Karatsuba Multiplication for Integers

Jakob Schulz, Emin Karayel

April 18, 2024

Abstract

We give a verified implementation of the Karatsuba Multiplication on Integers [1] as well as verified runtime bounds. Integers are represented as LSBF (least significant bit first) boolean lists, on which the algorithm by Karatsuba [1] is implemented. The running time of $O\left(n^{\log_2 3}\right)$ is verified using the Time Monad defined in [2].

Contents

1 Preliminaries

Some general preliminaries.

theory *Karatsuba-Preliminaries*
imports *Main Expander-Graphs.Extra-Congruence-Method HOL−Number-Theory.Residues*
begin

lemma *prop-ifI*:
assumes $Q \Longrightarrow P\ R$
assumes $\neg\ Q \Longrightarrow P\ S$

shows P (*if* Q *then* R *else* S)
using *assms* **by** *argo*

lemma *let-prop-cong*:
assumes $T = T'$
assumes $P\ (f\ T)\ (f'\ T')$
shows P (*let* $x = T$ *in* $f\ x$) (*let* $x = T'$ *in* $f'\ x$)
using *assms* **by** *simp*

lemma *set-subseteqD*:
assumes *set xs* $\subseteq A$
shows $\bigwedge i.\ i <$ *length xs* $\Longrightarrow$ *xs* ! $i \in A$
using *assms* **by** *fastforce*

lemma *set-subseteqI*:
assumes $\bigwedge i.\ i <$ *length xs* $\Longrightarrow$ *xs* ! $i \in A$
shows *set xs* $\subseteq A$
using *assms*
by (*metis in-set-conv-nth subsetI*)

lemma *Nat-max-le-sum*: *max* (a :: *nat*) $b \leq a + b$
by *simp*

lemma *upt-add-eq-append'*:
assumes $a \leq b$ $b \leq c$
shows $[a..<c] = [a..<b]$ @ $[b..<c]$
using *assms upt-add-eq-append*[*of* a b $c - b$] **by** *auto*

lemma *map-add-const-upt*: *map* ($\lambda j.\ j + c$) $[a..<b] = [a + c..<b + c]$
proof (*cases* $a < b$)
case *True*
then have *map* ($\lambda j.\ j + c$) $[a..<b]$ = *map* ($\lambda j.\ j + c$) (*map* ($\lambda j.\ j + a$) $[0..<b-a]$)
using *map-add-upt*[*of* a $b - a$] **by** *simp*
also have ... = *map* ($\lambda j.\ j + (a + c)$) $[0..<b-a]$
by *simp*
also have ... = $[a+c..<b+c]$
using *map-add-upt*[*of* $a + c$ $b - a$] *True* **by** *simp*
finally show *?thesis* **.**
next
case *False*
then show *?thesis* **by** *simp*
qed

lemma *filter-even-upt-even*: *filter even* $[0..<2*n]$ = *map* ((∗) 2) $[0..<n]$
by (*induction n*) *simp-all*
lemma *filter-even-upt-odd*: *filter even* $[0..<2*n + 1]$ = *map* ((∗) 2) $[0..<n + 1]$
by (*simp add*: *filter-even-upt-even*)

lemma *filter-odd-upt-even*: *filter odd* $[0..<2*n]$ = *map* ($\lambda i.\ 2*i + 1$) $[0..<n]$

```
  by (induction n) simp-all
lemma filter-odd-upt-odd: filter odd [0..<2*n + 1] = map (λi. 2*i + 1) [0..<n]
  by (simp add: filter-odd-upt-even)

lemma length-filter-even: length (filter even [0..<n]) = (if even n then n div 2 else
n div 2 + 1)
  by (induction n) simp-all
lemma length-filter-odd: length (filter odd [0..<n]) = n div 2
  by (induction n) simp-all

lemma filter-even-nth:
  assumes i < length (filter even [0..<n])
  shows filter even [0..<n] ! i = 2 * i
proof (cases even n)
  case True
  then obtain n' where n = 2 * n' by blast
  then show ?thesis using filter-even-upt-even[of n'] assms by auto
next
  case False
  then obtain n' where n = 2 * n' + 1 using oddE by blast
  show ?thesis
    using assms
    apply (simp only: ‹n = 2 * n' + 1› filter-even-upt-odd length-map nth-map)
    apply (intro arg-cong[where f = (*) 2])
    by (metis add-0 diff-zero length-upt nth-upt)
qed

lemma filter-odd-nth:
  assumes i < length (filter odd [0..<n])
  shows filter odd [0..<n] ! i = 2 * i + 1
proof (cases even n)
  case True
  then obtain n' where n = 2 * n' by blast
  then show ?thesis using filter-odd-upt-even assms by auto
next
  case False
  then obtain n' where n = 2 * n' + 1 using oddE by blast
  then show ?thesis
    using assms
    by (simp only: filter-odd-upt-odd length-map)
       (simp add: ‹n = 2 * n' + 1› length-filter-odd)
qed

fun sublist where
sublist 0 n xs = take n xs
| sublist (Suc m) (Suc n) (a # xs) = sublist m n xs
| sublist (Suc m) 0 xs = []
| sublist (Suc m) (Suc n) [] = []
```

lemma *length-sublist*[*simp*]: *length* (*sublist m n xs*) = *card* ({*m*..<*n*} ∩ {*0*..<*length xs*})
by (*induction m n xs rule*: *sublist.induct*) *simp-all*

lemma *length-sublist′*:
assumes *m* ≤ *n*
assumes *n* ≤ *length xs*
shows *length* (*sublist m n xs*) = *n* − *m*
using *assms* **by** *simp*

lemma *nth-sublist*:
assumes *m* ≤ *n*
assumes *n* ≤ *length xs*
assumes *i* < *n* − *m*
shows *sublist m n xs* ! *i* = *xs* ! (*m* + *i*)
using *assms*
by (*induction m n xs arbitrary*: *i rule*: *sublist.induct*) *simp-all*

lemma *filter-map-map2*:
assumes *length b* = *m*
assumes *length c* = *m*
shows [*f* (*b*!*i*) (*c*!*i*). *i* ← [*0*..<*m*]] = *map2 f b c*
using *assms* **by** (*intro nth-equalityI*) *simp-all*

fun *map3* **where**
map3 f (*x* # *xs*) (*y* # *ys*) (*z* # *zs*) = *f x y z* # *map3 f xs ys zs*
| *map3 f* - - - = []

lemma *map3-as-map*: *map3 f xs ys zs* = *map* (λ((*x*, *y*), *z*). *f x y z*) (*zip* (*zip xs ys*) *zs*)
by (*induction f xs ys zs rule*: *map3.induct*; *simp*)

lemma *filter-map-map3*:
assumes *length b* = *m*
assumes *length c* = *m*
shows [*f* (*b*!*i*) (*c*!*i*) *i*. *i* ← [*0*..<*m*]] = *map3 f b c* [*0*..<*m*]
using *assms*
apply (*intro nth-equalityI*)
unfolding *map3-as-map* **by** *simp-all*

fun *map4* **where**
map4 f (*x* # *xs*) (*y* # *ys*) (*z* # *zs*) (*w* # *ws*) = *f x y z w* # *map4 f xs ys zs ws*
| *map4 f* - - - - = []

lemma *map4-as-map*: *map4 f xs ys zs ws* = *map* (λ(((*x*,*y*),*z*),*w*). *f x y z w*) (*zip* (*zip* (*zip xs ys*) *zs*) *ws*)
by (*induction f xs ys zs ws rule*: *map4.induct*; *simp*)

lemma *nth-map2*:

assumes *i* < *length xs*
assumes *i* < *length ys*
shows *map2 f xs ys* ! *i* = *f* (*xs* ! *i*) (*ys* ! *i*)
using *assms* **by** *simp*
lemma *nth-map3*:
assumes *i* < *length xs*
assumes *i* < *length ys*
assumes *i* < *length zs*
shows *map3 f xs ys zs* ! *i* = *f* (*xs* ! *i*) (*ys* ! *i*) (*zs* ! *i*)
using *assms* **unfolding** *map3-as-map* **by** *simp*
lemma *nth-map4*:
assumes *i* < *length xs*
assumes *i* < *length ys*
assumes *i* < *length zs*
assumes *i* < *length ws*
shows *map4 f xs ys zs ws* ! *i* = *f* (*xs* ! *i*) (*ys* ! *i*) (*zs* ! *i*) (*ws* ! *i*)
using *assms* **unfolding** *map4-as-map* **by** *simp*
lemma *nth-map4'*:
assumes *i* < *l*
assumes *length xs* = *l*
assumes *length ys* = *l*
assumes *length zs* = *l*
assumes *length ws* = *l*
shows *map4 f xs ys zs ws* ! *i* = *f* (*xs* ! *i*) (*ys* ! *i*) (*zs* ! *i*) (*ws* ! *i*)
using *assms* **unfolding** *map4-as-map* **by** *simp*

lemma *map2-of-map-r*: *map2 f xs* (*map g ys*) = *map2* (λ*x y*. *f x* (*g y*)) *xs ys*
by (*intro nth-equalityI*) *simp-all*
lemma *map2-of-map-l*: *map2 f* (*map g xs*) *ys* = *map2* (λ*x y*. *f* (*g x*) *y*) *xs ys*
by (*intro nth-equalityI*) *simp-all*
lemma *map2-of-map2-r*: *map2 f xs* (*map2 g ys zs*) = *map3* (λ*x y z*. *f x* (*g y z*)) *xs ys zs*
unfolding *map3-as-map* **by** (*intro nth-equalityI*) *simp-all*
lemma *map-of-map3*: *map f* (*map3 g xs ys zs*) = *map3* (λ*x y z*. *f* (*g x y z*)) *xs ys zs*
unfolding *map3-as-map* **by** (*intro nth-equalityI*) *simp-all*

lemma *cyclic-index-lemma*:
fixes *n* :: *nat*
assumes σ < *n* ϱ < *n* *i* < *n*
shows (σ + ϱ) *mod n* = *i* ⟷ ϱ = (*n* + *i* − σ) *mod n*
proof
assume (σ + ϱ) *mod n* = *i*
then have (*int* σ + *int* ϱ) *mod* (*int n*) = *int i*
using *zmod-int* **by** *fastforce*
also have ... = (*int n* + *int i*) *mod int n*
using ‹*i* < *n*› **by** *auto*
finally have (*int* σ + *int* ϱ − *int* σ) *mod* (*int n*) = (*int n* + *int i* − *int* σ) *mod int n*

using *mod-diff-cong* **by** *blast*
then have $(int\ \varrho)\ mod\ (int\ n) = (int\ n + int\ i - int\ \sigma)\ mod\ (int\ n)$
by *simp*
also have ... $= (int\ (n + i - \sigma))\ mod\ (int\ n)$
using *assms* **by** (*simp add*: *int-ops*(*6*))
finally show $\varrho = (n + i - \sigma)\ mod\ n$
using *zmod-int assms* **by** (*metis mod-less of-nat-eq-iff*)
next
assume $\varrho = (n + i - \sigma)\ mod\ n$
then have $(\sigma + \varrho)\ mod\ n = (\sigma + (n + i - \sigma))\ mod\ n$
by *presburger*
also have ... $= (n + i)\ mod\ n$
using *assms* **by** *simp*
also have ... $= i$
using *assms* **by** *simp*
finally show $(\sigma + \varrho)\ mod\ n = i$ **.**
qed

lemma (**in** *residues*) *residues-minus-eq*: $x \ominus_R y = (x - y)\ mod\ m$
proof −
have $x \ominus_R y = x \oplus_R (\ominus_R y)$
using *a-minus-def* **by** *fast*
also have $\ominus_R y = (- y)\ mod\ m$
using *res-neg-eq*[*of y*] **.**
also have $x \oplus_R ((-y)\ mod\ m) = (x + ((-y)\ mod\ m))\ mod\ m$
by (*simp add*: *R-m-def residue-ring-def*)
also have ... $= (x - y)\ mod\ m$
by (*simp add*: *mod-add-right-eq*)
finally show *?thesis* **.**
qed

lemma *residue-ring-carrier-eq*: $\{0..(n::int) - 1\} = \{0..<n\}$
by *auto*

context *ring*
begin

fun *nat-embedding* :: $nat \Rightarrow 'a$ **where**
nat-embedding $0 = \mathbf{0}$
| *nat-embedding* $(Suc\ n) =$ *nat-embedding* $n \oplus \mathbf{1}$
fun *int-embedding* :: $int \Rightarrow 'a$ **where**
int-embedding $n = (if\ n \geq 0\ then$ *nat-embedding* $(nat\ n)\ else \ominus$ *nat-embedding* $(nat\ (-n)))$

lemma *nat-embedding-closed*[*simp*]: *nat-embedding* $x \in carrier\ R$
by (*induction x*)(*simp-all*)
lemma *int-embedding-closed*[*simp*]: *int-embedding* $x \in carrier\ R$
by *simp*

lemma *nat-embedding-a-hom*: *nat-embedding* $(x + y) =$ *nat-embedding* $x \oplus$ *nat-embedding* y
  apply (*induction x arbitrary*: *y*)
  using *a-comm a-assoc* **by** *simp-all*
lemma *nat-embedding-m-hom*: *nat-embedding* $(x * y) =$ *nat-embedding* $x \otimes$ *nat-embedding* y
  apply (*induction x arbitrary*: *y*)
  by (*simp-all add*: *nat-embedding-a-hom l-distr a-comm*)
lemma *nat-embedding-exp-hom*: *nat-embedding* $(x \mathbin{\char`^} y) =$ *nat-embedding* x [^] y
  apply (*induction y*)
  by (*simp-all add*: *nat-embedding-m-hom group-commutes-pow*)
lemma *int-embedding-neg-hom*: *int-embedding* $(- x) = \ominus$ *int-embedding* x
  by *simp*

end

lemma *int-exp-hom*: *int* x ^ i = *int* (x ^ i)
  by *simp*

end

2 Auxiliary Sum Lemmas

theory *Karatsuba-Sum-Lemmas*
  imports *Karatsuba-Preliminaries Expander-Graphs.Extra-Congruence-Method*
begin

lemma *sum-list-eq*: $(\bigwedge x.\ x \in$ *set xs* $\Longrightarrow f\ x = g\ x) \Longrightarrow$ *sum-list* (*map f xs*) = *sum-list* (*map g xs*)
  by (*rule arg-cong*[*OF list.map-cong0*])

lemma *sum-list-split-0*: $(\sum i \leftarrow [0..<Suc\ n].\ f\ i) = f\ 0 + (\sum i \leftarrow [1..<Suc\ n].\ f\ i)$
  using *upt-eq-Cons-conv*
proof −
  have $[0..<Suc\ n] = 0\ \#\ [1..<Suc\ n]$ **using** *upt-eq-Cons-conv* **by** *auto*
  then show *?thesis* **by** *simp*
qed
lemma *sum-list-index-trafo*: $(\sum i \leftarrow xs.\ f\ (g\ i)) = (\sum i \leftarrow map\ g\ xs.\ f\ i)$
  by (*induction xs*) *simp-all*
lemma *sum-list-index-shift*: $(\sum i \leftarrow [a..<b].\ f\ (i + c)) = (\sum i \leftarrow [a+c..<b+c].\ f\ i)$
proof −
  have $(\sum i \leftarrow [a..<b].\ f\ (i + c)) = (\sum i \leftarrow (map\ (\lambda j.\ j + c)\ [a..<b]).\ f\ i)$
    by (*intro sum-list-index-trafo*)
  also have $map\ (\lambda j.\ j + c)\ [a..<b] = [a+c..<b+c]$
    using *map-add-const-upt* **by** *simp*
  finally show *?thesis* **.**
qed

lemma *list-sum-index-shift*: $n = j - k \Longrightarrow (\sum i \leftarrow [k+1..<j+1].\ f\ i) = (\sum i \leftarrow [k..<j].\ f\ (i + 1))$
using *sum-list-index-trafo*[**where** $g = \lambda l.\ l + 1$ **and** $xs = [k..<j]$ **and** $f = f$, *symmetric*]
using *map-Suc-upt* **by** *simp*

lemma *list-sum-index-shift'*: $(\sum i \leftarrow [0..<m].\ a\ (i + c)) = (\sum i \leftarrow [c..<m+c].\ a\ i)$
by (*induction m arbitrary*: *a c*) *auto*

lemma *list-sum-index-concat*: $(\sum i \leftarrow [0..<m].\ a\ i) + (\sum i \leftarrow [m..<m+c].\ a\ i) = (\sum i \leftarrow [0..<m+c].\ a\ i)$
proof −
have $(\sum i \leftarrow [0..<m+c].\ a\ i) = (\sum i \leftarrow [0..<m]\ @\ [m..<m+c].\ a\ i)$
using *upt-add-eq-append*[*of 0 m c*] **by** *simp*
then show *?thesis* **using** *sum-list-append* **by** *simp*
qed

lemma *sum-list-linear*:
assumes $\bigwedge a\ b.\ f\ (a + b) = f\ a + f\ b$
assumes $f\ 0 = 0$
shows $f\ (\sum i \leftarrow xs.\ g\ i) = (\sum i \leftarrow xs.\ f\ (g\ i))$
using *assms*
by (*induction xs*) *simp-all*
lemma *sum-list-int*:
shows $int\ (\sum i \leftarrow xs.\ g\ i) = (\sum i \leftarrow xs.\ int\ (g\ i))$
by (*intro sum-list-linear int-ops*(*5*) *int-ops*(*1*))

lemma *sum-list-split-Suc*:
assumes $n = Suc\ n'$
shows $(\sum i \leftarrow [0..<n].\ f\ i) = (\sum i \leftarrow [0..<n'].\ f\ i) + f\ n'$
using *assms* **by** *simp*

lemma *sum-list-estimation-leq*:
assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i \leq B$
shows $(\sum i \leftarrow xs.\ f\ i) \leq length\ xs * B$
using *assms* **by** (*induction xs*)(*simp*, *fastforce*)

lemma *sum-list-estimation-le*:
assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i < B$
assumes $xs \neq []$
shows $(\sum i \leftarrow xs.\ f\ i) < length\ xs * B$
proof −
from ‹$xs \neq []$› **have** $length\ xs > 0$ **by** *simp*
from ‹$xs \neq []$› **obtain** x **where** $x \in set\ xs$ **by** *fastforce*
then have $B > 0$ **using** *assms*(*1*) **by** *fastforce*
then obtain B' **where** $B = Suc\ B'$ **using** *not0-implies-Suc* **by** *blast*
with *assms*(*1*) **have** $\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i \leq B'$ **by** *fastforce*

with *sum-list-estimation-leq* **have** $(\sum i \leftarrow xs.\ f\ i) \leq$ *length xs* $*\ B'$ **by** *blast*
also have ... $<$ *length xs* $*\ B$ **using** ‹$B = Suc\ B'$› ‹*length xs* > 0› **by** *simp*
finally show *?thesis* **.**
qed

