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 Although it is superseded by the model in *Amortized_Framework* that allows arbitrarily many parameters, it is still of interest because of its simplicity.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*: $inv(state\ f\ n)$
⟨*proof*⟩

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*: $(\sum i{<}n.\ t\ (f\ i)\ (state\ f\ i)) = (\sum i{<}n.\ a\ f\ i) - \Phi(state\ f\ n)$
⟨*proof*⟩

corollary *ta*: $(\sum i{<}n.\ t\ (f\ i)\ (state\ f\ i)) \leq (\sum i{<}n.\ a\ f\ i)$
⟨*proof*⟩

lemma *aa1*: $a\ f\ i \leq U\ (f\ i)\ (state\ f\ i)$
⟨*proof*⟩

lemma *ub*: $(\sum i{<}n.\ t\ (f\ i)\ (state\ f\ i)) \leq (\sum i{<}n.\ U\ (f\ i)\ (state\ f\ i))$
⟨*proof*⟩

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* ⇒ *real* **where**
t_{incr} [] = *1* |
t_{incr} (*False*#*bs*) = *1* |
t_{incr} (*True*#*bs*) = t_{incr} *bs* + *1*

definition *p_incr* :: *bool list* ⇒ *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*
⟨*proof*⟩

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*⟩

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*⟩

locale *table_insert* =
fixes *a* :: *real*

fixes $c :: real$
assumes $c1[arith]$: $c > 1$
assumes $ac2$: $a \geq c/(c - 1)$
begin

lemma ac: $a \geq 1/(c - 1)$
⟨*proof*⟩

lemma $a0[arith]$: $a>0$
⟨*proof*⟩

definition $b = 1/(c - 1)$

lemma $b0[arith]$: $b > 0$
⟨*proof*⟩

fun $ins :: nat * nat \Rightarrow nat * nat$ **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 $inv = \lambda(n,l).$ *if* $l=0$ *then* $n=0$ *else* $n \leq l \wedge (b/a)*l \leq n$
and $t = \lambda\_.\ t_{ins}$ **and** $\Phi = pins$ **and** $U = \lambda\_\ \_.\ a + 1$
⟨*proof*⟩

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* = λ_. *True*
and *t* = t_{stk} **and** Φ = *length* **and** *U* = λ*f* _. *case f of Push* _ ⇒ *2* | *Pop* _ ⇒ *0*
⟨*proof*⟩

1.4 Queue

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

datatype *'a* op_q = *Enq 'a* | *Deq*

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

fun nxt_q :: *'a* op_q ⇒ *'a queue* ⇒ *'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 ⇒ *'a queue* ⇒ *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* = λ_. *True*
and *t* = t_q **and** Φ = λ(*xs*,*ys*). *length xs* **and** *U* = λ*f* _. *case f of Enq* _ ⇒ *2* | *Deq* ⇒ *0*
⟨*proof*⟩

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

fun *nxt_q2* :: *'a* op_q ⇒ *'a queue* ⇒ *'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 ⇒ *'a queue* ⇒ *real* **where**
t_q2 (*Enq* _) (*xs*,*ys*) = *1* + (*if size xs* + *1* ≤ *size ys then 0 else size xs* + *1* + *size ys*) |
t_q2 Deq (*xs*,*ys*) = (*if size xs* ≤ *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*⟩

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*⟩

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* $\Rightarrow$ *nat*
fixes *exec* :: *'op* $\Rightarrow$ *'s list* $\Rightarrow$ *'s*
fixes *inv* :: *'s* $\Rightarrow$ *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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

corollary *t_sum_le_a_sum*: *wf ot* ⟹ *cost_sum ot* ≤ *acost_sum ot*
⟨*proof*⟩

lemma *a_le_U*: ⟦ ∀ *s* ∈ *set ss*. *inv s*; *length ss* = *arity f* ⟧ ⟹ *acost f ss* ≤ *U f ss*
⟨*proof*⟩

lemma *u_sum_le_U_sum*: *wf ot* ⟹ *acost_sum ot* ≤ *U_sum ot*
⟨*proof*⟩

corollary *t_sum_le_U_sum*: *wf ot* $\Longrightarrow$ *cost_sum ot* $\leq$ *U_sum ot*
⟨*proof*⟩

