next
  case 3 thus ?case by (auto split: op.split) linarith
qed

end

locale Dyn_Tab2 =
fixes a :: real
fixes c :: real
assumes c1[arith]: c > 1
assumes ac2: a ≥ c/(c − 1)
begin

lemma ac: a ≥ 1/(c − 1)
using ac2 by(simp add: field_simps)

lemma a0[arith]: a>0
proof−
```

```
  have 1/(c − 1) > 0 using ac by simp
  thus ?thesis by (metis ac dual_order.strict_trans1)
qed

definition b = 1/(c − 1)

lemma b0[arith]: b > 0
using ac by (simp add: b_def)

type_synonym tab = nat × nat

datatype op = Empty | Ins

fun arity :: op ⇒ nat where
arity Empty = 0 |
arity Ins = 1

fun ins :: tab ⇒ tab where
ins(n,l) = (n+1, if n<l then l else if l=0 then 1 else nat(ceiling(c∗l)))

fun exec :: op ⇒ tab list ⇒ tab where
exec Empty [] = (0::nat,0::nat) |
exec Ins [s] = ins s |
exec _ _ = (0,0)

fun cost :: op ⇒ tab list ⇒ nat where
cost Empty _ = 1 |
cost Ins [(n,l)] = (if n<l then 1 else n+1)

fun Φ :: tab ⇒ real where
Φ(n,l) = a∗n − b∗l

interpretation Amortized
where exec = exec and arity = arity
and inv = λ(n,l). if l=0 then n=0 else n ≤ l ∧ (b/a)∗l ≤ n
and cost = cost and Φ = Φ and U = λf _. case f of Empty ⇒ 1 | Ins ⇒
a + 1
proof (standard, goal_cases)
  case (1 ss f)
  show ?case
  proof (cases f)
    case Empty thus ?thesis using 1 by auto
  next
    case [simp]: Ins
```

```
    obtain n l where [simp]: ss = [(n,l)] using 1(2) by (auto)
    show ?thesis
    proof cases
      assume l=0 thus ?thesis using 1 ac
        by (simp add: b_def field_simps)
    next
      assume l≠0
      show ?thesis
      proof cases
        assume n<l
        thus ?thesis using 1 by(simp add: algebra_simps)
      next
        assume ¬ n<l
        hence [simp]: n=l using 1 ‹l≠0› by simp
        have 1: (b/a) * ceiling(c * l) ≤ real l + 1
        proof−
          have (b/a) * ceiling(c * l) = ceiling(c * l)/(a*(c − 1))
            by(simp add: b_def)
          also have ceiling(c * l) ≤ c*l + 1 by simp
          also have ... ≤ c*(real l+1) by (simp add: algebra_simps)
          also have ... / (a*(c − 1)) = (c/(a*(c − 1))) * (real l + 1) by
simp
         also have c/(a*(c − 1)) ≤ 1 using ac2 by (simp add: field_simps)
          finally show ?thesis by (simp add: divide_right_mono)
        qed
        have 2: real l + 1 ≤ ceiling(c * real l)
        proof−
          have real l + 1 = of_int(int(l)) + 1 by simp
          also have ... ≤ ceiling(c * real l) using ‹l ≠ 0›
            by(simp only: int_less_real_le[symmetric] less_ceiling_iff)
              (simp add: mult_less_cancel_right1)
          finally show ?thesis .
        qed
      from ‹l≠0› 1 2 show ?thesis by simp (simp add: not_le zero_less_mult_iff)
      qed
    qed
  qed
next
  case 2 thus ?case by(auto simp: field_simps split: if_splits prod.splits)
next
  case (3 ss f)
  show ?case
  proof (cases f)
    case Empty thus ?thesis using 3(2) by simp
```

```
  next
    case [simp]: Ins
    obtain n l where [simp]: ss = [(n,l)] using 3(2) by (auto)
    show ?thesis
    proof cases
      assume l=0 thus ?thesis using 3 by (simp)
    next
      assume [arith]: l≠0
      show ?thesis
      proof cases
        assume n<l
        thus ?thesis using 3 ac by(simp add: algebra_simps b_def)
      next
        assume ¬ n<l
        hence [simp]: n=l using 3 by simp
         have cost Ins [(n,l)] + Φ (ins (n,l)) − Φ(n,l) = l + a + 1 + (−
b∗ceiling(c∗l)) + b∗l
          using ‹l≠0›
          by(simp add: algebra_simps less_trans[of −1::real 0])
          also have − b ∗ ceiling(c∗l) ≤ − b ∗ (c∗l) by (simp add: ceil-
ing_correct)
        also have l + a + 1 + − b∗(c∗l) + b∗l = a + 1 + l∗(1 − b∗(c −
1))
          by (simp add: algebra_simps)
        also have b∗(c − 1) = 1 by(simp add: b_def)
        also have a + 1 + (real l)∗(1 − 1) = a+1 by simp
        finally show ?thesis by simp
      qed
    qed
  qed
qed

end
```

3.4 Dynamic tables: insert and delete

```
locale Dyn_Tab3
begin

type_synonym tab = nat × nat

datatype op = Empty | Ins | Del

fun arity :: op ⇒ nat where
```

```
arity Empty = 0 |
arity Ins = 1 |
arity Del = 1

fun exec :: op ⇒ tab list ⇒ tab where
exec Empty [] = (0::nat,0::nat) |
exec Ins [(n,l)] = (n+1, if n<l then l else if l=0 then 1 else 2∗l) |
exec Del [(n,l)] = (n−1, if n≤1 then 0 else if 4∗(n − 1)<l then l div 2 else
l)

fun cost :: op ⇒ tab list ⇒ nat where
cost Empty _ = 1 |
cost Ins [(n,l)] = (if n<l then 1 else n+1) |
cost Del [(n,l)] = (if n≤1 then 1 else if 4∗(n − 1)<l then n else 1)

interpretation Amortized
where arity = arity and exec = exec
and inv = λ(n,l). if l=0 then n=0 else n ≤ l ∧ l ≤ 4∗n
and cost = cost and Φ = (λ(n,l). if 2∗n < l then l/2 − n else 2∗n − l)
and U = λf _. case f of Empty ⇒ 1 | Ins ⇒ 3 | Del ⇒ 2
proof (standard, goal_cases)
  case (1 _ f) thus ?case by (cases f) (auto split: if_splits)
next
  case 2 thus ?case by(auto split: prod.splits)
next
  case (3 _ f) thus ?case
    by (cases f)(auto simp: field_simps split: prod.splits)
qed

end
```

3.5 Queue

See, for example, the book by Okasaki [6].

```
locale Queue
begin

datatype 'a op = Empty | Enq 'a | Deq

type_synonym 'a queue = 'a list ∗ 'a list

fun arity :: 'a op ⇒ nat where
arity Empty = 0 |
arity (Enq _) = 1 |
```

```
arity Deq = 1

fun exec :: 'a op ⇒ 'a queue list ⇒ 'a queue where
exec Empty [] = ([],[]) |
exec (Enq x) [(xs,ys)] = (x#xs,ys) |
exec Deq [(xs,ys)] = (if ys = [] then ([], tl(rev xs)) else (xs,tl ys))

fun cost :: 'a op ⇒ 'a queue list ⇒ nat where
cost Empty _ = 0 |
cost (Enq x) [(xs,ys)] = 1 |
cost Deq [(xs,ys)] = (if ys = [] then length xs else 0)

interpretation Amortized
where arity = arity and exec = exec and inv = λ_. True
and cost = cost and Φ = λ(xs,ys). length xs
and U = λf _. case f of Empty ⇒ 0 | Enq _ ⇒ 2 | Deq ⇒ 0
proof (standard, goal_cases)
  case 1 show ?case by simp
next
  case 2 thus ?case by (auto split: prod.splits)
next
  case 3 thus ?case by(auto split: op.split)
qed

end

locale Queue2
begin

datatype 'a op = Empty | Enq 'a | Deq

type_synonym 'a queue = 'a list ∗ 'a list

fun arity :: 'a op ⇒ nat where
arity Empty = 0 |
arity (Enq _) = 1 |
arity Deq = 1

fun adjust :: 'a queue ⇒ 'a queue where
adjust(xs,ys) = (if ys = [] then ([], rev xs) else (xs,ys))

fun exec :: 'a op ⇒ 'a queue list ⇒ 'a queue where
exec Empty [] = ([],[]) |
exec (Enq x) [(xs,ys)] = adjust(x#xs,ys) |
```

```
exec Deq [(xs,ys)] = adjust (xs, tl ys)

fun cost :: 'a op ⇒ 'a queue list ⇒ nat where
cost Empty _ = 0 |
cost (Enq x) [(xs,ys)] = 1 + (if ys = [] then size xs + 1 else 0) |
cost Deq [(xs,ys)] = (if tl ys = [] then size xs else 0)

interpretation Amortized
where arity = arity and exec = exec
and inv = λ_. True
and cost = cost and Φ = λ(xs,ys). size xs
and U = λf _. case f of Empty ⇒ 0 | Enq _ ⇒ 2 | Deq ⇒ 0
proof (standard, goal_cases)
  case (1 _ f) thus ?case by (cases f) (auto split: if_splits)
next
  case 2 thus ?case by (auto)
next
  case (3 _ f) thus ?case by(cases f) (auto split: if_splits)
qed

end

locale Queue3
begin

datatype 'a op = Empty | Enq 'a | Deq

type_synonym 'a queue = 'a list * 'a list

fun arity :: 'a op ⇒ nat where
arity Empty = 0 |
arity (Enq _) = 1 |
arity Deq = 1

fun balance :: 'a queue ⇒ 'a queue where
balance(xs,ys) = (if size xs ≤ size ys then (xs,ys) else ([], ys @ rev xs))

fun exec :: 'a op ⇒ 'a queue list ⇒ 'a queue where
exec Empty [] = ([],[]) |
exec (Enq x) [(xs,ys)] = balance(x#xs,ys) |
exec Deq [(xs,ys)] = balance (xs, tl ys)

fun cost :: 'a op ⇒ 'a queue list ⇒ nat where
cost Empty _ = 0 |
```

cost (*Enq x*) [(*xs*,*ys*)] = *1* + (*if size xs* + *1* ≤ *size ys then 0 else size xs* + *1* + *size ys*) |
cost Deq [(*xs*,*ys*)] = (*if size xs* ≤ *size ys* − *1 then 0 else size xs* + (*size ys* − *1*))

interpretation *Amortized*
where *arity* = *arity* **and** *exec* = *exec*
and *inv* = λ(*xs*,*ys*). *size xs* ≤ *size ys*
and *cost* = *cost* **and** Φ = λ(*xs*,*ys*). *2* ∗ *size xs*
and *U* = λ*f* _. *case f of Empty* ⇒ *0* | *Enq* _ ⇒ *3* | *Deq* ⇒ *0*
proof (*standard*, *goal_cases*)
  case (*1* _ *f*) **thus** *?case* **by** (*cases f*) (*auto split*: *if_splits*)
next
  case *2* **thus** *?case* **by** (*auto*)
next
  case (*3* _ *f*) **thus** *?case* **by**(*cases f*) (*auto split*: *prod.splits*)
qed

end

end
theory *Priority_Queue_ops_merge*
imports *Main*
begin

datatype *'a op* = *Empty* | *Insert 'a* | *Del_min* | *Merge*

fun *arity* :: *'a op* ⇒ *nat* **where**
arity Empty = *0* |
arity (*Insert* _) = *1* |
arity Del_min = *1* |
arity Merge = *2*

end

4 Skew Heap Analysis

theory *Skew_Heap_Analysis*
imports
  Complex_Main
  Skew_Heap.Skew_Heap
  Amortized_Framework
  Priority_Queue_ops_merge

begin

The following proof is a simplified version of the one by Kaldewaij and Schoenmakers [3].

right-heavy:

definition *rh* :: *'a tree => 'a tree => nat* **where**
rh l r = (*if size l < size r then 1 else 0*)

Function Γ in [3]: number of right-heavy nodes on left spine.

fun *lrh* :: *'a tree ⇒ nat* **where**
lrh Leaf = *0* |
lrh (*Node l _ r*) = *rh l r* + *lrh l*

Function Δ in [3]: number of not-right-heavy nodes on right spine.

fun *rlh* :: *'a tree ⇒ nat* **where**
rlh Leaf = *0* |
rlh (*Node l _ r*) = (*1* − *rh l r*) + *rlh r*

lemma *Gexp*: *2* ^ *lrh h* ≤ *size h* + *1*
by (*induction h*) (*auto simp*: *rh_def*)

corollary *Glog*: *lrh h* ≤ *log 2* (*size1 h*)
by (*metis Gexp le_log2_of_power size1_size*)

lemma *Dexp*: *2* ^ *rlh h* ≤ *size h* + *1*
by (*induction h*) (*auto simp*: *rh_def*)

corollary *Dlog*: *rlh h* ≤ *log 2* (*size1 h*)
by (*metis Dexp le_log2_of_power size1_size*)

function *t_merge* :: *'a*::*linorder heap ⇒ 'a heap ⇒ nat* **where**
t_merge Leaf h = *1* |
t_merge h Leaf = *1* |
t_merge (*Node l1 a1 r1*) (*Node l2 a2 r2*) =
(*if a1* ≤ *a2 then t_merge* (*Node l2 a2 r2*) *r1 else t_merge* (*Node l1 a1 r1*) *r2*) + *1*
by *pat_completeness auto*
termination
by (*relation measure* (λ(*x*, *y*). *size x* + *size y*)) *auto*

fun Φ :: *'a heap ⇒ int* **where**
Φ *Leaf* = *0* |
Φ (*Node l _ r*) = Φ *l* + Φ *r* + *rh l r*

lemma Φ_nneg: $\Phi\ t \geq 0$
by (*induction t*) *auto*

lemma *plus_log_le_2log_plus*: $[\![\ x > 0;\ y > 0;\ b > 1\ ]\!]$
$\Longrightarrow$ *log b x* + *log b y* $\leq$ *2* $*$ *log b* $(x + y)$
by(*subst mult_2*; *rule add_mono*; *auto*)

lemma *rh1*: *rh l r* $\leq$ *1*
by(*simp add*: *rh_def*)

lemma *amor_le_long*:
t_merge t1 t2 + Φ (*merge t1 t2*) $-$ Φ *t1* $-$ Φ *t2* $\leq$
lrh(*merge t1 t2*) + *rlh t1* + *rlh t2* + *1*
proof (*induction t1 t2 rule*: *merge.induct*)
case *1* **thus** *?case* **by** *simp*
next
case *2* **thus** *?case* **by** *simp*
next
case (*3 l1 a1 r1 l2 a2 r2*)
show *?case*
proof (*cases a1* $\leq$ *a2*)
case *True*
let *?t1* = *Node l1 a1 r1* **let** *?t2* = *Node l2 a2 r2* **let** *?m* = *merge ?t2 r1*
have *t_merge ?t1 ?t2* + Φ (*merge ?t1 ?t2*) $-$ Φ *?t1* $-$ Φ *?t2*
= *t_merge ?t2 r1* + *1* + Φ *?m* + Φ *l1* + *rh ?m l1* $-$ Φ *?t1* $-$ Φ *?t2*
using *True* **by** (*simp*)
also have ... = *t_merge ?t2 r1* + *1* + Φ *?m* + *rh ?m l1* $-$ Φ *r1* $-$ *rh l1 r1* $-$ Φ *?t2*
by *simp*
also have ... $\leq$ *lrh ?m* + *rlh ?t2* + *rlh r1* + *rh ?m l1* + *2* $-$ *rh l1 r1*
using *3.IH*(*1*)[*OF True*] **by** *linarith*
also have ... = *lrh ?m* + *rlh ?t2* + *rlh r1* + *rh ?m l1* + *1* + (*1* $-$ *rh l1 r1*)
using *rh1*[*of l1 r1*] **by** (*simp*)
also have ... = *lrh ?m* + *rlh ?t2* + *rlh ?t1* + *rh ?m l1* + *1*
by (*simp*)
also have ... = *lrh* (*merge ?t1 ?t2*) + *rlh ?t1* + *rlh ?t2* + *1*
using *True* **by**(*simp*)
finally show *?thesis* **.**
next
case *False* **with** *3* **show** *?thesis* **by** *auto*
qed

qed

lemma *amor_le*:
t_merge t1 t2 + Φ (*merge t1 t2*) − Φ *t1* − Φ *t2* ≤
lrh(*merge t1 t2*) + *rlh t1* + *rlh t2* + *1*
by(*induction t1 t2 rule*: *merge.induct*)(*auto*)

lemma *a_merge*:
t_merge t1 t2 + Φ(*merge t1 t2*) − Φ *t1* − Φ *t2* ≤
3 ∗ *log 2* (*size1 t1* + *size1 t2*) + *1* (**is** *?l* ≤ _)
proof −
have *?l* ≤ *lrh*(*merge t1 t2*) + *rlh t1* + *rlh t2* + *1* **using** *amor_le*[*of t1 t2*] **by** *arith*
also have ... = *real*(*lrh*(*merge t1 t2*)) + *rlh t1* + *rlh t2* + *1* **by** *simp*
also have ... = *real*(*lrh*(*merge t1 t2*)) + (*real*(*rlh t1*) + *rlh t2*) + *1* **by** *simp*
also have *rlh t1* ≤ *log 2* (*size1 t1*) **by**(*rule Dlog*)
also have *rlh t2* ≤ *log 2* (*size1 t2*) **by**(*rule Dlog*)
also have *lrh* (*merge t1 t2*) ≤ *log 2* (*size1*(*merge t1 t2*)) **by**(*rule Glog*)
also have *size1*(*merge t1 t2*) = *size1 t1* + *size1 t2* − *1* **by**(*simp add*: *size1_size*)
also have *log 2* (*size1 t1* + *size1 t2* − *1*) ≤ *log 2* (*size1 t1* + *size1 t2*) **by**(*simp add*: *size1_size*)
also have *log 2* (*size1 t1*) + *log 2* (*size1 t2*) ≤ *2* ∗ *log 2* (*real*(*size1 t1*) + (*size1 t2*))
by(*rule plus_log_le_2log_plus*) (*auto simp*: *size1_size*)
finally show *?thesis* **by**(*simp*)
qed

definition *t_insert* :: *'a*::*linorder* ⇒ *'a heap* ⇒ *int* **where**
t_insert a h = *t_merge* (*Node Leaf a Leaf*) *h* + *1*

lemma *a_insert*: *t_insert a h* + Φ(*Skew_Heap.insert a h*) − Φ *h* ≤ *3* ∗ *log 2* (*size1 h* + *2*) + *2*
using *a_merge*[*of Node Leaf a Leaf h*]
by (*simp add*: *numeral_eq_Suc t_insert_def Skew_Heap.insert_def rh_def*)

definition *t_del_min* :: (*'a*::*linorder*) *heap* ⇒ *int* **where**
t_del_min h = (*case h of Leaf* ⇒ *1* | *Node t1 a t2* ⇒ *t_merge t1 t2* + *1*)

lemma *a_del_min*: *t_del_min h* + Φ(*del_min h*) − Φ *h* ≤ *3* ∗ *log 2* (*size1 h* + *2*) + *2*
proof (*cases h*)
case *Leaf* **thus** *?thesis* **by** (*simp add*: *t_del_min_def*)