2.1 *semiring-1* Sums

lemma (**in** *semiring-1*) *of-bool-mult*: *of-bool* $x * a =$ (*if x then a else 0*)
by *simp*

lemma (**in** *semiring-1-cancel*) *of-bool-disj*: *of-bool* $(x \vee y) =$ *of-bool* $x\ +$ *of-bool* y $-$ *of-bool* $x\ *$ *of-bool* y
by *simp*
lemma (**in** *semiring-1*) *of-bool-disj-excl*: $\neg (x \wedge y) \Longrightarrow$ *of-bool* $(x \vee y) =$ *of-bool* $x\ +$ *of-bool* y
by *simp*

lemma (**in** *semiring-1*) *of-bool-var-swap*:
$(\sum i \leftarrow xs.$ *of-bool* $(i = j) * f\ i) = (\sum i \leftarrow xs.$ *of-bool* $(i = j) * f\ j)$
by (*induction xs*) *simp-all*
lemma $(\sum i \leftarrow xs.$ *of-bool* $(i = j) * f\ i) =$ *count-list xs j* $*\ f\ j$
by (*induction xs*) *simp-all*
lemma (**in** *semiring-1*) *of-bool-distinct*:
distinct xs $\Longrightarrow (\sum i \leftarrow xs.$ *of-bool* $(i = j) * f\ i\ j) =$ *of-bool* $(j \in$ *set xs*$) * f\ j\ j$
by (*induction xs*) *auto*
lemma (**in** *semiring-1*) *of-bool-distinct-in*:
distinct xs $\Longrightarrow j \in$ *set xs* $\Longrightarrow (\sum i \leftarrow xs.$ *of-bool* $(i = j) * f\ i\ j) = f\ j\ j$
using *of-bool-distinct*[*of xs j f*] *of-bool-mult* **by** *simp*

lemma (**in** *linordered-semiring-1*) *of-bool-sum-leq-1*:
assumes *distinct xs*
assumes $\bigwedge i\ j.\ i \in$ *set xs* $\Longrightarrow j \in$ *set xs* $\Longrightarrow P\ i \Longrightarrow P\ j \Longrightarrow i = j$
shows $(\sum l \leftarrow xs.$ *of-bool* $(P\ l)) \leq 1$
using *assms*
proof (*induction xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
consider $P\ a \mid \neg\ P\ a$ **by** *blast*
then show *?case*
proof *cases*
case *1*
then have r: $(\sum l \leftarrow a\ \#\ xs.$ *of-bool* $(P\ l)) = 1 + (\sum l \leftarrow xs.$ *of-bool* $(P\ l))$
by *simp*
have *of-bool* $(P\ l) = 0$ **if** $l \in$ *set xs* **for** l
proof $-$
from *that* **have** $a \neq l$ **using** *Cons* **by** *auto*
then have $\neg\ P\ l$ **using** *Cons* ‹$l \in$ *set xs*› *1* **by** *force*

then show *of-bool* (*P l*) = *0* **by** *simp*
qed
then have $(\sum l{\leftarrow}xs.\ \textit{of-bool}\ (P\ l)) = (\sum l{\leftarrow}xs.\ 0)$
using *list.map-cong0*[*of xs*] **by** *metis*
then show *?thesis* **using** *r* **by** *simp*
next
case *2*
then have $(\sum l{\leftarrow}a\ \#\ xs.\ \textit{of-bool}\ (P\ l)) = (\sum l{\leftarrow}xs.\ \textit{of-bool}\ (P\ l))$
by *simp*
then show *?thesis* **using** *Cons* **by** *simp*
qed
qed
instantiation *nat* :: *linordered-semiring-1*
begin
instance **..**
end

lemma (**in** *semiring-1*) *sum-list-mult-sum-list*: $(\sum i \leftarrow xs.\ f\ i) * (\sum j \leftarrow ys.\ g\ j)$
$= (\sum i \leftarrow xs.\ \sum j \leftarrow ys.\ f\ i * g\ j)$
by (*simp add*: *sum-list-const-mult sum-list-mult-const*)

lemma (**in** *semiring-1*) *semiring-1-sum-list-eq*:
$(\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i = g\ i) \Longrightarrow (\sum i \leftarrow xs.\ f\ i) = (\sum i \leftarrow xs.\ g\ i)$
using *arg-cong*[*OF list.map-cong0*] **by** *blast*

lemma (**in** *semiring-1*) *sum-swap*:
$(\sum i \leftarrow xs.\ (\sum j \leftarrow ys.\ f\ i\ j)) = (\sum j \leftarrow ys.\ (\sum i \leftarrow xs.\ f\ i\ j))$
proof (*induction xs*)
case (*Cons a xs*)
have $(\sum i \leftarrow (a\ \#\ xs).\ (\sum j \leftarrow ys.\ f\ i\ j)) = (\sum j \leftarrow ys.\ f\ a\ j) + (\sum i \leftarrow xs.\ (\sum j \leftarrow ys.\ f\ i\ j))$
by *simp*
also have ... $= (\sum j \leftarrow ys.\ f\ a\ j) + (\sum j \leftarrow ys.\ (\sum i \leftarrow xs.\ f\ i\ j))$
using *Cons* **by** *simp*
also have ... $= (\sum j \leftarrow ys.\ f\ a\ j + (\sum i \leftarrow xs.\ f\ i\ j))$
using *sum-list-addf*[*of* $\lambda j.\ f\ a\ j$ - *ys*] **by** *simp*
also have ... $= (\sum j \leftarrow ys.\ (\sum i \leftarrow (a\ \#\ xs).\ f\ i\ j))$ **by** *simp*
finally show *?case* **.**
qed *simp*

lemma (**in** *semiring-1*) *sum-append*:
$(\sum i \leftarrow (xs\ @\ ys).\ f\ i) = (\sum i \leftarrow xs.\ f\ i) + (\sum i \leftarrow ys.\ f\ i)$
by (*induction xs*) (*simp-all add*: *add.assoc*)

lemma (**in** *semiring-1*) *sum-append'*:
assumes $zs = xs\ @\ ys$
shows $(\sum i \leftarrow zs.\ f\ i) = (\sum i \leftarrow xs.\ f\ i) + (\sum i \leftarrow ys.\ f\ i)$
using *assms sum-append* **by** *blast*

2.1.1 Power Sums

lemma (**in** *semiring-1*) *sum-list-of-bool-filter*: ($\sum i \leftarrow xs.$ *of-bool* $(P\ i) * f\ i) =$ ($\sum i \leftarrow$ *filter* $P\ xs.\ f\ i)$
by (*induction xs*; *simp*)

lemma *upt-filter-less*: *filter* $(\lambda i.\ i < c)\ [a..<b] = [a..<min\ b\ c]$
by (*induction b*; *simp*)

lemma *upt-filter-geq*: *filter* $(\lambda i.\ i \geq c)\ [a..<b] = [max\ a\ c..<b]$
by (*induction b*; *simp*)

lemma (**in** *semiring-1*) *sum-list-of-bool-less*: ($\sum i \leftarrow [a..<b].$ *of-bool* $(i < c) * f\ i)$ $= (\sum i \leftarrow [a..<min\ b\ c].\ f\ i)$
unfolding *sum-list-of-bool-filter upt-filter-less* **by** (*rule refl*)

lemma (**in** *semiring-1*) *sum-list-of-bool-geq*: ($\sum i \leftarrow [a..<b].$ *of-bool* $(i \geq c) * f\ i)$ $= (\sum i \leftarrow [max\ a\ c..<b].\ f\ i)$
unfolding *sum-list-of-bool-filter upt-filter-geq* **by** (*rule refl*)

lemma (**in** *semiring-1*) *sum-list-of-bool-range*: ($\sum i \leftarrow [a..<b].$ *of-bool* $(i \in set$ $[c..<d]) * f\ i) =$
$(\sum i \leftarrow [max\ a\ c..<min\ b\ d].\ f\ i)$
proof −
have ($\sum i \leftarrow [a..<b].$ *of-bool* $(i \in set\ [c..<d]) * f\ i) =$
($\sum i \leftarrow [a..<b].$ *of-bool* $(i \geq c) *$ (*of-bool* $(i < d) * f\ i))$
by (*intro semiring-1-sum-list-eq*; *simp*)
then show *?thesis* **unfolding** *sum-list-of-bool-geq sum-list-of-bool-less* **.**
qed

lemma (**in** *comm-semiring-1*) *cauchy-product*:
$(\sum i \leftarrow [0..<n].\ f\ i) * (\sum j \leftarrow [0..<m].\ g\ j) =$
$(\sum k \leftarrow [0..<n + m - 1].\ \sum l \leftarrow [k + 1 - m..<min\ (k + 1)\ n].\ f\ l * g\ (k - l))$
proof −
have $(\sum i \leftarrow [0..<n].\ f\ i) * (\sum j \leftarrow [0..<m].\ g\ j) =$
$(\sum i \leftarrow [0..<n].\ \sum j \leftarrow [0..<m].\ f\ i * g\ j)$
unfolding *sum-list-mult-const*[*symmetric*]
unfolding *sum-list-const-mult*[*symmetric*]
by (*rule refl*)
also have ... $= (\sum i \leftarrow [0..<n].\ \sum j \leftarrow [0..<m].\ \sum k \leftarrow [0..<n + m - 1].$ *of-bool* $(k = i + j) * (f\ i * g\ j))$
by (*intro semiring-1-sum-list-eq of-bool-distinct-in*[*symmetric*]; *simp*)
also have ... $= (\sum k \leftarrow [0..<n + m - 1].\ \sum i \leftarrow [0..<n].\ \sum j \leftarrow [0..<m].$ *of-bool* $(k = i + j) * (f\ i * g\ j))$
unfolding *sum-swap*[**where** $xs = [0..<m]$ **and** $ys = [0..<n + m - 1]$]
unfolding *sum-swap*[**where** $xs = [0..<n]$ **and** $ys = [0..<n + m - 1]$]
by (*rule refl*)
also have ... $= (\sum k \leftarrow [0..<n + m - 1].\ \sum i \leftarrow [0..<n].\ \sum j \leftarrow [0..<m].$ *of-bool* $(k \geq i \wedge j = k - i) * (f\ i * g\ j))$

by (*intro semiring-1-sum-list-eq*; *simp*)
also have ... = (∑ k ← [0..<n + m − 1]. ∑ i ← [0..<n]. ∑ j ← [0..<m]. of-bool (j = k − i) ∗ (of-bool (k ≥ i) ∗ (f i ∗ g j)))
by (*intro semiring-1-sum-list-eq*; *simp*)
also have ... = (∑ k ← [0..<n + m − 1]. ∑ i ← [0..<n]. of-bool (k − i ∈ set [0..<m]) ∗ ((of-bool (k ≥ i) ∗ (f i ∗ g (k − i)))))
by (*intro semiring-1-sum-list-eq of-bool-distinct distinct-upt*)
also have ... = (∑ k ← [0..<n + m − 1]. ∑ i ← [0..<n]. of-bool (i ≥ k + 1 − m) ∗ ((of-bool (k + 1 > i) ∗ (f i ∗ g (k − i)))))
by (*intro semiring-1-sum-list-eq*; *auto*)
also have ... = (∑ k ← [0..<n + m − 1]. ∑ l ← [k + 1 − m..<min (k + 1) n]. f l ∗ g (k − l))
apply (*intro semiring-1-sum-list-eq*)
unfolding *sum-list-of-bool-geq sum-list-of-bool-less max-0L min.commute*[*of n*]
by (*rule refl*)
finally show *?thesis* **.**
qed

lemma (**in** *comm-semiring-1*) *power-sum-product*:
assumes m > 0
assumes n ≥ m
shows
(∑ i←[0..<n]. f i ∗ x ^ i) ∗ (∑ j←[0..<m]. g j ∗ x ^ j) =
(∑ k←[0..<m]. (∑ i←[0..<Suc k]. f i ∗ g (k − i)) ∗ x ^ k) +
(∑ k←[m..<n]. (∑ i←[Suc k − m..<Suc k]. f i ∗ g (k − i)) ∗ x ^ k) +
(∑ k←[n..<n + m − 1]. (∑ i←[Suc k − m..<n]. f i ∗ g (k − i)) ∗ x ^ k)
proof −
have 1: [0..<n + m − 1] = [0..<m] @ [m..<n] @ [n..<n + m − 1]
using *upt-add-eq-append'*[*of 0 m n + m − 1*] *upt-add-eq-append'*[*of m n n + m − 1*] *assms* **by** *simp*

have (∑ i ← [0..<n]. f i ∗ x ^ i) ∗ (∑ j ← [0..<m]. g j ∗ x ^ j) =
(∑ k ← [0..<n + m − 1]. ∑ l ← [k + 1 − m..<min (k + 1) n]. (f l ∗ x ^ l) ∗ (g (k − l) ∗ x ^ (k − l)))
by (*rule cauchy-product*)
also have ... = (∑ k ← [0..<n + m − 1]. ∑ l ← [k + 1 − m..<min (k + 1) n]. f l ∗ g (k − l) ∗ x ^ k)
apply (*intro semiring-1-sum-list-eq*)
using *mult.commute mult.assoc power-add*[*symmetric*]
by *simp*
also have ... = (∑ k ← [0..<n + m − 1]. (∑ l ← [k + 1 − m..<min (k + 1) n]. f l ∗ g (k − l)) ∗ x ^ k)
by (*intro semiring-1-sum-list-eq sum-list-mult-const*)
also have ... = (∑ k←[0..<m]. (∑ i←[k + 1 − m..<min (k + 1) n]. f i ∗ g (k − i)) ∗ x ^ k) +
(∑ k←[m..<n]. (∑ i←[k + 1 − m..<min (k + 1) n]. f i ∗ g (k − i)) ∗ x ^ k) +
(∑ k←[n..<n + m − 1]. (∑ i←[k + 1 − m..<min (k + 1) n]. f i ∗ g (k − i)) ∗ x ^ k)

```
    unfolding 1 sum-append add.assoc by (rule refl)
  also have ... = (∑ k←[0..<m]. (∑ i←[0..<Suc k]. f i ∗ g (k − i)) ∗ x ^ k) +
      (∑ k←[m..<n]. (∑ i←[Suc k − m..<Suc k]. f i ∗ g (k − i)) ∗ x ^ k) +
      (∑ k←[n..<n + m − 1]. (∑ i←[Suc k − m..<n]. f i ∗ g (k − i)) ∗ x ^ k)
   using assms by (intro-cong [cong-tag-2 (+)] more: semiring-1-sum-list-eq; simp)
  finally show ?thesis .
qed

lemma (in comm-semiring-1) power-sum-product-same-length:
  assumes n > 0
  shows (∑ i←[0..<n]. f i ∗ x ^ i) ∗ (∑ j←[0..<n]. g j ∗ x ^ j) =
  (∑ k←[0..<n]. (∑ i←[0..<Suc k]. f i ∗ g (k − i)) ∗ x ^ k) +
  (∑ k←[n..<2 ∗ n − 1]. (∑ i←[Suc k − n..<n]. f i ∗ g (k − i)) ∗ x ^ k)
  using power-sum-product[of n n f x g, OF assms order.refl]
  by (simp add: semiring-numeral-class.mult-2)

lemma (in semiring-1) sum-index-transformation:
  shows (∑ i ← xs. f (g i)) = (∑ j ← map g xs. f j)
  by (induction xs) simp-all

lemma (in comm-semiring-1) power-sum-split:
  fixes f :: nat ⇒ 'a
  fixes x :: 'a
  fixes c :: nat
  assumes j ≤ n
  shows (∑ i ← [0..<n]. f i ∗ x ^ (i ∗ c)) =
      (∑ i ← [0..<j]. f i ∗ x ^ (i ∗ c)) +
      x ^ (j ∗ c) ∗ (∑ i ← [0..<n − j]. f (j + i) ∗ x ^ (i ∗ c))
proof −
  have (λi. i + j) = (+) j by fastforce
  have (∑ i ← [0..<n]. f i ∗ x ^ (i ∗ c)) =
    (∑ i ← [0..<j]. f i ∗ x ^ (i ∗ c)) + (∑ i ← [j..<n]. f i ∗ x ^ (i ∗ c))
    apply (intro sum-append' upt-add-eq-append') using ‹j ≤ n› by auto
  also have (∑ i ← [j..<n]. f i ∗ x ^ (i ∗ c)) =
    (∑ i ← map ((+) j) [0..<n − j]. f i ∗ x ^ (i ∗ c))
    apply (intro-cong [cong-tag-1 sum-list, cong-tag-2 map] more: refl)
    using ‹j ≤ n› map-add-upt[of j n − j] ‹(λi. i + j) = (+) j› by simp
  also have ... = (∑ i ← [0..<n − j]. f (j + i) ∗ x ^ ((j + i) ∗ c))
    by (intro sum-index-transformation[symmetric])
  also have ... = (∑ i ← [0..<n − j]. x ^ (j ∗ c) ∗ (f (j + i) ∗ x ^ (i ∗ c)))
    apply (intro semiring-1-sum-list-eq)
    using mult.commute mult.assoc by (simp add: power-add add-mult-distrib)
  also have ... = x ^ (j ∗ c) ∗ (∑ i ← [0..<n − j]. (f (j + i) ∗ x ^ (i ∗ c)))
    by (intro sum-list-const-mult)
  finally show ?thesis .
qed
```