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* :: $'op \Rightarrow$ *nat* **and** *exec* **and** *inv* :: $'s \Rightarrow$ *bool* **and** *cost* Φ *U* +
fixes *exec′* :: $'op \Rightarrow$ $'t$ *list* $\Rightarrow$ $'t$
fixes *inv′* :: $'t \Rightarrow$ *bool*
fixes *cost′* :: $'op \Rightarrow$ $'t$ *list* $\Rightarrow$ *nat*
fixes *U′* :: $'op \Rightarrow$ $'t$ *list* $\Rightarrow$ *real*
fixes *hom* :: $'t \Rightarrow$ $'s$
assumes *exec′*: $[\![\forall s \in set\ ts.\ inv'\ s;\ length\ ts = arity\ f\ ]\!]$
  $\Longrightarrow$ *hom*(*exec′ f ts*) = *exec f* (*map hom ts*)
assumes *inv_exec′*: $[\![\forall s \in set\ ss.\ inv'\ s;\ length\ ss = arity\ f\ ]\!]$
  $\Longrightarrow$ *inv′*(*exec′ f ss*)
assumes *inv_hom*: *inv′ t* $\Longrightarrow$ *inv* (*hom t*)
assumes *cost′*: $[\![\forall s \in set\ ts.\ inv'\ s;\ length\ ts = arity\ f\ ]\!]$
  $\Longrightarrow$ *cost′ f ts* = *cost f* (*map hom ts*)
assumes *U′*: $[\![\forall s \in set\ ts.\ inv'\ s;\ length\ ts = arity\ f\ ]\!]$
  $\Longrightarrow$ *U′ f ts* = *U f* (*map hom ts*)
begin

sublocale *A′*: *Amortized arity exec′ inv′ cost′* Φ *o hom U′*
⟨*proof*⟩

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*
⟨*proof*⟩

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* = λ*f* _. *case f of Empty* ⇒ *1* | *Incr* ⇒ *2*
⟨*proof*⟩

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 = λf _. case f of Empty ⇒ 1 | Push _ ⇒ 2 | Pop _ ⇒ 0*
⟨*proof*⟩

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* = λ*f* _. *case f of Empty* ⇒ *1* | *Ins* ⇒ *3*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *a0*[*arith*]: *a*>*0*
⟨*proof*⟩

definition *b* = *1*/(*c* − *1*)

lemma *b0*[*arith*]: *b* > *0*
⟨*proof*⟩

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* = λ*f* _. *case f of Empty* ⇒ *1* | *Ins* ⇒ *a* + *1*
⟨*proof*⟩

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* = λ*f* _. *case f of Empty* ⇒ *1* | *Ins* ⇒ *3* | *Del* ⇒ *2*
⟨*proof*⟩

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* = λ*f* _. *case f of Empty* ⇒ *0* | *Enq* _ ⇒ *2* | *Deq* ⇒ *0*

⟨*proof*⟩

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* = λ*f* _. *case f of Empty* ⇒ *0* | *Enq _* ⇒ *2* | *Deq* ⇒ *0*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

corollary *Glog*: *lrh h ≤ log 2* (*size1 h*)
⟨*proof*⟩

lemma *Dexp*: *2 ^ rlh h ≤ size h + 1*
⟨*proof*⟩

corollary *Dlog*: *rlh h ≤ log 2* (*size1 h*)
⟨*proof*⟩

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*

⟨*proof*⟩
termination
⟨*proof*⟩

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

lemma Φ_*nneg*: Φ *t* ≥ *0*
⟨*proof*⟩

lemma *plus_log_le_2log_plus*: ⟦ *x* > *0*; *y* > *0*; *b* > *1* ⟧
⟹ *log b x* + *log b y* ≤ *2* ∗ *log b* (*x* + *y*)
⟨*proof*⟩