```
next
  case (Node t1 _ t2)
  have [arith]: log 2 (2 + (real (size t1) + real (size t2))) ≤
                log 2 (4 + (real (size t1) + real (size t2))) by simp
  from Node show ?thesis using a_merge[of t1 t2]
    by (simp add: size1_size t_del_min_def rh_def)
qed
```

4.0.1 Instantiation of Amortized Framework

```
lemma t_merge_nneg: t_merge h1 h2 ≥ 0
by(induction h1 h2 rule: t_merge.induct) auto

fun exec :: 'a::linorder op ⇒ 'a heap list ⇒ 'a heap where
exec Empty [] = Leaf |
exec (Insert a) [h] = Skew_Heap.insert a h |
exec Del_min [h] = del_min h |
exec Merge [h1,h2] = merge h1 h2

fun cost :: 'a::linorder op ⇒ 'a heap list ⇒ nat where
cost Empty [] = 1 |
cost (Insert a) [h] = t_merge (Node Leaf a Leaf) h |
cost Del_min [h] = (case h of Leaf ⇒ 1 | Node t1 a t2 ⇒ t_merge t1 t2) |
cost Merge [h1,h2] = t_merge h1 h2

fun U where
U Empty [] = 1 |
U (Insert _) [h] = 3 * log 2 (size1 h + 2) + 1 |
U Del_min [h] = 3 * log 2 (size1 h + 2) + 3 |
U Merge [h1,h2] = 3 * log 2 (size1 h1 + size1 h2) + 1

interpretation Amortized
where arity = arity and exec = exec and inv = λ_. True
and cost = cost and Φ = Φ and U = U
proof (standard, goal_cases)
  case 1 show ?case by simp
next
  case (2 h) show ?case using Φ_nneg[of h] by linarith
next
  case (3 ss f)
  show ?case
  proof (cases f)
    case Empty thus ?thesis using 3(2) by (auto)
  next
```

```
    case [simp]: (Insert a)
    obtain h where [simp]: ss = [h] using 3(2) by (auto)
    thus ?thesis using a_merge[of Node Leaf a Leaf h]
      by (simp add: numeral_eq_Suc insert_def rh_def t_merge_nneg)
  next
    case [simp]: Del_min
    obtain h where [simp]: ss = [h] using 3(2) by (auto)
    thus ?thesis
    proof (cases h)
      case Leaf with Del_min show ?thesis by simp
    next
      case (Node t1 _ t2)
      have [arith]: log 2 (2 + (real (size t1) + real (size t2))) ≤
              log 2 (4 + (real (size t1) + real (size t2))) by simp
      from Del_min Node show ?thesis using a_merge[of t1 t2]
        by (simp add: size1_size t_merge_nneg)
    qed
  next
    case [simp]: Merge
     obtain h1 h2 where ss = [h1,h2] using 3(2) by (auto simp: nu-
meral_eq_Suc)
    thus ?thesis using a_merge[of h1 h2] by (simp add: t_merge_nneg)
  qed
qed

end
theory Lemmas_log
imports Complex_Main
begin

lemma ld_sum_inequality:
  assumes x > 0 y > 0
  shows   log 2 x + log 2 y + 2 ≤ 2 * log 2 (x + y)
proof −
  have 0: 0 ≤ (x−y)^2 using assms by(simp)
  have 2 powr (2 + log 2 x + log 2 y) = 4 * x * y using assms
    by(simp add: powr_add)
  also have 4*x*y ≤ (x+y)^2 using 0 by(simp add: algebra_simps nu-
meral_eq_Suc)
  also have ... = 2 powr (log 2 (x + y) * 2) using assms
    by(simp add: powr_powr[symmetric] powr_numeral)
  finally show ?thesis by (simp add: mult_ac)
qed
```

lemma *ld_ld_1_less*:
  $[\![x > 0; y > 0]\!] \Longrightarrow 1 + log\ 2\ x + log\ 2\ y < 2 * log\ 2\ (x+y)$
using *ld_sum_inequality*[*of x y*] **by** *linarith*

lemma *ld_le_2ld*:
  assumes $x \geq 0\ y \geq 0$ **shows** $log\ 2\ (1+x+y) \leq 1 + log\ 2\ (1+x) + log\ 2\ (1+y)$
proof −
  have *1*: $1+x+y \leq (x+1)*(y+1)$ **using** *assms*
    by(*simp add*: *algebra_simps*)
  show *?thesis*
    apply(*rule powr_le_cancel_iff*[*of 2*, *THEN iffD1*])
     apply *simp*
    using *assms 1* **by**(*simp add*: *powr_add algebra_simps*)
qed

lemma *ld_ld_less2*: **assumes** $x \geq 2\ y \geq 2$
  shows $1 + log\ 2\ x + log\ 2\ y \leq 2 * log\ 2\ (x + y - 1)$
proof−
  from *assms* **have** $2*x \leq x*x\ 2*y \leq y*y$ **by** *simp_all*
  hence *1*: $2 * x * y \leq (x + y - 1)\hat{}2$
    by(*simp add*: *numeral_eq_Suc algebra_simps*)
  show *?thesis*
    apply(*rule powr_le_cancel_iff*[*of 2*, *THEN iffD1*])
     apply *simp*
   using *assms 1* **by**(*simp add*: *powr_add log_powr*[*symmetric*] *powr_numeral*)
qed

end

5 Splay Tree

5.1 Basics

theory *Splay_Tree_Analysis_Base*
imports
  Lemmas_log
  Splay_Tree.Splay_Tree
begin

declare *size1_size*[*simp*]

abbreviation $\varphi\ t == log\ 2\ (size1\ t)$

```
fun Φ :: 'a tree ⇒ real where
Φ Leaf = 0 |
Φ (Node l a r) = Φ l + Φ r + φ (Node l a r)


fun t_splay :: 'a::linorder ⇒ 'a tree ⇒ nat where
t_splay a Leaf = 1 |
t_splay a (Node l b r) =
  (if a=b
   then 1
   else if a < b
        then case l of
          Leaf ⇒ 1 |
          Node ll c lr ⇒
            (if a=c then 1
             else if a < c then if ll = Leaf then 1 else t_splay a ll + 1
                  else if lr = Leaf then 1 else t_splay a lr + 1)
        else case r of
          Leaf ⇒ 1 |
          Node rl c rr ⇒
            (if a=c then 1
             else if a < c then if rl = Leaf then 1 else t_splay a rl + 1
                  else if rr = Leaf then 1 else t_splay a rr + 1))

lemma t_splay_simps[simp]:
  t_splay a (Node l a r) = 1
  a<b ⟹ t_splay a (Node Leaf b r) = 1
  a<b ⟹ t_splay a (Node (Node ll a lr) b r) = 1
  a<b ⟹ a<c ⟹ t_splay a (Node (Node ll c lr) b r) =
   (if ll = Leaf then 1 else t_splay a ll + 1)
  a<b ⟹ c<a ⟹ t_splay a (Node (Node ll c lr) b r) =
   (if lr = Leaf then 1 else t_splay a lr + 1)
  b<a ⟹ t_splay a (Node l b Leaf) = 1
  b<a ⟹ t_splay a (Node l b (Node rl a rr)) = 1
  b<a ⟹ a<c ⟹ t_splay a (Node l b (Node rl c rr)) =
  (if rl=Leaf then 1 else t_splay a rl + 1)
  b<a ⟹ c<a ⟹ t_splay a (Node l b (Node rl c rr)) =
  (if rr=Leaf then 1 else t_splay a rr + 1)
by auto

declare t_splay.simps(2)[simp del]

fun t_splay_max :: 'a::linorder tree ⇒ nat where
```

```
t_splay_max Leaf = 1 |
t_splay_max (Node l b Leaf) = 1 |
t_splay_max (Node l b (Node rl c rr)) = (if rr=Leaf then 1 else t_splay_max
rr + 1)

definition t_delete :: 'a::linorder ⇒ 'a tree ⇒ nat where
t_delete a t = (if t = Leaf then 0 else t_splay a t + (case splay a t of
  Node l x r ⇒
    if x=a then case l of Leaf ⇒ 0 | _ ⇒ t_splay_max l
    else 0))

lemma ex_in_set_tree: t ≠ Leaf ⟹ bst t ⟹
  ∃ a' ∈ set_tree t. splay a' t = splay a t ∧ t_splay a' t = t_splay a t
proof(induction a t rule: splay.induct)
  case (6 a b c ll)
  hence splay a ll ≠ Leaf by simp
  then obtain lll u llr where [simp]: splay a ll = Node lll u llr
    by (metis tree.exhaust)
  have b < c bst ll using 6.prems by auto
  from 6.IH[OF ⟨ll ≠ Leaf⟩ ⟨bst ll⟩]
  obtain e where e ∈ set_tree ll splay e ll = splay a ll t_splay e ll = t_splay
a ll
    by blast
  moreover hence e<b using 6.prems(2) by auto
  ultimately show ?case using ⟨a<c⟩ ⟨a<b⟩ ⟨b<c⟩ ⟨bst ll⟩ by force
next
  case (8 b a c lr)
  hence splay a lr ≠ Leaf by simp
  then obtain lrl u lrr where [simp]: splay a lr = Node lrl u lrr
    by (metis tree.exhaust)
  have b < c bst lr using 8.prems by auto
  from 8.IH[OF ⟨lr ≠ Leaf⟩ ⟨bst lr⟩]
  obtain e where e ∈ set_tree lr splay e lr = splay a lr t_splay e lr = t_splay
a lr
    by blast
  moreover hence b<e & e<c using 8.prems(2) by simp
  ultimately show ?case using ⟨a<c⟩ ⟨b<a⟩ ⟨b<c⟩ ⟨bst lr⟩ by force
next
  case (11 c a b rl)
  hence splay a rl ≠ Leaf by simp
  then obtain rll u rlr where [simp]: splay a rl = Node rll u rlr
    by (metis tree.exhaust)
  have c < b bst rl using 11.prems by auto
  from 11.IH[OF ⟨rl ≠ Leaf⟩ ⟨bst rl⟩]
```

obtain *e* **where** *e* ∈ *set_tree rl splay e rl* = *splay a rl t_splay e rl* = *t_splay a rl*
by *blast*
moreover hence *c*<*e* & *e*<*b* **using** *11.prems* **by** *simp*
ultimately show *?case* **using** ⟨*c*<*a*⟩ ⟨*a*<*b*⟩ ⟨*c*<*b*⟩ ⟨*bst rl*⟩ **by** *force*
next
case (*14 c a b rr*)
hence *splay a rr* ≠ *Leaf* **by** *simp*
then obtain *rrl u rrr* **where** [*simp*]: *splay a rr* = *Node rrl u rrr*
by (*metis tree.exhaust*)
have *c* < *b bst rr* **using** *14.prems* **by** *auto*
from *14.IH*[*OF* ⟨*rr* ≠ *Leaf*⟩ ⟨*bst rr*⟩]
obtain *e* **where** *e* ∈ *set_tree rr splay e rr* = *splay a rr t_splay e rr* = *t_splay a rr*
by *blast*
moreover hence *b*<*e* **using** *14.prems*(*2*) **by** *simp*
ultimately show *?case* **using** ⟨*c*<*a*⟩ ⟨*b*<*a*⟩ ⟨*c*<*b*⟩ ⟨*bst rr*⟩ **by** *force*
qed (*auto simp*: *le_less*)

datatype *'a op* = *Empty* | *Splay 'a* | *Insert 'a* | *Delete 'a*

fun *arity* :: *'a*::*linorder op* ⇒ *nat* **where**
arity Empty = *0* |
arity (*Splay a*) = *1* |
arity (*Insert a*) = *1* |
arity (*Delete a*) = *1*

fun *exec* :: *'a*::*linorder op* ⇒ *'a tree list* ⇒ *'a tree* **where**
exec Empty [] = *Leaf* |
exec (*Splay a*) [*t*] = *splay a t* |
exec (*Insert a*) [*t*] = *Splay_Tree.insert a t* |
exec (*Delete a*) [*t*] = *Splay_Tree.delete a t*

fun *cost* :: *'a*::*linorder op* ⇒ *'a tree list* ⇒ *nat* **where**
cost Empty [] = *1* |
cost (*Splay a*) [*t*] = *t_splay a t* |
cost (*Insert a*) [*t*] = *t_splay a t* |
cost (*Delete a*) [*t*] = *t_delete a t*

end

5.2 Splay Tree Analysis

theory *Splay_Tree_Analysis*
imports
Splay_Tree_Analysis_Base
Amortized_Framework
begin

5.2.1 Analysis of splay

definition *a_splay* :: *'a::linorder* ⇒ *'a tree* ⇒ *real* **where**
a_splay a t = *t_splay a t* + Φ(*splay a t*) − Φ *t*

lemma *a_splay_simps*[*simp*]: *a_splay a* (*Node l a r*) = *1*
a<*b* ⟹ *a_splay a* (*Node* (*Node ll a lr*) *b r*) = φ (*Node lr b r*) − φ (*Node ll a lr*) + *1*
b<*a* ⟹ *a_splay a* (*Node l b* (*Node rl a rr*)) = φ (*Node rl b l*) − φ (*Node rl a rr*) + *1*
by(*auto simp add*: *a_splay_def algebra_simps*)

The following lemma is an attempt to prove a generic lemma that covers both zig-zig cases. However, the lemma is not as nice as one would like. Hence it is used only once, as a demo. Ideally the lemma would involve function *a_splay*, but that is impossible because this involves *splay* and thus depends on the ordering. We would need a truly symmetric version of *splay* that takes the ordering as an explicit argument. Then we could define all the symmetric cases by one final equation *splay2* (<) *t* = *splay2* (*not* ∘ (<)) (*mirror t*). This would simplify the code and the proofs.

lemma *zig_zig*: **fixes** *lx x rx lb b rb a ra u lb1 lb2*
defines [*simp*]: *X* == *Node lx* (*x*) *rx* **defines**[*simp*]: *B* == *Node lb b rb*
defines [*simp*]: *t* == *Node B a ra* **defines** [*simp*]: *A'* == *Node rb a ra*
defines [*simp*]: *t'* == *Node lb1 u* (*Node lb2 b A'*)
assumes *hyps*: *lb* ≠ ⟨⟩ **and** *IH*: *t_splay x lb* + Φ *lb1* + Φ *lb2* − Φ *lb* ≤ *2* ∗ φ *lb* − *3* ∗ φ *X* + *1* **and**
prems: *size lb* = *size lb1* + *size lb2* + *1 X* ∈ *subtrees lb*
shows *t_splay x lb* + Φ *t'* − Φ *t* ≤ *3* ∗ (φ *t* − φ *X*)
proof −
define *B'* **where** [*simp*]: *B'* = *Node lb2 b A'*
have *t_splay x lb* + Φ *t'* − Φ *t* = *t_splay x lb* + Φ *lb1* + Φ *lb2* − Φ *lb* + φ *B'* + φ *A'* − φ *B*
using *prems*
by(*auto simp*: *a_splay_def size_if_splay algebra_simps in_set_tree_if split*: *tree.split*)
also have ... ≤ *2* ∗ φ *lb* + φ *B'* + φ *A'* − φ *B* − *3* ∗ φ *X* + *1*
using *IH prems*(*2*) **by**(*auto simp*: *algebra_simps*)

