Amortized Complexity Verified

Tobias Nipkow

March 17, 2025

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 Although it is superseded by the model in *Amortized_Framework* that allows arbitrarily many parameters, it is still of interest because of its simplicity.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* ⇒ *'s* ⇒ *'s*
fixes *inv* :: *'s* ⇒ *bool*
fixes *T* :: *'o* ⇒ *'s* ⇒ *real*
fixes Φ :: *'s* ⇒ *real*
fixes *U* :: *'o* ⇒ *'s* ⇒ *real*
assumes *inv_init*: *inv init*
assumes *inv_nxt*: *inv s* ⟹ *inv*(*nxt f s*)
assumes *ppos*: *inv s* ⟹ Φ *s* ≥ *0*
assumes *p0*: Φ *init* = *0*
assumes *U*: *inv s* ⟹ *T f s* + Φ(*nxt f s*) − Φ *s* ≤ *U f s*
begin

fun *state* :: (*nat* ⇒ *'o*) ⇒ *nat* ⇒ *'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* ⇒ *'o*) ⇒ *nat* ⇒ *real* **where**
A f i = *T* (*f i*) (*state f i*) + Φ(*state f* (*i*+*1*)) − Φ(*state f i*)

lemma *aeq*: (∑ *i*<*n*. *T* (*f i*) (*state f i*)) = (∑ *i*<*n*. *A f i*) − Φ(*state f n*)
⟨*proof*⟩

corollary *TA*: (∑ *i*<*n*. *T* (*f i*) (*state f i*)) ≤ (∑ *i*<*n*. *A f i*)
⟨*proof*⟩

lemma *aa1*: *A f i* ≤ *U* (*f i*) (*state f i*)
⟨*proof*⟩

lemma *ub*: $(\sum i<n.\ T\ (f\ i)\ (state\ f\ i)) \le (\sum i<n.\ U\ (f\ i)\ (state\ f\ i))$
⟨*proof*⟩

end

1.1 Binary Counter

locale *BinCounter*
begin

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

lemma *A_incr*: *T_incr bs* + Φ(*incr bs*) − Φ *bs* = *2*
⟨*proof*⟩

interpretation *incr*: *Amortized*
where *init* = [] **and** *nxt* = %_. *incr* **and** *inv* = λ_. *True*
and *T* = λ_. *T_incr* **and** Φ = Φ **and** *U* = λ_ _. *2*
⟨*proof*⟩

thm *incr.ub*

end

1.2 Dynamic tables: insert only

locale *DynTable1*
begin

fun *ins* :: *nat*∗*nat* ⇒ *nat*∗*nat* **where**
ins (*n*,*l*) = (*n*+*1*, *if n*<*l then l else if l*=*0 then 1 else 2*∗*l*)

fun *T_ins* :: *nat*∗*nat* ⇒ *real* **where**

T_ins (*n*,*l*) = (*if n*<*l then 1 else if l*=*0 then 1 else n*+*1*)

fun *invar* :: *nat*∗*nat* ⇒ *bool* **where**
invar (*n*,*l*) = (*l*/*2* ≤ *n* ∧ *n* ≤ *l*)

fun Φ :: *nat*∗*nat* ⇒ *real* **where**
Φ (*n*,*l*) = *2*∗(*real n*) − *l*

interpretation *ins*: *Amortized*
where *init* = (*0*::*nat*,*0*::*nat*)
and *nxt* = λ_. *ins*
and *inv* = *invar*
and *T* = λ_. *T_ins* **and** Φ = Φ **and** *U* = λ_ _. *3*
⟨*proof*⟩

end

locale *table_insert* = *DynTable1* +
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*⟩

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 $T = \lambda\_.\ T\_ins$ **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** $inv = \lambda\_.\ True$
and $T = T\_stk$ **and** $\Phi = length$ **and** $U = \lambda f\ \_.\ case\ f\ of\ Push\ \_ \Rightarrow 2\ \mid$
$Pop\ \_ \Rightarrow 0$
⟨*proof*⟩

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**
$nxt\_q\ (Enq\ x)\ (xs,ys) = (x\#xs,ys)\ \mid$
$nxt\_q\ Deq\ (xs,ys) = (if\ ys = []\ then\ ([],\ tl(rev\ xs))\ else\ (xs,tl\ ys))$

fun $T\_q :: 'a\ op_q \Rightarrow 'a\ queue \Rightarrow real$ **where**
$T\_q\ (Enq\ x)\ (xs,ys) = 1\ \mid$
$T\_q\ Deq\ (xs,ys) = (if\ ys = []\ then\ length\ xs\ else\ 0)$

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*⟩

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 $inv = \lambda(xs,ys).$ *size xs* $\leq$ *size ys*
and $T = T\_q2$ **and** $\Phi = \lambda(xs,ys).\ 2 *$ *size xs*
and $U = \lambda f\ \_.$ *case f of Enq* $\_ \Rightarrow 3 \mid Deq \Rightarrow 0$
⟨*proof*⟩