2.2 *nat* Sums

```
lemma geo-sum-nat:
  assumes (q :: nat) > 1
  shows (q − 1) ∗ (∑ i ← [0..<n]. q ^ i) = q ^ n − 1
proof (induction n)
  case (Suc n)
  have (q − 1) ∗ (∑ i ← [0..<Suc n]. q ^ i) = (q − 1) ∗ (q ^ n + (∑ i ← [0..<n].
q ^ i))
    by simp
  also have ... = (q − 1) ∗ q ^ n + (q − 1) ∗ (∑ i ← [0..<n]. q ^ i)
    using add-mult-distrib mult.commute by metis
  also have ... = (q − 1) ∗ q ^ n + (q ^ n − 1)
    using Suc.IH by simp
  also have ... = q ∗ q ^ n − 1 using ‹q > 1› by (simp add: diff-mult-distrib)
  finally show ?case by simp
qed simp

lemma geo-sum-bound:
  assumes (q :: nat) > 1
  assumes ⋀i. i < n ⟹ f i < q
  shows (∑ i ← [0..<n]. f i ∗ q ^ i) < q ^ n
proof −
  from assms have ⋀i. i < n ⟹ f i ≤ (q − 1) by fastforce
  then have (∑ i ← [0..<n]. f i ∗ q ^ i) ≤ (∑ i ← [0..<n]. (q − 1) ∗ q ^ i)
    apply (intro sum-list-mono mult-le-mono1)
    using assms by simp
  also have ... = (q − 1) ∗ (∑ i ← [0..<n]. q ^ i)
    by (intro sum-list-const-mult)
  also have ... = q ^ n − 1
    by (intro geo-sum-nat assms)
  also have ... < q ^ n using ‹q > 1› by simp
  finally show ?thesis .
qed

lemma power-sum-nat-split-div-mod:
  assumes x > 1
  assumes c > 0
  assumes ⋀i. i < n ⟹ (f i :: nat) < x ^ c
  assumes j ≤ n
  shows (∑ i ← [0..<n]. f i ∗ x ^ (i ∗ c)) div x ^ (j ∗ c)
      = (∑ i ← [0..<n − j]. f (j + i) ∗ x ^ (i ∗ c))
        (∑ i ← [0..<n]. f i ∗ x ^ (i ∗ c)) mod x ^ (j ∗ c)
      = (∑ i ← [0..<j]. f i ∗ x ^ (i ∗ c))
proof −
  define sum where sum = (∑ i ← [0..<n]. f i ∗ x ^ (i ∗ c))
  then have sum = (∑ i ← [0..<j]. f i ∗ x ^ (i ∗ c)) +
      x ^ (j ∗ c) ∗ (∑ i ← [0..<n − j]. f (j + i) ∗ x ^ (i ∗ c))
    (is sum = ?sum1 + x ^ (j ∗ c) ∗ ?sum2)
    using power-sum-split ‹j ≤ n› by blast
```

```
  have ?sum1 = (∑ i ← [0..<j]. f i * (x ^ c) ^ i)
    apply (intro-cong [cong-tag-2 (*)] more: semiring-1-sum-list-eq refl)
    using power-mult mult.commute by metis
  also have ... < (x ^ c) ^ j
    apply (intro geo-sum-bound)
    subgoal using assms one-less-power by blast
    subgoal using assms by simp
    done
  finally have ?sum1 < x ^(j * c) by (simp add: power-mult mult.commute)
  then show sum mod x ^(j * c) = ?sum1 sum div (x ^(j * c)) = ?sum2 using
‹sum = ?sum1 + x ^(j * c) * ?sum2›
    using assms(1) by fastforce+
qed

lemma power-sum-nat-extract-coefficient:
  assumes x > 1
  assumes c > 0
  assumes ⋀i. i < n ⟹ (f i :: nat) < x ^ c
  assumes j < n
  shows ((∑ i ← [0..<n]. f i * x ^(i * c)) div x ^(j * c)) mod x ^ c = f j
proof −
  have (∑ i ← [0..<n]. f i * x ^(i * c)) div x ^(j * c) =
    (∑ i ← [0..<n − j]. f (j + i) * x ^(i * c)) (is ?sum = -)
    apply (intro power-sum-nat-split-div-mod(1) assms)
    using assms by simp-all
  moreover have ... mod x ^(1 * c) = (∑ i ← [0..<1]. f (j + i) * x ^(i * c))
    apply (intro power-sum-nat-split-div-mod(2) assms)
    using assms by simp-all
  ultimately show ?sum mod x ^ c = f j by simp
qed

lemma power-sum-nat-eq:
  assumes x > 1
  assumes c > 0
  assumes ⋀i. i < n ⟹ (f i :: nat) < x ^ c
  assumes ⋀i. i < n ⟹ g i < x ^ c
  assumes (∑ i ← [0..<n]. f i * x ^(i * c)) = (∑ i ← [0..<n]. g i * x ^(i * c))
    (is ?sumf = ?sumg)
  shows ⋀i. i < n ⟹ f i = g i
proof −
  fix i
  assume i < n
  then have f i = (?sumf div x ^(i * c)) mod x ^ c
    apply (intro power-sum-nat-extract-coefficient[symmetric] assms) by assump-
tion
  also have ... = (?sumg div x ^(i * c)) mod x ^ c
    using assms by argo
  also have ... = g i
   apply (intro power-sum-nat-extract-coefficient assms) using ‹i < n› by simp-all
```

finally show *f i* = *g i* **.**
qed

end

3 Sums in Monoids

theory *Monoid-Sums*
imports *HOL−Algebra.Ring Expander-Graphs.Extra-Congruence-Method Karatsuba-Preliminaries HOL−Library.Multiset HOL−Number-Theory.Residues Karatsuba-Sum-Lemmas*
begin

This section contains a version of *sum-list* for entries in some abelian monoid. Contrary to *sum-list*, which is defined for the type class *comm-monoid-add*, this version is for the locale *abelian-monoid*. After the definition, some simple lemmas about sums are proven for this sum function.

context *abelian-monoid*
begin

fun *monoid-sum-list* :: [′*c* ⇒ ′*a*, ′*c list*] ⇒ ′*a* **where**
monoid-sum-list f [] = **0**
| *monoid-sum-list f* (*x* # *xs*) = *f x* ⊕ *monoid-sum-list f xs*

lemma *monoid-sum-list f xs* = *foldr* (⊕) (*map f xs*) **0**
by (*induction xs*) *simp-all*

end

The syntactic sugar used for *finsum* is adapted accordingly.

syntax
-monoid-sum-list :: *index* ⇒ *idt* ⇒ ′*c list* ⇒ ′*c* ⇒ ′*a*
((*3*⨁ --←-. -) [*1000*, *0*, *51*, *10*] *10*)
translations
$\bigoplus_G$*i*←*xs*. *b* ⇌ *CONST abelian-monoid.monoid-sum-list G* (λ*i*. *b*) *xs*

context *abelian-monoid*
begin

lemma *monoid-sum-list-finsum*:
assumes ⋀*i*. *i* ∈ *set xs* ⟹ *f i* ∈ *carrier G*
assumes *distinct xs*
shows (⨁ *i* ← *xs*. *f i*) = (⨁ *i* ∈ *set xs*. *f i*)
using *assms*
proof (*induction xs*)
case *Nil*
then show *?case* **by** *simp*
next

```
  case (Cons a xs)
  then show ?case using finsum-insert[of set xs a f] by simp
qed

lemma monoid-sum-list-cong:
  assumes ⋀i. i ∈ set xs ⟹ f i = g i
  shows (⨁ i ← xs. f i) = (⨁ i ← xs. g i)
  using assms by (induction xs) simp-all

lemma monoid-sum-list-closed[simp]:
  assumes ⋀i. i ∈ set xs ⟹ f i ∈ carrier G
  shows (⨁ i ← xs. f i) ∈ carrier G
  using assms by (induction xs) simp-all

lemma monoid-sum-list-add-in:
  assumes ⋀i. i ∈ set xs ⟹ f i ∈ carrier G
  assumes ⋀i. i ∈ set xs ⟹ g i ∈ carrier G
  shows (⨁ i ← xs. f i) ⊕ (⨁ i ← xs. g i) =
            (⨁ i ← xs. f i ⊕ g i)
  using assms
proof (induction xs)
  case (Cons a xs)
  have (⨁ i ← (a # xs). f i) ⊕ (⨁ i ← (a # xs). g i)
     = (f a ⊕ (⨁ i ← xs. f i)) ⊕ (g a ⊕ (⨁ i ← xs. g i))
    by simp
  also have ... = (f a ⊕ g a) ⊕ ((⨁ i ← xs. f i) ⊕ (⨁ i ← xs. g i))
    using a-comm a-assoc Cons.prems by simp
  also have ... = (f a ⊕ g a) ⊕ (⨁ i ← xs. f i ⊕ g i)
    using Cons by simp
  finally show ?case by simp
qed simp

lemma monoid-sum-list-0[simp]: (⨁ i ← xs. 0) = 0
  by (induction xs) simp-all

lemma monoid-sum-list-swap:
  assumes[simp]: ⋀i j. i ∈ set xs ⟹ j ∈ set ys ⟹ f i j ∈ carrier G
  shows (⨁ i ← xs. (⨁ j ← ys. f i j)) =
          (⨁ j ← ys. (⨁ i ← xs. f i j))
  using assms
proof (induction xs arbitrary: ys)
  case (Cons a xs)
  have (⨁ i ← (a # xs). (⨁ j ← ys. f i j))
     = (⨁ j ← ys. f a j) ⊕ (⨁ i ← xs. (⨁ j ← ys. f i j))
    by simp
  also have ... = (⨁ j ← ys. f a j) ⊕ (⨁ j ← ys. (⨁ i ← xs. f i j))
    using Cons by simp
  also have ... = (⨁ j ← ys. f a j ⊕ (⨁ i ← xs. f i j))
    using monoid-sum-list-add-in[of ys λj. f a j λj. (⨁ i ← xs. f i j)] Cons.prems
```

```
by simp
  finally show ?case by simp
qed simp

lemma monoid-sum-list-index-transformation:
  (⨁ i ← (map g xs). f i) = (⨁ i ← xs. f (g i))
  by (induction xs) simp-all

lemma monoid-sum-list-index-shift-0:
  (⨁ i ← [c..<c+n]. f i) = (⨁ i ← [0..<n]. f (c + i))
  using monoid-sum-list-index-transformation[of f λi. c + i [0..<n]]
  by (simp add: add.commute map-add-upt)

lemma monoid-sum-list-index-shift:
  (⨁ l ← [a..<b]. f (l+c)) = (⨁ l ← [(a+c)..<(b+c)]. f l)
  using monoid-sum-list-index-transformation[of f λi. i + c [a..<b]]
  by (simp add: map-add-const-upt)

lemma monoid-sum-list-app:
  assumes ⋀i. i ∈ set xs ⟹ f i ∈ carrier G
  assumes ⋀i. i ∈ set ys ⟹ f i ∈ carrier G
  shows (⨁ i ← xs @ ys. f i) = (⨁ i ← xs. f i) ⊕ (⨁ i ← ys. f i)
  using assms
  by (induction xs) (simp-all add: a-assoc)

lemma monoid-sum-list-app':
  assumes ⋀i. i ∈ set xs ⟹ f i ∈ carrier G
  assumes ⋀i. i ∈ set ys ⟹ f i ∈ carrier G
  assumes xs @ ys = zs
  shows (⨁ i ← zs. f i) = (⨁ i ← xs. f i) ⊕ (⨁ i ← ys. f i)
  using monoid-sum-list-app assms by blast

lemma monoid-sum-list-extract:
  assumes ⋀i. i ∈ set xs ⟹ f i ∈ carrier G
  assumes ⋀i. i ∈ set ys ⟹ f i ∈ carrier G
  assumes f x ∈ carrier G
  shows (⨁ i ← xs @ x # ys. f i) = f x ⊕ (⨁ i ← (xs @ ys). f i)
proof −
  have (⨁ i ← xs @ x # ys. f i) = (⨁ i ← xs. f i) ⊕ f x ⊕ (⨁ i ← ys. f i)
    using assms monoid-sum-list-app[of xs f x # ys]
    using a-assoc by auto
  also have ... = f x ⊕ ((⨁ i ← xs. f i) ⊕ (⨁ i ← ys. f i))
    using assms a-assoc a-comm by auto
  finally show ?thesis using monoid-sum-list-app[of xs f ys] assms by algebra
qed

lemma monoid-sum-list-Suc:
  assumes ⋀i. i < Suc r ⟹ f i ∈ carrier G
  shows (⨁ i ← [0..<Suc r]. f i) = (⨁ i ← [0..<r]. f i) ⊕ f r
```

  using *assms monoid-sum-list-app*[*of* [*0*..<*r*] *f* [*r*]]
  by *simp*

lemma *bij-betw-diff-singleton*: $a \in A \Longrightarrow b \in B \Longrightarrow$ *bij-betw f A B* $\Longrightarrow f\ a = b$ $\Longrightarrow$ *bij-betw f* $(A - \{a\})\ (B - \{b\})$
 by (*metis* (*no-types*, *lifting*) *DiffE Diff-Diff-Int Diff-cancel Diff-empty Int-insert-right-if1 Un-Diff-Int notIn-Un-bij-betw3 singleton-iff*)

lemma $a \in A \Longrightarrow$ *bij-betw f A B* $\Longrightarrow$ *bij-betw f* $(A - \{a\})\ (B - \{f\ a\})$
  using *bij-betw-diff-singleton*[*of a A f a B f*]
  by (*simp add*: *bij-betwE*)

lemma *monoid-sum-list-multiset-eq*:
  assumes *mset xs* = *mset ys*
  assumes $\bigwedge i.\ i \in$ *set xs* $\Longrightarrow f\ i \in$ *carrier G*
  shows $(\bigoplus i \leftarrow xs.\ f\ i) = (\bigoplus i \leftarrow ys.\ f\ i)$
  using *assms*
proof (*induction xs arbitrary*: *ys*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then have $a \in$ *set ys* **using** *mset-eq-setD* **by** *fastforce*
  then obtain *ys1 ys2* **where** *ys* = *ys1* @ *a* # *ys2* **by** (*meson split-list*)
  with *Cons.prems* **have** *1*: *mset xs* = *mset* (*ys1* @ *ys2*) **by** *simp*
  from *Cons.prems mset-eq-setD* **have** $\bigwedge i.\ i \in$ *set ys* $\Longrightarrow f\ i \in$ *carrier G* **by** *blast*
  then have[*simp*]: $\bigwedge i.\ i \in$ *set ys1* $\Longrightarrow f\ i \in$ *carrier G* $f\ a \in$ *carrier G* $\bigwedge i.\ i \in$ *set ys2* $\Longrightarrow f\ i \in$ *carrier G*
    using ‹*ys* = *ys1* @ *a* # *ys2*› **by** *simp-all*
  from *1* **have** $(\bigoplus i \leftarrow xs.\ f\ i) = (\bigoplus i \leftarrow (ys1\ @\ ys2).\ f\ i)$
    using *Cons* **by** *simp*
  also have ... $= (\bigoplus i \leftarrow ys1.\ f\ i) \oplus (\bigoplus i \leftarrow ys2.\ f\ i)$
    by (*intro monoid-sum-list-app*) *simp-all*
  also have $f\ a \oplus ... = (\bigoplus i \leftarrow ys1.\ f\ i) \oplus (f\ a \oplus (\bigoplus i \leftarrow ys2.\ f\ i))$
    using *a-comm a-assoc monoid-sum-list-closed* **by** *simp*
  also have ... $= (\bigoplus i \leftarrow ys1.\ f\ i) \oplus (\bigoplus i \leftarrow (a\ \#\ ys2).\ f\ i)$
    by *simp*
  also have ... $= (\bigoplus i \leftarrow ys.\ f\ i)$
    unfolding ‹*ys* = *ys1* @ *a* # *ys2*›
    by (*intro monoid-sum-list-app*[*symmetric*]) *auto*
  finally show *?case* **by** *simp*
qed
lemma *monoid-sum-list-index-permutation*:
  assumes *distinct xs*
  assumes *distinct ys* ∨ *length xs* = *length ys*
  assumes *bij-betw f* (*set xs*) (*set ys*)
  assumes $\bigwedge i.\ i \in$ *set ys* $\Longrightarrow g\ i \in$ *carrier G*
  shows $(\bigoplus i \leftarrow ys.\ g\ i) = (\bigoplus i \leftarrow xs.\ g\ (f\ i))$
  using *assms*

```
proof (induction xs arbitrary: ys)
  case Nil
  then have ys = [] using bij-betw-same-card by fastforce
  then show ?case by simp
next
  case (Cons a xs)
  then have length ys = length (a # xs) distinct ys
   by (metis bij-betw-same-card distinct-card, metis bij-betw-same-card distinct-card
card-distinct)

  have 0: ⋀i. i ∈ set (a # xs) ⟹ g (f i) ∈ carrier G
  proof −
    fix i
    assume i ∈ set (a # xs)
    then have f i ∈ set ys using Cons.prems(3) by (simp add: bij-betw-apply)
    then show g (f i) ∈ carrier G using Cons.prems(4) by blast
  qed

  define b where b = f a
  then have b ∈ set ys using Cons(4) bij-betw-apply by fastforce
  then obtain ys1 ys2 where ys = ys1 @ b # ys2 by (meson split-list)
  then have b ∉ set ys1 b ∉ set ys2 using ‹distinct ys› by simp-all
  have bij-betw f (set xs) (set (ys1 @ ys2))
    using ‹ys = ys1 @ b # ys2› Cons(4) b-def
    using bij-betw-diff-singleton[of a set (a # xs) f a set ys f]
    using Cons.prems(1) ‹distinct ys› by auto
  moreover have length (ys1 @ ys2) = length xs using ‹length ys = length (a #
xs)› ‹ys = ys1 @ b # ys2›
    by simp
   ultimately have 1: (⨁ i ← (ys1@ys2). g i) = (⨁ i ← xs. g (f i)) using
Cons.IH[of ys1@ys2] Cons.prems(4)
    using Cons.prems(1) 0 ‹ys = ys1 @ b # ys2› by auto

  have (⨁ i ← (a # xs). g (f i)) = g b ⊕ (⨁ i ← xs. g (f i))
    using ‹b = f a› by simp
  also have ... = g b ⊕ (⨁ i ← (ys1@ys2). g i) using 1 by simp
  also have ... = (⨁ i ← (ys1@b#ys2). g i)
    apply (intro monoid-sum-list-extract[symmetric])
    using Cons.prems(4) ‹ys = ys1 @ b # ys2› by simp-all
  finally show (⨁ i ← ys. g i) = (⨁ i ← (a # xs). g (f i))
    using ‹ys = ys1 @ b # ys2› by simp
qed

lemma monoid-sum-list-split:
  assumes[simp]: ⋀i. i < b + c ⟹ f i ∈ carrier G
  shows (⨁ l ← [0..<b]. f l) ⊕ (⨁ l ← [b..< b + c]. f l) = (⨁ l ← [0..< b + c].
f l)
  using monoid-sum-list-app[of [0..<b] f [b..< b + c], symmetric]
  using upt-add-eq-append[of 0 b c]
```

```
  by simp

lemma monoid-sum-list-splice:
  assumes[simp]: ⋀i. i < 2 ∗ n ⟹ f i ∈ carrier G
  shows (⨁ i ← [0..< 2 ∗ n]. f i) = (⨁ i ← [0..<n]. f (2∗i)) ⊕ (⨁ i ← [0..<n].
f (2∗i+1))
proof −
  let ?es = filter even [0..< 2 ∗ n]
  let ?os = filter odd [0..< 2 ∗ n]
  have 1: (⨁ i ← [0..< 2 ∗ n]. f i) = (⨁ i ∈ {0..< 2 ∗ n}. f i)
    using monoid-sum-list-finsum[of [0..< 2 ∗ n] f] by simp

  let ?E = {i ∈ {0..<2∗n}. even i}
  let ?O = {i ∈ {0..<2∗n}. odd i}
  have ?E ∩ ?O = {} by blast
  moreover have ?E ∪ ?O = {0..<2∗n} by blast
  ultimately have (⨁ i ∈ {0..<2∗n}. f i) = (⨁ i ∈ ?E. f i) ⊕ (⨁ i ∈ ?O. f i)
    using finsum-Un-disjoint[of ?E ?O f] assms by auto
  moreover have ?E = set ?es ?O = set ?os by simp-all
  ultimately have (⨁ i ∈ {0..<2∗n}. f i) = (⨁ i ∈ set ?es. f i) ⊕ (⨁ i ∈ set
?os. f i)
    by presburger
  also have (⨁ i ∈ set ?es. f i) = (⨁ i ← ?es. f i)
    using monoid-sum-list-finsum[of ?es f] by simp
  also have ... = (⨁ i ← [0..<n]. f (2∗i))
    using monoid-sum-list-index-transformation[of f λi. 2 ∗ i [0..<n]]
    using filter-even-upt-even
    by algebra
  also have (⨁ i ∈ set ?os. f i) = (⨁ i ← ?os. f i)
    using monoid-sum-list-finsum[of ?os f] by simp
  also have ... = (⨁ i ← [0..<n]. f (2∗i + 1))
    using monoid-sum-list-index-transformation[of f λi. 2 ∗ i + 1 [0..<n]]
    using filter-odd-upt-even
    by algebra
  finally show ?thesis using 1 by presburger
qed

lemma monoid-sum-list-even-odd-split:
  assumes even (n::nat)
  assumes ⋀i. i < n ⟹ f i ∈ carrier G
  shows (⨁ i ← [0..<n]. f i) = (⨁ i ← [0..< n div 2]. f (2∗i)) ⊕ (⨁ i ← [0..<
n div 2]. f (2∗i+1))
  using assms monoid-sum-list-splice by auto

end

context abelian-group
begin
```