```
  also have ... ≤ φ lb + φ B' + φ A' − 3 * φ X + 1 by(simp)
  also have ... ≤ φ B' + 2 * φ t − 3 * φ X
    using prems ld_ld_1_less[of size1 lb size1 A']
    by(simp add: size_if_splay)
  also have ... ≤ 3 * φ t − 3 * φ X
    using prems by(simp add: size_if_splay)
  finally show ?thesis by simp
qed

lemma a_splay_ub: ⟦ bst t; Node lx x rx : subtrees t ⟧
  ⟹ a_splay x t ≤ 3 * (φ t − φ(Node lx x rx)) + 1
proof(induction x t rule: splay.induct)
  case 1 thus ?case by simp
next
  case 2 thus ?case by auto
next
  case 4 hence False by(fastforce dest: in_set_tree_if) thus ?case ..
next
  case 5 hence False by(fastforce dest: in_set_tree_if) thus ?case ..
next
  case 7 hence False by(fastforce dest: in_set_tree_if) thus ?case ..
next
  case 10 hence False by(fastforce dest: in_set_tree_if) thus ?case ..
next
  case 12 hence False by(fastforce dest: in_set_tree_if) thus ?case ..
next
  case 13 hence False by(fastforce dest: in_set_tree_if) thus ?case ..
next
  case (3 b a lb rb ra)
  let ?A = Node (Node lb b rb) a ra
  have b ∉ set_tree ra using 3.prems(1) by auto
  with 3 show ?case using
    log_le_cancel_iff[of 2 size1(Node rb a ra) size1 ?A]
    log_le_cancel_iff[of 2 size1(Node lb b rb) size1 ?A]
    by (auto simp: algebra_simps simp del:log_le_cancel_iff)
next
  case (9 a b la lb rb)
  let ?A = ⟨la, a, ⟨lb, b, rb⟩⟩
  have b ∉ set_tree la using 9.prems(1) by auto
  with 9 show ?case using
    log_le_cancel_iff[of 2 size1(Node la a lb) size1 ?A]
    log_le_cancel_iff[of 2 size1(Node lb b rb) size1 ?A]
    by (auto simp: algebra_simps simp del:log_le_cancel_iff)
next
```

case (*6 x b a lb rb ra*)
hence *0*: $x \notin set\_tree\ rb \wedge x \notin set\_tree\ ra$ **using** *6.prems*(*1*) **by** *auto*
hence *1*: $x \in set\_tree\ lb$ **using** *6.prems* ‹*x*<*b*› **by** (*auto*)
obtain *lu u ru* **where** *sp*: $splay\ x\ lb = Node\ lu\ u\ ru$
using *splay_not_Leaf*[*OF* ‹*lb* ≠ *Leaf*›] **by** *blast*
let $?X = Node\ lx\ x\ rx$ **let** $?B = Node\ lb\ b\ rb$ **let** $?A = Node\ ?B\ a\ ra$
let $?R = lb$ **let** $?R' = Node\ lu\ u\ ru$
let $?A' = Node\ rb\ a\ ra$ **let** $?B' = Node\ ru\ b\ ?A'$
have $a\_splay\ x\ ?A = a\_splay\ x\ ?R + \varphi\ ?B' + \varphi\ ?A' - \varphi\ ?B - \varphi\ ?R' + 1$
using *6.prems 1 sp*
by(*auto simp*: *a_splay_def size_if_splay algebra_simps split*: *tree.split*)
also have $\ldots \le 3 * \varphi\ ?R + \varphi\ ?B' + \varphi\ ?A' - \varphi\ ?B - \varphi\ ?R' - 3 * \varphi\ ?X + 2$
using *6 0* **by**(*auto simp*: *algebra_simps*)
also have $\ldots = 2 * \varphi\ ?R + \varphi\ ?B' + \varphi\ ?A' - \varphi\ ?B - 3 * \varphi\ ?X + 2$
using *sp* **by**(*simp add*: *size_if_splay*)
also have $\ldots \le \varphi\ ?R + \varphi\ ?B' + \varphi\ ?A' - 3 * \varphi\ ?X + 2$ **by**(*simp*)
also have $\ldots \le \varphi\ ?B' + 2 * \varphi\ ?A - 3 * \varphi\ ?X + 1$
using *sp ld_ld_1_less*[*of size1 ?R size1 ?A'*]
by(*simp add*: *size_if_splay*)
also have $\ldots \le 3 * \varphi\ ?A - 3 * \varphi\ ?X + 1$
using *sp* **by**(*simp add*: *size_if_splay*)
finally show *?case* **by** *simp*
next
case (*8 b x a rb lb ra*)
hence *0*: $x \notin set\_tree\ lb \wedge x \notin set\_tree\ ra$
using *8.prems*(*1*) ‹*x* < *a*› **by**(*auto*)
hence *1*: $x \in set\_tree\ rb$ **using** *8.prems* ‹*b*<*x*› ‹*x*<*a*› **by** (*auto*)
obtain *lu u ru* **where** *sp*: $splay\ x\ rb = Node\ lu\ u\ ru$
using *splay_not_Leaf*[*OF* ‹*rb* ≠ *Leaf*›] **by** *blast*
let $?X = Node\ lx\ x\ rx$ **let** $?B = Node\ lb\ b\ rb$ **let** $?A = Node\ ?B\ a\ ra$
let $?R = rb$ **let** $?R' = Node\ lu\ u\ ru$
let $?B' = Node\ lb\ b\ lu$ **let** $?A' = Node\ ru\ a\ ra$
have $a\_splay\ x\ ?A = a\_splay\ x\ ?R + \varphi\ ?B' + \varphi\ ?A' - \varphi\ ?B - \varphi\ ?R' + 1$
using *8.prems 1 sp*
by(*auto simp*: *a_splay_def size_if_splay algebra_simps split*: *tree.split*)
also have $\ldots \le 3 * \varphi\ ?R + \varphi\ ?B' + \varphi\ ?A' - \varphi\ ?B - \varphi\ ?R' - 3 * \varphi\ ?X + 2$
using *8 0* **by**(*auto simp*: *algebra_simps*)
also have $\ldots = 2 * \varphi\ rb + \varphi\ ?B' + \varphi\ ?A' - \varphi\ ?B - 3 * \varphi\ ?X + 2$
using *sp* **by**(*simp add*: *size_if_splay*)
also have $\ldots \le \varphi\ rb + \varphi\ ?B' + \varphi\ ?A' - 3 * \varphi\ ?X + 2$ **by**(*simp*)

```
  also have ... ≤ φ rb + 2 * φ ?A − 3 * φ ?X + 1
    using sp ld_ld_1_less[of size1 ?B' size1 ?A']
    by(simp add: size_if_splay)
  also have ... ≤ 3 * φ ?A − 3 * φ ?X + 1 by(simp)
  finally show ?case by simp
next
  case (11 a x b lb la rb)
  hence 0: x ∉ set_tree rb ∧ x ∉ set_tree la
    using 11.prems(1) ‹a<x› by (auto)
  hence 1: x ∈ set_tree lb using 11.prems ‹a<x› ‹x<b› by (auto)
  obtain lu u ru where sp: splay x lb = Node lu u ru
    using splay_not_Leaf[OF ‹lb ≠ Leaf›] by blast
  let ?X = Node lx x rx let ?B = Node lb b rb  let ?A = Node la a ?B
  let ?R = lb  let ?R' = Node lu u ru
  let ?B' = Node ru b rb  let ?A' = Node la a lu
  have a_splay x ?A = a_splay x ?R + φ ?B' + φ ?A' − φ ?B − φ ?R' +
1
    using 11.prems 1 sp
    by(auto simp: a_splay_def size_if_splay algebra_simps split: tree.split)
  also have ... ≤ 3 * φ ?R + φ ?B' + φ ?A' − φ ?B − φ ?R' − 3 * φ
?X + 2
    using 11 0 by(auto simp: algebra_simps)
  also have ... = 2 * φ ?R + φ ?B' + φ ?A' − φ ?B − 3 * φ ?X + 2
    using sp by(simp add: size_if_splay)
  also have ... ≤ φ ?R + φ ?B' + φ ?A' − 3 * φ ?X + 2 by(simp)
  also have ... ≤ φ ?R + 2 * φ ?A − 3 * φ ?X + 1
    using sp ld_ld_1_less[of size1 ?B' size1 ?A']
    by(simp add: size_if_splay algebra_simps)
  also have ... ≤ 3 * φ ?A − 3 * φ ?X + 1 by(simp)
  finally show ?case by simp
next
  case (14 a x b rb la lb)
  hence 0: x ∉ set_tree lb ∧ x ∉ set_tree la
    using 14.prems(1) ‹b<x› by(auto)
  hence 1: x ∈ set_tree rb using 14.prems ‹b<x› ‹a<x› by (auto)
  obtain lu u ru where sp: splay x rb = Node lu u ru
    using splay_not_Leaf[OF ‹rb ≠ Leaf›] by blast
  from zig_zig[of rb x ru lu lx rx _ b lb a la] 14 sp 0
  show ?case by(auto simp: a_splay_def size_if_splay algebra_simps)

qed

lemma a_splay_ub2: assumes bst t a : set_tree t
shows a_splay a t ≤ 3 * (φ t − 1) + 1
```

proof −
from *assms*(*2*) **obtain** *la ra* **where** *N*: *Node la a ra* : *subtrees t*
by (*metis set_treeE*)
have *a_splay a t* ≤ *3* ∗ (*φ t* − *φ*(*Node la a ra*)) + *1* **by**(*rule a_splay_ub*[*OF assms*(*1*) *N*])
also have ... ≤ *3* ∗ (*φ t* − *1*) + *1* **by**(*simp add*: *field_simps*)
finally show *?thesis* **.**
qed

lemma *a_splay_ub3*: **assumes** *bst t* **shows** *a_splay a t* ≤ *3* ∗ *φ t* + *1*
proof *cases*
assume *t* = *Leaf* **thus** *?thesis* **by**(*simp add*: *a_splay_def*)
next
assume *t* ≠ *Leaf*
from *ex_in_set_tree*[*OF this assms*] **obtain** *a′* **where**
a′: *a′* ∈ *set_tree t splay a′ t* = *splay a t t_splay a′ t* = *t_splay a t*
by *blast*
have [*arith*]: *log 2 2* > *0* **by** *simp*
show *?thesis* **using** *a_splay_ub2*[*OF assms a′*(*1*)] **by**(*simp add*: *a_splay_def a′ log_divide*)
qed

5.2.2 Analysis of insert

lemma *amor_insert*: **assumes** *bst t*
shows *t_splay a t* + Φ(*Splay_Tree.insert a t*) − Φ *t* ≤ *4* ∗ *log 2* (*size1 t*) + *2* (**is** *?l* ≤ *?r*)
proof *cases*
assume *t* = *Leaf* **thus** *?thesis* **by**(*simp*)
next
assume *t* ≠ *Leaf*
then obtain *l e r* **where** [*simp*]: *splay a t* = *Node l e r*
by (*metis tree.exhaust splay_Leaf_iff*)
let *?t* = *real*(*t_splay a t*)
let *?Plr* = Φ *l* + Φ *r* **let** *?Ps* = Φ *t*
let *?slr* = *real*(*size1 l*) + *real*(*size1 r*) **let** *?LR* = *log 2* (*1* + *?slr*)
have *opt*: *?t* + Φ (*splay a t*) − *?Ps* ≤ *3* ∗ *log 2* (*real* (*size1 t*)) + *1*
using *a_splay_ub3*[*OF* ‹*bst t*›, *simplified a_splay_def*, *of a*] **by** (*simp*)
from *less_linear*[*of e a*]
show *?thesis*
proof (*elim disjE*)
assume *e*=*a*
have *nneg*: *log 2* (*1* + *real* (*size t*)) ≥ *0* **by** *simp*
thus *?thesis* **using** ‹*t* ≠ *Leaf*› *opt* ‹*e*=*a*›

```
    apply(simp add: algebra_simps) using nneg by arith
  next
    let ?L = log 2 (real(size1 l) + 1)
    assume e<a hence e ≠ a by simp
    hence ?l = (?t + ?Plr − ?Ps) + ?L + ?LR
      using ⟨t ≠ Leaf⟩ ⟨e<a⟩ by(simp)
    also have ?t + ?Plr − ?Ps ≤ 2 * log 2 ?slr + 1
      using opt size_splay[of a t,symmetric] by(simp)
    also have ?L ≤ log 2 ?slr by(simp)
    also have ?LR ≤ log 2 ?slr + 1
    proof −
      have ?LR ≤ log 2 (2 * ?slr) by (simp add:)
      also have ... ≤ log 2 ?slr + 1
        by (simp add: log_mult del:distrib_left_numeral)
      finally show ?thesis .
    qed
    finally show ?thesis using size_splay[of a t,symmetric] by (simp)
  next
    let ?R = log 2 (2 + real(size r))
    assume a<e hence e ≠ a by simp
    hence ?l = (?t + ?Plr − ?Ps) + ?R + ?LR
      using ⟨t ≠ Leaf⟩ ⟨a<e⟩ by(simp)
    also have ?t + ?Plr − ?Ps ≤ 2 * log 2 ?slr + 1
      using opt size_splay[of a t,symmetric] by(simp)
    also have ?R ≤ log 2 ?slr by(simp)
    also have ?LR ≤ log 2 ?slr + 1
    proof −
      have ?LR ≤ log 2 (2 * ?slr) by (simp add:)
      also have ... ≤ log 2 ?slr + 1
        by (simp add: log_mult del:distrib_left_numeral)
      finally show ?thesis .
    qed
    finally show ?thesis using size_splay[of a t, symmetric] by simp
  qed
qed
```

5.2.3 Analysis of delete

```
definition a_splay_max :: 'a::linorder tree ⇒ real where
a_splay_max t = t_splay_max t + Φ(splay_max t) − Φ t

lemma a_splay_max_ub: ⟦ bst t; t ≠ Leaf ⟧ ⟹ a_splay_max t ≤ 3 * (φ t
− 1) + 1
proof(induction t rule: splay_max.induct)
```

```
  case 1 thus ?case by (simp)
next
  case (2 l b)
  thus ?case using one_le_log_cancel_iff[of 2 size1 l + 1]
    by (simp add: a_splay_max_def del: one_le_log_cancel_iff)
next
  case (3 l b rl c rr)
  show ?case
  proof cases
    assume rr = Leaf
    thus ?thesis
      using one_le_log_cancel_iff[of 2 1 + size1 rl]
        one_le_log_cancel_iff[of 2 1 + size1 l + size1 rl]
        log_le_cancel_iff[of 2 size1 l + size1 rl 1 + size1 l + size1 rl]
     by (auto simp: a_splay_max_def field_simps
           simp del: log_le_cancel_iff one_le_log_cancel_iff)
  next
    assume rr ≠ Leaf
    then obtain l' u r' where sp: splay_max rr = Node l' u r'
      using splay_max_Leaf_iff tree.exhaust by blast
    hence 1: size rr = size l' + size r' + 1
      using size_splay_max[of rr,symmetric] by(simp)
    let ?C = Node rl c rr  let ?B = Node l b ?C
    let ?B' = Node l b rl  let ?C' = Node ?B' c l'
    have a_splay_max ?B = a_splay_max rr + φ ?B' + φ ?C' − φ rr − φ
?C + 1 using 3.prems sp 1
      by(auto simp add: a_splay_max_def)
    also have ... ≤ 3 * (φ rr − 1) + φ ?B' + φ ?C' − φ rr − φ ?C + 2
      using 3 ‹rr ≠ Leaf› by auto
    also have ... = 2 * φ rr + φ ?B' + φ ?C' − φ ?C − 1 by simp
    also have ... ≤ φ rr + φ ?B' + φ ?C' − 1 by simp
    also have ... ≤ 2 * φ ?B + φ ?C' − 2
      using ld_ld_1_less[of size1 ?B' size1 rr] by(simp add:)
    also have ... ≤ 3 * φ ?B − 2 using 1 by simp
    finally show ?case by simp
  qed
qed

lemma a_splay_max_ub3: assumes bst t shows a_splay_max t ≤ 3 * φ t
+ 1
proof cases
  assume t = Leaf thus ?thesis by(simp add: a_splay_max_def)
next
  assume t ≠ Leaf
```

```
  have [arith]: log 2 2 > 0 by simp
  show ?thesis using a_splay_max_ub[OF assms ‹t ≠ Leaf›] by(simp add:
a_splay_max_def)
qed

lemma amor_delete: assumes bst t
shows t_delete a t + Φ(Splay_Tree.delete a t) − Φ t ≤ 6 ∗ log 2 (size1 t)
+ 2
proof (cases t)
  case Leaf thus ?thesis by(simp add: Splay_Tree.delete_def t_delete_def)
next
  case [simp]: (Node ls x rs)
  then obtain l e r where sp[simp]: splay a (Node ls x rs) = Node l e r
    by (metis tree.exhaust splay_Leaf_iff)
  let ?t = real(t_splay a t)
  let ?Plr = Φ l + Φ r  let ?Ps = Φ t
  let ?slr = real(size1 l) + real(size1 r) let ?LR = log 2 (1 + ?slr)
  let ?lslr = log 2 (real (size ls) + (real (size rs) + 2))
  have ?lslr ≥ 0 by simp
  have opt: ?t + Φ (splay a t) − ?Ps ≤ 3 ∗ log 2 (real (size1 t)) + 1
    using a_splay_ub3[OF ‹bst t›, simplified a_splay_def, of a]
    by (simp add: field_simps)
  show ?thesis
  proof (cases e=a)
    case False thus ?thesis
    using opt apply(simp add: Splay_Tree.delete_def t_delete_def field_simps)
      using ‹?lslr ≥ 0› by arith
  next
    case [simp]: True
    show ?thesis
    proof (cases l)
      case Leaf
      have 1: log 2 (real (size r) + 2) ≥ 0 by(simp)
      show ?thesis
         using Leaf opt apply(simp add: Splay_Tree.delete_def t_delete_def
field_simps)
        using 1 ‹?lslr ≥ 0› by arith
    next
      case (Node ll y lr)
      then obtain l' y' r' where [simp]: splay_max (Node ll y lr) = Node
l' y' r'
      using splay_max_Leaf_iff tree.exhaust by blast
      have bst l using bst_splay[OF ‹bst t›, of a] by simp
      have Φ r' ≥ 0 apply (induction r') by (auto)
```

```
    have optm: real(t_splay_max l) + Φ (splay_max l) − Φ l ≤ 3 ∗ φ l +
1
      using a_splay_max_ub3[OF ‹bst l›, simplified a_splay_max_def] by
(simp add: field_simps Node)
    have 1: log 2 (2+(real(size l')+real(size r))) ≤ log 2 (2+(real(size
l)+real(size r)))
      using size_splay_max[of l] Node by simp
    have 2: log 2 (2 + (real (size l') + real (size r'))) ≥ 0 by simp
    have 3: log 2 (size1 l' + size1 r) ≤ log 2 (size1 l' + size1 r') + log 2
?slr
      apply simp using 1 2 by arith
    have 4: log 2 (real(size ll) + (real(size lr) + 2)) ≤ ?lslr
      using size_if_splay[OF sp] Node by simp
    show ?thesis using add_mono[OF opt optm] Node 3
      apply(simp add: Splay_Tree.delete_def t_delete_def field_simps)
      using 4 ‹Φ r' ≥ 0› by arith
  qed
 qed
qed
```

5.2.4 Overall analysis