1.5 Dynamic tables: insert and delete

datatype $op_{tb} = Ins \mid Del$

locale *DynTable2* = *DynTable1*
begin

fun *del* :: $nat*nat \Rightarrow nat*nat$ **where**
del $(n,l) = (n - 1,$ *if* $n=1$ *then* 0 *else if* $4*(n - 1)<l$ *then l div* 2 *else* $l)$

fun *T_del* :: $nat*nat \Rightarrow$ *real* **where**
T_del $(n,l) = ($*if* $n=1$ *then* 1 *else if* $4*(n - 1)<l$ *then n else* $1)$

fun *nxt_tb* :: $op_{tb} \Rightarrow nat*nat \Rightarrow nat*nat$ **where**
nxt_tb Ins = *ins* |
nxt_tb Del = *del*

fun *T_tb* :: $op_{tb} \Rightarrow nat*nat \Rightarrow$ *real* **where**

T_tb Ins = *T_ins* |
T_tb Del = *T_del*

fun *invar* :: *nat*∗*nat* ⇒ *bool* **where**
invar (*n*,*l*) = (*n* ≤ *l*)

fun Φ :: *nat*∗*nat* ⇒ *real* **where**
Φ (*n*,*l*) = (*if n* < *l*/*2 then l*/*2* − *n else 2*∗*n* − *l*)

interpretation *tb*: *Amortized*
where *init* = (*0*,*0*) **and** *nxt* = *nxt_tb*
and *inv* = *invar*
and *T* = *T_tb* **and** Φ = Φ
and *U* = λ*f _*. *case f of Ins* ⇒ *3* | *Del* ⇒ *2*
⟨*proof*⟩

end

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 :: $'op\ rose\_tree \Rightarrow bool$ **where**
$wf\ (T\ f\ ts) = (length\ ts = arity\ f \wedge (\forall t \in set\ ts.\ wf\ t))$

fun $state$ :: $'op\ rose\_tree \Rightarrow 's$ **where**
$state\ (T\ f\ ts) = exec\ f\ (map\ state\ ts)$

lemma inv_state: $wf\ ot \Longrightarrow inv(state\ ot)$
⟨*proof*⟩

definition $acost$ :: $'op \Rightarrow 's\ list \Rightarrow real$ **where**
$acost\ f\ ss = cost\ f\ ss + \Phi\ (exec\ f\ ss) - sum\_list\ (map\ \Phi\ ss)$

fun $acost\_sum$ :: $'op\ rose\_tree \Rightarrow real$ **where**
$acost\_sum\ (T\ f\ ts) = acost\ f\ (map\ state\ ts) + sum\_list\ (map\ acost\_sum\ ts)$

fun $cost\_sum$ :: $'op\ rose\_tree \Rightarrow real$ **where**
$cost\_sum\ (T\ f\ ts) = cost\ f\ (map\ state\ ts) + sum\_list\ (map\ cost\_sum\ ts)$

fun U_sum :: $'op\ rose\_tree \Rightarrow real$ **where**
$U\_sum\ (T\ f\ ts) = U\ f\ (map\ state\ ts) + sum\_list\ (map\ U\_sum\ ts)$

lemma $t\_sum\_a\_sum$: $wf\ ot \Longrightarrow cost\_sum\ ot = acost\_sum\ ot - \Phi(state\ ot)$
⟨*proof*⟩

corollary $t\_sum\_le\_a\_sum$: $wf\ ot \Longrightarrow cost\_sum\ ot \le acost\_sum\ ot$
⟨*proof*⟩

lemma a_le_U: $[\![\ \forall s \in set\ ss.\ inv\ s;\ length\ ss = arity\ f\ ]\!] \Longrightarrow acost\ f\ ss \le U\ f\ ss$
⟨*proof*⟩

lemma $a\_sum\_le\_U\_sum$: $wf\ ot \Longrightarrow acost\_sum\ ot \le U\_sum\ ot$
⟨*proof*⟩

corollary $t\_sum\_le\_U\_sum$: $wf\ ot \Longrightarrow cost\_sum\ ot \le U\_sum\ ot$
⟨*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 $inv' = inv \circ hom$: 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* $exec'\ inv'\ cost'\ \Phi\ o\ hom\ U'$
$\langle proof \rangle$

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* = λ*f* _. *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 = λf _. 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 = λf _. 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 = λf _. 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
HOL−Data_Structures.Define_Time_Function
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 t* ≤ *size t* + *1*
⟨*proof*⟩

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

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

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

time_fun *merge*

fun Φ :: *'a tree* ⇒ *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*: $[\![x > 0; y > 0; b > 1]\!]$
$\Longrightarrow log\ b\ x + log\ b\ y \leq 2 * log\ b\ (x + y)$
⟨*proof*⟩

lemma *rh1*: $rh\ l\ r \leq 1$
⟨*proof*⟩

lemma *amor_le_long*:
$T\_merge\ t1\ t2 + \Phi\ (merge\ t1\ t2) - \Phi\ t1 - \Phi\ t2 \leq$
$lrh(merge\ t1\ t2) + rlh\ t1 + rlh\ t2 + 1$
⟨*proof*⟩

lemma *amor_le*:
$T\_merge\ t1\ t2 + \Phi\ (merge\ t1\ t2) - \Phi\ t1 - \Phi\ t2 \leq$
$lrh(merge\ t1\ t2) + rlh\ t1 + rlh\ t2 + 1$
⟨*proof*⟩

lemma *a_merge*:
$T\_merge\ t1\ t2 + \Phi(merge\ t1\ t2) - \Phi\ t1 - \Phi\ t2 \leq$
$3 * log\ 2\ (size1\ t1 + size1\ t2) + 1$ (**is** $?l \leq \_$)
⟨*proof*⟩