lemma *rh1*: *rh l r* ≤ *1*
⟨*proof*⟩

lemma *amor_le_long*:
t_merge t1 t2 + Φ (*merge t1 t2*) − Φ *t1* − Φ *t2* ≤
lrh(*merge t1 t2*) + *rlh t1* + *rlh t2* + *1*
⟨*proof*⟩

lemma *amor_le*:
t_merge t1 t2 + Φ (*merge t1 t2*) − Φ *t1* − Φ *t2* ≤
lrh(*merge t1 t2*) + *rlh t1* + *rlh t2* + *1*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

4.0.1 Instantiation of Amortized Framework

lemma *t_merge_nneg*: $t\_merge\ h1\ h2 \geq 0$
⟨*proof*⟩

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*⟩

end
theory *Lemmas_log*
imports *Complex_Main*
begin

lemma *ld_sum_inequality*:
assumes $x > 0\ y > 0$
shows $log\ 2\ x + log\ 2\ y + 2 \leq 2 * log\ 2\ (x + y)$
⟨*proof*⟩

lemma *ld_ld_1_less*:
$[\![x > 0;\ y > 0\ ]\!] \Longrightarrow 1 + log\ 2\ x + log\ 2\ y < 2 * log\ 2\ (x+y)$
⟨*proof*⟩

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*⟩

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*⟩

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

fun *t_splay* :: *'a*::*linorder* $\Rightarrow$ *'a tree* $\Rightarrow$ *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 $\Rightarrow 1$ |
Node ll c lr $\Rightarrow$
(*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)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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* $(not \circ (<))$ $(mirror\ t)$. 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*: $lb \neq \langle\rangle$ **and** *IH*: $t\_splay\ x\ lb + \Phi\ lb1 + \Phi\ lb2 - \Phi\ lb \leq 2 * \varphi\ lb - 3 * \varphi\ X + 1$ **and**
prems: $size\ lb = size\ lb1 + size\ lb2 + 1\ X \in subtrees\ lb$
shows $t\_splay\ x\ lb + \Phi\ t' - \Phi\ t \leq 3 * (\varphi\ t - \varphi\ X)$
⟨*proof*⟩

lemma *a_splay_ub*: $[\![\ bst\ t;\ Node\ lx\ x\ rx : subtrees\ t\ ]\!]$
$\Longrightarrow a\_splay\ x\ t \leq 3 * (\varphi\ t - \varphi(Node\ lx\ x\ rx)) + 1$
⟨*proof*⟩

lemma *a_splay_ub2*: **assumes** $bst\ t\ a : set\_tree\ t$
shows $a\_splay\ a\ t \leq 3 * (\varphi\ t - 1) + 1$
⟨*proof*⟩

lemma *a_splay_ub3*: **assumes** *bst t* **shows** $a\_splay\ a\ t \leq 3 * \varphi\ t + 1$
⟨*proof*⟩

5.2.2 Analysis of insert

lemma *amor_insert*: **assumes** *bst t*
shows $t\_splay\ a\ t + \Phi(Splay\_Tree.insert\ a\ t) - \Phi\ t \leq 4 * log\ 2\ (size1\ t) + 2$ (**is** $?l \leq ?r$)
⟨*proof*⟩

5.2.3 Analysis of delete

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

lemma *a_splay_max_ub*: $[\![\ bst\ t;\ t \neq Leaf\ ]\!] \Longrightarrow a\_splay\_max\ t \leq 3 * (\varphi\ t - 1) + 1$
⟨*proof*⟩

lemma *a_splay_max_ub3*: **assumes** *bst t* **shows** $a\_splay\_max\ t \leq 3 * \varphi\ t + 1$
⟨*proof*⟩