```
fun U where
U Empty [] = 1 |
U (Splay _) [t] = 3 ∗ log 2 (size1 t) + 1 |
U (Insert _) [t] = 4 ∗ log 2 (size1 t) + 2 |
U (Delete _) [t] = 6 ∗ log 2 (size1 t) + 2

interpretation Amortized
where arity = arity and exec = exec and inv = bst
and cost = cost and Φ = Φ and U = U
proof (standard, goal_cases)
 case (1 ss f) show ?case
 proof (cases f)
  case Empty thus ?thesis using 1 by auto
 next
  case (Splay a)
  then obtain t where ss = [t] bst t using 1 by auto
  with Splay bst_splay[OF ‹bst t›, of a] show ?thesis
   by (auto split: tree.split)
 next
  case (Insert a)
  then obtain t where ss = [t] bst t using 1 by auto
  with bst_splay[OF ‹bst t›, of a] Insert show ?thesis
```

by (*auto simp*: *splay_bstL*[*OF* ‹*bst t*›] *splay_bstR*[*OF* ‹*bst t*›] *split*: *tree.split*)
next
case (*Delete a*)
then obtain *t* **where** *ss* = [*t*] *bst t* **using** *1* **by** *auto*
with *1 Delete* **show** *?thesis* **by**(*simp add*: *bst_delete*)
qed
next
case (*2 t*) **thus** *?case* **by** (*induction t*) *auto*
next
case (*3 ss f*)
show *?case* (**is** *?l* ≤ *?r*)
proof(*cases f*)
case *Empty* **thus** *?thesis* **using** *3*(*2*) **by**(*simp add*: *a_splay_def*)
next
case (*Splay a*)
then obtain *t* **where** *ss* = [*t*] *bst t* **using** *3* **by** *auto*
thus *?thesis* **using** *Splay a_splay_ub3*[*OF* ‹*bst t*›] **by**(*simp add*: *a_splay_def*)
next
case [*simp*]: (*Insert a*)
then obtain *t* **where** [*simp*]: *ss* = [*t*] **and** *bst t* **using** *3* **by** *auto*
thus *?thesis* **using** *amor_insert*[*of t a*] **by** *auto*
next
case [*simp*]: (*Delete a*)
then obtain *t* **where** [*simp*]: *ss* = [*t*] **and** *bst t* **using** *3* **by** *auto*
thus *?thesis* **using** *amor_delete*[*of t a*] **by** *auto*
qed
qed

end

5.3 Splay Tree Analysis (Optimal)

theory *Splay_Tree_Analysis_Optimal*
imports
Splay_Tree_Analysis_Base
Amortized_Framework
HOL−Library.Sum_of_Squares
begin

This analysis follows Schoenmakers [7].

5.3.1 Analysis of splay

locale *Splay_Analysis* =

fixes α :: *real* **and** β :: *real*
assumes *a1*[*arith*]: $\alpha > 1$
assumes *A1*: $[\![1 \le x;\ 1 \le y;\ 1 \le z]\!] \Longrightarrow$
$(x+y) * (y+z)$ *powr* $\beta \le (x+y)$ *powr* $\beta * (x+y+z)$
assumes *A2*: $[\![1 \le l';\ 1 \le r';\ 1 \le lr;\ 1 \le r]\!] \Longrightarrow$
$\alpha * (l'+r') * (lr+r)$ *powr* $\beta * (lr+r'+r)$ *powr* β
$\le (l'+r')$ *powr* $\beta * (l'+lr+r')$ *powr* $\beta * (l'+lr+r'+r)$
assumes *A3*: $[\![1 \le l';\ 1 \le r';\ 1 \le ll;\ 1 \le r]\!] \Longrightarrow$
$\alpha * (l'+r') * (l'+ll)$ *powr* $\beta * (r'+r)$ *powr* β
$\le (l'+r')$ *powr* $\beta * (l'+ll+r')$ *powr* $\beta * (l'+ll+r'+r)$
begin

lemma *nl2*: $[\![\ ll \ge 1;\ lr \ge 1;\ r \ge 1\ ]\!] \Longrightarrow$
$log\ \alpha\ (ll + lr) + \beta * log\ \alpha\ (lr + r)$
$\le \beta * log\ \alpha\ (ll + lr) + log\ \alpha\ (ll + lr + r)$
apply(*rule powr_le_cancel_iff*[*THEN iffD1*, *OF a1*])
apply(*simp add*: *powr_add mult.commute*[*of* β] *powr_powr*[*symmetric*] *A1*)
done

definition φ :: *'a tree* $\Rightarrow$ *'a tree* $\Rightarrow$ *real* **where**
$\varphi\ t1\ t2 = \beta * log\ \alpha\ (size1\ t1 + size1\ t2)$

fun Φ :: *'a tree* $\Rightarrow$ *real* **where**
$\Phi\ Leaf = 0\ |$
$\Phi\ (Node\ l\ \_\ r) = \Phi\ l + \Phi\ r + \varphi\ l\ r$

definition *A* :: *'a*::*linorder* $\Rightarrow$ *'a tree* $\Rightarrow$ *real* **where**
$A\ a\ t = t\_splay\ a\ t + \Phi(splay\ a\ t) - \Phi\ t$

lemma *A_simps*[*simp*]: $A\ a\ (Node\ l\ a\ r) = 1$
$a<b \Longrightarrow A\ a\ (Node\ (Node\ ll\ a\ lr)\ b\ r) = \varphi\ lr\ r - \varphi\ lr\ ll + 1$
$b<a \Longrightarrow A\ a\ (Node\ l\ b\ (Node\ rl\ a\ rr)) = \varphi\ rl\ l - \varphi\ rr\ rl + 1$
by(*auto simp add*: *A_def* φ*_def algebra_simps*)

lemma *A_ub*: $[\![$ *bst t*; *Node la a ra* : *subtrees t* $]\!]$
$\Longrightarrow A\ a\ t \le log\ \alpha\ ((size1\ t)/(size1\ la + size1\ ra)) + 1$
proof(*induction a t rule*: *splay.induct*)
case *1* **thus** *?case* **by** *simp*
next
case *2* **thus** *?case* **by** *auto*
next
case *4* **hence** *False* **by**(*fastforce dest*: *in_set_tree_if*) **thus** *?case* **..**

next
case *5* **hence** *False* **by**(*fastforce dest*: *in_set_tree_if*) **thus** *?case* **..**
next
case *7* **hence** *False* **by**(*fastforce dest*: *in_set_tree_if*) **thus** *?case* **..**
next
case *10* **hence** *False* **by**(*fastforce dest*: *in_set_tree_if*) **thus** *?case* **..**
next
case *12* **hence** *False* **by**(*fastforce dest*: *in_set_tree_if*) **thus** *?case* **..**
next
case *13* **hence** *False* **by**(*fastforce dest*: *in_set_tree_if*) **thus** *?case* **..**
next
case (*3 b a lb rb ra*)
have *b* ∉ *set_tree ra* **using** *3.prems*(*1*) **by** *auto*
thus *?case* **using** *3.prems*(*1*,*2*) *nl2*[*of size1 lb size1 rb size1 ra*]
by (*auto simp*: *φ_def log_divide algebra_simps*)
next
case (*9 a b la lb rb*)
have *b* ∉ *set_tree la* **using** *9.prems*(*1*) **by** *auto*
thus *?case* **using** *9.prems*(*1*,*2*) *nl2*[*of size1 rb size1 lb size1 la*]
by (*auto simp add*: *φ_def log_divide algebra_simps*)
next
case (*6 x b a lb rb ra*)
hence *0*: *x* ∉ *set_tree rb* ∧ *x* ∉ *set_tree ra* **using** *6.prems*(*1*) **by** *auto*
hence *1*: *x* ∈ *set_tree lb* **using** *6.prems* ‹*x*<*b*› **by** (*auto*)
then obtain *lu u ru* **where** *sp*: *splay x lb* = *Node lu u ru*
using *6.prems*(*1*,*2*) **by**(*cases splay x lb*) *auto*
have *b* < *a* **using** *6.prems*(*1*,*2*) **by** (*auto*)
let *?lu* = *real* (*size1 lu*) **let** *?ru* = *real* (*size1 ru*)
let *?rb* = *real*(*size1 rb*) **let** *?r* = *real*(*size1 ra*)
have *1* + *log* α (*?lu* + *?ru*) + β ∗ *log* α (*?rb* + *?r*) + β ∗ *log* α (*?rb* + *?ru* + *?r*) ≤
β ∗ *log* α (*?lu* + *?ru*) + β ∗ *log* α (*?lu* + *?rb* + *?ru*) + *log* α (*?lu* + *?rb* + *?ru* + *?r*) (**is** *?L*≤*?R*)
proof(*rule powr_le_cancel_iff*[*THEN iffD1*, *OF a1*])
show α *powr ?L* ≤ α *powr ?R* **using** *A2*[*of ?lu ?ru ?rb ?r*]
by(*simp add*: *powr_add add_ac mult.commute*[*of* β] *powr_powr*[*symmetric*])
qed
thus *?case* **using** *6 0 1 sp*
by(*auto simp*: *A_def φ_def size_if_splay algebra_simps log_divide*)
next
case (*8 b x a rb lb ra*)
hence *0*: *x* ∉ *set_tree lb* ∧ *x* ∉ *set_tree ra* **using** *8.prems*(*1*) **by**(*auto*)
hence *1*: *x* ∈ *set_tree rb* **using** *8.prems* ‹*b*<*x*› ‹*x*<*a*› **by** (*auto*)
then obtain *lu u ru* **where** *sp*: *splay x rb* = *Node lu u ru*

```
    using 8.prems(1,2) by(cases splay x rb) auto
  let ?lu = real (size1 lu) let ?ru = real (size1 ru)
  let ?lb = real(size1 lb) let ?r = real(size1 ra)
  have 1 + log α (?lu + ?ru) + β * log α (?lu + ?lb) + β * log α (?ru +
?r) ≤
        β * log α (?lu + ?ru) + β * log α (?lu + ?lb + ?ru) + log α (?lu
+ ?lb + ?ru + ?r) (is ?L≤?R)
  proof(rule powr_le_cancel_iff[THEN iffD1, OF a1])
     show α powr ?L ≤ α powr ?R using A3[of ?lu ?ru ?lb ?r]
       by(simp add: powr_add mult.commute[of β] powr_powr[symmetric])
  qed
  thus ?case using 8 0 1 sp
    by(auto simp add: A_def size_if_splay φ_def log_divide algebra_simps)
next
  case (11 a x b lb la rb)
  hence 0: x ∉ set_tree rb ∧ x ∉ set_tree la using 11.prems(1) by (auto)
  hence 1: x ∈ set_tree lb using 11.prems ⟨a<x⟩ ⟨x<b⟩ by (auto)
  then obtain lu u ru where sp: splay x lb = Node lu u ru
    using 11.prems(1,2) by(cases splay x lb) auto
  let ?lu = real (size1 lu) let ?ru = real (size1 ru)
  let ?l = real(size1 la)  let ?rb = real(size1 rb)
  have 1 + log α (?lu + ?ru) + β * log α (?lu + ?l) + β * log α (?ru +
?rb) ≤
        β * log α (?lu + ?ru) + β * log α (?lu + ?ru + ?rb) + log α (?lu
+ ?l + ?ru + ?rb) (is ?L≤?R)
  proof(rule powr_le_cancel_iff[THEN iffD1, OF a1])
    show α powr ?L ≤ α powr ?R using A3[of ?ru ?lu ?rb ?l]
      by(simp add: powr_add mult.commute[of β] powr_powr[symmetric])
        (simp add: algebra_simps)
  qed
  thus ?case using 11 0 1 sp
    by(auto simp add: A_def size_if_splay φ_def log_divide algebra_simps)
next
  case (14 a x b rb la lb)
  hence 0: x ∉ set_tree lb ∧ x ∉ set_tree la using 14.prems(1) by(auto)
  hence 1: x ∈ set_tree rb using 14.prems ⟨a<x⟩ ⟨b<x⟩ by (auto)
  then obtain lu u ru where sp: splay x rb = Node lu u ru
    using 14.prems(1,2) by(cases splay x rb) auto
  let ?la = real(size1 la)  let ?lb = real(size1 lb)
  let ?lu = real (size1 lu) let ?ru = real (size1 ru)
  have 1 + log α (?lu + ?ru) + β * log α (?la + ?lb) + β * log α (?lu +
?la + ?lb) ≤
        β * log α (?lu + ?ru) + β * log α (?lu + ?lb + ?ru) + log α (?lu
+ ?lb + ?ru + ?la) (is ?L≤?R)
```

```
  proof(rule powr_le_cancel_iff[THEN iffD1, OF a1])
    show α powr ?L ≤ α powr ?R using A2[of ?ru ?lu ?lb ?la]
    by(simp add: powr_add add_ac mult.commute[of β] powr_powr[symmetric])
  qed
  thus ?case using 14 0 1 sp
   by(auto simp add: A_def size_if_splay φ_def log_divide algebra_simps)
qed

lemma A_ub2: assumes bst t a : set_tree t
shows A a t ≤ log α ((size1 t)/2) + 1
proof −
  from assms(2) obtain la ra where N: Node la a ra : subtrees t
    by (metis set_treeE)
  have A a t ≤ log α ((size1 t)/(size1 la + size1 ra)) + 1
    by(rule A_ub[OF assms(1) N])
  also have ... ≤ log α ((size1 t)/2) + 1 by(simp add: field_simps)
  finally show ?thesis by simp
qed

lemma A_ub3: assumes bst t shows A a t ≤ log α (size1 t) + 1
proof cases
  assume t = Leaf thus ?thesis by(simp add: A_def)
next
  assume t ≠ Leaf
  from ex_in_set_tree[OF this assms] obtain a' where
    a': a' ∈ set_tree t  splay a' t = splay a t  t_splay a' t = t_splay a t
    by blast
  have [arith]: log α 2 > 0 by simp
 show ?thesis using A_ub2[OF assms a'(1)] by(simp add: A_def a' log_divide)
qed


definition Am :: 'a::linorder tree ⇒ real where
Am t = t_splay_max t + Φ(splay_max t) − Φ t

lemma Am_simp3': ⟦ c<b; bst rr; rr ≠ Leaf⟧ ⟹
  Am (Node l c (Node rl b rr)) =
  (case splay_max rr of Node rrl _ rrr ⇒
   Am rr + φ rrl (Node l c rl) + φ l rl − φ rl rr − φ rrl rrr + 1)
by(auto simp: Am_def φ_def size_if_splay_max algebra_simps neq_Leaf_iff
split: tree.split)

lemma Am_ub: ⟦ bst t; t ≠ Leaf ⟧ ⟹ Am t ≤ log α ((size1 t)/2) + 1
proof(induction t rule: splay_max.induct)
```

```
  case 1 thus ?case by (simp)
next
  case 2 thus ?case by (simp add: Am_def)
next
  case (3 l b rl c rr)
  show ?case
  proof cases
    assume rr = Leaf
    thus ?thesis
     using nl2[of 1 size1 rl size1 l] log_le_cancel_iff[of α 2 2 + real(size rl)]
      by(auto simp: Am_def φ_def log_divide field_simps
           simp del: log_le_cancel_iff)
  next
    assume rr ≠ Leaf
    then obtain l' u r' where sp: splay_max rr = Node l' u r'
      using splay_max_Leaf_iff tree.exhaust by blast
    hence 1: size rr = size l' + size r' + 1
      using size_splay_max[of rr] by(simp)
    let ?l = real (size1 l) let ?rl = real (size1 rl)
    let ?l' = real (size1 l') let ?r' = real (size1 r')
    have 1 + log α (?l' + ?r') + β * log α (?l + ?rl) + β * log α (?l' +
?l + ?rl) ≤
      β * log α (?l' + ?r') + β * log α (?l' + ?rl + ?r') + log α (?l' + ?rl
+ ?r' + ?l) (is ?L≤?R)
    proof(rule powr_le_cancel_iff[THEN iffD1, OF a1])
      show α powr ?L ≤ α powr ?R using A2[of ?r' ?l' ?rl ?l]
     by(simp add: powr_add add.commute add.left_commute mult.commute[of
β] powr_powr[symmetric])
    qed
    thus ?case using 3 sp 1 ‹rr ≠ Leaf›
      by(auto simp add: Am_simp3' φ_def log_divide algebra_simps)
  qed
qed