Command *time_fun* does not work for *skew_heap.insert* and *skew_heap.del_min* because they are the result of a locale and not what they seem. However, their manual definition is trivial:

definition $T\_insert :: 'a::linorder \Rightarrow 'a\ tree \Rightarrow int$ **where**
$T\_insert\ a\ t = T\_merge\ (Node\ Leaf\ a\ Leaf)\ t$

lemma *a_insert*: $T\_insert\ a\ t + \Phi(skew\_heap.insert\ a\ t) - \Phi\ t \leq 3 * log\ 2\ (size1\ t + 2) + 1$
⟨*proof*⟩

definition $T\_del\_min :: ('a::linorder)\ tree \Rightarrow int$ **where**
$T\_del\_min\ t = (case\ t\ of\ Leaf \Rightarrow 0 \mid Node\ t1\ a\ t2 \Rightarrow T\_merge\ t1\ t2)$

lemma *a_del_min*: $T\_del\_min\ t + \Phi(skew\_heap.del\_min\ t) - \Phi\ t \leq 3 * log\ 2\ (size1\ t + 2) + 1$
⟨*proof*⟩

4.0.1 Instantiation of Amortized Framework

lemma *T_merge_nneg*: $T\_merge\ t1\ t2 \geq 0$
⟨*proof*⟩

```
fun exec :: 'a::linorder op ⇒ 'a tree list ⇒ 'a tree where
exec Empty [] = Leaf |
exec (Insert a) [t] = skew_heap.insert a t |
exec Del_min [t] = skew_heap.del_min t |
exec Merge [t1,t2] = merge t1 t2

fun cost :: 'a::linorder op ⇒ 'a tree list ⇒ nat where
cost Empty [] = 1 |
cost (Insert a) [t] = T_merge (Node Leaf a Leaf) t + 1 |
cost Del_min [t] = (case t of Leaf ⇒ 1 | Node t1 a t2 ⇒ T_merge t1 t2
+ 1) |
cost Merge [t1,t2] = T_merge t1 t2

fun U where
U Empty [] = 1 |
U (Insert _) [t] = 3 * log 2 (size1 t + 2) + 2 |
U Del_min [t] = 3 * log 2 (size1 t + 2) + 2 |
U Merge [t1,t2] = 3 * log 2 (size1 t1 + size1 t2) + 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 ≤ 2 * log 2 (x + y)
⟨proof⟩

lemma ld_ld_1_less:
  ⟦x > 0; y > 0 ⟧ ⟹ 1 + log 2 x + log 2 y < 2 * log 2 (x+y)
⟨proof⟩


lemma ld_le_2ld:
  assumes x ≥ 0 y ≥ 0 shows log 2 (1+x+y) ≤ 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
HOL−Data_Structures.Define_Time_Function
begin

declare *size1_size*[*simp*]

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

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

time_fun *cmp*
time_fun *splay* **equations** *splay.simps*(*1*) *splay_code*

lemma *T_splay_simps*[*simp*]:
T_splay a (*Node l a r*) = *1*
x<*b* ⟹ *T_splay x* (*Node Leaf b CD*) = *1*
a<*b* ⟹ *T_splay a* (*Node* (*Node A a B*) *b CD*) = *1*
x<*a* ⟹ *x*<*b* ⟹ *T_splay x* (*Node* (*Node A a B*) *b CD*) =
(*if A* = *Leaf then 1 else T_splay x A* + *1*)
x<*b* ⟹ *a*<*x* ⟹ *T_splay x* (*Node* (*Node A a B*) *b CD*) =
(*if B* = *Leaf then 1 else T_splay x B* + *1*)
b<*x* ⟹ *T_splay x* (*Node AB b Leaf*) = *1*
b<*a* ⟹ *T_splay a* (*Node AB b* (*Node C a D*)) = *1*
b<*x* ⟹ *x*<*c* ⟹ *T_splay x* (*Node AB b* (*Node C c D*)) =
(*if C*=*Leaf then 1 else T_splay x C* + *1*)
b<*x* ⟹ *c*<*x* ⟹ *T_splay x* (*Node AB b* (*Node C c D*)) =
(*if D*=*Leaf then 1 else T_splay x D* + *1*)
⟨*proof*⟩

declare *T_splay.simps(2)[simp del]*

time_fun *insert*

lemma *T_insert_simp*: $T\_insert\ x\ t = (if\ t = Leaf\ then\ 0\ else\ T\_splay\ x\ t)$
⟨*proof*⟩

time_fun *splay_max*

time_fun *delete*

lemma *ex_in_set_tree*: $t \neq Leaf \Longrightarrow bst\ t \Longrightarrow$
$\exists\, x' \in set\_tree\ t.\ splay\ x'\ t = splay\ x\ t \wedge T\_splay\ x'\ t = T\_splay\ x\ t$
⟨*proof*⟩

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

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

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

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

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* (λ*x y*. ¬ *x* < *y*) (*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*⟩