lemma *monoid-sum-list-minus-in*:
  assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i \in carrier\ G$
  shows $\ominus\ (\bigoplus i \leftarrow xs.\ f\ i) = (\bigoplus i \leftarrow xs.\ \ominus f\ i)$
  using *assms* **by** (*induction xs*) (*simp-all add*: *minus-add*)

lemma *monoid-sum-list-diff-in*:
  assumes[*simp*]: $\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i \in carrier\ G$
  assumes[*simp*]: $\bigwedge i.\ i \in set\ xs \Longrightarrow g\ i \in carrier\ G$
  shows $(\bigoplus i \leftarrow xs.\ f\ i) \ominus (\bigoplus i \leftarrow xs.\ g\ i) =$
      $(\bigoplus i \leftarrow xs.\ f\ i \ominus g\ i)$
proof −
  have $(\bigoplus i \leftarrow xs.\ f\ i) \ominus (\bigoplus i \leftarrow xs.\ g\ i) = (\bigoplus i \leftarrow xs.\ f\ i) \oplus (\ominus\ (\bigoplus i \leftarrow xs.\ g$
$i))$
    unfolding *minus-eq* **by** *simp*
  also have ... $= (\bigoplus i \leftarrow xs.\ f\ i) \oplus (\bigoplus i \leftarrow xs.\ \ominus g\ i)$
    using *monoid-sum-list-minus-in*[*of xs g*] **by** *simp*
  also have ... $= (\bigoplus i \leftarrow xs.\ f\ i \oplus (\ominus g\ i))$
    using *monoid-sum-list-add-in*[*of xs f*] **by** *simp*
  finally show *?thesis* **unfolding** *minus-eq* **.**
qed

end

context *ring*
begin

lemma *monoid-sum-list-const*:
  assumes[*simp*]: $c \in carrier\ R$
  shows $(\bigoplus i \leftarrow xs.\ c) = (nat\text{-}embedding\ (length\ xs)) \otimes c$
  apply (*induction xs*)
  using *a-comm l-distr* **by** *auto*

lemma *monoid-sum-list-in-right*:
  assumes $y \in carrier\ R$
  assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i \in carrier\ R$
  shows $(\bigoplus i \leftarrow xs.\ f\ i \otimes y) = (\bigoplus i \leftarrow xs.\ f\ i) \otimes y$
  using *assms* **by** (*induction xs*) (*simp-all add*: *l-distr*)

lemma *monoid-sum-list-in-left*:
  assumes $y \in carrier\ R$
  assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i \in carrier\ R$
  shows $(\bigoplus i \leftarrow xs.\ y \otimes f\ i) = y \otimes (\bigoplus i \leftarrow xs.\ f\ i)$
  using *assms* **by** (*induction xs*) (*simp-all add*: *r-distr*)

lemma *monoid-sum-list-prod*:
  assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i \in carrier\ R$
  assumes $\bigwedge i.\ i \in set\ ys \Longrightarrow g\ i \in carrier\ R$
  shows $(\bigoplus i \leftarrow xs.\ f\ i) \otimes (\bigoplus j \leftarrow ys.\ g\ j) = (\bigoplus i \leftarrow xs.\ (\bigoplus j \leftarrow ys.\ f\ i \otimes g\ j))$
proof −

have (⨁ *i* ← *xs*. *f i*) ⊗ (⨁ *j* ← *ys*. *g j*) = (⨁ *i* ← *xs*. *f i* ⊗ (⨁ *j* ← *ys*. *g j*))
apply (*intro monoid-sum-list-in-right*[*symmetric*])
using *assms* **by** *simp-all*
also have ... = (⨁ *i* ← *xs*. (⨁ *j* ← *ys*. *f i* ⊗ *g j*))
apply (*intro monoid-sum-list-cong monoid-sum-list-in-left*[*symmetric*])
using *assms* **by** *simp-all*
finally show *?thesis* **.**
qed

3.1 Kronecker delta

definition *delta* **where**
delta i j = (*if i* = *j then* **1** *else* **0**)

lemma *delta-closed*[*simp*]: *delta i j* ∈ *carrier R*
unfolding *delta-def* **by** *simp*

lemma *delta-sym*: *delta i j* = *delta j i*
unfolding *delta-def* **by** *simp*

lemma *delta-refl*[*simp*]: *delta i i* = **1**
unfolding *delta-def* **by** *simp*

lemma *monoid-sum-list-delta*[*simp*]:
assumes[*simp*]: ⋀*i*. *i* < *n* ⟹ *f i* ∈ *carrier R*
assumes[*simp*]: *j* < *n*
shows (⨁ *i* ← [*0*..<*n*]. *delta i j* ⊗ *f i*) = *f j*
proof −
from *assms* **have** *0*: [*0*..<*n*] = [*0*..<*j*] @ *j* # [*Suc j*..<*n*]
by (*metis le-add1 le-add-same-cancel1 less-imp-add-positive upt-add-eq-append upt-conv-Cons*)
then have [*0*..<*n*] = [*0*..<*j*] @ [*j*] @ [*Suc j*..<*n*]
by *simp*
moreover have *1*: ⋀*i*. *i* ∈ *set* [*0*..<*j*] ⟹ *delta i j* ⊗ *f i* ∈ *carrier R*
using *0 assms delta-closed m-closed atLeastLessThan-iff*
by (*metis le-add1 less-imp-add-positive linorder-le-less-linear set-upt upt-conv-Nil*)
moreover have *2*: ⋀*i*. *i* ∈ *set* ([*j*] @ [*Suc j*..<*n*]) ⟹ *delta i j* ⊗ *f i* ∈ *carrier R*
using *0 assms delta-closed m-closed*
by *auto*
ultimately have (⨁ *i* ← [*0*..<*n*]. *delta i j* ⊗ *f i*) = (⨁ *i* ← [*0*..<*j*]. *delta i j* ⊗ *f i*) ⊕ (⨁ *i* ← [*j*] @ [*Suc j*..<*n*]. *delta i j* ⊗ *f i*)
using *monoid-sum-list-app*[*of* [*0*..<*j*] λ*i*. *delta i j* ⊗ *f i* [*j*] @ [*Suc j*..<*n*]]
by *presburger*
also have (⨁ *i* ← [*j*] @ [*Suc j*..<*n*]. *delta i j* ⊗ *f i*) = (⨁ *i* ← [*j*]. *delta i j* ⊗ *f i*) ⊕ (⨁ *i* ← [*Suc j*..<*n*]. *delta i j* ⊗ *f i*)
using *2 monoid-sum-list-app*[*of* [*j*] λ*i*. *delta i j* ⊗ *f i* [*Suc j*..<*n*]]
by *simp*
also have (⨁ *i* ← [*0*..<*j*]. *delta i j* ⊗ *f i*) = **0**
using *monoid-sum-list-0*[*of* [*0*..<*j*]] *monoid-sum-list-cong*[*of* [*0*..<*j*] λ*i*. **0** λ*i*.

delta i j ⊗ *f i*]
  unfolding *delta-def* **using** ‹*j* < *n*› **by** *simp*
 also have (⨁ *i* ← [*Suc j*..<*n*]. *delta i j* ⊗ *f i*) = **0**
  using *monoid-sum-list-0*[*of* [*Suc j*..<*n*]] *monoid-sum-list-cong*[*of* [*Suc j*..<*n*] λ*i*. **0** λ*i*. *delta i j* ⊗ *f i*]
  unfolding *delta-def* **by** *simp*
 also have (⨁ *i* ← [*j*]. *delta i j* ⊗ *f i*) = *f j* **by** *simp*
 finally show *?thesis* **by** *simp*
qed

lemma *mononid-sum-list-only-delta*[*simp*]:
 j < *n* ⟹ (⨁ *i* ← [*0*..<*n*]. *delta i j*) = **1**
 using *monoid-sum-list-delta*[*of n* λ*i*. **1** *j*] **by** *simp*

3.2 Power sums

lemma *geo-monoid-list-sum*:
 assumes[*simp*]: *x* ∈ *carrier R*
 shows (**1** ⊖ *x*) ⊗ (⨁ *l* ← [*0*..<*r*]. *x* [^] *l*) = (**1** ⊖ *x* [^] *r*)
 using *assms*
proof (*induction r*)
 case *0*
 then show *?case* **using** *assms* **by** (*simp*, *algebra*)
next
 case (*Suc r*)
 have (**1** ⊖ *x*) ⊗ (⨁ *l* ← [(*0*::*nat*)..< *Suc r*]. *x* [^] *l*) = (**1** ⊖ *x*) ⊗ (*x* [^] *r* ⊕ (⨁ *l* ← [*0*..<*r*]. *x* [^] *l*))
  using *monoid-sum-list-Suc*[*of r* λ*l*. *x* [^] *l*] *a-comm*
  by *simp*

 also have ... = (**1** ⊖ *x*) ⊗ *x* [^] *r* ⊕ (**1** ⊖ *x*) ⊗ (⨁ *l* ← [*0*..<*r*]. *x* [^] *l*)
  using *r-distr* **by** *auto*
 also have ... = *x* [^] *r* ⊖ *x* [^] (*Suc r*) ⊕ (**1** ⊖ *x*) ⊗ (⨁ *l* ← [*0*..<*r*]. *x* [^] *l*)
  apply (*intro arg-cong2*[**where** *f* = (⊕)] *refl*)
  unfolding *minus-eq*
   l-distr[*OF one-closed a-inv-closed*[*OF* ‹*x* ∈ *carrier R*›] *nat-pow-closed*[*OF* ‹*x* ∈ *carrier R*›]]
  using ‹*x* ∈ *carrier R*›
  using *l-minus nat-pow-Suc2* **by** *force*
 also have ... = *x* [^] *r* ⊖ *x* [^] (*Suc r*) ⊕ (**1** ⊖ *x* [^] *r*)
  using *Suc* **by** *presburger*
 also have ... = **1** ⊖ *x* [^] (*Suc r*)
  using *one-closed minus-add assms nat-pow-closed*[*of x*] **by** *algebra*
 finally show *?case* **.**
qed

rewrite *?x* ∈ *carrier R* ⟹ (*?x* [^] *?n*) [^] *?m* = *?x* [^] (*?n* ∗ *?m*) and *?a* ∗ *?b* = *?b* ∗ *?a* inside power sum

lemma *monoid-pow-sum-nat-pow-pow*:

assumes $x \in$ *carrier R*
shows $(\bigoplus i \leftarrow xs.\ f\ i \otimes x\ [\urcorner]\ ((g\ i :: nat) * h\ i)) = (\bigoplus i \leftarrow xs.\ f\ i \otimes (x\ [\urcorner]\ h\ i)$
$[\urcorner]\ g\ i)$
apply (*intro-cong* [*cong-tag-2* (⊗)] *more*: *monoid-sum-list-cong refl*)
using *nat-pow-pow*[*OF assms*] **by** (*simp add*: *mult.commute*)

end

context *cring*
begin

Split a power sum at some term

lemma *monoid-pow-sum-list-split*:
assumes $l + k = n$
assumes $\bigwedge i.\ i < n \Longrightarrow f\ i \in carrier\ R$
assumes $x \in carrier\ R$
shows $(\bigoplus i \leftarrow [0..<n].\ f\ i \otimes x\ [\urcorner]\ i) =$
$(\bigoplus i \leftarrow [0..<l].\ f\ i \otimes x\ [\urcorner]\ i) \oplus$
$x\ [\urcorner]\ l \otimes (\bigoplus i \leftarrow [0..<k].\ f\ (l + i) \otimes x\ [\urcorner]\ i)$
proof −
have $(\bigoplus i \leftarrow [0..<n].\ f\ i \otimes x\ [\urcorner]\ i) =$
$(\bigoplus i \leftarrow [0..<l].\ f\ i \otimes x\ [\urcorner]\ i) \oplus$
$(\bigoplus i \leftarrow [l..<n].\ f\ i \otimes x\ [\urcorner]\ i)$
apply (*intro monoid-sum-list-app′ m-closed nat-pow-closed upt-add-eq-append′*[*symmetric*])
using *assms* **by** *simp-all*
also have $(\bigoplus i \leftarrow [l..<n].\ f\ i \otimes x\ [\urcorner]\ i) =$
$(\bigoplus i \leftarrow [0..<k].\ f\ (l + i) \otimes x\ [\urcorner]\ (l + i))$
using *monoid-sum-list-index-shift-0*[*of - l n−l*] ‹$l + k = n$›
by *fastforce*
also have ... $= (\bigoplus i \leftarrow [0..<k].\ x\ [\urcorner]\ l \otimes (f\ (l + i) \otimes x\ [\urcorner]\ i))$
apply (*intro monoid-sum-list-cong*)
using *assms m-comm m-assoc nat-pow-mult*[*symmetric*, *OF* ‹$x \in carrier\ R$›]
by *simp*
also have ... $= x\ [\urcorner]\ l \otimes (\bigoplus i \leftarrow [0..<k].\ f\ (l + i) \otimes x\ [\urcorner]\ i)$
apply (*intro monoid-sum-list-in-left m-closed nat-pow-closed*)
using *assms* **by** *simp-all*
finally show *?thesis* **.**
qed

split power sum at term, more general

lemma *monoid-pow-sum-split*:
assumes $l + k = n$
assumes $\bigwedge i.\ i < n \Longrightarrow f\ i \in carrier\ R$
assumes $x \in carrier\ R$
shows $(\bigoplus i \leftarrow [0..<n].\ f\ i \otimes x\ [\urcorner]\ (i * c)) =$
$(\bigoplus i \leftarrow [0..<l].\ f\ i \otimes x\ [\urcorner]\ (i * c)) \oplus$
$x\ [\urcorner]\ (l * c) \otimes (\bigoplus i \leftarrow [0..<k].\ f\ (l + i) \otimes x\ [\urcorner]\ (i * c))$
proof −
have $(\bigoplus i \leftarrow [0..<n].\ f\ i \otimes x\ [\urcorner]\ (i * c)) = (\bigoplus i \leftarrow [0..<n].\ f\ i \otimes (x\ [\urcorner]\ c)\ [\urcorner]$

i)
  by (*intro monoid-pow-sum-nat-pow-pow* ‹*x* ∈ *carrier R*›)
 also have ... = (⨁ *i* ← [*0..<l*]. *f i* ⊗ (*x* [^] *c*) [^] *i*) ⊕
  (*x* [^] *c*) [^] *l* ⊗ (⨁ *i* ← [*0..<k*]. *f* (*l* + *i*) ⊗ (*x* [^] *c*) [^] *i*)
  by (*intro monoid-pow-sum-list-split assms nat-pow-closed*) *argo*
 also have ... = (⨁ *i* ← [*0..<l*]. *f i* ⊗ *x* [^] (*i* ∗ *c*)) ⊕
  x [^] (*c* ∗ *l*) ⊗ (⨁ *i* ← [*0..<k*]. *f* (*l* + *i*) ⊗ *x* [^] (*i* ∗ *c*))
 by (*intro-cong* [*cong-tag-2* (⊕), *cong-tag-2* (⊗)] *more*: *monoid-pow-sum-nat-pow-pow*[*symmetric*] *nat-pow-pow* ‹*x* ∈ *carrier R*›)
 also have ... = (⨁ *i* ← [*0..<l*]. *f i* ⊗ *x* [^] (*i* ∗ *c*)) ⊕
  x [^] (*l* ∗ *c*) ⊗ (⨁ *i* ← [*0..<k*]. *f* (*l* + *i*) ⊗ *x* [^] (*i* ∗ *c*))
   by (*intro-cong* [*cong-tag-2* (⊕), *cong-tag-2* (⊗), *cong-tag-2* ([^])] *more*: *refl mult.commute*)
 finally show *?thesis* **.**
qed

3.2.1 Algebraic operations

addition

lemma *monoid-pow-sum-add*:
 assumes ⋀*i*. *i* ∈ *set xs* ⟹ *f i* ∈ *carrier R*
 assumes ⋀*i*. *i* ∈ *set xs* ⟹ *g i* ∈ *carrier R*
 assumes *x* ∈ *carrier R*
 shows (⨁ *i* ← *xs*. *f i* ⊗ *x* [^] (*i*::*nat*)) ⊕ (⨁ *i* ← *xs*. *g i* ⊗ *x* [^] *i*) = (⨁ *i* ← *xs*. (*f i* ⊕ *g i*) ⊗ *x* [^] *i*)
proof −
 have (⨁ *i* ← *xs*. *f i* ⊗ *x* [^] *i*) ⊕ (⨁ *i* ← *xs*. *g i* ⊗ *x* [^] *i*) =
  (⨁ *i* ← *xs*. (*f i* ⊗ *x* [^] *i*) ⊕ (*g i* ⊗ *x* [^] *i*))
   apply (*intro monoid-sum-list-add-in m-closed nat-pow-closed assms*) **by** *assumption+*
 also have ... = (⨁ *i* ← *xs*. (*f i* ⊕ *g i*) ⊗ *x* [^] *i*)
  apply (*intro monoid-sum-list-cong l-distr*[*symmetric*] *nat-pow-closed assms*) **by** *assumption+*
 finally show *?thesis* **.**
qed

lemma *monoid-pow-sum-add'*:
 assumes ⋀*i*. *i* ∈ *set xs* ⟹ *f i* ∈ *carrier R*
 assumes ⋀*i*. *i* ∈ *set xs* ⟹ *g i* ∈ *carrier R*
 assumes *x* ∈ *carrier R*
shows (⨁ *i* ← *xs*. *f i* ⊗ *x* [^] ((*i*::*nat*) ∗ *c*)) ⊕ (⨁ *i* ← *xs*. *g i* ⊗ *x* [^] (*i* ∗ *c*)) = (⨁ *i* ← *xs*. (*f i* ⊕ *g i*) ⊗ *x* [^] (*i* ∗ *c*))
proof −
 have (⨁ *i* ← *xs*. *f i* ⊗ *x* [^] ((*i*::*nat*) ∗ *c*)) ⊕ (⨁ *i* ← *xs*. *g i* ⊗ *x* [^] (*i* ∗ *c*)) =
  (⨁ *i* ← *xs*. (*f i* ⊗ (*x* [^] *c*) [^] *i*)) ⊕ (⨁ *i* ← *xs*. (*g i* ⊗ (*x* [^] *c*) [^] *i*))
 by (*intro-cong* [*cong-tag-2* (⊕)] *more*: *monoid-pow-sum-nat-pow-pow* ‹*x* ∈ *carrier R*›)
 also have ... = (⨁ *i* ← *xs*. (*f i* ⊕ *g i*) ⊗ (*x* [^] *c*) [^] *i*)
  apply (*intro monoid-pow-sum-add nat-pow-closed*) **using** *assms* **by** *simp-all*