lemma Am_ub3: assumes bst t shows Am t ≤ log α (size1 t) + 1
proof cases
  assume t = Leaf thus ?thesis by(simp add: Am_def)
next
  assume t ≠ Leaf
  have [arith]: log α 2 > 0 by simp
  show ?thesis using Am_ub[OF assms ‹t ≠ Leaf›] by(simp add: Am_def
log_divide)
qed
```

end

5.3.2 Optimal Interpretation

lemma *mult_root_eq_root*:
$n{>}0 \Longrightarrow y \geq 0 \Longrightarrow root\ n\ x * y = root\ n\ (x * (y\ \hat{}\ n))$
by(*simp add*: *real_root_mult real_root_pos2*)

lemma *mult_root_eq_root2*:
$n{>}0 \Longrightarrow y \geq 0 \Longrightarrow y * root\ n\ x = root\ n\ ((y\ \hat{}\ n) * x)$
by(*simp add*: *real_root_mult real_root_pos2*)

lemma *powr_inverse_numeral*:
$0 < x \Longrightarrow x\ powr\ (1\ /\ numeral\ n) = root\ (numeral\ n)\ x$
by (*simp add*: *root_powr_inverse*)

lemmas *root_simps* = *mult_root_eq_root mult_root_eq_root2 powr_inverse_numeral*

lemma *nl31*: $[\![\ (l'::real) \geq 1;\ r' \geq 1;\ lr \geq 1;\ r \geq 1\ ]\!] \Longrightarrow$
$4 * (l' + r') * (lr + r) \leq (l' + lr + r' + r)\ \hat{}\ 2$
by (*sos* (((*A*<*0* * *R*<*1*) + (*R*<*1* * (*R*<*1* * [*r* + ~*1**l'* + *lr* + ~*1***r'*] ^ *2*)))))

lemma *nl32*: **assumes** $(l'::real) \geq 1\ r' \geq 1\ lr \geq 1\ r \geq 1$
shows $4 * (l' + r') * (lr + r) * (lr + r' + r) \leq (l' + lr + r' + r)\ \hat{}\ 3$
proof −
have *1*: $lr + r' + r \leq l' + lr + r' + r$ **using** *assms* **by** *arith*
have *2*: $0 \leq (l' + lr + r' + r)\ \hat{}\ 2$ **by** (*rule zero_le_power2*)
have *3*: $0 \leq lr + r' + r$ **using** *assms* **by** *arith*
from *mult_mono*[*OF nl31*[*OF assms*] *1 2 3*] **show** *?thesis*
by(*simp add*: *ac_simps numeral_eq_Suc*)
qed

lemma *nl3*: **assumes** $(l'::real) \geq 1\ r' \geq 1\ lr \geq 1\ r \geq 1$
shows $4 * (l' + r')\ \hat{}\ 2 * (lr + r) * (lr + r' + r)$
$\leq (l' + lr + r') * (l' + lr + r' + r)\ \hat{}\ 3$
proof−
have *1*: $l' + r' \leq l' + lr + r'$ **using** *assms* **by** *arith*
have *2*: $0 \leq (l' + lr + r' + r)\ \hat{}\ 3$ **using** *assms* **by** *simp*
have *3*: $0 \leq l' + r'$ **using** *assms* **by** *arith*
from *mult_mono*[*OF nl32*[*OF assms*] *1 2 3*] **show** *?thesis*
unfolding *power2_eq_square* **by** (*simp only*: *ac_simps*)
qed

lemma *nl41*: **assumes** $(l'::real) \geq 1\ r' \geq 1\ ll \geq 1\ r \geq 1$
shows $4 * (l' + ll) * (r' + r) \leq (l' + ll + r' + r)\hat{}2$
using *assms* **by** (*sos* (((*A*<*0* * *R*<*1*) + (*R*<*1* * (*R*<*1* * [*r* + ~*1**l' + ~*1***ll* + *r'*]^*2*)))))

lemma *nl42*: **assumes** $(l'::real) \geq 1\ r' \geq 1\ ll \geq 1\ r \geq 1$
shows $4 * (l' + r') * (l' + ll) * (r' + r) \leq (l' + ll + r' + r)\hat{}3$
proof −
  have *1*: $l' + r' \leq l' + ll + r' + r$ **using** *assms* **by** *arith*
  have *2*: $0 \leq (l' + ll + r' + r)\hat{}2$ **by** (*rule zero_le_power2*)
  have *3*: $0 \leq l' + r'$ **using** *assms* **by** *arith*
  from *mult_mono*[*OF nl41*[*OF assms*] *1 2 3*] **show** *?thesis*
    by(*simp add*: *ac_simps numeral_eq_Suc del*: *distrib_right_numeral*)
qed

lemma *nl4*: **assumes** $(l'::real) \geq 1\ r' \geq 1\ ll \geq 1\ r \geq 1$
shows $4 * (l' + r')\hat{}2 * (l' + ll) * (r' + r)$
  $\leq (l' + ll + r') * (l' + ll + r' + r)\hat{}3$
proof−
  have *1*: $l' + r' \leq l' + ll + r'$ **using** *assms* **by** *arith*
  have *2*: $0 \leq (l' + ll + r' + r)\hat{}3$ **using** *assms* **by** *simp*
  have *3*: $0 \leq l' + r'$ **using** *assms* **by** *arith*
  from *mult_mono*[*OF nl42*[*OF assms*] *1 2 3*] **show** *?thesis*
    unfolding *power2_eq_square* **by** (*simp only*: *ac_simps*)
qed

lemma *cancel*: $x>(0::real) \Longrightarrow c * x\ \hat{}\ 2 * y * z \leq u * v \Longrightarrow c * x\ \hat{}\ 3 * y * z \leq x * u * v$
by(*simp add*: *power2_eq_square power3_eq_cube*)

interpretation *S34*: *Splay_Analysis root 3 4 1/3*
proof (*standard*, *goal_cases*)
  case *2* **thus** *?case*
    by(*simp add*: *root_simps*)
      (*auto simp*: *numeral_eq_Suc split_mult_pos_le intro*!: *mult_mono*)
next
  case *3* **thus** *?case* **by**(*simp add*: *root_simps cancel nl3*)
next
  case *4* **thus** *?case* **by**(*simp add*: *root_simps cancel nl4*)
qed *simp*

lemma *log4_log2*: *log 4 x = log 2 x / 2*

```
proof −
  have log 4 x = log (2^2) x by simp
  also have ... = log 2 x / 2 by(simp only: log_base_pow)
  finally show ?thesis .
qed

declare log_base_root[simp]

lemma A34_ub: assumes bst t
shows S34.A a t ≤ (3/2) * log 2 (size1 t) + 1
proof −
  have S34.A a t ≤ log (root 3 4) (size1 t) + 1 by(rule S34.A_ub3[OF
assms])
  also have ... = (3/2) * log 2 (size t + 1) + 1 by(simp add: log4_log2)
  finally show ?thesis by simp
qed

lemma Am34_ub: assumes bst t
shows S34.Am t ≤ (3/2) * log 2 (size1 t) + 1
proof −
  have S34.Am t ≤ log (root 3 4) (size1 t) + 1 by(rule S34.Am_ub3[OF
assms])
  also have ... = (3/2) * log 2 (size1 t) + 1 by(simp add: log4_log2)
  finally show ?thesis by simp
qed
```

5.3.3 Overall analysis

```
fun U where
U Empty [] = 1 |
U (Splay _) [t] = (3/2) * log 2 (size1 t) + 1 |
U (Insert _) [t] = 2 * log 2 (size1 t) + 3/2 |
U (Delete _) [t] = 3 * log 2 (size1 t) + 2

interpretation Amortized
where arity = arity and exec = exec and inv = bst
and cost = cost and Φ = S34.Φ and U = U
proof (standard, goal_cases)
  case (1 ss f) show ?case
  proof (cases f)
    case Empty thus ?thesis using 1 by auto
  next
    case (Splay a)
    then obtain t where ss = [t] bst t using 1 by auto
```

```
    with Splay bst_splay[OF ‹bst t›, of a] show ?thesis
      by (auto split: tree.split)
  next
    case (Insert a)
    then obtain t where ss = [t] bst t using 1 by auto
    with bst_splay[OF ‹bst t›, of a] Insert show ?thesis
       by (auto simp: splay_bstL[OF ‹bst t›] splay_bstR[OF ‹bst t›] split:
tree.split)
  next
    case (Delete a)
    then obtain t where ss = [t] bst t using 1 by auto
    with 1 Delete show ?thesis by(simp add: bst_delete)
  qed
next
  case (2 t) show ?case by(induction t)(simp_all add: S34.φ_def)
next
  case (3 ss f)
  show ?case (is ?l ≤ ?r)
  proof(cases f)
    case Empty thus ?thesis using 3(2) by(simp add: S34.A_def)
  next
    case (Splay a)
    then obtain t where ss = [t] bst t using 3 by auto
    thus ?thesis using S34.A_ub3[OF ‹bst t›] Splay
      by(simp add: S34.A_def log4_log2)
  next
    case [simp]: (Insert a)
    obtain t where [simp]: ss = [t] and bst t using 3 by auto
    show ?thesis
    proof cases
      assume t = Leaf thus ?thesis by(simp add: S34.φ_def log4_log2)
    next
      assume t ≠ Leaf
      then obtain l e r where [simp]: splay a t = Node l e r
        by (metis tree.exhaust splay_Leaf_iff)
      let ?t = real(t_splay a t)
      let ?Plr = S34.Φ l + S34.Φ r  let ?Ps = S34.Φ t
      let ?slr = real(size1 l) + real(size1 r) let ?LR = log 2 (1 + ?slr)
       have opt: ?t + S34.Φ (splay a t) − ?Ps ≤ 3/2 ∗ log 2 (real (size1
t)) + 1
        using S34.A_ub3[OF ‹bst t›, simplified S34.A_def, of a]
        by (simp add: log4_log2)
      from less_linear[of e a]
      show ?thesis
```

```
    proof (elim disjE)
      assume e=a
      have nneg: log 2 (1 + real (size t)) ≥ 0 by simp
      thus ?thesis using ⟨t ≠ Leaf⟩ opt ⟨e=a⟩
        apply(simp add: field_simps) using nneg by arith
    next
      let ?L = log 2 (real(size1 l) + 1)
      assume e<a hence e ≠ a by simp
      hence ?l = (?t + ?Plr − ?Ps) + ?L / 2 + ?LR / 2
        using ⟨t ≠ Leaf⟩ ⟨e<a⟩ by(simp add: S34.φ_def log4_log2)
      also have ?t + ?Plr − ?Ps ≤ log 2 ?slr + 1
        using opt size_splay[of a t,symmetric]
        by(simp add: S34.φ_def log4_log2)
      also have ?L/2 ≤ log 2 ?slr / 2 by(simp)
      also have ?LR/2 ≤ log 2 ?slr / 2 + 1/2
      proof −
        have ?LR/2 ≤ log 2 (2 * ?slr) / 2 by simp
        also have ... ≤ log 2 ?slr / 2 + 1/2
          by (simp add: log_mult del:distrib_left_numeral)
        finally show ?thesis .
      qed
      finally show ?thesis using size_splay[of a t,symmetric] by simp
    next
      let ?R = log 2 (2 + real(size r))
      assume a<e hence e ≠ a by simp
      hence ?l = (?t + ?Plr − ?Ps) + ?R / 2 + ?LR / 2
        using ⟨t ≠ Leaf⟩ ⟨a<e⟩ by(simp add: S34.φ_def log4_log2)
      also have ?t + ?Plr − ?Ps ≤ log 2 ?slr + 1
        using opt size_splay[of a t,symmetric]
        by(simp add: S34.φ_def log4_log2)
      also have ?R/2 ≤ log 2 ?slr / 2 by(simp)
      also have ?LR/2 ≤ log 2 ?slr / 2 + 1/2
      proof −
        have ?LR/2 ≤ log 2 (2 * ?slr) / 2 by simp
        also have ... ≤ log 2 ?slr / 2 + 1/2
          by (simp add: log_mult del:distrib_left_numeral)
        finally show ?thesis .
      qed
      finally show ?thesis using size_splay[of a t,symmetric] by simp
    qed
  qed
next
  case [simp]: (Delete a)
  obtain t where [simp]: ss = [t] and bst t using 3 by auto
```

```
    show ?thesis
    proof (cases t)
      case Leaf thus ?thesis
        by(simp add: Splay_Tree.delete_def t_delete_def S34.φ_def log4_log2)
    next
      case [simp]: (Node ls x rs)
     then obtain l e r where sp[simp]: splay a (Node ls x rs) = Node l e r
        by (metis tree.exhaust splay_Leaf_iff)
      let ?t = real(t_splay a t)
      let ?Plr = S34.Φ l + S34.Φ r  let ?Ps = S34.Φ t
      let ?slr = real(size1 l) + real(size1 r) let ?LR = log 2 (1 + ?slr)
      let ?lslr = log 2 (real (size ls) + (real (size rs) + 2))
      have ?lslr ≥ 0 by simp
      have opt: ?t + S34.Φ (splay a t) − ?Ps ≤ 3/2 ∗ log 2 (real (size1
t)) + 1
        using S34.A_ub3[OF ‹bst t›, simplified S34.A_def, of a]
        by (simp add: log4_log2 field_simps)
      show ?thesis
      proof (cases e=a)
        case False thus ?thesis using opt
          apply(simp add: Splay_Tree.delete_def t_delete_def field_simps)
          using ‹?lslr ≥ 0› by arith
      next
        case [simp]: True
        show ?thesis
        proof (cases l)
          case Leaf
          have S34.φ Leaf r ≥ 0 by(simp add: S34.φ_def)
          thus ?thesis using Leaf opt
            apply(simp add: Splay_Tree.delete_def t_delete_def field_simps)
            using ‹?lslr ≥ 0› by arith
        next
          case (Node ll y lr)
          then obtain l' y' r' where [simp]:
            splay_max (Node ll y lr) = Node l' y' r'
            using splay_max_Leaf_iff tree.exhaust by blast
          have bst l using bst_splay[OF ‹bst t›, of a] by simp
            have S34.Φ r' ≥ 0 apply (induction r') by (auto simp add:
S34.φ_def)
          have optm: real(t_splay_max l) + S34.Φ (splay_max l) − S34.Φ l
            ≤ 3/2 ∗ log 2 (real (size1 l)) + 1
            using S34.Am_ub3[OF ‹bst l›, simplified S34.Am_def]
            by (simp add: log4_log2 field_simps Node)
          have 1: log 4 (2+(real(size l')+real(size r))) ≤
```

```
      log 4 (2+(real(size l)+real(size r)))
      using size_splay_max[of l] Node by simp
    have 2: log 4 (2 + (real (size l') + real (size r'))) ≥ 0 by simp
    have 3: S34.φ l' r ≤ S34.φ l' r' + S34.φ l r
      apply(simp add: S34.φ_def) using 1 2 by arith
    have 4: log 2 (real(size ll) + (real(size lr) + 2)) ≤ ?lslr
      using size_if_splay[OF sp] Node by simp
    show ?thesis using add_mono[OF opt optm] Node 3
      apply(simp add: Splay_Tree.delete_def t_delete_def field_simps)
      using 4 ‹S34.Φ r' ≥ 0› by arith
  qed
 qed
qed
qed

end
theory Priority_Queue_ops
imports Main
begin

datatype 'a op = Empty | Insert 'a | Del_min

fun arity :: 'a op ⇒ nat where
arity Empty = 0 |
arity (Insert _) = 1 |
arity Del_min = 1

end
```

6 Splay Heap

```
theory Splay_Heap_Analysis
imports
  Splay_Tree.Splay_Heap
  Amortized_Framework
  Priority_Queue_ops
  Lemmas_log
begin
```

Timing functions must be kept in sync with the corresponding functions on splay heaps.

```
fun t_part :: 'a::linorder ⇒ 'a tree ⇒ nat where
t_part p Leaf = 1 |
```

```
t_part p (Node l a r) =
  (if a ≤ p then
     case r of
       Leaf ⇒ 1 |
       Node rl b rr ⇒ if b ≤ p then t_part p rr + 1 else t_part p rl + 1
   else case l of
       Leaf ⇒ 1 |
       Node ll b lr ⇒ if b ≤ p then t_part p lr + 1 else t_part p ll + 1)

definition t_in :: 'a::linorder ⇒ 'a tree ⇒ nat where
t_in x h = t_part x h

fun t_dm :: 'a::linorder tree ⇒ nat where
t_dm Leaf = 1 |
t_dm (Node Leaf _ r) = 1 |
t_dm (Node (Node ll a lr) b r) = (if ll=Leaf then 1 else t_dm ll + 1)

abbreviation φ t == log 2 (size1 t)

fun Φ :: 'a tree ⇒ real where
Φ Leaf = 0 |
Φ (Node l a r) = Φ l + Φ r + φ (Node l a r)

lemma amor_del_min: t_dm t + Φ (del_min t) − Φ t ≤ 2 * φ t + 1
proof(induction t rule: t_dm.induct)
  case (3 ll a lr b r)
  let ?t = Node (Node ll a lr) b r
  show ?case
  proof cases
    assume [simp]: ll = Leaf
    have 1: log 2 (real (size1 lr) + real (size1 r))
        ≤ 3 * log 2 (1 + (real (size1 lr) + real (size1 r))) (is ?l ≤ 3 * ?r)
    proof −
      have ?l ≤ ?r by(simp add: size1_size)
      also have ... ≤ 3 * ?r by(simp)
      finally show ?thesis .
    qed
    have 2: log 2 (1 + real (size1 lr)) ≥ 0 by simp
    thus ?case apply simp using 1 2 by linarith
  next
    assume ll[simp]: ¬ ll = Leaf
    let ?l' = del_min ll
    let ?s = Node ll a lr  let ?t = Node ?s b r
    let ?s' = Node lr b r  let ?t' = Node ?l' a ?s'
```

```
    have 0: φ ?t' ≤ φ ?t by(simp add: size1_size)
    have 1: φ ll < φ ?s by(simp add: size1_size)
     have 2: log 2 (size1 ll + size1 ?s') ≤ log 2 (size1 ?t) by(simp add:
size1_size)
    have t_dm ?t + Φ (del_min ?t) − Φ ?t
       = 1 + t_dm ll + Φ (del_min ?t) − Φ ?t by simp
    also have ... ≤ 2 + 2 * φ ll + Φ ll − Φ ?l' + Φ (del_min ?t) − Φ ?t
      using 3 ll by linarith
    also have ... = 2 + 2 * φ ll + φ ?t' + φ ?s' − φ ?t − φ ?s by(simp)
    also have ... ≤ 2 + φ ll + φ ?s' using 0 1 by linarith
    also have ... < 2 * φ ?t + 1 using 2 ld_ld_1_less[of size1 ll size1 ?s']
      by (simp add: size1_size)
    finally show ?case by simp
  qed
qed auto

lemma zig_zig:
fixes s u r r1' r2' T a b
defines t == Node s a (Node u b r) and t' == Node (Node s a u) b r1'
assumes size r1' ≤ size r
    t_part p r + Φ r1' + Φ r2' − Φ r ≤ 2 * φ r + 1
shows t_part p r + 1 + Φ t' + Φ r2' − Φ t ≤ 2 * φ t + 1
proof −
  have 1: φ r ≤ φ (Node u b r) by (simp add: size1_size)
  have 2: log 2 (real (size1 s + size1 u + size1 r1')) ≤ φ t
    using assms(3) by (simp add: t_def size1_size)
  from ld_ld_1_less[of size1 s + size1 u size1 r]
  have 1 + φ r + log 2 (size1 s + size1 u) ≤ 2 * log 2 (size1 s + size1 u
+ size1 r)
    by(simp add: size1_size)
  thus ?thesis using assms 1 2 by (simp add: algebra_simps)
qed

lemma zig_zag:
fixes s u r r1' r2' a b
defines t ≡ Node s a (Node r b u) and t1' == Node s a r1' and t2' ≡
Node u b r2'
assumes size r = size r1' + size r2'
    t_part p r + Φ r1' + Φ r2' − Φ r ≤ 2 * φ r + 1
shows t_part p r + 1 + Φ t1' + Φ t2' − Φ t ≤ 2 * φ t + 1
proof −
  have 1: φ r ≤ φ (Node u b r) by (simp add: size1_size)
  have 2: φ r ≤ φ t by (simp add: t_def size1_size)
  from ld_ld_less2[of size1 s + size1 r1' size1 u + size1 r2']
```