lemma *A_splay_ub*: ⟦ *bst t*; *Node l x r* : *subtrees t* ⟧
⟹ *A_splay x t* ≤ *3* ∗ (φ *t* − φ(*Node l x r*)) + *1*
⟨*proof*⟩

lemma *A_splay_ub2*: **assumes** *bst t x* : *set_tree t*
shows *A_splay x t* ≤ *3* ∗ (φ *t* − *1*) + *1*
⟨*proof*⟩

lemma *A_splay_ub3*: **assumes** *bst t* **shows** *A_splay x t* ≤ *3* ∗ φ *t* + *1*
⟨*proof*⟩

5.2.2 Analysis of insert

lemma *amor_insert*: **assumes** *bst t*
shows *T_insert x t* + Φ(*Splay_Tree.insert x t*) − Φ *t* ≤ *4* ∗ *log 2* (*size1 t*) + *2* (**is** *?l* ≤ *?r*)
⟨*proof*⟩

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*: $t \neq Leaf \Longrightarrow A\_splay\_max\ t \leq 3 * (\varphi\ t - 1) + 1$
⟨*proof*⟩

lemma *A_splay_max_ub3*: $A\_splay\_max\ t \leq 3 * \varphi\ t + 1$
⟨*proof*⟩

lemma *amor_delete*: **assumes** *bst t*
shows $T\_delete\ a\ t + \Phi(Splay\_Tree.delete\ a\ t) - \Phi\ t \leq 6 * log\ 2\ (size1\ t) + 2$
⟨*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*) + *3* |
U (*Delete* _) [*t*] = *6* ∗ *log 2* (*size1 t*) + *3*

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*]: $\alpha > 1$
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*: $[\![1 \le l';\ 1 \le r';\ 1 \le lr;\ 1 \le r]\!] \Longrightarrow$
$\alpha * (l'+r') * (lr+r)\ powr\ \beta * (lr+r'+r)\ powr\ \beta$
$\le (l'+r')\ powr\ \beta * (l'+lr+r')\ powr\ \beta * (l'+lr+r'+r)$
assumes *A3*: $[\![1 \le l';\ 1 \le r';\ 1 \le ll;\ 1 \le r]\!] \Longrightarrow$
$\alpha * (l'+r') * (l'+ll)\ powr\ \beta * (r'+r)\ powr\ \beta$
$\le (l'+r')\ powr\ \beta * (l'+ll+r')\ powr\ \beta * (l'+ll+r'+r)$
begin

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

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

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

definition *A* :: *'a::linorder* ⇒ *'a tree* ⇒ *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*: $[\![\ bst\ t;\ Node\ la\ a\ ra : subtrees\ t\ ]\!]$
$\Longrightarrow A\ a\ t \le log\ \alpha\ ((size1\ t)/(size1\ la + size1\ ra)) + 1$
⟨*proof*⟩

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

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

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*)
⟨*proof*⟩

lemma *Am_ub*: ⟦ *bst t*; *t* ≠ *Leaf* ⟧ ⟹ *Am t* ≤ *log* α ((*size1 t*)/*2*) + *1*
⟨*proof*⟩

lemma *Am_ub3*: **assumes** *bst t* **shows** *Am t* ≤ *log* α (*size1 t*) + *1*
⟨*proof*⟩

end

5.3.2 Optimal Interpretation

lemma *mult_root_eq_root*:
n>0 ⟹ *y* ≥ *0* ⟹ *root n x* ∗ *y* = *root n* (*x* ∗ (*y* ^ *n*))
⟨*proof*⟩

lemma *mult_root_eq_root2*:
n>0 ⟹ *y* ≥ *0* ⟹ *y* ∗ *root n x* = *root n* ((*y* ^ *n*) ∗ *x*)
⟨*proof*⟩

lemma *powr_inverse_numeral*:
0 < *x* ⟹ *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*) ≥ *1*; *r'* ≥ *1*; *lr* ≥ *1*; *r* ≥ *1* ⟧ ⟹
4 ∗ (*l'* + *r'*) ∗ (*lr* + *r*) ≤ (*l'* + *lr* + *r'* + *r*)^*2*
⟨*proof*⟩

lemma *nl32*: **assumes** (*l'::real*) ≥ *1* *r'* ≥ *1* *lr* ≥ *1* *r* ≥ *1*
shows *4* ∗ (*l'* + *r'*) ∗ (*lr* + *r*) ∗ (*lr* + *r'* + *r*) ≤ (*l'* + *lr* + *r'* + *r*)^*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
HOL−Data_Structures.Define_Time_Function
begin

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

time_fun *partition*

time_fun *insert*

time_fun *del_min*

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_del_min 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_partition p r + Φ *r1'* + Φ *r2'* − Φ *r* ≤ *2* ∗ φ *r* + *1*
shows *T_partition 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_partition p r + Φ *r1'* + Φ *r2'* − Φ *r* ≤ *2* ∗ φ *r* + *1*