also have ... = $(\bigoplus i \leftarrow xs.\ (f\ i \oplus g\ i) \otimes x\ [\urcorner]\ (i * c))$
by (*intro monoid-pow-sum-nat-pow-pow*[*symmetric*] ‹$x \in carrier\ R$›)
finally show *?thesis* **.**
qed

unary minus

lemma *monoid-pow-sum-minus*:
assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i \in carrier\ R$
assumes $x \in carrier\ R$
shows $\ominus\ (\bigoplus i \leftarrow xs.\ f\ i \otimes x\ [\urcorner]\ (i :: nat)) = (\bigoplus i \leftarrow xs.\ (\ominus\ f\ i) \otimes x\ [\urcorner]\ i)$
proof −
have $\ominus\ (\bigoplus i \leftarrow xs.\ f\ i \otimes x\ [\urcorner]\ (i :: nat)) = (\bigoplus i \leftarrow xs.\ \ominus\ (f\ i \otimes x\ [\urcorner]\ (i :: nat)))$
apply (*intro monoid-sum-list-minus-in m-closed nat-pow-closed assms*) **by** *assumption*
also have ... = $(\bigoplus i \leftarrow xs.\ (\ominus\ f\ i) \otimes x\ [\urcorner]\ i)$
apply (*intro monoid-sum-list-cong l-minus*[*symmetric*] *nat-pow-closed assms*) **by** *assumption*
finally show *?thesis* **.**
qed

minus

lemma *monoid-pow-sum-diff*:
assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i \in carrier\ R$
assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow g\ i \in carrier\ R$
assumes $x \in carrier\ R$
shows $(\bigoplus i \leftarrow xs.\ f\ i \otimes x\ [\urcorner]\ (i::nat)) \ominus (\bigoplus i \leftarrow xs.\ g\ i \otimes x\ [\urcorner]\ (i :: nat)) =$
$(\bigoplus i \leftarrow xs.\ (f\ i \ominus g\ i) \otimes x\ [\urcorner]\ i)$
using *assms*
by (*simp add*: *minus-eq monoid-pow-sum-add*[*symmetric*] *monoid-pow-sum-minus*)

lemma *monoid-pow-sum-diff'*:
assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i \in carrier\ R$
assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow g\ i \in carrier\ R$
assumes $x \in carrier\ R$
shows $(\bigoplus i \leftarrow xs.\ f\ i \otimes x\ [\urcorner]\ ((i::nat) * c)) \ominus (\bigoplus i \leftarrow xs.\ g\ i \otimes x\ [\urcorner]\ (i * c)) =$
$(\bigoplus i \leftarrow xs.\ (f\ i \ominus g\ i) \otimes x\ [\urcorner]\ (i * c))$
proof −
have $(\bigoplus i \leftarrow xs.\ f\ i \otimes x\ [\urcorner]\ ((i::nat) * c)) \ominus (\bigoplus i \leftarrow xs.\ g\ i \otimes x\ [\urcorner]\ (i * c)) =$
$(\bigoplus i \leftarrow xs.\ f\ i \otimes (x\ [\urcorner]\ c)\ [\urcorner]\ i) \ominus (\bigoplus i \leftarrow xs.\ g\ i \otimes (x\ [\urcorner]\ c)\ [\urcorner]\ i)$
by (*intro-cong* [*cong-tag-2* ($\lambda i\ j.\ i \ominus j$)] *more*: *monoid-pow-sum-nat-pow-pow* ‹$x \in carrier\ R$›)
also have ... = $(\bigoplus i \leftarrow xs.\ (f\ i \ominus g\ i) \otimes (x\ [\urcorner]\ c)\ [\urcorner]\ i)$
apply (*intro monoid-pow-sum-diff nat-pow-closed*) **using** *assms* **by** *simp-all*
also have ... = $(\bigoplus i \leftarrow xs.\ (f\ i \ominus g\ i) \otimes x\ [\urcorner]\ (i * c))$
by (*intro monoid-pow-sum-nat-pow-pow*[*symmetric*] ‹$x \in carrier\ R$›)
finally show *?thesis* **.**
qed

end

3.3 *monoid-sum-list* in the context *residues*

context *residues*
begin

lemma *monoid-sum-list-eq-sum-list*:
$(\bigoplus_R\ i \leftarrow xs.\ f\ i) = (\sum i \leftarrow xs.\ f\ i)\ mod\ m$
  apply (*induction xs*)
  subgoal by (*simp add*: *zero-cong*)
  subgoal for *a xs* **by** (*simp add*: *mod-add-right-eq res-add-eq*)
  done

lemma *monoid-sum-list-mod-in*:
$(\bigoplus_R\ i \leftarrow xs.\ f\ i) = (\bigoplus_R\ i \leftarrow xs.\ (f\ i)\ mod\ m)$
  by (*induction xs*) (*simp-all add*: *mod-add-left-eq res-add-eq*)

lemma *monoid-sum-list-eq-sum-list'*:
$(\bigoplus_R\ i \leftarrow xs.\ f\ i\ mod\ m) = (\sum i \leftarrow xs.\ f\ i)\ mod\ m$
  using *monoid-sum-list-eq-sum-list monoid-sum-list-mod-in* **by** *metis*

end

end

4 The *estimation* tactic

theory *Estimation-Method*
  imports *Main HOL−Eisbach.Eisbach-Tools*
begin

A few useful lemmas for working with inequalities.

lemma *if-prop-cong*:
  assumes $C = C'$
  assumes $C \Longrightarrow P\ A\ A'$
  assumes $\neg\ C \Longrightarrow P\ B\ B'$
  shows $P\ (if\ C\ then\ A\ else\ B)\ (if\ C'\ then\ A'\ else\ B')$
  using *assms* **by** *simp*

lemma *if-leqI*:
  assumes $C \Longrightarrow A \leq t$
  assumes $\neg\ C \Longrightarrow B \leq t$
  shows $(if\ C\ then\ A\ else\ B) \leq t$
  using *assms* **by** *simp*

lemma *if-le-max*:
  $(if\ C\ then\ (t1\ ::\ 'a\ ::\ linorder)\ else\ t2) \leq max\ t1\ t2$
  by *simp*

Prove some inequality by showing a chain of inequalities via an intermediate term.

```
method itrans for step :: 'a :: order =
  (match conclusion in s ≤ t for s t :: 'a ⇒ ‹rule order.trans[of s step t]›)
```

A collection of monotonicity intro rules that will be automatically used by *estimation*.

```
lemmas mono-intros =
  order.refl add-mono diff-mono mult-le-mono max.mono min.mono power-increasing
power-mono
  iffD2[OF Suc-le-mono] if-prop-cong[where P = (≤)] Nat.le0 one-le-numeral
```

Try to apply a given estimation rule *estimate* in a forward-manner.

```
method estimation uses estimate =
  (match estimate in ⋀a. f a ≤ h a (multi) for f h ⇒ ‹
    match conclusion in g f ≤ t for g and t :: nat ⇒
    ‹rule order.trans[of g f g h t], intro mono-intros refl estimate››

  | x ≤ y for x y ⇒ ‹
    match conclusion in g x ≤ t for g and t :: nat ⇒
    ‹rule order.trans[of g x g y t], intro mono-intros refl estimate››)

end
theory Time-Monad-Extended
  imports Root-Balanced-Tree.Time-Monad
begin
```

5 Some Automation for *Root-Balanced-Tree.Time-Monad*

A bit of automation for statements involving the *time* component.

```
lemma time-bind-tm: time (s ⤜ f) = time s + time (f (val s))
  unfolding bind-tm-def
  by (simp split: tm.splits)

lemma time-tick: time (tick s) = 1
  by (simp add: tick-def)

lemmas tm-time-simps[simp] = time-bind-tm time-return time-tick if-distrib[of
time]

lemma bind-tm-cong[fundef-cong]:
  assumes f1 = f2
  assumes g1 (val f1) = g2 (val f2)
  shows f1 ⤜ g1 = f2 ⤜ g2
  using assms unfolding bind-tm-def
  by (auto split: tm.splits)
```

Introduce *val-simp* as named theorem. The idea is to collect simplification rules for the *Time-Monad.val* component that can be unfolded on their own.

named-theorems *val-simp*
declare *val-simps*[*val-simp*]

end
theory *Main-TM*
  imports *Main Time-Monad-Extended Estimation-Method*
begin

6 Running Time Formalization for some functions available in *Main*

6.1 Functions on *bool*

6.1.1 Not

fun *Not-tm* :: *bool* ⇒ *bool tm* **where**
Not-tm True =1 return False
| *Not-tm False =1 return True*

lemma *val-Not-tm*[*simp*, *val-simp*]: *val* (*Not-tm x*) = *Not x*
  by (*cases x*; *simp*)

lemma *time-Not-tm*[*simp*]: *time* (*Not-tm x*) = *1*
  by (*cases x*; *simp*)

6.1.2 disj / conj

definition *disj-tm* **where** *disj-tm x y =1 return* (*x* ∨ *y*)
definition *conj-tm* **where** *conj-tm x y =1 return* (*x* ∧ *y*)

lemma *val-disj-tm*[*simp*, *val-simp*]: *val* (*disj-tm x y*) = (*x* ∨ *y*)
  by (*simp add*: *disj-tm-def*)
lemma *time-disj-tm*[*simp*]: *time* (*disj-tm x y*) = *1*
  by (*simp add*: *disj-tm-def*)
lemma *val-conj-tm*[*simp*, *val-simp*]: *val* (*conj-tm x y*) = (*x* ∧ *y*)
  by (*simp add*: *conj-tm-def*)
lemma *time-conj-tm*[*simp*]: *time* (*conj-tm x y*) = *1*
  by (*simp add*: *conj-tm-def*)

6.1.3 equal

fun *equal-bool-tm* :: *bool* ⇒ *bool* ⇒ *bool tm* **where**
equal-bool-tm True p =1 return p
| *equal-bool-tm False p =1 Not-tm p*

lemma *val-equal-bool-tm*[*simp*, *val-simp*]: *val* (*equal-bool-tm x y*) = (*x* = *y*)
  by (*cases x*; *simp*)

lemma *time-equal-bool-tm-le*: *time* (*equal-bool-tm x y*) ≤ *2*

by (*cases x*; *simp*)

6.2 Functions involving pairs

6.2.1 *fst* / *snd*

fun *fst-tm* :: *'a* × *'b* ⇒ *'a tm* **where**
fst-tm (*x*, *y*) =*1 return x*
fun *snd-tm* :: *'a* × *'b* ⇒ *'b tm* **where**
snd-tm (*x*, *y*) =*1 return y*

lemma *val-fst-tm*[*simp*, *val-simp*]: *val* (*fst-tm p*) = *fst p*
by (*subst prod.collapse*[*symmetric*], *unfold fst-tm.simps*, *simp*)
lemma *time-fst-tm*[*simp*]: *time* (*fst-tm p*) = *1*
by (*subst prod.collapse*[*symmetric*], *unfold fst-tm.simps*, *simp*)
lemma *val-snd-tm*[*simp*, *val-simp*]: *val* (*snd-tm p*) = *snd p*
by (*subst prod.collapse*[*symmetric*], *unfold snd-tm.simps*, *simp*)
lemma *time-snd-tm*[*simp*]: *time* (*snd-tm p*) = *1*
by (*subst prod.collapse*[*symmetric*], *unfold snd-tm.simps*, *simp*)

6.3 Functions on *nat*

6.3.1 (+)

fun *plus-nat-tm* :: *nat* ⇒ *nat* ⇒ *nat tm* **where**
plus-nat-tm (*Suc m*) *n* =*1 plus-nat-tm m* (*Suc n*)
| *plus-nat-tm 0 n* =*1 return n*

lemma *val-plus-nat-tm*[*simp*, *val-simp*]: *val* (*plus-nat-tm m n*) = *m* + *n*
by (*induction m n rule*: *plus-nat-tm.induct*) *simp-all*

lemma *time-plus-nat-tm*[*simp*]: *time* (*plus-nat-tm m n*) = *m* + *1*
by (*induction m n rule*: *plus-nat-tm.induct*) *simp-all*

6.3.2 (∗)

fun *times-nat-tm* :: *nat* ⇒ *nat* ⇒ *nat tm* **where**
times-nat-tm 0 n =*1 return 0*
| *times-nat-tm* (*Suc m*) *n* =*1 do* {
r ← *times-nat-tm m n*;
plus-nat-tm n r
}

lemma *val-times-nat-tm*[*simp*]: *val* (*times-nat-tm m n*) = *m* ∗ *n*
by (*induction m n rule*: *times-nat-tm.induct*) *simp-all*

lemma *time-times-nat-tm*[*simp*]: *time* (*times-nat-tm m n*) = *m* ∗ (*n* + *2*) + *1*
by (*induction m n rule*: *times-nat-tm.induct*) *simp-all*

6.3.3 (^)

```
fun power-nat-tm :: nat ⇒ nat ⇒ nat tm where
power-nat-tm a 0 =1 return 1
| power-nat-tm a (Suc n) =1 do {
    r ← power-nat-tm a n;
    times-nat-tm a r
  }

lemma val-power-nat-tm[simp, val-simp]: val (power-nat-tm a n) = a ^ n
  by (induction a n rule: power-nat-tm.induct) simp-all

lemma time-power-nat-tm-aux0: time (power-nat-tm 0 n) = 2 * n + 1
  by (induction n) simp-all

lemma time-power-nat-tm-aux1: time (power-nat-tm 1 n) = 5 * n + 1
  by (induction n) simp-all

lemma time-power-nat-tm-aux2:
  assumes m ≥ 2
  shows time (power-nat-tm m n) ≤ (2 * n + m ^ n) * m + 2 * n + 1
proof (induction n)
  case 0
  then have time (power-nat-tm m 0) = 1 by simp
  then show ?case by simp
next
  case (Suc n)
  have time (power-nat-tm m (Suc n)) ≤ time (power-nat-tm m n) + (m ^ n +
2) * m + 2
    by simp
  also have ... ≤ (2 * n + m ^ n) * m + 2 * n + 1 + (m ^ n + 2) * m + 2
    using Suc by simp
  also have ... = (2 * n + m ^ n) * m + (m ^ n + 2) * m + 2 * Suc n + 1
    by simp
  also have ... = (2 * Suc n + 2 * m ^ n) * m + 2 * Suc n + 1
    using add-mult-distrib by simp
  also have ... ≤ (2 * Suc n + m ^ Suc n) * m + 2 * Suc n + 1
    using assms by simp
  finally show ?case .
qed

lemma time-power-nat-tm-le: time (power-nat-tm m n) ≤ 3 * m ^ Suc n + 5 * n
+ 1
proof −
  consider m = 0 | m = 1 | m ≥ 2 by linarith
  then show ?thesis
  proof cases
    case 1
    then show ?thesis using time-power-nat-tm-aux0[of n] by simp
  next
```

```
    case 2
    then show ?thesis using time-power-nat-tm-aux1[of n] by simp
  next
    case 3
    then have 2 ^ n ≤ m ^ n using power-mono by fast
    moreover have n < 2 ^ n by simp
    ultimately have n-le-m-pow-n: n ≤ m ^ n by linarith
    have time (power-nat-tm m n) ≤ (2 ∗ m ^ n + m ^ n) ∗ m + 2 ∗ n + 1
      apply (estimation estimate: time-power-nat-tm-aux2[OF 3, of n])
      using n-le-m-pow-n by simp
    also have ... = 3 ∗ m ^ Suc n + 2 ∗ n + 1 by simp
    finally show ?thesis by simp
  qed
qed

lemma time-power-nat-tm-2-le: time (power-nat-tm 2 n) ≤ 12 ∗ 2 ^ n
proof −
  have time (power-nat-tm 2 n) ≤ 3 ∗ 2 ^ Suc n + 5 ∗ n + 1
    by (fact time-power-nat-tm-le)
  also have ... ≤ 3 ∗ 2 ^ Suc n + 5 ∗ 2 ^ n + 2 ^ n
    apply (intro add-mono mult-le-mono order.refl)
    using less-exp[of n] by simp-all
  finally show ?thesis by simp
qed
```

6.3.4 (−)

```
fun minus-nat-tm :: nat ⇒ nat ⇒ nat tm where
minus-nat-tm m 0 =1 return m
| minus-nat-tm 0 m =1 return 0
| minus-nat-tm (Suc m) (Suc n) =1 minus-nat-tm m n

lemma val-minus-nat-tm[simp, val-simp]: val (minus-nat-tm m n) = m − n
  by (induction m n rule: minus-nat-tm.induct) simp-all

lemma time-minus-nat-tm[simp]: time (minus-nat-tm m n) = min m n + 1
  by (induction m n rule: minus-nat-tm.induct) simp-all
```

6.3.5 (<) / (≤)

```
fun less-eq-nat-tm :: nat ⇒ nat ⇒ bool tm and less-nat-tm :: nat ⇒ nat ⇒ bool
tm where
less-eq-nat-tm (Suc m) n =1 less-nat-tm m n
| less-eq-nat-tm 0 n =1 return True
| less-nat-tm m (Suc n) =1 less-eq-nat-tm m n
| less-nat-tm m 0 =1 return False

lemma val-less-eq-nat-tm[simp, val-simp]: (val (less-eq-nat-tm n m) = (n ≤ m))
  and val-less-nat-tm[simp, val-simp]: (val (less-nat-tm m n) = (m < n))
  by (induction m and n rule: less-eq-nat-tm-less-nat-tm.induct) auto
```

lemma *time-less-eq-nat-tm-aux*: *time* (*less-eq-nat-tm* (*m* + *k*) (*n* + *k*)) = *2* ∗ *k* + *time* (*less-eq-nat-tm m n*)
  by (*induction k*) *simp-all*
lemma *time-less-nat-tm-aux*: *time* (*less-nat-tm* (*m* + *k*) (*n* + *k*)) = *2* ∗ *k* + *time* (*less-nat-tm m n*)
  by (*induction k*) *simp-all*

lemma *time-less-eq-nat-tm*: *time* (*less-eq-nat-tm n m*) = *2* ∗ *min n m* + *1* + *of-bool* (*m* < *n*)
proof (*cases m* < *n*)
  case *True*
  then obtain *k* **where** *n* = *m* + *Suc k* **by** (*metis add-Suc-right less-natE*)
  then have *time* (*less-eq-nat-tm n m*) = *2* ∗ *m* + *2*
    using *time-less-eq-nat-tm-aux*[*of Suc k m 0*] **by** (*simp add*: *add.commute*)
  then show *?thesis* **using** *True* **by** *simp*
next
  case *False*
  then obtain *k* **where** *m* = *n* + *k* **using** *nat-le-iff-add*[*of n m*] **by** *auto*
  then have *time* (*less-eq-nat-tm n m*) = *2* ∗ *n* + *1*
    using *time-less-eq-nat-tm-aux*[*of 0 n k*] **by** (*simp add*: *add.commute*)
  then show *?thesis* **using** *False* **by** *simp*
qed
lemma *time-less-nat-tm*: *time* (*less-nat-tm m n*) = *2* ∗ *min m n* + *1* + *of-bool* (*m* < *n*)
proof (*cases m* < *n*)
  case *True*
  then obtain *k* **where** *n* = *m* + *Suc k* **by** (*metis add-Suc-right less-natE*)
  then have *time* (*less-nat-tm m n*) = *2* ∗ *m* + *2*
    using *time-less-nat-tm-aux*[*of 0 m Suc k*] **by** (*simp add*: *add.commute*)
  then show *?thesis* **using** *True* **by** *simp*
next
  case *False*
  then obtain *k* **where** *m* = *n* + *k* **using** *nat-le-iff-add*[*of n m*] **by** *auto*
  then have *time* (*less-nat-tm m n*) = *2* ∗ *n* + *1*
    using *time-less-nat-tm-aux*[*of k n 0*] **by** (*simp add*: *add.commute*)
  then show *?thesis* **using** *False* **by** *simp*
qed

lemma *time-less-eq-nat-tm-le*: *time* (*less-eq-nat-tm n m*) ≤ *2* ∗ *min n m* + *2*
  by (*simp add*: *time-less-eq-nat-tm*)
lemma *time-less-nat-tm-le*: *time* (*less-nat-tm m n*) ≤ *2* ∗ *min m n* + *2*
  by (*simp add*: *time-less-nat-tm*)