```
  have 1 + log 2 (size1 s + size1 r1') + log 2 (size1 u + size1 r2') ≤ 2 *
φ t
    by(simp add: assms(4) size1_size t_def ac_simps)
  thus ?thesis using assms 1 2 by (simp add: algebra_simps)
qed

lemma amor_partition: bst_wrt (≤) t ⟹ partition p t = (l',r')
  ⟹ t_part p t + Φ l' + Φ r' − Φ t ≤ 2 * log 2 (size1 t) + 1
proof(induction p t arbitrary: l' r' rule: partition.induct)
  case 1 thus ?case by simp
next
  case (2 p l a r)
  show ?case
  proof cases
    assume a ≤ p
    show ?thesis
    proof (cases r)
      case Leaf thus ?thesis using ‹a ≤ p› 2.prems by fastforce
    next
      case [simp]: (Node rl b rr)
      let ?t = Node l a r
      show ?thesis
      proof cases
        assume b ≤ p
        with ‹a ≤ p› 2.prems obtain rrl
          where 0: partition p rr = (rrl, r') l' = Node (Node l a rl) b rrl
          by (auto split: tree.splits prod.splits)
        have size rrl ≤ size rr
          using size_partition[OF 0(1)] by (simp add: size1_size)
        with 0 ‹a ≤ p› ‹b ≤ p› 2.prems(1) 2.IH(1)[OF _ Node , of rrl r']
          zig_zig[where s=l and u=rl and r=rr and r1'=rrl and r2'=r'
and p=p, of a b]
        show ?thesis by (simp add: algebra_simps)
      next
        assume ¬ b ≤ p
        with ‹a ≤ p› 2.prems obtain rll rlr
          where 0: partition p rl = (rll, rlr) l' = Node l a rll r' = Node rlr
b rr
          by (auto split: tree.splits prod.splits)
        from 0 ‹a ≤ p› ‹¬ b ≤ p› 2.prems(1) 2.IH(2)[OF _ Node, of rll rlr]
          size_partition[OF 0(1)]
          zig_zag[where s=l and u=rr and r=rl and r1'=rll and r2'=rlr
and p=p, of a b]
        show ?thesis by (simp add: algebra_simps)
```

```
    qed
   qed
  next
   assume ¬ a ≤ p
   show ?thesis
   proof (cases l)
    case Leaf thus ?thesis using ⟨¬ a ≤ p⟩ 2.prems by fastforce
   next
    case [simp]: (Node ll b lr)
    let ?t = Node l a r
    show ?thesis
    proof cases
     assume b ≤ p
     with ⟨¬ a ≤ p⟩ 2.prems obtain lrl lrr
      where 0: partition p lr = (lrl, lrr) l' = Node ll b lrl r' = Node lrr
a r
      by (auto split: tree.splits prod.splits)
     from 0 ⟨¬ a ≤ p⟩ ⟨b ≤ p⟩ 2.prems(1) 2.IH(3)[OF _ Node, of lrl lrr]
      size_partition[OF 0(1)]
      zig_zag[where s=r and u=ll and r=lr and r1'=lrr and r2'=lrl
and p=p, of a b]
     show ?thesis by (auto simp: algebra_simps)
    next
     assume ¬ b ≤ p
     with ⟨¬ a ≤ p⟩ 2.prems obtain llr
      where 0: partition p ll = (l',llr) r' = Node llr b (Node lr a r)
      by (auto split: tree.splits prod.splits)
     have size llr ≤ size ll
      using size_partition[OF 0(1)] by (simp add: size1_size)
     with 0 ⟨¬ a ≤ p⟩ ⟨¬ b ≤ p⟩ 2.prems(1) 2.IH(4)[OF _ Node, of l'
llr]
      zig_zig[where s=r and u=lr and r=ll and r1'=llr and r2'=l'
and p=p, of a b]
     show ?thesis by (auto simp: algebra_simps)
    qed
   qed
  qed
qed

fun exec :: 'a::linorder op ⇒ 'a tree list ⇒ 'a tree where
exec Empty [] = Leaf |
exec (Insert a) [t] = insert a t |
exec Del_min [t] = del_min t
```

```
fun cost :: 'a::linorder op ⇒ 'a tree list ⇒ nat where
cost Empty [] = 1 |
cost (Insert a) [t] = t_in a t |
cost Del_min [t] = t_dm t

fun U where
U Empty [] = 1 |
U (Insert _) [t] = 3 * log 2 (size1 t + 1) + 1 |
U Del_min [t] = 2 * φ t + 1

interpretation Amortized
where arity = arity and exec = exec and inv = bst_wrt (≤)
and cost = cost and Φ = Φ and U = U
proof (standard, goal_cases)
  case (1 _ f) thus ?case
    by(cases f)
      (auto simp: insert_def bst_del_min dest!: bst_partition split: prod.splits)
next
  case (2 h) thus ?case by(induction h) (auto simp: size1_size)
next
  case (3 s f)
  show ?case
  proof (cases f)
    case Empty with 3 show ?thesis by(auto)
  next
    case Del_min with 3 show ?thesis by(auto simp: amor_del_min)
  next
    case [simp]: (Insert x)
    then obtain t where [simp]: s = [t] bst_wrt (≤) t using 3 by auto
    { fix l r assume 1: partition x t = (l,r)
      have log 2 (1 + size t) < log 2 (2 + size t) by simp
     with 1 amor_partition[OF ‹bst_wrt (≤) t› 1] size_partition[OF 1] have
?thesis
        by(simp add: t_in_def insert_def algebra_simps size1_size
            del: log_less_cancel_iff) }
    thus ?thesis by(simp add: insert_def split: prod.split)
  qed
qed

end
```

7 Pairing Heaps

7.1 Binary Tree Representation

theory *Pairing_Heap_Tree_Analysis*
imports
Pairing_Heap.Pairing_Heap_Tree
Amortized_Framework
Priority_Queue_ops_merge
Lemmas_log
begin

Verification of logarithmic bounds on the amortized complexity of pairing heaps [2, 1].

fun *len* :: *'a tree* ⇒ *nat* **where**
len Leaf = *0*
| *len* (*Node* _ _ *r*) = *1* + *len r*

fun Φ :: *'a tree* ⇒ *real* **where**
Φ *Leaf* = *0*
| Φ (*Node l* _ *r*) = Φ *l* + Φ *r* + *log 2* (*1* + *size l* + *size r*)

lemma *link_size*[*simp*]: *size* (*link h*) = *size h*
by (*cases h rule*: *link.cases*) *simp_all*

lemma *size_pass*$_1$: *size* (*pass*$_1$ *h*) = *size h*
by (*induct h rule*: *pass*$_1$*.induct*) *simp_all*

lemma *size_pass*$_2$: *size* (*pass*$_2$ *h*) = *size h*
by (*induct h rule*: *pass*$_2$*.induct*) *simp_all*

lemma *size_merge*:
is_root h1 ⟹ *is_root h2* ⟹ *size* (*merge h1 h2*) = *size h1* + *size h2*
by (*simp split*: *tree.splits*)

lemma Δ Φ*_insert*: *is_root h* ⟹ Φ (*insert x h*) − Φ *h* ≤ *log 2* (*size h* + *1*)
by (*simp split*: *tree.splits*)

lemma Δ Φ*_merge*:
assumes *h1* = *Node lx x Leaf h2* = *Node ly y Leaf*
shows Φ (*merge h1 h2*) − Φ *h1* − Φ *h2* ≤ *log 2* (*size h1* + *size h2*) + *1*
proof −
let *?hs* = *Node lx x* (*Node ly y Leaf*)
have Φ (*merge h1 h2*) = Φ (*link ?hs*) **using** *assms* **by** *simp*

```
  also have ... = Φ lx + Φ ly + log 2 (size lx + size ly + 1) + log 2 (size
lx + size ly + 2)
    by (simp add: algebra_simps)
  also have ... = Φ lx + Φ ly + log 2 (size lx + size ly + 1) + log 2 (size
h1 + size h2)
     using assms by simp
  finally have Φ (merge h1 h2) = ... .
  have Φ (merge h1 h2) − Φ h1 − Φ h2 =
   log 2 (size lx + size ly + 1) + log 2 (size h1 + size h2)
   − log 2 (size lx + 1) − log 2 (size ly + 1)
     using assms by (simp add: algebra_simps)
  also have ... ≤ log 2 (size h1 + size h2) + 1
    using ld_le_2ld[of size lx size ly] assms by (simp add: algebra_simps)
  finally show ?thesis .
qed

fun upperbound :: 'a tree ⇒ real where
  upperbound Leaf = 0
| upperbound (Node _ _ Leaf) = 0
| upperbound (Node lx _ (Node ly _ Leaf)) = 2∗log 2 (size lx + size ly + 2)

| upperbound (Node lx _ (Node ly _ ry)) = 2∗log 2 (size lx + size ly + size
ry + 2)
    − 2∗log 2 (size ry) − 2 + upperbound ry

lemma ΔΦ_pass1_upperbound: Φ (pass1 hs) − Φ hs ≤ upperbound hs
proof (induction hs rule: upperbound.induct)
  case (3 lx x ly y)
  have log 2 (size lx + size ly + 1) − log 2 (size ly + 1)
    ≤ log 2 (size lx + size ly + 1) by simp
  also have ... ≤ log 2 (size lx + size ly + 2) by simp
  also have ... ≤ 2∗... by simp
  finally show ?case by (simp add: algebra_simps)
next
  case (4 lx x ly y lz z rz)
  let ?ry = Node lz z rz
  let ?rx = Node ly y ?ry
  let ?h = Node lx x ?rx
  let ?t =log 2 (size lx + size ly + 1) − log 2 (size ly + size ?ry + 1)
  have Φ (pass1 ?h) − Φ ?h ≤ ?t + upperbound ?ry
    using 4.IH by (simp add: size_pass1 algebra_simps)
  moreover have log 2 (size lx + size ly + 1) + 2 ∗ log 2 (size ?ry) + 2
    ≤ 2 ∗ log 2 (size ?h) + log 2 (size ly + size ?ry + 1) (is ?l ≤ ?r)
  proof −
```

```
    have ?l ≤ 2 * log 2 (size lx + size ly + size ?ry + 1) + log 2 (size
?ry)
      using ld_sum_inequality [of size lx + size ly + 1 size ?ry] by simp
    also have ... ≤ 2 * log 2 (size lx + size ly + size ?ry + 2) + log 2
(size ?ry)
      by simp
    also have ... ≤ ?r by simp
    finally show ?thesis .
  qed
  ultimately show ?case by simp
qed simp_all

lemma ΔΦ_pass1: assumes hs ≠ Leaf
  shows Φ (pass₁ hs) − Φ hs ≤ 2*log 2 (size hs) − len hs + 2
proof −
  from assms have upperbound hs ≤ 2*log 2 (size hs) − len hs + 2
    by (induct hs rule: upperbound.induct) (simp_all add: algebra_simps)
  thus ?thesis using ΔΦ_pass1_upperbound [of hs] order_trans by blast
qed

lemma ΔΦ_pass2: hs ≠ Leaf ⟹ Φ (pass₂ hs) − Φ hs ≤ log 2 (size hs)
proof (induction hs)
  case (Node lx x rx)
  thus ?case
  proof (cases rx)
    case 1: (Node ly y ry)
    let ?h = Node lx x rx
    obtain la a where 2: pass₂ rx = Node la a Leaf
      using pass₂_struct 1 by force
    hence 3: size rx = size ... using size_pass₂ by metis
    have link: Φ(link(Node lx x (pass₂ rx))) − Φ lx − Φ (pass₂ rx) =
          log 2 (size lx + size rx + 1) + log 2 (size lx + size rx) − log 2
(size rx)
      using 2 3 by (simp add: algebra_simps)
    have Φ (pass₂ ?h) − Φ ?h =
        Φ (link (Node lx x (pass₂ rx))) − Φ lx − Φ rx − log 2 (size lx +
size rx + 1)
      by (simp)
    also have ... = Φ (pass₂ rx) − Φ rx + log 2 (size lx + size rx) − log
2 (size rx)
      using link by linarith
    also have ... ≤ log 2 (size lx + size rx)
      using Node.IH 1 by simp
    also have ... ≤ log 2 (size ?h) using 1 by simp
```

```
    finally show ?thesis .
  qed simp
qed simp

lemma ΔΦ_mergepairs: assumes hs ≠ Leaf
  shows Φ (merge_pairs hs) − Φ hs ≤ 3 ∗ log 2 (size hs) − len hs + 2
proof −
  have pass1 hs ≠ Leaf by (metis assms eq_size_0 size_pass1)
  with assms ΔΦ_pass1[of hs] ΔΦ_pass2[of pass1 hs]
  show ?thesis by (auto simp add: size_pass1 pass12_merge_pairs)
qed

lemma ΔΦ_del_min: assumes lx ≠ Leaf
shows Φ (del_min (Node lx x Leaf)) − Φ (Node lx x Leaf)
  ≤ 3∗log 2 (size lx) − len lx + 2
proof −
  let ?h = Node lx x Leaf
  let ?ΔΦ1 = Φ lx − Φ ?h
  let ?ΔΦ2 = Φ(pass2(pass1 lx)) − Φ lx
  let ?ΔΦ = Φ (del_min ?h) − Φ ?h
  have lx ≠ Leaf ⟹ Φ(pass2(pass1 lx)) − Φ (pass1 lx) ≤ log 2 (size lx)
    using ΔΦ_pass2 [of pass1 lx] by(metis eq_size_0 size_pass1)
  moreover have lx ≠ Leaf ⟹ Φ (pass1 lx) − Φ lx ≤ 2∗... − len lx +
2
    using ΔΦ_pass1 by metis
  moreover have ?ΔΦ ≤ ?ΔΦ2 by simp
  ultimately show ?thesis using assms by linarith
qed

lemma is_root_merge:
  is_root h1 ⟹ is_root h2 ⟹ is_root (merge h1 h2)
by (simp split: tree.splits)

lemma is_root_insert: is_root h ⟹ is_root (insert x h)
by (simp split: tree.splits)

lemma is_root_del_min:
  assumes is_root h shows is_root (del_min h)
proof (cases h)
  case [simp]: (Node lx x rx)
  have rx = Leaf using assms by simp
  thus ?thesis
  proof (cases lx)
    case (Node ly y ry)
```

```
    then obtain la a ra where pass1 lx = Node a la ra
      using pass1_struct by blast
    moreover obtain lb b where pass2 ... = Node b lb Leaf
      using pass2_struct by blast
    ultimately show ?thesis using assms by simp
  qed simp
qed simp

lemma pass1_len: len (pass1 h) ≤ len h
by (induct h rule: pass1.induct) simp_all

fun exec :: 'a :: linorder op ⇒ 'a tree list ⇒ 'a tree where
exec Empty [] = Leaf |
exec Del_min [h] = del_min h |
exec (Insert x) [h] = insert x h |
exec Merge [h1,h2] = merge h1 h2

fun t_pass1 :: 'a tree ⇒ nat where
  t_pass1 Leaf = 1
| t_pass1 (Node _ _ Leaf) = 1
| t_pass1 (Node _ _ (Node _ _ ry)) = t_pass1 ry + 1

fun t_pass2 :: 'a tree ⇒ nat where
  t_pass2 Leaf = 1
| t_pass2 (Node _ _ rx) = t_pass2 rx + 1

fun cost :: 'a :: linorder op ⇒ 'a tree list ⇒ nat where
  cost Empty [] = 1
| cost Del_min [Leaf] = 1
| cost Del_min [Node lx _ _] = t_pass2 (pass1 lx) + t_pass1 lx
| cost (Insert a) _ = 1
| cost Merge _ = 1

fun U :: 'a :: linorder op ⇒ 'a tree list ⇒ real where
  U Empty [] = 1
| U (Insert a) [h] = log 2 (size h + 1) + 1
| U Del_min [h] = 3*log 2 (size h + 1) + 4
| U Merge [h1,h2] = log 2 (size h1 + size h2 + 1) + 2

interpretation Amortized
where arity = arity and exec = exec and cost = cost and inv = is_root
and Φ = Φ and U = U
proof (standard, goal_cases)
  case (1 _ f) thus ?case using is_root_insert is_root_del_min is_root_merge
```