shows *T_partition 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_partition 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 [] = *0* |
cost (*Insert a*) [*t*] = *T_insert a t* |
cost Del_min [*t*] = *T_del_min t*

fun *U* **where**
U Empty [] = *0* |
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
HOL−Data_Structures.Define_Time_Function
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].

7.1.1 Analysis

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 x r*) = *log 2* (*size* (*Node l x r*)) + Φ *l* + Φ *r*

lemma *link_size*[*simp*]: *size* (*link hp*) = *size hp*
⟨*proof*⟩

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

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

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

⟨*proof*⟩

lemma $\Delta\Phi$_*insert*: *is_root hp* $\Longrightarrow$ Φ (*insert x hp*) $-$ Φ *hp* $\leq$ *log 2* (*size hp* + *1*)
⟨*proof*⟩

lemma $\Delta\Phi$_*merge*:
assumes *h1* = *Node hs1 x1 Leaf h2* = *Node hs2 x2 Leaf*
shows Φ (*merge h1 h2*) $-$ Φ *h1* $-$ Φ *h2* $\leq$ *log 2* (*size h1* + *size h2*) + *1*
⟨*proof*⟩

fun *ub_pass*$_1$ :: *'a tree* $\Rightarrow$ *real* **where**
ub_pass$_1$ (*Node _ _ Leaf*) = *0*
| *ub_pass*$_1$ (*Node hs1 _* (*Node hs2 _ Leaf*)) = *2*∗*log 2* (*size hs1* + *size hs2* + *2*)
| *ub_pass*$_1$ (*Node hs1 _* (*Node hs2 _ hs*)) = *2*∗*log 2* (*size hs1* + *size hs2* + *size hs* + *2*)
$-$ *2*∗*log 2* (*size hs*) $-$ *2* + *ub_pass*$_1$ *hs*

lemma $\Delta\Phi$_*pass1_ub_pass1*: *hs* $\neq$ *Leaf* $\Longrightarrow$ Φ (*pass*$_1$ *hs*) $-$ Φ *hs* $\leq$ *ub_pass*$_1$ *hs*
⟨*proof*⟩

lemma $\Delta\Phi$_*pass1*: **assumes** *hs* $\neq$ *Leaf*
shows Φ (*pass*$_1$ *hs*) $-$ Φ *hs* $\leq$ *2*∗*log 2* (*size hs*) $-$ *len hs* + *2*
⟨*proof*⟩

lemma $\Delta\Phi$_*pass2*: *hs* $\neq$ *Leaf* $\Longrightarrow$ Φ (*pass*$_2$ *hs*) $-$ Φ *hs* $\leq$ *log 2* (*size hs*)
⟨*proof*⟩

lemma $\Delta\Phi$_*del_min*: **assumes** *hs* $\neq$ *Leaf*
shows Φ (*del_min* (*Node hs x Leaf*)) $-$ Φ (*Node hs x Leaf*)
$\leq$ *3*∗*log 2* (*size hs*) $-$ *len hs* + *2*
⟨*proof*⟩

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

7.1.2 Putting it all together (boiler plate)

fun *exec* :: *'a* :: *linorder op* $\Rightarrow$ *'a tree list* $\Rightarrow$ *'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

time_fun *link*

lemma *T_link_0[simp]: T_link h = 0*
⟨*proof*⟩

time_fun *pass*$_1$

time_fun *pass*$_2$

time_fun *del_min*

time_fun *merge*

lemma *T_merge_0[simp]: T_merge h1 h2 = 0*
⟨*proof*⟩

time_fun *insert*

fun *cost* :: *'a :: linorder op ⇒ 'a tree list ⇒ nat* **where**
  cost Empty [] = 0
| *cost Del_min [hp] = T_del_min hp*
| *cost (Insert a) [hp] = T_insert a hp*
| *cost Merge [h1,h2] = T_merge h1 h2*

fun *U* :: *'a :: linorder op ⇒ 'a tree list ⇒ real* **where**
  U Empty [] = 0
| *U (Insert a) [h] = log 2 (size h + 1)*
| *U Del_min [h] = 3∗log 2 (size h + 1) + 4*
| *U Merge [h1,h2] = log 2 (size h1 + size h2 + 1) + 1*

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 Binary Tree Representation (Simplified)

theory *Pairing_Heap_Tree_Analysis2*
imports
  HOL−Data_Structures.Define_Time_Function

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. As in [2, 1], except that the treatment of $pass_1$ is simplified.

7.2.1 Analysis

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 x r*) = *log 2* (*size* (*Node l x r*)) + Φ *l* + Φ *r*