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*⟩

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*⟩

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*]: α > *1*
assumes *A1*: ⟦*1* ≤ *x*; *1* ≤ *y*; *1* ≤ *z*⟧ ⟹
(*x*+*y*) ∗ (*y*+*z*) *powr* β ≤ (*x*+*y*) *powr* β ∗ (*x*+*y*+*z*)
assumes *A2*: ⟦*1* ≤ *l*′; *1* ≤ *r*′; *1* ≤ *lr*; *1* ≤ *r*⟧ ⟹
α ∗ (*l*′+*r*′) ∗ (*lr*+*r*) *powr* β ∗ (*lr*+*r*′+*r*) *powr* β
≤ (*l*′+*r*′) *powr* β ∗ (*l*′+*lr*+*r*′) *powr* β ∗ (*l*′+*lr*+*r*′+*r*)
assumes *A3*: ⟦*1* ≤ *l*′; *1* ≤ *r*′; *1* ≤ *ll*; *1* ≤ *r*⟧ ⟹
α ∗ (*l*′+*r*′) ∗ (*l*′+*ll*) *powr* β ∗ (*r*′+*r*) *powr* β
≤ (*l*′+*r*′) *powr* β ∗ (*l*′+*ll*+*r*′) *powr* β ∗ (*l*′+*ll*+*r*′+*r*)
begin

lemma *nl2*: $[\![\ ll \geq 1;\ lr \geq 1;\ r \geq 1\ ]\!] \Longrightarrow$
$log\ \alpha\ (ll + lr) + \beta * log\ \alpha\ (lr + r)$
$\leq \beta * log\ \alpha\ (ll + lr) + log\ \alpha\ (ll + lr + r)$
⟨*proof*⟩

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

fun $\Phi :: 'a\ tree \Rightarrow real$ **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*]: $A\ a\ (Node\ l\ a\ r) = 1$
$a<b \Longrightarrow A\ a\ (Node\ (Node\ ll\ a\ lr)\ b\ r) = \varphi\ lr\ r - \varphi\ lr\ ll + 1$
$b<a \Longrightarrow A\ a\ (Node\ l\ b\ (Node\ rl\ a\ rr)) = \varphi\ rl\ l - \varphi\ rr\ rl + 1$
⟨*proof*⟩

lemma *A_ub*: $[\![\ bst\ t;\ Node\ la\ a\ ra : subtrees\ t\ ]\!]$
$\Longrightarrow A\ a\ t \leq log\ \alpha\ ((size1\ t)/(size1\ la + size1\ ra)) + 1$
⟨*proof*⟩

lemma *A_ub2*: **assumes** $bst\ t\ a : set\_tree\ t$
shows $A\ a\ t \leq log\ \alpha\ ((size1\ t)/2) + 1$
⟨*proof*⟩

lemma *A_ub3*: **assumes** $bst\ t$ **shows** $A\ a\ t \leq log\ \alpha\ (size1\ t) + 1$
⟨*proof*⟩

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

lemma *Am_simp3'*: $[\![\ c<b;\ bst\ rr;\ rr \neq Leaf]\!] \Longrightarrow$
$Am\ (Node\ l\ c\ (Node\ rl\ b\ rr)) =$
$(case\ splay\_max\ rr\ of\ Node\ rrl\ \_\ rrr \Rightarrow$
$Am\ rr + \varphi\ rrl\ (Node\ l\ c\ rl) + \varphi\ l\ rl - \varphi\ rl\ rr - \varphi\ rrl\ rrr + 1)$
⟨*proof*⟩

lemma *Am_ub*: $[\![\ bst\ t;\ t \neq Leaf\ ]\!] \Longrightarrow Am\ t \leq log\ \alpha\ ((size1\ t)/2) + 1$
⟨*proof*⟩

lemma *Am_ub3*: **assumes** *bst t* **shows** $Am\ t \leq log\ \alpha\ (size1\ t) + 1$
⟨*proof*⟩