```
    by (cases f) (auto simp: numeral_eq_Suc)
next
  case (2 s) show ?case by (induct s) simp_all
next
  case (3 ss f) show ?case
  proof (cases f)
    case Empty with 3 show ?thesis by(auto)
  next
    case (Insert x)
    then obtain h where ss = [h] is_root h using 3 by auto
    thus ?thesis using Insert ΔΦ_insert 3 by auto
  next
    case [simp]: (Del_min)
    then obtain h where [simp]: ss = [h] using 3 by auto
    show ?thesis
    proof (cases h)
      case [simp]: (Node lx x rx)
      have t_pass2 (pass1 lx) + t_pass1 lx ≤ len lx + 2
        by (induct lx rule: pass1.induct) simp_all
      hence cost f ss ≤ ... by simp
      moreover have Φ (del_min h) − Φ h ≤ 3∗log 2 (size h + 1) − len
lx + 2
      proof (cases lx)
        case [simp]: (Node ly y ry)
        have Φ (del_min h) − Φ h ≤ 3∗log 2 (size lx) − len lx + 2
          using ΔΦ_del_min[of lx x] 3 by simp
        also have ... ≤ 3∗log 2 (size h + 1) − len lx + 2 by fastforce
        finally show ?thesis by blast
      qed (insert 3, simp)
      ultimately show ?thesis by auto
    qed simp
  next
    case [simp]: Merge
   then obtain h1 h2 where [simp]: ss = [h1,h2] and 1: is_root h1 is_root
h2
      using 3 by (auto simp: numeral_eq_Suc)
    show ?thesis
    proof (cases h1)
      case Leaf thus ?thesis by (cases h2) auto
    next
      case h1: Node
      show ?thesis
      proof (cases h2)
        case Leaf thus ?thesis using h1 by simp
```

next
case *h2*: *Node*
have Φ (*merge h1 h2*) − Φ *h1* − Φ *h2* ≤ *log 2* (*real* (*size h1* + *size h2*)) + *1*
apply(*rule* ΔΦ_*merge*) **using** *h1 h2 1* **by** *auto*
also have ... ≤ *log 2* (*size h1* + *size h2* + *1*) + *1* **by** (*simp add*: *h1*)
finally show *?thesis* **by**(*simp add*: *algebra_simps*)
qed
qed
qed
qed

end

7.2 Okasaki's Pairing Heap

theory *Pairing_Heap_List1_Analysis*
imports
Pairing_Heap.Pairing_Heap_List1
Amortized_Framework
Priority_Queue_ops_merge
Lemmas_log
begin

Amortized analysis of pairing heaps as defined by Okasaki [6].

fun *hps* **where**
hps (*Hp* _ *hs*) = *hs*

lemma *merge_Empty*[*simp*]: *merge heap.Empty h* = *h*
by(*cases h*) *auto*

lemma *merge2*: *merge* (*Hp x lx*) *h* = (*case h of heap.Empty* ⇒ *Hp x lx* | (*Hp y ly*) ⇒
(*if x* < *y then Hp x* (*Hp y ly* # *lx*) *else Hp y* (*Hp x lx* # *ly*)))
by(*auto split*: *heap.split*)

lemma *pass1_Nil_iff*: *pass*$_1$ *hs* = [] ⟷ *hs* = []
by(*cases hs rule*: *pass*$_1$.*cases*) *auto*

7.2.1 Invariant

fun *no_Empty* :: 'a :: *linorder heap* ⇒ *bool* **where**
no_Empty heap.Empty = *False* |
no_Empty (*Hp x hs*) = (∀ *h* ∈ *set hs*. *no_Empty h*)

abbreviation *no_Emptys* :: *'a* :: *linorder heap list* ⇒ *bool* **where**
no_Emptys hs ≡ ∀ *h* ∈ *set hs*. *no_Empty h*

fun *is_root* :: *'a* :: *linorder heap* ⇒ *bool* **where**
is_root heap.Empty = *True* |
is_root (*Hp x hs*) = *no_Emptys hs*

lemma *is_root_if_no_Empty*: *no_Empty h* ⟹ *is_root h*
by(*cases h*) *auto*

lemma *no_Emptys_hps*: *no_Empty h* ⟹ *no_Emptys*(*hps h*)
by(*induction h*) *auto*

lemma *no_Empty_merge*: ⟦ *no_Empty h1*; *no_Empty h2*⟧ ⟹ *no_Empty* (*merge h1 h2*)
by (*cases* (*h1*,*h2*) *rule*: *merge.cases*) *auto*

lemma *is_root_merge*: ⟦ *is_root h1*; *is_root h2*⟧ ⟹ *is_root* (*merge h1 h2*)
by (*cases* (*h1*,*h2*) *rule*: *merge.cases*) *auto*

lemma *no_Emptys_pass1*:
no_Emptys hs ⟹ *no_Emptys* (*pass*$_1$ *hs*)
by(*induction hs rule*: *pass*$_1$.*induct*)(*auto simp*: *no_Empty_merge*)

lemma *is_root_pass2*: *no_Emptys hs* ⟹ *is_root*(*pass*$_2$ *hs*)
proof(*induction hs*)
case (*Cons* _ *hs*)
show *?case*
proof *cases*
assume *hs* = [] **thus** *?thesis* **using** *Cons* **by** (*auto simp*: *is_root_if_no_Empty*)
next
assume *hs* ≠ [] **thus** *?thesis* **using** *Cons* **by**(*auto simp*: *is_root_merge is_root_if_no_Empty*)
qed
qed *simp*

7.2.2 Complexity

fun *size_hp* :: *'a heap* ⇒ *nat* **where**
size_hp heap.Empty = *0* |
size_hp (*Hp x hs*) = *sum_list*(*map size_hp hs*) + *1*

abbreviation *size_hps* **where**
size_hps hs ≡ *sum_list*(*map size_hp hs*)

fun Φ_*hps* :: *'a heap list* ⇒ *real* **where**
Φ_*hps* [] = *0* |
Φ_*hps* (*heap.Empty* # *hs*) = Φ_*hps hs* |
Φ_*hps* (*Hp x hsl* # *hsr*) =
Φ_*hps hsl* + Φ_*hps hsr* + *log 2* (*size_hps hsl* + *size_hps hsr* + *1*)

fun Φ :: *'a heap* ⇒ *real* **where**
Φ *heap.Empty* = *0* |
Φ (*Hp* _ *hs*) = Φ_*hps hs* + *log 2* (*size_hps*(*hs*)+*1*)

lemma Φ_*hps_ge0*: Φ_*hps hs* ≥ *0*
by (*induction hs rule*: Φ_*hps.induct*) *auto*

lemma *no_Empty_ge0*: *no_Empty h* ⟹ *size_hp h* > *0*
by(*cases h*) *auto*

declare *algebra_simps*[*simp*]

lemma Φ_*hps1*: Φ_*hps* [*h*] = Φ *h*
by(*cases h*) *auto*

lemma *size_hp_merge*: *size_hp*(*merge h1 h2*) = *size_hp h1* + *size_hp h2*
by (*induction h1 h2 rule*: *merge.induct*) *simp_all*

lemma $pass_1$_*size*[*simp*]: *size_hps* ($pass_1$ *hs*) = *size_hps hs*
by (*induct hs rule*: $pass_1$.*induct*) (*simp_all add*: *size_hp_merge*)

lemma Δ Φ_*insert*:
Φ (*Pairing_Heap_List1.insert x h*) − Φ *h* ≤ *log 2* (*size_hp h* + *1*)
by(*cases h*)(*auto simp*: *size_hp_merge*)

lemma Δ Φ_*merge*:
Φ (*merge h1 h2*) − Φ *h1* − Φ *h2*
≤ *log 2* (*size_hp h1* + *size_hp h2* + *1*) + *1*
proof(*induction h1 h2 rule*: *merge.induct*)
case (*3 x lx y ly*)
thus *?case*
using *ld_le_2ld*[*of size_hps lx size_hps ly*]
log_le_cancel_iff[*of 2 size_hps lx* + *size_hps ly* + *2 size_hps lx* + *size_hps ly* + *3*]
by (*auto simp del*: *log_le_cancel_iff*)

qed *auto*

fun *sum_ub* :: *'a heap list* ⇒ *real* **where**
sum_ub [] = *0*
| *sum_ub* [_] = *0*
| *sum_ub* [*h1*, *h2*] = *2*∗*log 2* (*size_hp h1* + *size_hp h2*)
| *sum_ub* (*h1* # *h2* # *hs*) = *2*∗*log 2* (*size_hp h1* + *size_hp h2* + *size_hps hs*)
− *2*∗*log 2* (*size_hps hs*) − *2* + *sum_ub hs*

lemma ΔΦ*_pass1_sum_ub*: *no_Emptys hs* ⟹
Φ*_hps* (*pass*$_1$ *hs*) − Φ*_hps hs* ≤ *sum_ub hs* (**is** _ ⟹ *?P hs*)
proof (*induction hs rule*: *sum_ub.induct*)
case (*3 h1 h2*)
then obtain *x hsx y hsy* **where** [*simp*]: *h1* = *Hp x hsx h2* = *Hp y hsy*
by *simp* (*meson no_Empty.elims*(*2*))
have *0*: ⋀*x y*::*real*. *0* ≤ *x* ⟹ *x* ≤ *y* ⟹ *x* ≤ *2*∗*y* **by** *linarith*
show *?case* **using** *3* **by** (*auto simp add*: *add_increasing 0*)
next
case (*4 h1 h2 h3 hs*)
hence *IH*: *?P*(*h3*#*hs*) **by** *auto*
from *4.prems* **obtain** *x hsx y hsy* **where** [*simp*]: *h1* = *Hp x hsx h2* = *Hp y hsy*
by *simp* (*meson no_Empty.elims*(*2*))
from *4.prems* **have** *s3*: *size_hp h3* > *0*
apply *auto* **using** *size_hp.elims* **by** *force*
let *?ry* = *h3* # *hs*
let *?rx* = *Hp y hsy* # *?ry*
let *?h* = *Hp x hsx* # *?rx*
have Φ*_hps*(*pass*$_1$ *?h*) − Φ*_hps ?h*
≤ *log 2* (*1* + *size_hps hsx* + *size_hps hsy*) − *log 2* (*1* + *size_hps hsy* + *size_hps ?ry*) + *sum_ub ?ry*
using *IH* **by** *simp*
also have *log 2* (*1* + *size_hps hsx* + *size_hps hsy*) − *log 2* (*1* + *size_hps hsy* + *size_hps ?ry*)
≤ *2*∗*log 2* (*size_hps ?h*) − *2*∗*log 2* (*size_hps ?ry*) − *2*
proof −
have *log 2* (*1* + *size_hps hsx* + *size_hps hsy*) + *log 2* (*size_hps ?ry*) − *2*∗*log 2* (*size_hps ?h*)
= *log 2* ((*1* + *size_hps hsx* + *size_hps hsy*)/(*size_hps ?h*)) + *log 2* (*size_hps ?ry* / *size_hps ?h*)
using *s3* **by** (*simp add*: *log_divide*)
also have ... ≤ −*2*
proof −

```
    have 2 + ...
     ≤ 2*log 2 ((1 + size_hps hsx + size_hps hsy) / size_hps ?h +  size_hps
?ry / size_hps ?h)
       using ld_sum_inequality [of (1 + size_hps hsx + size_hps hsy) /
size_hps ?h (size_hps ?ry / size_hps ?h)] using s3 by simp
   also have ... ≤ 0 by (simp add: field_simps log_divide add_pos_nonneg)
    finally show ?thesis by linarith
  qed
  finally have log 2 (1 + size_hps hsx + size_hps hsy) + log 2 (size_hps
?ry) + 2
    ≤  2*log 2 (size_hps ?h) by simp
   moreover have log 2 (size_hps ?ry) ≤ log 2 (size_hps ?rx) using s3
by simp
  ultimately have log 2 (1 + size_hps hsx + size_hps hsy) − ...
    ≤  2*log 2 (size_hps ?h) − 2*log 2 (size_hps ?ry) − 2 by linarith
  thus ?thesis by simp
 qed
 finally show ?case by (simp)
qed simp_all

lemma ΔΦ_pass1: assumes hs ≠ [] no_Emptys hs
 shows Φ_hps (pass1 hs) − Φ_hps hs ≤ 2 * log 2 (size_hps hs) − length
hs + 2
proof −
 have sum_ub hs ≤ 2 * log 2 (size_hps hs) − length hs + 2
  using assms by (induct hs rule: sum_ub.induct) (auto dest: no_Empty_ge0)
 thus ?thesis using ΔΦ_pass1_sum_ub[OF assms(2)] by linarith
qed

lemma size_hps_pass2: hs ≠ [] ⟹ no_Emptys hs ⟹
 no_Empty(pass2 hs) & size_hps hs = size_hps(hps(pass2 hs))+1
apply(induction hs rule: Φ_hps.induct)
 apply (fastforce simp: merge2 split: heap.split)+
done

lemma ΔΦ_pass2: hs ≠ [] ⟹ no_Emptys hs ⟹
 Φ (pass2 hs) − Φ_hps hs ≤ log 2 (size_hps hs)
proof (induction hs)
 case (Cons h hs)
 thus ?case
 proof cases
  assume hs = []
  thus ?thesis using Cons by (auto simp add: Φ_hps1 dest: no_Empty_ge0)
 next
```

```
    assume ∗: hs ≠ []
    obtain x hs2 where [simp]: h = Hp x hs2
      using Cons.prems(2)by simp (meson no_Empty.elims(2))
    show ?thesis
    proof (cases pass2 hs)
      case Empty thus ?thesis using Φ_hps_ge0[of hs]
        by(simp add: add_increasing xt1(3)[OF mult_2, OF add_increasing])
    next
      case (Hp y hs3)
     with Cons ∗ size_hps_pass2[of hs] show ?thesis by(auto simp: add_mono)
    qed
  qed
qed simp

lemma ΔΦ_del_min: assumes hps h ≠ [] no_Empty h
  shows Φ (del_min h) − Φ h
  ≤ 3 ∗ log 2 (size_hps(hps h)) − length(hps h) + 2
proof −
  obtain x hs where [simp]: h = Hp x hs using assms(2) by (cases h)
auto
  let ?ΔΦ1 = Φ_hps(hps h) − Φ h
  let ?ΔΦ2 = Φ(pass2(pass1 (hps h))) − Φ_hps (hps h)
  let ?ΔΦ = Φ (del_min h) − Φ h
 have Φ(pass2(pass1(hps h))) − Φ_hps (pass1(hps h)) ≤ log 2 (size_hps(hps
h))
    using ΔΦ_pass2[of pass1(hps h)] using assms
    by (auto simp: pass1_Nil_iff no_Emptys_pass1 dest: no_Emptys_hps)
 moreover have Φ_hps (pass1 (hps h)) − Φ_hps (hps h) ≤ 2∗... − length
(hps h) + 2
    using ΔΦ_pass1[OF assms(1) no_Emptys_hps[OF assms(2)]] by blast
  moreover have ?ΔΦ1 ≤ 0 by simp
  moreover have ?ΔΦ = ?ΔΦ1 + ?ΔΦ2 by simp
  ultimately show ?thesis by linarith
qed


fun exec :: 'a :: linorder op ⇒ 'a heap list ⇒ 'a heap where
exec Empty [] = heap.Empty |
exec Del_min [h] = del_min h |
exec (Insert x) [h] = Pairing_Heap_List1.insert x h |
exec Merge [h1,h2] = merge h1 h2

fun t_pass1 :: 'a heap list ⇒ nat where
  t_pass1 [] = 1
```

```
| t_pass1 [_] = 1
| t_pass1 (_ # _ # hs) = 1 + t_pass1 hs

fun t_pass2 :: 'a heap list ⇒ nat where
 t_pass2 [] = 1
| t_pass2 (_ # hs) = 1 + t_pass2 hs

fun cost :: 'a :: linorder op ⇒ 'a heap list ⇒ nat where
cost Empty _ = 1 |
cost Del_min [heap.Empty] = 1 |
cost Del_min [Hp x hs] = t_pass2 (pass_1 hs) + t_pass1 hs |
cost (Insert a) _ = 1 |
cost Merge _ = 1

fun U :: 'a :: linorder op ⇒ 'a heap list ⇒ real where
U Empty _ = 1 |
U (Insert a) [h] = log 2 (size_hp h + 1) + 1 |
U Del_min [h] = 3*log 2 (size_hp h + 1) + 4 |
U Merge [h1,h2] = log 2 (size_hp h1 + size_hp h2 + 1) + 2

interpretation pairing: Amortized
where arity = arity and exec = exec and cost = cost and inv = is_root
and Φ = Φ and U = U
proof (standard, goal_cases)
  case (1 ss f) show ?case
  proof (cases f)
    case Empty with 1 show ?thesis by simp
  next
    case Insert thus ?thesis using 1 by(auto simp: is_root_merge)
  next
    case Merge
    thus ?thesis using 1 by(auto simp: is_root_merge numeral_eq_Suc)
  next
    case [simp]: Del_min
    then obtain h where [simp]: ss = [h] using 1 by auto
    show ?thesis
    proof (cases h)
      case [simp]: (Hp _ hs)
      show ?thesis
      proof cases
        assume hs = [] thus ?thesis by simp
      next
        assume hs ≠ [] thus ?thesis
          using 1(1) no_Emptys_pass1 by (auto intro: is_root_pass2)
```

```
    qed
   qed simp
  qed
next
  case (2 s) show ?case by (cases s) (auto simp: Φ_hps_ge0)
next
  case (3 ss f) show ?case
  proof (cases f)
    case Empty with 3 show ?thesis by(auto)
  next
    case Insert thus ?thesis using ΔΦ_insert 3 by auto
  next
    case [simp]: Del_min
    then obtain h where [simp]: ss = [h] using 3 by auto
    show ?thesis
    proof (cases h)
      case [simp]: (Hp x hs)
      have t_pass2 (pass_1 hs) + t_pass1 hs ≤ 2 + length hs
        by (induct hs rule: pass_1.induct) simp_all
      hence  cost f ss ≤ ... by simp
      moreover have  Φ (del_min h) − Φ h ≤ 3∗log 2 (size_hp h + 1) −
length hs + 2
      proof (cases hs = [])
        case False
        hence Φ (del_min h) − Φ h ≤ 3∗log 2 (size_hps hs) − length hs + 2
          using  ΔΦ_del_min[of h] 3(1) by simp
        also have ... ≤ 3∗log 2 (size_hp h + 1) − length hs + 2
          using False 3(1) size_hps_pass2 by fastforce
        finally show ?thesis .
      qed simp
      ultimately show ?thesis by simp
    qed simp
  next
    case [simp]: Merge
    then obtain h1 h2 where [simp]: ss = [h1, h2]
      using 3 by(auto simp: numeral_eq_Suc)
    show ?thesis
    proof (cases h1 = heap.Empty ∨ h2 = heap.Empty)
      case True thus ?thesis by auto
    next
      case False
      then obtain x1 x2 hs1 hs2 where [simp]: h1 = Hp x1 hs1 h2 = Hp
x2 hs2
        by (meson hps.cases)
```

have Φ (*merge h1 h2*) $-$ Φ *h1* $-$ Φ *h2* $\leq$ *log 2* (*size_hp h1* + *size_hp h2* + *1*) + *1*
using $\Delta\Phi$*_merge*[*of h1 h2*] **by** *simp*
thus *?thesis* **by**(*simp*)
qed
qed
qed

end

7.3 Transfer of Tree Analysis to List Representation

theory *Pairing_Heap_List1_Analysis2*
imports
Pairing_Heap_List1_Analysis
Pairing_Heap_Tree_Analysis
begin

This theory transfers the amortized analysis of the tree-based pairing heaps to Okasaki's pairing heaps.

abbreviation *is_root'* == *Pairing_Heap_List1_Analysis.is_root*
abbreviation *del_min'* == *Pairing_Heap_List1.del_min*
abbreviation *insert'* == *Pairing_Heap_List1.insert*
abbreviation *merge'* == *Pairing_Heap_List1.merge*
abbreviation $pass_1'$ == *Pairing_Heap_List1.*$pass_1$
abbreviation $pass_2'$ == *Pairing_Heap_List1.*$pass_2$
abbreviation t_{pass1}' == *Pairing_Heap_List1_Analysis.*t_{pass1}
abbreviation t_{pass2}' == *Pairing_Heap_List1_Analysis.*t_{pass2}

fun *homs* :: *'a heap list* $\Rightarrow$ *'a tree* **where**
homs [] = *Leaf* |
homs (*Hp x lhs # rhs*) = *Node* (*homs lhs*) *x* (*homs rhs*)

fun *hom* :: *'a heap* $\Rightarrow$ *'a tree* **where**
hom heap.Empty = *Leaf* |
hom (*Hp x hs*) = (*Node* (*homs hs*) *x Leaf*)

lemma *homs_pass1'*: *no_Emptys hs* $\Longrightarrow$ *homs*($pass_1'$ *hs*) = $pass_1$ (*homs hs*)
apply(*induction hs rule*: *Pairing_Heap_List1.*$pass_1$*.induct*)
apply *simp*
subgoal for *h*
apply(*case_tac h*)
apply (*auto*)
done

subgoal for *h1 h2*
apply(*case_tac h1*)
apply *simp*
apply(*case_tac h2*)
apply (*auto*)
done
done

lemma *hom_merge'*: ⟦ *no_Emptys lhs*; *Pairing_Heap_List1_Analysis.is_root h*⟧
⟹ *hom* (*merge'* (*Hp x lhs*) *h*) = *link* ⟨*homs lhs*, *x*, *hom h*⟩
by(*cases h*) *auto*

lemma *hom_pass2'*: *no_Emptys hs* ⟹ *hom*(*pass*$_2$' *hs*) = *pass*$_2$ (*homs hs*)
by(*induction hs rule*: *homs.induct*) (*auto simp*: *hom_merge' is_root_pass2*)

lemma *del_min'*: *is_root' h* ⟹ *hom*(*del_min' h*) = *del_min* (*hom h*)
by(*cases h*)
(*auto simp*: *homs_pass1' hom_pass2' no_Emptys_pass1 is_root_pass2*)

lemma *insert'*: *is_root' h* ⟹ *hom*(*insert' x h*) = *insert x* (*hom h*)
by(*cases h*)(*auto*)

lemma *merge'*:
⟦ *is_root' h1*; *is_root' h2* ⟧ ⟹ *hom*(*merge' h1 h2*) = *merge* (*hom h1*) (*hom h2*)
apply(*cases h1*)
apply(*simp*)
apply(*cases h2*)
apply(*auto*)
done

lemma *t_pass1'*: *no_Emptys hs* ⟹ t_{pass1}' *hs* = t_{pass1}(*homs hs*)
apply(*induction hs rule*: *Pairing_Heap_List1.pass*$_1$.*induct*)
apply *simp*
subgoal for *h*
apply(*case_tac h*)
apply (*auto*)
done
subgoal for *h1 h2*
apply(*case_tac h1*)
apply *simp*
apply(*case_tac h2*)
apply (*auto*)