lemma *link_size*[*simp*]: *size* (*link hp*) = *size hp*
⟨*proof*⟩

lemma *size_pass*$_1$: *size* ($pass_1$ *hp*) = *size hp*
⟨*proof*⟩

lemma *size_pass*$_2$: *size* ($pass_2$ *hp*) = *size hp*
⟨*proof*⟩

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

lemma ΔΦ_*insert*: *is_root hp* ⟹ Φ (*insert x hp*) − Φ *hp* ≤ *log 2* (*size hp* + *1*)
⟨*proof*⟩

lemma ΔΦ_*merge*:
assumes *h1* = *Node hs1 x1 Leaf h2* = *Node hs2 x2 Leaf*
shows Φ (*merge h1 h2*) − Φ *h1* − Φ *h2* ≤ *log 2* (*size h1* + *size h2*) + *1*
⟨*proof*⟩

lemma ΔΦ_*pass1*: Φ ($pass_1$ *hs*) − Φ *hs* ≤ *2* ∗ *log 2* (*size hs* + *1*) − *len hs* + *2*
⟨*proof*⟩

lemma $\Delta\Phi$_*pass2*: $hs \neq Leaf \Longrightarrow \Phi\ (pass_2\ hs) - \Phi\ hs \leq log\ 2\ (size\ hs)$
⟨*proof*⟩

corollary $\Delta\Phi$_*pass2'*: $\Phi\ (pass_2\ hs) - \Phi\ hs \leq log\ 2\ (size\ hs + 1)$
⟨*proof*⟩

lemma $\Delta\Phi$_*del_min*:
$\Phi\ (del\_min\ (Node\ hs\ x\ Leaf)) - \Phi\ (Node\ hs\ x\ Leaf)$
$\leq 2 * log\ 2\ (size\ hs + 1) - len\ hs + 2$
⟨*proof*⟩

lemma $pass_1$_*len*: $len\ (pass_1\ h) \leq len\ h$
⟨*proof*⟩

7.2.2 Putting it all together (boiler plate)

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*

time_fun *link*

lemma *T_link_0*[*simp*]: *T_link h* = *0*
⟨*proof*⟩

time_fun $pass_1$

time_fun $pass_2$

time_fun *del_min*

time_fun *merge*

lemma *T_merge_0*[*simp*]: *T_merge h1 h2* = *0*
⟨*proof*⟩

time_fun *insert*

lemma *A_del_min*: **assumes** *is_root h*
shows $T\_del\_min\ h + \Phi(del\_min\ h) - \Phi\ h \leq 2 * log\ 2\ (size\ h + 1) + 4$
⟨*proof*⟩

lemma *A_insert*: *is_root h* $\Longrightarrow$ *T_insert a h* + Φ(*insert a h*) − Φ *h* ≤ *log 2* (*size h* + *1*)
⟨*proof*⟩

lemma *A_merge*: **assumes** *is_root h1 is_root h2*
shows *T_merge h1 h2* + Φ(*merge h1 h2*) − Φ *h1* − Φ *h2* ≤ *log 2* (*size h1* + *size h2* + *1*) + *1*
⟨*proof*⟩

fun *cost* :: *'a* :: *linorder op* ⇒ *'a tree list* ⇒ *nat* **where**
  cost Empty [] = *0*
| *cost Del_min* [*h*] = *T_del_min h*
| *cost* (*Insert a*) [*h*] = *T_insert a h*
| *cost Merge* [*h1*,*h2*] = *T_merge h1 h2*

fun *U* :: *'a* :: *linorder op* ⇒ *'a tree list* ⇒ *real* **where**
  U Empty [] = *0*
| *U* (*Insert a*) [*h*] = *log 2* (*size h* + *1*)
| *U Del_min* [*h*] = *2* ∗ *log 2* (*size h* + *1*) + *4*
| *U Merge* [*h1*,*h2*] = *log 2* (*size h1* + *size h2* + *1*) + *1*

interpretation *Amortized*
where *arity* = *arity* **and** *exec* = *exec* **and** *cost* = *cost* **and** *inv* = *is_root* **and** Φ = Φ **and** *U* = *U*
⟨*proof*⟩

end

7.3 Okasaki's Pairing Heap

theory *Pairing_Heap_List1_Analysis*
imports
  HOL−Data_Structures.Define_Time_Function
  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*⟩

7.3.1 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*⟩

7.3.2 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*

time_fun *merge*

lemma *T_merge_0*[*simp*]: *T_merge h1 h2* = *0*

⟨*proof*⟩

time_fun *insert*

time_fun $pass_1$

time_fun $pass_2$

time_fun *del_min*

fun *cost* :: *'a* :: *linorder op* ⇒ *'a heap list* ⇒ *nat* **where**
cost Empty _ = 0 |
cost Del_min [hp] = T_del_min hp |
cost (Insert a) [hp] = T_insert a hp |
cost Merge [hp1,hp2] = T_merge hp1 hp2

fun *U* :: *'a* :: *linorder op* ⇒ *'a heap list* ⇒ *real* **where**
U Empty _ = 0 |
U (Insert a) [h] = log 2 (size_hp h + 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) + 1

interpretation *pairing*: *Amortized*
where *arity = arity* **and** *exec = exec* **and** *cost = cost* **and** *inv = is_root*
and Φ = Φ **and** *U = U*