end

5.3.2 Optimal Interpretation

lemma *mult_root_eq_root*:
$n{>}0 \Longrightarrow y \geq 0 \Longrightarrow root\ n\ x * y = root\ n\ (x * (y\ \hat{}\ n))$
⟨*proof*⟩

lemma *mult_root_eq_root2*:
$n{>}0 \Longrightarrow y \geq 0 \Longrightarrow y * root\ n\ x = root\ n\ ((y\ \hat{}\ n) * x)$
⟨*proof*⟩

lemma *powr_inverse_numeral*:
$0 < x \Longrightarrow x\ powr\ (1\ /\ numeral\ n) = root\ (numeral\ n)\ x$
⟨*proof*⟩

lemmas *root_simps* = *mult_root_eq_root mult_root_eq_root2 powr_inverse_numeral*

lemma *nl31*: $[\![\ (l'::real) \geq 1;\ r' \geq 1;\ lr \geq 1;\ r \geq 1\ ]\!] \Longrightarrow$
$4 * (l' + r') * (lr + r) \leq (l' + lr + r' + r)\ \hat{}\ 2$
⟨*proof*⟩

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*⟩

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*⟩

lemma *nl41*: **assumes** $(l'::real) \geq 1\ r' \geq 1\ ll \geq 1\ r \geq 1$
shows $4 * (l' + ll) * (r' + r) \leq (l' + ll + r' + r)\ \hat{}\ 2$
⟨*proof*⟩

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*⟩

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*⟩

lemma *cancel*: $x>(0::real) \Longrightarrow c * x\ \hat{}\ 2 * y * z \leq u * v \Longrightarrow c * x\ \hat{}\ 3 * y * z \leq x * u * v$
⟨*proof*⟩

interpretation *S34*: *Splay_Analysis root 3 4 1/3*
⟨*proof*⟩

lemma *log4_log2*: *log 4 x = log 2 x / 2*
⟨*proof*⟩

declare *log_base_root*[*simp*]

lemma *A34_ub*: **assumes** *bst t*
shows *S34.A a t* $\leq$ *(3/2) * log 2 (size1 t) + 1*
⟨*proof*⟩

lemma *Am34_ub*: **assumes** *bst t*
shows *S34.Am t* $\leq$ *(3/2) * log 2 (size1 t) + 1*
⟨*proof*⟩

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*⟩

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* $\Rightarrow$ *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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

lemma *size_pass*$_1$: *size* (*pass*$_1$ *h*) = *size h*
⟨*proof*⟩

lemma *size_pass*$_2$: *size* (*pass*$_2$ *h*) = *size h*
⟨*proof*⟩

lemma *size_merge*:
is_root h1 ⟹ *is_root h2* ⟹ *size* (*merge h1 h2*) = *size h1* + *size h2*
⟨*proof*⟩

lemma Δ Φ_*insert*: *is_root h* ⟹ Φ (*insert x h*) − Φ *h* ≤ *log 2* (*size h* + *1*)

⟨*proof*⟩

lemma $\Delta\Phi$_*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*⟩

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 $\Delta\Phi$_*pass1_upperbound*: Φ ($pass_1$ *hs*) − Φ *hs* ≤ *upperbound hs*
⟨*proof*⟩

lemma $\Delta\Phi$_*pass1*: **assumes** *hs* ≠ *Leaf*
shows Φ ($pass_1$ *hs*) − Φ *hs* ≤ *2*∗*log 2* (*size hs*) − *len hs* + *2*
⟨*proof*⟩

lemma $\Delta\Phi$_*pass2*: *hs* ≠ *Leaf* ⟹ Φ ($pass_2$ *hs*) − Φ *hs* ≤ *log 2* (*size hs*)
⟨*proof*⟩

lemma $\Delta\Phi$_*mergepairs*: **assumes** *hs* ≠ *Leaf*
shows Φ (*merge_pairs hs*) − Φ *hs* ≤ *3* ∗ *log 2* (*size hs*) − *len hs* + *2*
⟨*proof*⟩

lemma $\Delta\Phi$_*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*⟩

lemma *is_root_merge*:
is_root h1 ⟹ *is_root h2* ⟹ *is_root* (*merge h1 h2*)
⟨*proof*⟩

lemma *is_root_insert*: *is_root h* ⟹ *is_root* (*insert x h*)
⟨*proof*⟩

lemma *is_root_del_min*:
assumes *is_root h* **shows** *is_root* (*del_min h*)