6.3.6 (=)

fun *equal-nat-tm* :: *nat* ⇒ *nat* ⇒ *bool tm* **where**
equal-nat-tm 0 0 =*1 return True*
| *equal-nat-tm* (*Suc x*) *0* =*1 return False*

```
| equal-nat-tm 0 (Suc y) =1 return False
| equal-nat-tm (Suc x) (Suc y) =1 equal-nat-tm x y

lemma val-equal-nat-tm[simp, val-simp]: val (equal-nat-tm x y) = (x = y)
  by (induction x y rule: equal-nat-tm.induct) simp-all

lemma time-equal-nat-tm: time (equal-nat-tm x y) = min x y + 1
  by (induction x y rule: equal-nat-tm.induct) simp-all
```

6.3.7 *max*

```
fun max-nat-tm :: nat ⇒ nat ⇒ nat tm where
max-nat-tm x y =1 do {
  b ← less-eq-nat-tm x y;
  if b then return y else return x
}

lemma val-max-nat-tm[simp, val-simp]: val (max-nat-tm x y) = max x y
  by simp

lemma time-max-nat-tm: time (max-nat-tm x y) = 2 ∗ min x y + 2 + of-bool (y
< x)
  by (simp add: time-less-eq-nat-tm)

lemma time-max-nat-tm-le: time (max-nat-tm x y) ≤ 2 ∗ min x y + 3
  unfolding time-max-nat-tm by simp
```

6.3.8 (*div*) / (*mod*)

```
fun divmod-nat-tm :: nat ⇒ nat ⇒ (nat × nat) tm where
divmod-nat-tm m n =1 do {
  n0 ← equal-nat-tm n 0;
  m-lt-n ← less-nat-tm m n;
  b ← disj-tm n0 m-lt-n;
  if b then return (0, m) else do {
    m-minus-n ← minus-nat-tm m n;
    (q, r) ← divmod-nat-tm m-minus-n n;
    return (Suc q, r)
  }
}
declare divmod-nat-tm.simps[simp del]

lemma val-divmod-nat-tm[simp, val-simp]: val (divmod-nat-tm m n) = Euclidean-Rings.divmod-nat
m n
proof (induction m n rule: divmod-nat-tm.induct)
  case (1 m n)
  show ?case
  proof (cases n = 0 ∨ m < n)
    case True
```

then show *?thesis* **unfolding** *divmod-nat-tm.simps*[*of m n*] **by** (*simp add*: *Euclidean-Rings.divmod-nat-if*)
next
case *False*
then have *val* (*divmod-nat-tm m n*) = (*let* (*q*, *r*) = *val* (*divmod-nat-tm* (*m* − *n*) *n*) *in* (*Suc q*, *r*))
unfolding *divmod-nat-tm.simps*[*of m n*]
by (*simp add*: *Let-def split*: *prod.splits*)
also have ... = (*let* (*q*, *r*) = *Euclidean-Rings.divmod-nat* (*m* − *n*) *n in* (*Suc q*, *r*))
using *1 False* **by** *simp*
also have ... = *Euclidean-Rings.divmod-nat m n*
unfolding *Euclidean-Rings.divmod-nat-if*[*of m n*]
by (*simp add*: *False split*: *prod.splits*)
finally show *?thesis* **.**
qed
qed

lemma *time-divmod-nat-tm-aux*:
assumes *r* < *n*
assumes *n* > *0*
shows *time* (*divmod-nat-tm* (*n* ∗ *k* + *r*) *n*) = *5* ∗ *k* + *3* ∗ *n* ∗ *k* + *time* (*divmod-nat-tm r n*)
using *assms*
proof (*induction k*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc k*)
then show *?case*
unfolding *divmod-nat-tm.simps*[*of n* ∗ (*Suc k*) + *r n*]
by (*simp add*: *time-equal-nat-tm time-less-nat-tm split*: *prod.splits*)
qed

lemma *time-divmod-nat-tm-le*: *time* (*divmod-nat-tm m n*) ≤ *8* ∗ *m* + *2* ∗ *n* + *5*
proof (*cases n* = *0* ∨ *m* < *n*)
case *True*
have *time* (*divmod-nat-tm m n*) = *time* (*equal-nat-tm n 0*) + *time* (*less-nat-tm m n*) + *2*
unfolding *divmod-nat-tm.simps*[*of m n*]
by (*simp add*: *True*)
also have ... ≤ *2* ∗ *min m n* + *5*
apply (*subst time-equal-nat-tm*)
apply (*estimation estimate*: *time-less-nat-tm-le*)
by *simp*
finally show *?thesis* **by** *simp*
next
case *False*

define *k r* **where** *k* = *m div n r* = *m mod n*
then have *krn*: *m* = *n* ∗ *k* + *r* **by** *simp*
from *k-r-def* **have** *r* < *n* **using** *False* **by** *simp*
have *time* (*divmod-nat-tm m n*) = *5* ∗ *k* + *3* ∗ *n* ∗ *k* + *time* (*divmod-nat-tm r n*)
apply (*subst krn*, *intro time-divmod-nat-tm-aux*, *intro* ‹*r* < *n*›)
using *False* **by** *simp*
also have *time* (*divmod-nat-tm r n*) = *time* (*equal-nat-tm n 0*) + *time* (*less-nat-tm r n*) + *2*
unfolding *divmod-nat-tm.simps*[*of r n*]
by (*simp add*: ‹*r* < *n*›)
also have ... ≤ *2* ∗ *min r n* + *5*
apply (*subst time-equal-nat-tm*)
apply (*estimation estimate*: *time-less-nat-tm-le*)
by *simp*
finally have *time* (*divmod-nat-tm m n*) ≤ *5* ∗ *k* + *3* ∗ *n* ∗ *k* + *2* ∗ *n* + *5*
by *simp*
also have ... ≤ *5* ∗ *k* + *3* ∗ *m* + *2* ∗ *n* + *5*
using *k-r-def* **by** *simp*
also have ... ≤ *8* ∗ *m* + *2* ∗ *n* + *5*
using *k-r-def* **by** *simp*
finally show *?thesis* **.**
qed

definition *divide-nat-tm* :: *nat* ⇒ *nat* ⇒ *nat tm* **where**
divide-nat-tm m n =*1* *divmod-nat-tm m n* ⋙= *fst-tm*

lemma *val-divide-nat-tm*[*simp*, *val-simp*]: *val* (*divide-nat-tm m n*) = *m div n*
by (*simp add*: *divide-nat-tm-def Euclidean-Rings.divmod-nat-def*)

lemma *time-divide-nat-tm-le*: *time* (*divide-nat-tm m n*) ≤ *8* ∗ *m* + *2* ∗ *n* + *7*
using *time-divmod-nat-tm-le*[*of m n*] **by** (*simp add*: *divide-nat-tm-def*)

definition *mod-nat-tm* :: *nat* ⇒ *nat* ⇒ *nat tm* **where**
mod-nat-tm m n =*1* *divmod-nat-tm m n* ⋙= *snd-tm*

lemma *val-mod-nat-tm*[*simp*, *val-simp*]: *val* (*mod-nat-tm m n*) = *m mod n*
by (*simp add*: *mod-nat-tm-def Euclidean-Rings.divmod-nat-def*)

lemma *time-mod-nat-tm-le*: *time* (*mod-nat-tm m n*) ≤ *8* ∗ *m* + *2* ∗ *n* + *7*
using *time-divmod-nat-tm-le*[*of m n*] **by** (*simp add*: *mod-nat-tm-def*)

definition *dvd-tm* **where** *dvd-tm a b* =*1* *do* {
b-mod-a ← *mod-nat-tm b a*;
equal-nat-tm b-mod-a 0
}

6.3.9 (*dvd*)

lemma *val-dvd-tm*[*simp*, *val-simp*]: *val* (*dvd-tm a b*) = (*a dvd b*)
unfolding *dvd-tm-def dvd-eq-mod-eq-0* **by** *simp*

lemma *time-dvd-tm-le*: *time* (*dvd-tm a b*) ≤ *8* ∗ *b* + *2* ∗ *a* + *9*
unfolding *dvd-tm-def tm-time-simps val-mod-nat-tm time-equal-nat-tm*
using *time-mod-nat-tm-le*[*of b a*] **by** *simp*

6.3.10 *even* / *odd*

definition *even-tm* **where** *even-tm a* = *dvd-tm 2 a*

lemma *val-even-tm*[*simp*, *val-simp*]: *val* (*even-tm a*) = *even a*
unfolding *even-tm-def* **by** *simp*

lemma *time-even-tm-le*: *time* (*even-tm a*) ≤ *8* ∗ *a* + *13*
unfolding *even-tm-def tm-time-simps*
using *time-dvd-tm-le*[*of 2 a*] **by** *simp*

definition *odd-tm* **where** *odd-tm a* = *dvd-tm 2 a* ⪢= *Not-tm*

lemma *val-odd-tm*[*simp*, *val-simp*]: *val* (*odd-tm a*) = *odd a*
unfolding *odd-tm-def* **by** *simp*

lemma *time-odd-tm-le*: *time* (*odd-tm a*) ≤ *8* ∗ *a* + *14*
unfolding *odd-tm-def tm-time-simps*
using *time-dvd-tm-le*[*of 2 a*] **by** *simp*

6.4 List functions

6.4.1 *take*

```
fun take-tm :: nat ⇒ 'a list ⇒ 'a list tm where
take-tm n [] =1 return []
| take-tm n (x # xs) =1 (case n of 0 ⇒ return [] | Suc m ⇒
  do {
    r ← take-tm m xs;
    return (x # r)
  })
```

lemma *val-take-tm*[*simp*, *val-simp*]: *val* (*take-tm n xs*) = *take n xs*
by (*induction n xs rule*: *take-tm.induct*) (*simp-all split*: *nat.splits*)

lemma *time-take-tm*: *time* (*take-tm n xs*) = *min n* (*length xs*) + *1*
by (*induction n xs rule*: *take-tm.induct*) (*simp-all split*: *nat.splits*)

lemma *time-take-tm-le*: *time* (*take-tm n xs*) ≤ *n* + *1*
by (*simp add*: *time-take-tm*)

6.4.2 *drop*

```
fun drop-tm :: nat ⇒ 'a list ⇒ 'a list tm where
drop-tm n [] =1 return []
| drop-tm n (x # xs) =1 (case n of 0 ⇒ return (x # xs) | Suc m ⇒
    do {
      r ← drop-tm m xs;
      return r
    })

lemma val-drop-tm[simp, val-simp]: val (drop-tm n xs) = drop n xs
  by (induction n xs rule: drop-tm.induct) (simp-all split: nat.splits)

lemma time-drop-tm: time (drop-tm n xs) = min n (length xs) + 1
  by (induction n xs rule: drop-tm.induct) (simp-all split: nat.splits)

lemma time-drop-tm-le: time (drop-tm n xs) ≤ n + 1
  by (simp add: time-drop-tm)
```

6.4.3 (@)

```
fun append-tm :: 'a list ⇒ 'a list ⇒ 'a list tm where
append-tm [] ys =1 return ys
| append-tm (x # xs) ys =1 do {
  r ← append-tm xs ys;
  return (x # r)
}

lemma val-append-tm[simp, val-simp]: val (append-tm xs ys) = append xs ys
  by (induction xs ys rule: append-tm.induct) simp-all

lemma time-append-tm[simp]: time (append-tm xs ys) = length xs + 1
  by (induction xs ys rule: append-tm.induct) simp-all
```

6.4.4 *fold*

```
fun fold-tm where
fold-tm f [] s =1 return s
| fold-tm f (x # xs) s =1 do {
    r ← f x s;
    fold-tm f xs r
  }

lemma val-fold-tm[simp, val-simp]: val (fold-tm f xs s) = fold (λx y. val (f x y))
xs s
  by (induction xs s rule: fold-tm.induct; simp)

lemma time-fold-tm-Cons: time (fold-tm (λx y. return (x # y)) xs s) = length xs
+ 1
  by (induction xs arbitrary: s; simp)
```

6.4.5 *rev*

definition *rev-tm* **where** *rev-tm xs =1 fold-tm (λx y. return (x # y)) xs []*

lemma *val-rev-tm[simp, val-simp]: val (rev-tm xs) = rev xs*
by *(induction xs; simp add: rev-tm-def fold-Cons-rev)*

lemma *time-rev-tm-le[simp]: time (rev-tm xs) = length xs + 2*
unfolding *rev-tm-def* **using** *time-fold-tm-Cons* **by** *auto*

6.4.6 *replicate*

fun *replicate-tm :: nat ⇒ 'a ⇒ 'a list tm* **where**
replicate-tm 0 x =1 return []
| replicate-tm (Suc n) x =1 do {
r ← replicate-tm n x;
return (x # r)
}

lemma *val-replicate-tm[simp, val-simp]: val (replicate-tm n x) = replicate n x*
by *(induction n x rule: replicate-tm.induct) simp-all*

lemma *time-replicate-tm: time (replicate-tm n x) = n + 1*
by *(induction n x rule: replicate-tm.induct) simp-all*

6.4.7 *length*

fun *gen-length-tm :: nat ⇒ 'a list ⇒ nat tm* **where**
gen-length-tm n [] =1 return n
| gen-length-tm n (x # xs) =1 gen-length-tm (Suc n) xs

lemma *val-gen-length-tm[simp, val-simp]: val (gen-length-tm n xs) = List.gen-length n xs*
by *(induction n xs rule: gen-length-tm.induct) (simp-all add: List.gen-length-def)*

lemma *time-gen-length-tm[simp]: time (gen-length-tm n xs) = length xs + 1*
by *(induction n xs rule: gen-length-tm.induct) simp-all*

definition *length-tm :: 'a list ⇒ nat tm* **where**
length-tm xs = gen-length-tm 0 xs

lemma *val-length-tm[simp, val-simp]: val (length-tm xs) = length xs*
by *(simp add: length-tm-def length-code)*

lemma *time-length-tm[simp]: time (length-tm xs) = length xs + 1*
by *(simp add: length-tm-def)*

6.4.8 *List.null*

fun *null-tm :: 'a list ⇒ bool tm* **where**

null-tm [] =*1 return True*
| *null-tm* (*x* # *xs*) =*1 return False*

lemma *val-null-tm*[*simp*, *val-simp*]: *val* (*null-tm xs*) = *List.null xs*
by (*cases xs*; *simp add*: *List.null-def*)

lemma *time-null-tm*[*simp*]: *time* (*null-tm xs*) = *1*
by (*cases xs*; *simp*)

6.4.9 *butlast*

fun *butlast-tm* :: *'a list* ⇒ *'a list tm* **where**
butlast-tm [] =*1 return* []
| *butlast-tm* (*x* # *xs*) =*1 do* {
    b ← *null-tm xs*;
    if b then return [] *else do* {
      r ← *butlast-tm xs*;
      return (*x* # *r*)
    }
  }

lemma *val-butlast-tm*[*simp*, *val-simp*]: *val* (*butlast-tm xs*) = *butlast xs*
by (*induction xs rule*: *butlast-tm.induct*) (*simp-all add*: *List.null-def*)

lemma *time-butlast-tm*: *time* (*butlast-tm xs*) = *2* ∗ (*length xs* − *1*) + *1* + *of-bool* (*length xs* ≥ *1*)
by (*induction xs rule*: *butlast-tm.induct*) (*auto simp*: *List.null-def not-less-eq-eq*)

lemma *time-butlast-tm-le*: *time* (*butlast-tm xs*) ≤ *2* ∗ *length xs* + *1*
unfolding *time-butlast-tm* **by** (*cases xs*; *simp*)

6.4.10 *map*

fun *map-tm* :: (*'a* ⇒ *'b tm*) ⇒ *'a list* ⇒ *'b list tm* **where**
map-tm f [] =*1 return* []
| *map-tm f* (*x* # *xs*) =*1 do* {
    r ← *f x*;
    rs ← *map-tm f xs*;
    return (*r* # *rs*)
  }

lemma *val-map-tm*[*simp*, *val-simp*]: *val* (*map-tm f xs*) = *map* (λ*x*. *val* (*f x*)) *xs*
by (*induction f xs rule*: *map-tm.induct*) *simp-all*

lemma *time-map-tm*: *time* (*map-tm f xs*) = (∑ *i* ← *xs*. *time* (*f i*)) + *length xs* + *1*
by (*induction f xs rule*: *map-tm.induct*) (*simp-all*)

lemma *time-map-tm-constant*:
assumes ⋀*i*. *i* ∈ *set xs* ⟹ *time* (*f i*) = *c*

shows *time* (*map-tm f xs*) = (*c* + *1*) ∗ *length xs* + *1*
proof −
have *time* (*map-tm f xs*) = (∑ *i* ← *xs*. *time* (*f i*)) + *length xs* + *1*
by (*simp add*: *time-map-tm*)
also have ... = (∑ *i* ← *xs*. *c*) + *length xs* + *1*
using *assms iffD2*[*OF map-eq-conv*, *of xs*] **by** *metis*
also have ... = *c* ∗ *length xs* + *length xs* + *1*
using *sum-list-triv*[*of c xs*] **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *time-map-tm-bounded*:
assumes ⋀*i*. *i* ∈ *set xs* ⟹ *time* (*f i*) ≤ *c*
shows *time* (*map-tm f xs*) ≤ (*c* + *1*) ∗ *length xs* + *1*
proof −
have *time* (*map-tm f xs*) = (∑ *i* ← *xs*. *time* (*f i*)) + *length xs* + *1*
by (*simp add*: *time-map-tm*)
also have ... ≤ (∑ *i* ← *xs*. *c*) + *length xs* + *1*
by (*intro add-mono order.refl sum-list-mono assms*) *argo*
also have ... = *c* ∗ *length xs* + *length xs* + *1*
using *sum-list-triv*[*of c xs*] **by** *simp*
finally show *?thesis* **by** *simp*
qed