⟨*proof*⟩

end

7.4 Okasaki's Pairing Heaps via Tree Potential

theory *Pairing_Heap_List1_Analysis1*
imports
Pairing_Heap_List1_Analysis
HOL−Library.Tree_Multiset
begin

This theory analyses Okasaki heaps by defining the potential as a composition of mapping the heaps to trees and the standard tree potential.

datatype_compat *heap*

7.4.1 Analysis

fun *trees* :: *'a heap list* ⇒ *'a tree* **where**
trees [] = Leaf |

$trees\ (Hp\ x\ lhs\ \#\ rhs) = Node\ (trees\ lhs)\ x\ (trees\ rhs)$

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

fun Φ :: $'a\ tree \Rightarrow real$ **where**
$\Phi\ Leaf = 0$
$|\ \Phi\ (Node\ l\ x\ r) = log\ 2\ \ (size\ (Node\ l\ x\ r)) + \Phi\ l + \Phi\ r$

abbreviation Φ' :: $'a\ heap \Rightarrow real$ **where**
$\Phi'\ h \equiv \Phi(tree\ h)$

abbreviation Φ'' :: $'a\ heap\ list \Rightarrow real$ **where**
$\Phi''\ hs \equiv \Phi(trees\ hs)$

lemma Φ''_ge0: $no\_Emptys\ hs \Longrightarrow \Phi''\ hs \geq 0$
$\langle proof \rangle$

abbreviation $size'\ h \equiv size(tree\ h)$
abbreviation $size''\ hs \equiv size(trees\ hs)$

lemma $\Delta\Phi\_insert$: $is\_root\ hp \Longrightarrow \Phi'\ (insert\ x\ hp) - \Phi'\ hp \leq log\ 2\ \ (size'\ hp + 1)$
$\langle proof \rangle$

lemma $\Delta\Phi\_merge$:
$\Phi'\ (merge\ h1\ h2) - \Phi'\ h1\ - \Phi'\ h2 \leq log\ 2\ \ (size'\ h1 + size'\ h2 + 1) + 1$
$\langle proof \rangle$

lemma no_EmptyD: $no\_Empty\ h \Longrightarrow \exists\, x\ hs.\ h = Hp\ x\ hs$
$\langle proof \rangle$

lemma $size\_trees\_pass_1$: $no\_Emptys\ hs \Longrightarrow size''(pass_1\ \ hs) = \ size''\ hs$
$\langle proof \rangle$

lemma $\Delta\Phi\_pass1$: $no\_Emptys\ hs \Longrightarrow \Phi''\ (pass_1\ \ hs) - \Phi''\ hs\ \ \leq 2 * log\ 2\ \ (size''\ hs + 1) - length\ hs + 2$
$\langle proof \rangle$

lemma $pass_2\_struct$: $no\_Empty\ h \Longrightarrow \exists\, x\ hs'.\ pass_2\ \ (h\ \#\ hs) = Hp\ x\ hs'$
$\langle proof \rangle$

lemma $size'\_merge$: $size'\ (merge\ (Hp\ x\ hs1)\ h2) = size'(Hp\ x\ hs1) + size'$

h2
⟨*proof*⟩

lemma *size_pass*$_2$: *no_Emptys hs* ⟹ *size′* (*pass*$_2$ *hs*) = *size″ hs*
⟨*proof*⟩

lemma ΔΦ*_pass2*: *hs* ≠ [] ⟹ *no_Emptys hs* ⟹ Φ′ (*pass*$_2$ *hs*) − Φ″ *hs*
≤ *log 2* (*size″ hs*)
⟨*proof*⟩

lemma *trees_not_Leaf*: *hs* ≠ [] ⟹ *no_Emptys hs* ⟹ *trees hs* ≠ *Leaf*
⟨*proof*⟩

corollary ΔΦ*_pass2′*: **assumes** *no_Emptys hs*
shows Φ′ (*pass*$_2$ *hs*) − Φ″ *hs* ≤ *log 2* (*size″ hs* + *1*)
⟨*proof*⟩

lemma ΔΦ*_del_min*: **assumes** *no_Emptys hs*
shows Φ′ (*del_min* (*Hp x hs*)) − Φ′ (*Hp x hs*)
  ≤ *2* ∗ *log 2* (*size″ hs* + *1*) − *length hs* + *2*
⟨*proof*⟩

7.4.2 Putting it all together (boiler plate)

fun *U* :: *′a* :: *linorder op* ⇒ *′a heap list* ⇒ *real* **where**
U Empty _ = *0* |
U (*Insert a*) [*h*] = *log 2* (*size′ h* + *1*) |
U Del_min [*h*] = *2*∗*log 2* (*size′ h* + *1*) + *4* |
U Merge [*h1*,*h2*] = *log 2* (*size′ h1* + *size′ h2* + *1*) + *1*

interpretation *pairing0*: *Amortized*
where *arity* = *arity* **and** *exec* = *exec* **and** *cost* = *cost* **and** *inv* = *is_root*
and Φ = Φ′ **and** *U* = *U*
⟨*proof*⟩

end

7.5 Okasaki's Pairing Heaps via Transfer from Tree Analysis

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_*$pass_1$
abbreviation $T_{pass2}{}'$ == *Pairing_Heap_List1_Analysis.T_*$pass_2$

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_*$pass_1$(*homs hs*)

⟨*proof*⟩

lemma *T_pass2'*: *no_Emptys hs* $\Longrightarrow$ T_{pass2}' *hs* = T_pass_2(*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