⟨*proof*⟩

lemma *pass*$_1$*_len*: *len* (*pass*$_1$ *h*) ≤ *len h*
⟨*proof*⟩

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} (*pass*$_1$ *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*⟩

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*
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *pass1_Nil_iff*: *pass*$_1$ *hs* = [] ⟷ *hs* = []
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *no_Emptys_hps*: *no_Empty h* ⟹ *no_Emptys*(*hps h*)
⟨*proof*⟩

lemma *no_Empty_merge*: ⟦ *no_Empty h1*; *no_Empty h2*⟧ ⟹ *no_Empty*
(*merge h1 h2*)
⟨*proof*⟩

lemma *is_root_merge*: ⟦ *is_root h1*; *is_root h2*⟧ ⟹ *is_root* (*merge h1 h2*)
⟨*proof*⟩

lemma *no_Emptys_pass1*:
no_Emptys hs ⟹ *no_Emptys* (*pass*$_1$ *hs*)
⟨*proof*⟩

lemma *is_root_pass2*: *no_Emptys hs* ⟹ *is_root*(*pass*$_2$ *hs*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *no_Empty_ge0*: *no_Empty h* ⟹ *size_hp h* > *0*
⟨*proof*⟩

declare *algebra_simps*[*simp*]

lemma Φ*_hps1*: Φ*_hps* [*h*] = Φ *h*
⟨*proof*⟩

lemma *size_hp_merge*: *size_hp*(*merge h1 h2*) = *size_hp h1* + *size_hp h2*
⟨*proof*⟩

lemma *pass$_1$_size*[*simp*]: *size_hps* (*pass$_1$ hs*) = *size_hps hs*
⟨*proof*⟩

lemma ΔΦ_*insert*:
Φ (*Pairing_Heap_List1.insert x h*) − Φ *h* ≤ *log 2* (*size_hp h* + *1*)
⟨*proof*⟩

lemma ΔΦ_*merge*:
Φ (*merge h1 h2*) − Φ *h1* − Φ *h2*
≤ *log 2* (*size_hp h1* + *size_hp h2* + *1*) + *1*
⟨*proof*⟩

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*⟩

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

lemma *size_hps_pass2*: *hs* ≠ [] ⟹ *no_Emptys hs* ⟹
no_Empty(*pass$_2$ hs*) & *size_hps hs* = *size_hps*(*hps*(*pass$_2$ hs*))+*1*
⟨*proof*⟩

lemma ΔΦ_*pass2*: *hs* ≠ [] ⟹ *no_Emptys hs* ⟹
Φ (*pass$_2$ hs*) − Φ_*hps hs* ≤ *log 2* (*size_hps hs*)
⟨*proof*⟩

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*⟩

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*⟩

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* ⇒ *'a tree* **where**
homs [] = *Leaf* |
homs (*Hp x lhs # rhs*) = *Node* (*homs lhs*) *x* (*homs rhs*)

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

lemma *homs_pass1'*: *no_Emptys hs* ⟹ *homs*($pass_1'$ *hs*) = $pass_1$ (*homs hs*)
⟨*proof*⟩