6.4.11 *foldl*

fun *foldl-tm* :: (′*a* ⇒ ′*b* ⇒ ′*a tm*) ⇒ ′*a* ⇒ ′*b list* ⇒ ′*a tm* **where**
foldl-tm f a [] =*1 return a*
| *foldl-tm f a* (*x* # *xs*) =*1 do* {
r ← *f a x*;
foldl-tm f r xs
}

lemma *val-foldl-tm*[*simp*, *val-simp*]: *val* (*foldl-tm f a xs*) = *foldl* (λ*x y*. *val* (*f x y*)) *a xs*
by (*induction f a xs rule*: *foldl-tm.induct*; *simp*)

6.4.12 *concat*

fun *concat-tm* **where**
concat-tm [] =*1 return* []
| *concat-tm* (*x* # *xs*) =*1 do* {
r ← *concat-tm xs*;
append-tm x r
}

lemma *val-concat-tm*[*simp*, *val-simp*]: *val* (*concat-tm xs*) = *concat xs*
by (*induction xs*; *simp*)

lemma *time-concat-tm*[*simp*]: *time* (*concat-tm xs*) = *1* + *2* ∗ *length xs* + *length* (*concat xs*)
  by (*induction xs*; *simp*)

6.4.13 (!)

fun *nth-tm* **where**
nth-tm (*x* # *xs*) *0* =*1* *return x*
| *nth-tm* (*x* # *xs*) (*Suc i*) =*1* *nth-tm xs i*
| *nth-tm* [] - =*1* *undefined*

lemma *val-nth-tm*[*simp*, *val-simp*]:
  assumes *i* < *length xs*
  shows *val* (*nth-tm xs i*) = *xs* ! *i*
  using *assms*
proof (*induction i arbitrary*: *xs*)
  case *0*
  then show *?case* **using** *length-greater-0-conv*[*of xs*] *neq-Nil-conv*[*of xs*] **by** *auto*
next
  case (*Suc i*)
  then obtain *x xs′* **where** *xsr*: *xs* = *x* # *xs′* **by** (*meson Suc-lessE length-Suc-conv*)
  then have *i* < *length xs′* **using** *Suc.prems* **by** *simp*
  from *Suc.IH*[*OF this*] **show** *?case* **unfolding** *xsr* **by** *simp*
qed

lemma *time-nth-tm*[*simp*]:
  assumes *i* < *length xs*
  shows *time* (*nth-tm xs i*) = *i* + *1*
  using *assms*
proof (*induction i arbitrary*: *xs*)
  case *0*
  then show *?case* **using** *length-greater-0-conv*[*of xs*] *neq-Nil-conv*[*of xs*] **by** *auto*
next
  case (*Suc i*)
  then obtain *x xs′* **where** *xsr*: *xs* = *x* # *xs′* **by** (*meson Suc-lessE length-Suc-conv*)
  then have *i* < *length xs′* **using** *Suc.prems* **by** *simp*
  from *Suc.IH*[*OF this*] **show** *?case* **unfolding** *xsr* **by** *simp*
qed

6.4.14 *zip*

fun *zip-tm* :: *′a list* ⇒ *′b list* ⇒ (*′a* × *′b*) *list tm* **where**
zip-tm xs [] =*1* *return* []
| *zip-tm* [] *ys* =*1* *return* []
| *zip-tm* (*x* # *xs*) (*y* # *ys*) =*1* *do* { *rs* ← *zip-tm xs ys*; *return* ((*x*, *y*) # *rs*) }

lemma *val-zip-tm*[*simp*, *val-simp*]: *val* (*zip-tm xs ys*) = *zip xs ys*
  by (*induction xs ys rule*: *zip-tm.induct*; *simp*)

lemma *time-zip-tm*[*simp*]: *time* (*zip-tm xs ys*) = *min* (*length xs*) (*length ys*) + *1*

by (*induction xs ys rule*: *zip-tm.induct*; *simp*)

6.4.15 *map2*

definition *map2-tm* **where**
map2-tm f xs ys =1 do {
  xys ← *zip-tm xs ys*;
  map-tm (λ(*x*,*y*). *f x y*) *xys*
}

lemma *val-map2-tm*[*simp*, *val-simp*]: *val* (*map2-tm f xs ys*) = *map2* (λ*x y*. *val* (*f x y*)) *xs ys*
  unfolding *map2-tm-def* **by** (*simp split*: *prod.splits*)

lemma *time-map2-tm-bounded*:
  assumes *length xs* = *length ys*
  assumes ⋀*x y*. *x* ∈ *set xs* ⟹ *y* ∈ *set ys* ⟹ *time* (*f x y*) ≤ *c*
  shows *time* (*map2-tm f xs ys*) ≤ (*c* + *2*) ∗ *length xs* + *3*
proof −
  have *time* (*map2-tm f xs ys*) = *length xs* + *2* + *time* (*map-tm* (λ(*x*, *y*). *f x y*) (*zip xs ys*))
    unfolding *map2-tm-def* **by** (*simp add*: *assms*)
  also have ... ≤ *length xs* + *2* + ((*c* + *1*) ∗ *length* (*zip xs ys*) + *1*)
    apply (*intro add-mono order.refl time-map-tm-bounded*)
    using *assms* **by** (*auto split*: *prod.splits elim*: *in-set-zipE*)
  also have ... = (*c* + *2*) ∗ *length xs* + *3*
    using *assms* **by** *simp*
  finally show *?thesis* **.**
qed

6.4.16 *upt*

function *upt-tm* **where**
upt-tm i j =1 do {
  b ← *less-nat-tm i j*;
  (*if b then do* {
    rs ← *upt-tm* (*Suc i*) *j*;
    return (*i* # *rs*)
  } *else return* [])
}
  by *pat-completeness auto*
termination by (*relation Wellfounded.measure* (λ(*i*, *j*). *j* − *i*)) *simp-all*
declare *upt-tm.simps*[*simp del*]

lemma *val-upt-tm*[*simp*, *val-simp*]: *val* (*upt-tm i j*) = [*i*..<*j*]
  apply (*induction i j rule*: *upt-tm.induct*)
  subgoal for *i j*
    by (*cases i* < *j*; *simp add*: *upt-tm.simps*[*of i j*] *upt-conv-Cons*)
  done
lemma *time-upt-tm-le*: *time* (*upt-tm i j*) ≤ (*j* − *i*) ∗ (*2* ∗ *j* + *3*) + *2* ∗ *j* + *2*

```
proof (induction i j rule: upt-tm.induct)
  case (1 i j)
  then show ?case
  proof (cases i < j)
    case True
    then have time (upt-tm i j) = (2 * i + 3) + time (upt-tm (Suc i) j)
     unfolding upt-tm.simps[of i j] tm-time-simps by (simp add: time-less-nat-tm)
    also have ... ≤ (2 * j + 3) + ((j − Suc i) * (2 * j + 3) + 2 * j + 2)
      apply (intro add-mono mult-le-mono order.refl)
      subgoal using True by simp
      subgoal using 1 True by simp
      done
    also have ... = (j − Suc i + 1) * (2 * j + 3) + 2 * j + 2
      by simp
    also have j − Suc i + 1 = (j − i)
      using True by simp
    finally show ?thesis .
  next
    case False
    then show ?thesis by (simp add: upt-tm.simps[of i j] time-less-nat-tm)
  qed
qed

lemma time-upt-tm-le': time (upt-tm i j) ≤ 2 * j * j + 5 * j + 2
  apply (intro order.trans[OF time-upt-tm-le[of i j]])
  apply (estimation estimate: diff-le-self)
  by (simp add: add-mult-distrib2)
```

6.5 Syntactic sugar

```
consts equal-tm :: 'a ⇒ 'a ⇒ bool tm
adhoc-overloading equal-tm equal-nat-tm
adhoc-overloading equal-tm equal-bool-tm

consts plus-tm :: 'a ⇒ 'a ⇒ 'a tm
adhoc-overloading plus-tm plus-nat-tm

consts times-tm :: 'a ⇒ 'a ⇒ 'a tm
adhoc-overloading times-tm times-nat-tm

consts power-tm :: 'a ⇒ nat ⇒ 'a tm
adhoc-overloading power-tm power-nat-tm

consts minus-tm :: 'a ⇒ 'a ⇒ 'a tm
adhoc-overloading minus-tm minus-nat-tm

consts less-tm :: 'a ⇒ 'a ⇒ bool tm
adhoc-overloading less-tm less-nat-tm
```

consts *less-eq-tm* :: $'a \Rightarrow 'a \Rightarrow$ *bool tm*
adhoc-overloading *less-eq-tm less-eq-nat-tm*

consts *divide-tm* :: $'a \Rightarrow 'a \Rightarrow 'a$ *tm*
adhoc-overloading *divide-tm divide-nat-tm*

consts *mod-tm* :: $'a \Rightarrow 'a \Rightarrow 'a$ *tm*
adhoc-overloading *mod-tm mod-nat-tm*

bundle *main-tm-syntax*
begin
  notation *equal-tm* (**infixl** $=_t$ *51*)
  notation *Not-tm* ($\neg_t$ - [*40*] *40*)
  notation *conj-tm* (**infixr** $\wedge_t$ *35*)
  notation *disj-tm* (**infixr** $\vee_t$ *30*)
  notation *append-tm* (**infixr** $@_t$ *65*)
  notation *plus-tm* (**infixl** $+_t$ *65*)
  notation *times-tm* (**infixl** $*_t$ *70*)
  notation *power-tm* (**infixr** $\widehat{}_t$ *80*)
  notation *minus-tm* (**infixl** $-_t$ *65*)
  notation *less-tm* (**infix** $<_t$ *50*)
  notation *less-eq-tm* (**infix** $\leq_t$ *50*)
  notation *mod-tm* (**infixl** mod_t *70*)
  notation *divide-tm* (**infixl** div_t *70*)
  notation *dvd-tm* (**infix** dvd_t *50*)
end

bundle *no-main-tm-syntax*
begin
  no-notation *equal-tm* (**infixl** $=_t$ *51*)
  no-notation *Not-tm* ($\neg_t$ - [*40*] *40*)
  no-notation *conj-tm* (**infixr** $\wedge_t$ *35*)
  no-notation *disj-tm* (**infixr** $\vee_t$ *30*)
  no-notation *append-tm* (**infixr** $@_t$ *65*)
  no-notation *plus-tm* (**infixl** $+_t$ *65*)
  no-notation *times-tm* (**infixl** $*_t$ *70*)
  no-notation *power-tm* (**infixr** $\widehat{}_t$ *80*)
  no-notation *minus-tm* (**infixl** $-_t$ *65*)
  no-notation *less-tm* (**infix** $<_t$ *50*)
  no-notation *less-eq-tm* (**infix** $\leq_t$ *50*)
  no-notation *mod-tm* (**infixl** mod_t *70*)
  no-notation *divide-tm* (**infixl** div_t *70*)
  no-notation *dvd-tm* (**infix** dvd_t *50*)
end

unbundle *main-tm-syntax*

end

7 Representations

7.1 Abstract Representations

theory *Abstract-Representations*
  imports *Main*
begin

Idea: some type $'a$ is represented non-uniquely by some type $'b$. The function f produces a unique representant.

locale *abstract-representation* =
  fixes *from-type* :: $'a \Rightarrow 'b$
  fixes *to-type* :: $'b \Rightarrow 'a$
  fixes *f* :: $'b \Rightarrow 'b$
  assumes *to-from*: *to-type* ∘ *from-type* = *id*
  assumes *from-to*: *from-type* ∘ *to-type* = *f*
begin

lemma *to-from-elem*[*simp*]: *to-type* (*from-type x*) = *x*
  using *to-from* **by** (*metis comp-apply id-apply*)
lemma *from-to-elem*: *from-type* (*to-type x*) = *f x*
  using *from-to* **by** (*metis comp-apply*)

lemma *f-idem*: *f* ∘ *f* = *f*
proof –
  have *f* ∘ *f* = *from-type* ∘ *to-type* ∘ *from-type* ∘ *to-type*
    using *from-to* **by** *fastforce*
  also have ... = *from-type* ∘ *to-type*
    using *to-from* **by** (*simp add*: *rewriteR-comp-comp*)
  finally show *?thesis* **using** *from-to* **by** *simp*
qed

corollary *f-idem-elem*[*simp*]: *f* (*f x*) = *f x*
  using *f-idem* **by** (*metis comp-apply*)

lemma *f-from*: *f* ∘ *from-type* = *from-type*
proof –
  have *f* ∘ *from-type* = *from-type* ∘ *to-type* ∘ *from-type*
    using *from-to* **by** *simp*
  also have ... = *from-type*
    using *to-from* **by** (*simp add*: *rewriteR-comp-comp*)
  finally show *?thesis* **.**
qed

corollary *f-from-elem*[*simp*]: *f* (*from-type x*) = *from-type x*
  using *f-from* **by** (*metis comp-apply*)

lemma *to-f*: *to-type* ∘ *f* = *to-type*
proof –
  have *to-type* ∘ *f* = *to-type* ∘ *from-type* ∘ *to-type*

using *from-to* **by** *fastforce*
also have ... = *to-type* **using** *to-from* **by** *simp*
finally show *?thesis* **.**
qed

corollary *to-f-elem*[*simp*]: *to-type* (*f x*) = *to-type x*
using *to-f* **by** (*metis comp-apply*)

lemma *f-fixed-point-iff*: *f x* = *x* ⟷ (∃ *y*. *x* = *from-type y*)
proof
assume *f x* = *x*
then show ∃ *y*. *x* = *from-type y* **using** *from-to-elem* **by** *metis*
next
assume ∃ *y*. *x* = *from-type y*
then obtain *y* **where** *x* = *from-type y* **by** *blast*
then show *f x* = *x* **by** *simp*
qed

lemma *f-fixed-point-iff'*: *f x* = *x* ⟷ *x* = *from-type* (*to-type x*)
using *from-to* **by** *auto*

lemma *range-f-range-from*: *range f* = *range from-type*
proof (*standard*; *standard*)
fix *x*
assume *x* ∈ *range f*
then obtain *x'* **where** *x* = *f x'* **by** *blast*
then have *f x* = *x* **by** *simp*
then show *x* ∈ *range from-type* **using** *f-fixed-point-iff* **by** *blast*
next
fix *x*
assume *x* ∈ *range from-type*
then obtain *y* **where** *x* = *from-type y* **by** *blast*
then have *f x* = *x* **using** *f-fixed-point-iff* **by** *simp*
then show *x* ∈ *range f* **by** (*metis rangeI*)
qed

lemma *to-eq-iff-f-eq*: *to-type x* = *to-type y* ⟷ *f x* = *f y*
proof
show *to-type x* = *to-type y* ⟹ *f x* = *f y* **using** *from-to-elem*[*symmetric*] **by** *simp*
next
show *f x* = *f y* ⟹ *to-type x* = *to-type y* **using** *to-f-elem* **by** *metis*
qed

lemma *from-inj*: *inj from-type*
using *to-from* **by** (*metis inj-on-id inj-on-imageI2*)

end

lemma *from-to-f-criterion*:

assumes *to-type* ∘ *from-type* = *id*
assumes *f* ∘ *from-type* = *from-type*
assumes ⋀*x y*. *to-type x* = *to-type y* ⟹ *f x* = *f y*
shows *from-type* ∘ *to-type* = *f*
proof
fix *x*
have *to-type* (*from-type* (*to-type x*)) = *to-type x*
using *assms*(*1*) **by** (*metis comp-apply id-apply*)
hence *f* (*from-type* (*to-type x*)) = *f x*
using *assms*(*3*) **by** *metis*
hence *from-type* (*to-type x*) = *f x*
using *assms*(*2*) **by** (*metis comp-apply*)
thus (*from-type* ∘ *to-type*) *x* = *f x*
by (*metis comp-apply*)
qed

end

7.2 Abstract Representations 2

theory *Abstract-Representations-2*
imports *Main*
begin

Idea: a subset *represented-set* of some type *′a* is represented non-uniquely by some type *′b*.

locale *abstract-representation-2* =
fixes *from-type* :: *′a* ⇒ *′b*
fixes *to-type* :: *′b* ⇒ *′a*
fixes *represented-set* :: *′a set*
assumes *to-from*: ⋀*x*. *x* ∈ *represented-set* ⟹ *to-type* (*from-type x*) = *x*
assumes *to-type-in-represented-set*: ⋀*y*. *to-type y* ∈ *represented-set*
begin

definition *reduce* **where**
reduce x ≡ *from-type* (*to-type x*)

abbreviation *reduced* **where**
reduced x ≡ *reduce x* = *x*

lemma *reduce-reduce*[*simp*]: *reduced* (*reduce x*)
unfolding *reduce-def*
by (*simp add*: *to-from to-type-in-represented-set*)

definition *representations* **where**
representations ≡ *from-type* ‘ *represented-set*

lemma *range-reduce*: *representations* = *range reduce*
unfolding *representations-def reduce-def*

image-def
apply (*intro equalityI subsetI*)
subgoal for *x*
proof −
assume $x \in \{y.\ \exists x{\in}represented\text{-}set.\ y = from\text{-}type\ x\}$
then have $\exists y{\in}represented\text{-}set.\ x = from\text{-}type\ y$ **by** *simp*
then obtain *y* **where** $x = from\text{-}type\ y$ $y \in represented\text{-}set$ **by** *blast*
then have $to\text{-}type\ x = y$ **using** *to-from* **by** *simp*
then have $x = from\text{-}type\ (to\text{-}type\ x)$ **using** ‹$x = from\text{-}type\ y$› **by** *simp*
then show *?thesis* **by** *blast*
qed
subgoal for *x*
using *to-type-in-represented-set* **by** *blast*
done

corollary *reduced-from-type*[*simp*]: $x \in represented\text{-}set \Longrightarrow reduced\ (from\text{-}type\ x)$
using *range-reduce representations-def reduce-reduce* **by** *force*

lemma *to-type-reduce*: $to\text{-}type\ (reduce\ x) = to\text{-}type\ x$
unfolding *reduce-def*
by (*simp add*: *to-from to-type-in-represented-set*)

lemma *reduced-iff*: $reduced\ x \longleftrightarrow (\exists y{\in}represented\text{-}set.\ x = from\text{-}type\ y)$
apply *standard*
subgoal
using *reduce-def to-type-in-represented-set* **by** *metis*
subgoal
by *fastforce*
done

lemma *to-eq-iff-f-eq*: $to\text{-}type\ x = to\text{-}type\ y \longleftrightarrow reduce\ x = reduce\ y$
proof
show $to\text{-}type\ x = to\text{-}type\ y \Longrightarrow reduce\ x = reduce\ y$ **unfolding** *reduce-def* **by** *simp*
next
show $reduce\ x = reduce\ y \Longrightarrow to\text{-}type\ x = to\text{-}type\ y$ **using** *to-type-reduce* **by** *metis*
qed

lemma *from-inj*: *inj-on from-type represented-set*
unfolding *inj-on-def*
apply *standard+*
subgoal for *x y*
using *to-from*[*of x*, *symmetric*] *to-from*[*of y*] **by** *simp*
done

corollary *from-bij-betw*: *bij-betw from-type represented-set representations*
unfolding *representations-def*
using *from-inj*

by (*simp add*: *inj-on-imp-bij-betw*)

lemma *correctness-to-from*:
fixes $h :: 'a \Rightarrow 'a \Rightarrow 'a$
fixes $g :: 'b \Rightarrow 'b \Rightarrow 'b$
assumes $\bigwedge x\ y.\ \textit{to-type}\ (g\ x\ y) = h\ (\textit{to-type}\ x)\ (\textit{to-type}\ y)$
shows $\bigwedge x\ y.\ x \in \textit{represented-set} \Longrightarrow y \in \textit{represented-set} \Longrightarrow \textit{reduce}\ (g\ (\textit{from-type}\ x)\ (\textit{from-type}\ y)) = \textit{from-type}\ (h\ x\ y)$
proof −
fix *x y*
assume $x \in \textit{represented-set}$ $y \in \textit{represented-set}$
have $\textit{reduce}\ (g\ (\textit{from-type}\ x)\ (\textit{from-type}\ y)) = \textit{from-type}\ (\textit{to-type}\ (g\ (\textit{from-type}\ x)\ (\textit{from-type}\ y)))$
unfolding *reduce-def* **by** *simp*
also have ... $= \textit{from-type}\ (h\ (\textit{to-type}\ (\textit{from-type}\ x))\ (\textit{to-type}\ (\textit{from-type}\ y)))$
using *assms* **by** *simp*
also have ... $= \textit{from-type}\ (h\ x\ y)$
using *to-from* ‹$x \in \textit{represented-set}$› ‹$y \in \textit{represented-set}$› **by** *simp*
finally show $\textit{reduce}\ (g\ (\textit{from-type}\ x)\ (\textit{from-type}\ y)) = \textit{from-type}\ (h\ x\ y)$ **.**
qed

end

lemma *from-to-f-criterion*:
assumes $\bigwedge x.\ x \in \textit{represented-set} \Longrightarrow \textit{to-type}\ (\textit{from-type}\ x) = x$
assumes $\bigwedge x.\ x \in \textit{represented-set} \Longrightarrow f\ (\textit{from-type}\ x) = \textit{from-type}\ x$
assumes $\bigwedge x\ y.\ \textit{to-type}\ x = \textit{to-type}\ y \Longrightarrow f\ x = f\ y$
assumes $\bigwedge y.\ \textit{to-type}\ y \in \textit{represented-set}$
shows $\bigwedge x.\ \textit{from-type}\ (\textit{to-type}\ x) = f\ x$
proof −
fix *x*
have $\textit{to-type}\ (\textit{from-type}\ (\textit{to-type}\ x)) = \textit{to-type}\ x$
using *assms*(*1*) *assms*(*4*) **by** *simp*
hence $f\ (\textit{from-type}\ (\textit{to-type}\ x)) = f\ x$
using *assms*(*3*) **by** *metis*
thus $\textit{from-type}\ (\textit{to-type}\ x) = f\ x$
using *assms*(*2*) *assms*(*4*) **by** *simp*
qed

end
theory *Nat-LSBF*
imports *Main ../Preliminaries/Karatsuba-Sum-Lemmas Abstract-Representations HOL−Library.Log-Nat*
begin