7.6 Okasaki's Pairing Heap (Modified)

theory *Pairing_Heap_List2_Analysis*
imports
  Pairing_Heap.Pairing_Heap_List2
  Amortized_Framework
  Priority_Queue_ops_merge
  Lemmas_log
  HOL−Data_Structures.Define_Time_Function
begin

Amortized analysis of a modified version of the pairing heaps defined by Okasaki [6]. Simplified version of proof in the Nipkow and Brinkop paper.

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*

consts *sz* :: *'a* ⇒ *nat*

overloading
size_hps ≡ *sz* :: *'a hp list* ⇒ *nat*
size_hp ≡ *sz* :: *'a hp* ⇒ *nat*
size_heap ≡ *sz* :: *'a heap* ⇒ *nat*
begin

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* = *sz*(*hps h*) + *1*

definition *size_heap* :: *'a heap* ⇒ *nat* **where**
[*simp*]: *size_heap* ≡ *lift_hp 0 sz*

end

consts Φ :: *'a* ⇒ *real*

overloading
Φ*_hps* ≡ Φ :: *'a hp list* ⇒ *real*
Φ*_hp* ≡ Φ :: *'a hp* ⇒ *real*
Φ*_heap* ≡ Φ :: *'a heap* ⇒ *real*
begin

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* (*sz hsl* + *sz hsr* + *1*)

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

definition Φ*_heap* :: *'a heap* ⇒ *real* **where**
[*simp*]: Φ*_heap* ≡ *lift_hp 0* Φ

end

lemma Φ*_hps_ge0*: Φ (*hs*::_ *hp list*) ≥ *0*
⟨*proof*⟩

declare *algebra_simps*[*simp*]

lemma *sz_hps_Cons*[*simp*]: *sz*(*h* # *hs*) = *sz* (*h*::_ *hp*) + *sz 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 *sz_hps_link*: $sz(hps\ (link\ h1\ h2)) = sz\ h1 + sz\ h2 - 1$
⟨*proof*⟩

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

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

lemma ΔΦ*_insert*:
$\Phi\ (Pairing\_Heap\_List2.insert\ x\ h) - \Phi\ h \leq log\ 2\ (sz\ h + 1)$
⟨*proof*⟩

lemma ΔΦ*_link*: $\Phi\ (link\ h1\ h2) - \Phi\ h1 - \Phi\ h2 \leq 2 * log\ 2\ (sz\ h1 + sz\ h2)$
⟨*proof*⟩

lemma ΔΦ*_pass1*: $\Phi\ (pass_1\ hs) - \Phi\ hs \leq 2 * log\ 2\ (sz\ hs + 1) - length\ hs + 2$
⟨*proof*⟩

lemma *size_hps_pass2*: $sz(pass_2\ hs) = sz\ hs$
⟨*proof*⟩

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

corollary ΔΦ*_pass2'*: $\Phi\ (pass_2\ hs) - \Phi\ hs \leq log\ 2\ (sz\ hs + 1)$
⟨*proof*⟩

lemma ΔΦ*_del_min*:
shows $\Phi\ (del\_min\ (Some\ h)) - \Phi\ (Some\ h)$
$\leq 2 * log\ 2\ (sz(hps\ h) + 1) - length\ (hps\ h) + 2$
⟨*proof*⟩

time_fun *link*

lemma *T_link_0*[*simp*]: $T\_link\ h1\ h2 = 0$
⟨*proof*⟩

time_fun $pass_1$

time_fun $pass_2$

time_fun *del_min*

time_fun *Pairing_Heap_List2.insert*

lemma *T_insert_0*[*simp*]: *T_insert a h = 0*
⟨*proof*⟩

time_fun *merge*

lemma *T_merge_0*[*simp*]: *T_merge h1 h2 = 0*
⟨*proof*⟩

lemma *A_insert*: *T_insert a ho* + Φ(*Pairing_Heap_List2.insert a ho*) − Φ *ho* ≤ *log 2* (*sz ho* + *1*)
⟨*proof*⟩

lemma *A_merge*:
T_merge ho1 ho2 + Φ (*merge ho1 ho2*) − Φ *ho1* − Φ *ho2* ≤ *2* ∗ *log 2* (*sz ho1* + *sz ho2* + *1*)
⟨*proof*⟩

lemma *A_del_min*:
T_del_min ho + Φ (*del_min ho*) − Φ *ho* ≤ *2*∗*log 2* (*sz ho* + *1*) + *4*
⟨*proof*⟩

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 *cost* :: ′*a* :: *linorder op* ⇒ ′*a heap list* ⇒ *nat* **where**
cost Empty _ = *0* |
cost Del_min [*h*] = *T_del_min h* |
cost (*Insert a*) [*h*] = *T_insert a h* |
cost Merge [*h1*,*h2*] = *T_merge h1 h2*

fun *U* :: ′*a* :: *linorder op* ⇒ ′*a heap list* ⇒ *real* **where**
U Empty _ = *0* |
U (*Insert a*) [*h*] = *log 2* (*sz h* + *1*) |
U Del_min [*h*] = *2*∗*log 2* (*sz h* + *1*) + *4* |
U Merge [*h1*,*h2*] = *2*∗*log 2* (*sz h1* + *sz h2* + *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.