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*⟩
⟨*proof*⟩

lemma *hom_pass2'*: *no_Emptys hs* ⟹ *hom*($pass_2'$ *hs*) = $pass_2$ (*homs hs*)
⟨*proof*⟩

lemma *del_min'*: *is_root' h* ⟹ *hom*(*del_min' h*) = *del_min* (*hom h*)
⟨*proof*⟩

lemma *insert'*: *is_root' h* ⟹ *hom*(*insert' x h*) = *insert x* (*hom h*)
⟨*proof*⟩

lemma *merge'*:
⟦ *is_root' h1*; *is_root' h2* ⟧ ⟹ *hom*(*merge' h1 h2*) = *merge* (*hom h1*) (*hom h2*)
⟨*proof*⟩

lemma *t_pass1'*: *no_Emptys hs* ⟹ t_{pass1}' *hs* = t_{pass1}(*homs hs*)
⟨*proof*⟩

lemma *t_pass2'*: *no_Emptys hs* ⟹ t_{pass2}' *hs* = t_{pass2}(*homs hs*)
⟨*proof*⟩

lemma *size_hp*: *is_root' h* $\Longrightarrow$ *size_hp h* = *size* (*hom h*)
⟨*proof*⟩

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*⟩

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*]: $\Phi\_hp\ h = \Phi\_hps\ (hps\ h) + log\ 2\ (size\_hps(hps(h))+1)$

abbreviation Φ :: *'a heap* ⇒ *real* **where**
$\Phi \equiv lift\_hp\ 0\ \Phi\_hp$

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

lemma Φ_hps_ge0: $\Phi\_hps\ hs \geq 0$
⟨*proof*⟩

declare *algebra_simps*[*simp*]

lemma *size_hps_Cons*[*simp*]: *size_hps*(*h # hs*) = *size_hp h* + *size_hps hs*
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *size_hps_link*: *size_hps*(*hps* (*link h1 h2*)) = *size_hp h1* + *size_hp h2*
− *1*
⟨*proof*⟩

lemma $pass_1\_size$[*simp*]: *size_hps* ($pass_1$ *hs*) = *size_hps hs*
⟨*proof*⟩

lemma $pass_2\_None$[*simp*]: $pass_2\ hs = None \longleftrightarrow hs = []$
⟨*proof*⟩

lemma $\Delta\Phi\_insert$:
Φ (*Pairing_Heap_List2.insert x h*) − Φ *h* ≤ *log 2* (*size_heap h* + *1*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma $\Delta\Phi\_pass1$: **assumes** $hs \neq []$
shows $\Phi\_hps\ (pass_1\ hs) - \Phi\_hps\ hs \leq 2 * log\ 2\ (size\_hps\ hs) - length\ hs + 2$
⟨*proof*⟩

lemma $size\_hps\_pass2$: $pass_2\ hs = Some\ h \Longrightarrow size\_hps\ hs = size\_hps(hps\ h)+1$
⟨*proof*⟩

lemma $\Delta\Phi\_pass2$: $hs \neq [] \Longrightarrow \Phi\ (pass_2\ hs) - \Phi\_hps\ hs \leq log\ 2\ (size\_hps\ hs)$
⟨*proof*⟩

lemma $\Delta\Phi\_del\_min$: **assumes** $hps\ h \neq []$
shows $\Phi\ (del\_min\ (Some\ h)) - \Phi\ (Some\ h)$
$\leq 3 * log\ 2\ (size\_hps(hps\ h)) - length(hps\ h) + 2$
⟨*proof*⟩

fun $exec :: 'a :: linorder\ op \Rightarrow 'a\ heap\ list \Rightarrow 'a\ heap$ **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 $t_{pass1} :: 'a\ hp\ list \Rightarrow nat$ **where**
$t_{pass1}\ [] = 1$
$| t_{pass1}\ [\_] = 1$
$| t_{pass1}\ (\_ \#\ \_ \#\ hs) = 1 + t_{pass1}\ hs$

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

fun $cost :: 'a :: linorder\ op \Rightarrow 'a\ heap\ list \Rightarrow nat$ **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** $\Phi = \Phi$ **and** *U = U*
⟨*proof*⟩

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.