```
done
done

lemma t_pass2': no_Emptys hs ⟹ t_pass2' hs = t_pass2(homs hs)
by(induction hs rule: homs.induct) (auto simp: hom_merge' is_root_pass2)

lemma size_hp: is_root' h ⟹ size_hp h = size (hom h)
proof(induction h)
  case (Hp _ hs) thus ?case
    apply(induction hs rule: homs.induct)
     apply simp
    apply force
    apply simp
    done
qed simp

interpretation Amortized2
where arity = arity and exec = exec and inv = is_root
and cost = cost and Φ = Φ and U = U
and hom = hom
and exec' = Pairing_Heap_List1_Analysis.exec
and cost' = Pairing_Heap_List1_Analysis.cost and inv' = is_root'
and U' = Pairing_Heap_List1_Analysis.U
proof (standard, goal_cases)
  case (1 _ f) thus ?case
    by (cases f)(auto simp: merge' del_min' numeral_eq_Suc)
next
  case (2 ts f)
  show ?case
  proof(cases f)
    case [simp]: Del_min
    then obtain h where [simp]: ts = [h] using 2 by auto
    show ?thesis using 2
      by(cases h) (auto simp: is_root_pass2 no_Emptys_pass1)
  qed (insert 2,
      auto simp: Pairing_Heap_List1_Analysis.is_root_merge numeral_eq_Suc)
next
  case (3 t) thus ?case by (cases t) (auto)
next
  case (4 ts f) show ?case
  proof (cases f)
    case [simp]: Del_min
    then obtain h where [simp]: ts = [h] using 4 by auto
    show ?thesis using 4
```

by (*cases h*)(*auto simp*: *t_pass1'* *t_pass2'* *no_Emptys_pass1* *homs_pass1'*)
qed (*insert 4*, *auto*)
next
case (*5 _ f*) **thus** *?case* **by**(*cases f*) (*auto simp*: *size_hp numeral_eq_Suc*)
qed

end

7.4 Okasaki's Pairing Heap (Modified)

theory *Pairing_Heap_List2_Analysis*
imports
Pairing_Heap.Pairing_Heap_List2
Amortized_Framework
Priority_Queue_ops_merge
Lemmas_log
begin

Amortized analysis of a modified version of the pairing heaps defined by Okasaki [6].

fun *lift_hp* :: *'b* ⇒ (*'a hp* ⇒ *'b*) ⇒ *'a heap* ⇒ *'b* **where**
lift_hp c f None = *c* |
lift_hp c f (*Some h*) = *f h*

fun *size_hps* :: *'a hp list* ⇒ *nat* **where**
size_hps(*Hp x hsl # hsr*) = *size_hps hsl* + *size_hps hsr* + *1* |
size_hps [] = *0*

definition *size_hp* :: *'a hp* ⇒ *nat* **where**
[*simp*]: *size_hp h* = *size_hps*(*hps h*) + *1*

fun Φ*_hps* :: *'a hp list* ⇒ *real* **where**
Φ*_hps* [] = *0* |
Φ*_hps* (*Hp x hsl # hsr*) = Φ*_hps hsl* + Φ*_hps hsr* + *log 2* (*size_hps hsl* + *size_hps hsr* + *1*)

definition Φ*_hp* :: *'a hp* ⇒ *real* **where**
[*simp*]: Φ*_hp h* = Φ*_hps* (*hps h*) + *log 2* (*size_hps*(*hps*(*h*))+*1*)

abbreviation Φ :: *'a heap* ⇒ *real* **where**
Φ ≡ *lift_hp 0* Φ*_hp*

abbreviation *size_heap* :: *'a heap* ⇒ *nat* **where**
size_heap ≡ *lift_hp 0 size_hp*

lemma Φ_hps_ge0: $\Phi\_hps\ hs \geq 0$
by (*induction hs rule*: *size_hps.induct*) *auto*

declare *algebra_simps*[*simp*]

lemma *size_hps_Cons*[*simp*]: $size\_hps(h \# hs) = size\_hp\ h + size\_hps\ hs$
by(*cases h*) *simp*

lemma *link2*: *link* (*Hp x lx*) *h* = (*case h of* (*Hp y ly*) ⇒
(*if x* < *y then Hp x* (*Hp y ly* # *lx*) *else Hp y* (*Hp x lx* # *ly*)))
by(*simp split*: *hp.split*)

lemma *size_hps_link*: $size\_hps(hps\ (link\ h1\ h2)) = size\_hp\ h1 + size\_hp\ h2 - 1$
by (*induction rule*: *link.induct*) *simp_all*

lemma $pass_1\_size$[*simp*]: $size\_hps\ (pass_1\ hs) = size\_hps\ hs$
by (*induct hs rule*: $pass_1.induct$) (*simp_all add*: *size_hps_link*)

lemma $pass_2\_None$[*simp*]: $pass_2\ hs = None \longleftrightarrow hs = []$
by(*cases hs*) *auto*

lemma $\Delta\Phi\_insert$:
Φ (*Pairing_Heap_List2.insert x h*) − Φ *h* ≤ *log 2* (*size_heap h* + *1*)
by(*induct h*)(*auto simp*: *link2 split*: *hp.split*)

lemma $\Delta\Phi\_link$: $\Phi\_hp\ (link\ h1\ h2) - \Phi\_hp\ h1 - \Phi\_hp\ h2 \leq 2 * log\ 2\ (size\_hp\ h1 + size\_hp\ h2)$
by (*induction h1 h2 rule*: *link.induct*) (*simp add*: *add_increasing*)

fun *sum_ub* :: *'a hp list* ⇒ *real* **where**
sum_ub [] = *0*
| *sum_ub* [*Hp* _ _] = *0*
| *sum_ub* [*Hp* _ *lx*, *Hp* _ *ly*] = *2*log 2* (*2* + *size_hps lx* + *size_hps ly*)
| *sum_ub* (*Hp* _ *lx* # *Hp* _ *ly* # *ry*) = *2*log 2* (*2* + *size_hps lx* + *size_hps ly* + *size_hps ry*)
− *2*log 2* (*size_hps ry*) − *2* + *sum_ub ry*

lemma $\Delta\Phi\_pass1\_sum\_ub$: $\Phi\_hps\ (pass_1\ h) - \Phi\_hps\ h\ \leq sum\_ub\ h$
proof (*induction h rule*: *sum_ub.induct*)
case (*3 lx x ly y*)
have *0*: ⋀*x y*::*real*. *0* ≤ *x* ⟹ *x* ≤ *y* ⟹ *x* ≤ *2*y* **by** *linarith*

show *?case* **by** (*simp add*: *add_increasing 0*)
next
case (*4 x hsx y hsy z hsize_hp*)
let *?ry* = *z* # *hsize_hp*
let *?rx* = *Hp y hsy* # *?ry*
let *?h* = *Hp x hsx* # *?rx*
have Φ_*hps*(*pass*$_1$ *?h*) − Φ_*hps ?h*
≤ *log 2* (*1* + *size_hps hsx* + *size_hps hsy*) − *log 2* (*1* + *size_hps hsy* + *size_hps ?ry*) + *sum_ub ?ry*
using *4.IH* **by** *simp*
also have *log 2* (*1* + *size_hps hsx* + *size_hps hsy*) − *log 2* (*1* + *size_hps hsy* + *size_hps ?ry*)
≤ *2*∗*log 2* (*size_hps ?h*) − *2*∗*log 2* (*size_hps ?ry*) − *2*
proof −
have *log 2* (*1* + *size_hps hsx* + *size_hps hsy*) + *log 2* (*size_hps ?ry*) − *2*∗*log 2* (*size_hps ?h*)
= *log 2* ((*1* + *size_hps hsx* + *size_hps hsy*)/(*size_hps ?h*)) + *log 2* (*size_hps ?ry* / *size_hps ?h*)
by (*simp add*: *log_divide*)
also have ... ≤ −*2*
proof −
have *2* + ...
≤ *2*∗*log 2* ((*1* + *size_hps hsx* + *size_hps hsy*) / *size_hps ?h* + *size_hps ?ry* / *size_hps ?h*)
using *ld_sum_inequality* [*of* (*1* + *size_hps hsx* + *size_hps hsy*) / *size_hps ?h* (*size_hps ?ry* / *size_hps ?h*)] **by** *simp*
also have ... ≤ *0* **by** (*simp add*: *field_simps log_divide add_pos_nonneg*)
finally show *?thesis* **by** *linarith*
qed
finally have *log 2* (*1* + *size_hps hsx* + *size_hps hsy*) + *log 2* (*size_hps ?ry*) + *2*
≤ *2*∗*log 2* (*size_hps ?h*) **by** *simp*
moreover have *log 2* (*size_hps ?ry*) ≤ *log 2* (*size_hps ?rx*) **by** *simp*
ultimately have *log 2* (*1* + *size_hps hsx* + *size_hps hsy*) − ...
≤ *2*∗*log 2* (*size_hps ?h*) − *2*∗*log 2* (*size_hps ?ry*) − *2* **by** *linarith*
thus *?thesis* **by** *simp*
qed
finally show *?case* **by** (*simp*)
qed *simp_all*

lemma Δ Φ_*pass1*: **assumes** *hs* ≠ []
shows Φ_*hps* (*pass*$_1$ *hs*) − Φ_*hps hs* ≤ *2* ∗ *log 2* (*size_hps hs*) − *length hs* + *2*

```
proof −
  have sum_ub hs ≤ 2 ∗ log 2 (size_hps hs) − length hs + 2
    using assms by (induct hs rule: sum_ub.induct) (simp_all)
  thus ?thesis using ΔΦ_pass1_sum_ub[of hs] by linarith
qed

lemma size_hps_pass2: pass₂ hs = Some h ⟹ size_hps hs = size_hps(hps h)+1
apply(induction hs arbitrary: h rule: Φ_hps.induct)
apply (auto simp: link2 split: option.split hp.split)
done

lemma ΔΦ_pass2: hs ≠ [] ⟹ Φ (pass₂ hs) − Φ_hps hs ≤ log 2 (size_hps hs)
proof (induction hs)
  case (Cons h hs)
  thus ?case
  proof −
    obtain x hs2 where [simp]: h = Hp x hs2 by (metis hp.exhaust)
    show ?thesis
    proof (cases pass₂ hs)
      case [simp]: (Some h2)
      obtain y hs3 where [simp]: h2 = Hp y hs3 by (metis hp.exhaust)
      from size_hps_pass2[OF Some] Cons show ?thesis
        by(cases hs=[])(auto simp: add_mono)
    qed simp
  qed
qed simp

lemma ΔΦ_del_min: assumes hps h ≠ []
  shows Φ (del_min (Some h)) − Φ (Some h)
  ≤ 3 ∗ log 2 (size_hps(hps h)) − length(hps h) + 2
proof −
  let ?ΔΦ₁ = Φ_hps(hps h) − Φ_hp h
  let ?ΔΦ₂ = Φ(pass₂(pass₁ (hps h))) − Φ_hps (hps h)
  let ?ΔΦ = Φ (del_min (Some h)) − Φ (Some h)
 have Φ(pass₂(pass₁(hps h))) − Φ_hps (pass₁(hps h)) ≤ log 2 (size_hps(hps h))
    using ΔΦ_pass2[of pass₁(hps h)] using size_hps.elims assms by force
 moreover have Φ_hps (pass₁ (hps h)) − Φ_hps (hps h) ≤ 2∗... − length (hps h) + 2
    using ΔΦ_pass1[OF assms] by blast
  moreover have ?ΔΦ₁ ≤ 0 by (cases h) simp
  moreover have ?ΔΦ = ?ΔΦ₁ + ?ΔΦ₂ by (cases h) simp
```

ultimately show *?thesis* **by** *linarith*
qed

fun *exec* :: *'a* :: *linorder op* ⇒ *'a heap list* ⇒ *'a heap* **where**
exec Empty [] = *None* |
exec Del_min [*h*] = *del_min h* |
exec (*Insert x*) [*h*] = *Pairing_Heap_List2.insert x h* |
exec Merge [*h1*,*h2*] = *merge h1 h2*

fun t_{pass1} :: *'a hp list* ⇒ *nat* **where**
t_{pass1} [] = *1*
| t_{pass1} [_] = *1*
| t_{pass1} (_ # _ # *hs*) = *1* + t_{pass1} *hs*

fun t_{pass2} :: *'a hp list* ⇒ *nat* **where**
t_{pass2} [] = *1*
| t_{pass2} (_ # *hs*) = *1* + t_{pass2} *hs*

fun *cost* :: *'a* :: *linorder op* ⇒ *'a heap list* ⇒ *nat* **where**
cost Empty _ = *1* |
cost Del_min [*None*] = *1* |
cost Del_min [*Some*(*Hp x hs*)] = *1* + t_{pass2} ($pass_1$ *hs*) + t_{pass1} *hs* |
cost (*Insert a*) _ = *1* |
cost Merge _ = *1*

fun *U* :: *'a* :: *linorder op* ⇒ *'a heap list* ⇒ *real* **where**
U Empty _ = *1* |
U (*Insert a*) [*h*] = *log 2* (*size_heap h* + *1*) + *1* |
U Del_min [*h*] = *3*∗*log 2* (*size_heap h* + *1*) + *5* |
U Merge [*h1*,*h2*] = *2*∗*log 2* (*size_heap h1* + *size_heap h2* + *1*) + *1*

interpretation *pairing*: *Amortized*
where *arity* = *arity* **and** *exec* = *exec* **and** *cost* = *cost* **and** *inv* = λ_. *True*
and Φ = Φ **and** *U* = *U*
proof (*standard*, *goal_cases*)
case (*2 s*) **show** *?case* **by** (*cases s*) (*auto simp*: Φ_*hps_ge0*)
next
case (*3 ss f*) **show** *?case*
proof (*cases f*)
case *Empty* **with** *3* **show** *?thesis* **by**(*auto*)
next
case *Insert*
thus *?thesis* **using** *Insert* ΔΦ_*insert 3* **by** *auto*

```
  next
    case [simp]: Del_min
    then obtain ho where [simp]: ss = [ho] using 3 by auto
    show ?thesis
    proof (cases ho)
      case [simp]: (Some h)
        show ?thesis
        proof (cases h)
        case [simp]: (Hp x hs)
        have t_pass2 (pass_1 hs) + t_pass1 hs ≤ 2 + length hs
          by (induct hs rule: pass_1.induct) simp_all
        hence  cost f ss ≤ 1 + ... by simp
        moreover have  Φ (del_min ho) − Φ ho ≤ 3∗log 2 (size_heap ho +
1) − length hs + 2
        proof (cases hs = [])
          case False
          hence Φ (del_min ho) − Φ ho ≤ 3∗log 2 (size_hps hs) − length hs
+ 2
            using  ΔΦ_del_min[of h] by simp
          also have ... ≤ 3∗log 2 (size_heap ho + 1) − length hs + 2
            using False size_hps.elims by force
          finally show ?thesis .
        qed simp
        ultimately show ?thesis by simp
      qed
    qed simp
  next
    case [simp]: Merge
    then obtain ho1 ho2 where [simp]: ss = [ho1, ho2]
      using 3 by(auto simp: numeral_eq_Suc)
    show ?thesis
    proof (cases ho1 = None ∨ ho2 = None)
      case True thus ?thesis by auto
    next
      case False
       then obtain h1 h2 where [simp]: ho1 = Some h1 ho2 = Some h2
by auto
       have Φ (merge ho1 ho2) − Φ ho1 − Φ ho2 ≤ 2 ∗ log 2 (size_heap
ho1 + size_heap ho2)
        using ΔΦ_link[of h1 h2] by simp
      also have ... ≤ 2 ∗ log 2 (size_hp h1 + size_hp h2 + 1) by (simp)
      finally show ?thesis by(simp)
    qed
  qed
```

qed *simp*

end

References

[1] H. Brinkop. Verifikation der amortisierten Laufzeit von Pairing Heaps in Isabelle, 2015. Bachelor's Thesis, Fakultät für Informatik, Technische Universität München.

[2] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[3] A. Kaldewaij and B. Schoenmakers. The derivation of a tighter bound for top-down skew heaps. *Information Processing Letters*, 37:265–271, 1991.

[4] T. Nipkow. Amortized complexity verified. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving (ITP 2015)*, volume 9236 of *LNCS*, pages 310–324. Springer, 2015.

[5] T. Nipkow and H. Brinkop. Amortized complexity verified, 2016. Submitted for publication.

[6] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[7] B. Schoenmakers. A systematic analysis of splaying. *Information Processing Letters*, 45:41–50, 1993.