8 Representing *nat* in LSBF

In this theory, a representation of *nat* is chosen and simple algorithms implemented thereon.

lemma *list-isolate-nth*: *i* < *length xs* ⟹ ∃ *xs1 xs2*. *xs* = *xs1* @ (*xs* ! *i*) # *xs2* ∧ *length xs1* = *i*
  using *id-take-nth-drop* **by** *fastforce*

lemma *list-is-replicate-iff*: *xs* = *replicate* (*length xs*) *x* ⟷ (∀ *i* ∈ {*0*..<*length xs*}. *xs* ! *i* = *x*)
proof
  assume *1*: *xs* = *replicate* (*length xs*) *x*
  show ∀ *i* ∈ {*0*..<*length xs*}. *xs* ! *i* = *x*
    using *1 nth-replicate*[*of - length xs x*] **by** *auto*
next
  assume ∀ *i* ∈ {*0*..<*length xs*}. *xs* ! *i* = *x*
  then have ∀ *i* ∈ {*0*..<*length xs*}. *xs* ! *i* = (*replicate* (*length xs*) *x*) ! *i*
    using *nth-replicate* **by** *auto*
  then show *xs* = *replicate* (*length xs*) *x*
    using *nth-equalityI*[*of xs replicate* (*length xs*) *x*] **by** *simp*
qed

lemma *list-is-replicate-iff2*: *xs* = *replicate* (*length xs*) *x* ⟷ *set xs* = {*x*} ∨ *xs* = []
  by (*metis empty-replicate length-0-conv replicate-eqI set-replicate singleton-iff*)

lemma *set-bool-list*: *set xs* ⊆ {*True*, *False*}
  by *auto*
lemma *bool-list-is-replicate-if*:
  assumes *a* ∉ *set xs* **shows** *xs* = *replicate* (*length xs*) (¬ *a*)
proof (*intro iffD2*[*OF list-is-replicate-iff2*])
  from *assms set-bool-list* **have** *set xs* ⊆ {¬ *a*} **by** *fastforce*
  then have *set xs* = {¬ *a*} ∨ *set xs* = {} **by** (*meson subset-singletonD*)
  then show *set xs* = {¬ *a*} ∨ *xs* = [] **by** *simp*
qed

lemma *bit-strong-decomp-2*: ∃ *ys zs*. *xs* = *ys* @ *a* # *zs* ⟹ ∃ *ys′ n*. *xs* = *ys′* @ *a* # (*replicate n* (¬ *a*))
proof −
  assume ∃ *ys zs*. *xs* = *ys* @ *a* # *zs*
  then have *a* ∈ *set xs* **by** *auto*
  from *split-list-last*[*OF this*] **obtain** *ys zs* **where** *xs* = *ys* @ *a* # *zs a* ∉ *set zs* **by** *blast*
  from *this*(*2*) **have** *zs* = *replicate* (*length zs*) (¬*a*)
    by (*intro bool-list-is-replicate-if*)
  with ‹*xs* = *ys* @ *a* # *zs*› **show** *?thesis* **by** *blast*
qed

lemma *bit-strong-decomp-1*: ∃ *ys zs*. *xs* = *ys* @ *a* # *zs* ⟹ ∃ *ys′ n*. *xs* = (*replicate*

n (¬ *a*) @ *a* # *ys′*)
proof −
  assume ∃ *ys zs*. *xs* = *ys* @ *a* # *zs*
  then obtain *ys zs* **where** *xs* = *ys* @ *a* # *zs* **by** *blast*
  then have *rev xs* = *rev zs* @ [*a*] @ *rev ys* **by** *simp*
  then obtain *n ys′* **where** *rev xs* = *ys′* @ [*a*] @ *replicate n* (¬ *a*)
    using *bit-strong-decomp-2*[*of rev xs a*] **by** *auto*
  then have *xs* = *replicate n* (¬ *a*) @ [*a*] @ *rev ys′*
   by (*metis append-assoc rev-append rev-replicate rev-rev-ident rev-singleton-conv*)
  thus *?thesis* **by** *auto*
qed

8.1 Type definition

type-synonym *nat-lsbf* = *bool list*

8.2 Conversions

fun *eval-bool* :: *bool* ⇒ *nat* **where**
eval-bool True = *1*
| *eval-bool False* = *0*

lemma *eval-bool-is-of-bool*[*simp*]: *eval-bool* = *of-bool*
  by *auto*

lemma *eval-bool-leq-1*: *eval-bool a* ≤ *1*
  by (*cases a*) *simp-all*

lemma *eval-bool-inj*: *eval-bool a* = *eval-bool b* ⟹ *a* = *b*
  by (*cases a*; *cases b*) *simp-all*

fun *to-nat* :: *nat-lsbf* ⇒ *nat* **where**
to-nat [] = *0*
| *to-nat* (*x*#*xs*) = (*eval-bool x*) + *2* ∗ *to-nat xs*

fun *from-nat* :: *nat* ⇒ *nat-lsbf* **where**
from-nat 0 = []
| *from-nat x* = (*if x mod 2* = *0 then False else True*)#(*from-nat* (*x div 2*))

value *from-nat 103*
value *to-nat* (*from-nat 103*)

lemma *to-nat-from-nat*[*simp*]: *to-nat* (*from-nat x*) = *x*
proof (*induction x rule*: *less-induct*)
  case (*less x*)
  consider *x* = *0* | *x* > *0* **by** *auto*
  then show *?case*
  proof (*cases*)
    case *1*
    then show *?thesis* **by** *simp*

```
  next
    case 2
    then have to-nat (from-nat x) = eval-bool (if x mod 2 = 0 then False else
True) + 2 * to-nat (from-nat (x div 2))
      by (metis from-nat.elims nat-less-le to-nat.simps(2))
    also have ... = (x mod 2) + 2 * to-nat (from-nat (x div 2))
      by simp
    also have ... = (x mod 2) + 2 * (x div 2)
      using less 2 by simp
    also have ... = x by simp
    finally show ?thesis .
  qed
qed

lemma to-nat-explicitly: to-nat xs = (∑ i ← [0..<length xs]. eval-bool (xs ! i) * 2
^ i)
proof (induction xs rule: to-nat.induct)
  case 1
  then show ?case by simp
next
  case (2 x xs)
  let ?xs = λi. eval-bool ((x # xs) ! i)
  have (∑ i←[0..<length (x # xs)]. ?xs i * 2 ^ i)
    = ?xs 0 + (∑ i←[1..<length (x # xs)]. ?xs i * 2 ^ i)
    by (simp add: upt-rec)
  also have ... = ?xs 0 + (∑ i←[0..<length xs]. ?xs (i + 1) * 2 ^ (i + 1))
    using list-sum-index-shift[of - length xs 0 λi. ?xs i * 2 ^ i] by simp
  also have ... = ?xs 0 + 2 * (∑ i←[0..<length xs]. ?xs (i + 1) * 2 ^ i)
    by (simp add: sum-list-const-mult mult.left-commute)
  also have ... = ?xs 0 + 2 * to-nat xs
    using 2 by simp
  also have ... = to-nat (x # xs) by simp
  finally show ?case by simp
qed

lemma to-nat-app: to-nat (xs @ ys) = to-nat xs + (2 ^ length xs) * to-nat ys
  by (induction xs) auto

lemma to-nat-length-upper-bound: to-nat xs ≤ 2 ^ (length xs) − 1
proof (induction xs)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  then have to-nat (a # xs) = eval-bool a + 2 * to-nat xs by simp
  also have ... ≤ eval-bool a + 2 * (2 ^ (length xs) − 1) using Cons.IH by simp
  also have ... ≤ 1 + 2 * (2 ^ (length xs) − 1) using eval-bool-leq-1[of a] by
simp
  also have ... = 1 + (2 ^ (length xs + 1) − 1 − 1) by simp
```

also have ... = 2 ^ ($length$ xs + 1) − 1
apply ($intro$ add-$diff$-$inverse$-nat)
using $power$-$increasing$[of 1 $length$ xs + 1 2::nat]
by ($simp$ add: $add.commute$)
finally show *?case* **by** *simp*
qed
lemma *to-nat-length-bound*: *to-nat xs* < *2* ^ *length xs*
using *to-nat-length-upper-bound*[*of xs*]
using *le-eq-less-or-eq* **by** *fastforce*
lemma *to-nat-length-lower-bound*: *to-nat* (*xs* @ [*True*]) ≥ *2* ^ *length xs*
by (*induction xs*) *auto*

lemma *to-nat-replicate-false*[*simp*]: *to-nat* (*replicate n False*) = *0*
by (*induction n*) *simp-all*

lemma *to-nat-one-bit*[*simp*]: *to-nat* (*replicate n False* @ [*True*]) = *2* ^ *n*
by (*simp add*: *to-nat-app*)

lemma *to-nat-replicate-true*[*simp*]: *to-nat* (*replicate n True*) = *2* ^ *n* − *1*
proof (*induction n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
have *2* ^ (*Suc n*) ≥ (*2* :: *nat*) **by** *simp*
hence *1*: *2* ^ (*Suc n*) − *1* ≥ (*1* :: *nat*) **by** *linarith*
have *to-nat* (*replicate* (*Suc n*) *True*) = *1* + *2* ∗ *to-nat* (*replicate n True*)
by *simp*
also have ... = *1* + *2* ∗ (*2* ^ *n* − *1*)
using *Suc.IH* **by** *simp*
also have ... = *2* ^ (*Suc n*) − *1*
using *le-add-diff-inverse*[*of 1 2* ^ (*Suc n*) − *1*]
using *1* **by** *simp*
finally show *?case* **.**
qed

lemma *to-nat xs* = *0* ⟷ (∃ *n*. *xs* = *replicate n False*)
proof
show *to-nat xs* = *0* ⟹ ∃ *n*. *xs* = *replicate n False*
proof (*induction xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then have *a* = *False to-nat xs* = *0* **by** *auto*
then obtain *n* **where** *xs* = *replicate n False* **using** *Cons.IH* **by** *auto*
hence *a* # *xs* = *replicate* (*Suc n*) *False* **using** ‹*a* = *False*› **by** *simp*
then show *?case* **by** *blast*
qed

show $\exists\, n.\ xs = replicate\ n\ False \Longrightarrow to\text{-}nat\ xs = 0$
  using *to-nat-replicate-false* **by** *auto*
qed

lemma *to-nat-app-replicate*[*simp*]: $to\text{-}nat\ (xs\ @\ replicate\ n\ False) = to\text{-}nat\ xs$
 by (*induction xs*) *auto*

lemma *change-bit-ineq*: $length\ xs = length\ ys \Longrightarrow to\text{-}nat\ (xs\ @\ False\ \#\ zs) < to\text{-}nat\ (ys\ @\ True\ \#\ zs)$
proof −
 assume $length\ xs = length\ ys$
 have $to\text{-}nat\ (xs\ @\ False\ \#\ zs) = to\text{-}nat\ xs + 2\ \hat{}\ (length\ xs + 1) * to\text{-}nat\ zs$
  using *to-nat-app-replicate*[*of xs 1*] *to-nat-app* **by** *simp*
 also have ... $\leq 2\ \hat{}\ (length\ xs) - 1 + 2\ \hat{}\ (length\ xs + 1) * to\text{-}nat\ zs$
  using *to-nat-length-upper-bound*[*of xs*] **by** *linarith*
 also have ... $< 2\ \hat{}\ (length\ xs) + 2\ \hat{}\ (length\ xs + 1) * to\text{-}nat\ zs$ **by** *simp*
 also have ... $= 2\ \hat{}\ (length\ ys) + 2\ \hat{}\ (length\ ys + 1) * to\text{-}nat\ zs$
  using ‹$length\ xs = length\ ys$› **by** *simp*
 also have ... $\leq to\text{-}nat\ (ys\ @\ [True]) + 2\ \hat{}\ (length\ ys + 1) * to\text{-}nat\ zs$
  using *to-nat-length-lower-bound*[*of ys*] **by** *simp*
 also have ... $= to\text{-}nat\ (ys\ @\ True\ \#\ zs)$
  using *to-nat-app* **by** *simp*
 finally show *?thesis* **.**
qed

lemma *to-nat-ineq-imp-False-bit*: $to\text{-}nat\ xs < 2\ \hat{}\ length\ xs - 1 \Longrightarrow \exists\, ys\ zs.\ xs = ys\ @\ False\ \#\ zs$
proof (*rule ccontr*)
 assume $\nexists\, ys\ zs.\ xs = ys\ @\ False\ \#\ zs$
 then have $\forall\, i \in \{0..<length\ xs\}.\ xs\ !\ i = True$
  by (*metis*(*full-types*) *atLeastLessThan-iff in-set-conv-decomp-first in-set-conv-nth*)
 then have $xs = replicate\ (length\ xs)\ True$ **using** *list-is-replicate-iff* **by** *fast*
 then have $to\text{-}nat\ xs = 2\ \hat{}\ length\ xs - 1$ **using** *to-nat-replicate-true* **by** *metis*
 thus $to\text{-}nat\ xs < 2\ \hat{}\ length\ xs - 1 \Longrightarrow False$ **by** *simp*
qed

lemma *to-nat-bound-to-length-bound*: $to\text{-}nat\ xs \geq 2\ \hat{}\ n \Longrightarrow length\ xs \geq n + 1$
proof (*rule ccontr*)
 assume $to\text{-}nat\ xs \geq 2\ \hat{}\ n$
 assume $\neg\ n + 1 \leq length\ xs$
 then have $n \geq length\ xs$ **by** *simp*
 then have $to\text{-}nat\ xs \geq 2\ \hat{}\ length\ xs$ **using** ‹$to\text{-}nat\ xs \geq 2\ \hat{}\ n$›
  using *power-increasing le-trans one-le-numeral* **by** *meson*
 then show *False* **using** *to-nat-length-bound*[*of xs*] **by** *simp*
qed

lemma *to-nat-drop-take*: $to\text{-}nat\ xs = to\text{-}nat\ (take\ k\ xs) + 2\ \hat{}\ k * to\text{-}nat\ (drop\ k\ xs)$
proof −

have *xs* = *take k xs* @ *drop k xs* **by** *simp*
then have *to-nat xs* = *to-nat* (*take k xs*) + *2* ^ (*length* (*take k xs*)) ∗ *to-nat* (*drop k xs*)
using *to-nat-app* **by** *metis*
also have *2* ^ (*length* (*take k xs*)) ∗ *to-nat* (*drop k xs*) = *2* ^ *k* ∗ *to-nat* (*drop k xs*)
by (*cases length xs* < *k*) *simp-all*
finally show *?thesis* **.**
qed

lemma *to-nat-take*: *to-nat* (*take k xs*) = *to-nat xs mod 2* ^ *k*
proof −
have *to-nat xs* = *to-nat* (*take k xs*) + *2* ^ *k* ∗ *to-nat* (*drop k xs*)
by (*simp add*: *to-nat-drop-take*)
then have *to-nat xs mod 2* ^ *k* = *to-nat* (*take k xs*) *mod 2* ^ *k* **by** *simp*
moreover have *to-nat* (*take k xs*) < *2* ^ *k*
using *to-nat-length-bound*[*of take k xs*] *length-take*[*of k xs*]
by (*metis add-leD1 leI min-absorb2 min-def to-nat-bound-to-length-bound*)
ultimately show *?thesis* **by** *simp*
qed

lemma *to-nat-drop*: *to-nat* (*drop k xs*) = *to-nat xs div 2* ^ *k*
proof −
have *to-nat xs* = *to-nat xs mod 2* ^ *k* + *2* ^ *k* ∗ *to-nat* (*drop k xs*)
using *to-nat-drop-take*[*of xs k*] *to-nat-take*[*of k xs*] **by** *argo*
then have *to-nat xs div 2* ^ *k* = *to-nat* (*drop k xs*)
by (*metis add.right-neutral bits-mod-div-trivial div-mult-self2 power-not-zero zero-neq-numeral*)
thus *?thesis* **by** *rule*
qed

lemma *to-nat-nth-True-bound*:
assumes *i* < *length xs*
assumes *xs* ! *i* = *True*
shows *to-nat xs* ≥ *2* ^ *i*
proof −
from *assms* **have** *xs* = (*take i xs* @ [*True*]) @ *drop* (*Suc i*) *xs*
using *id-take-nth-drop* **by** *fastforce*
then show *to-nat xs* ≥ *2* ^ *i*
using *to-nat-app*[*of - drop* (*Suc i*) *xs*] *to-nat-length-lower-bound*[*of take i xs*] ‹*i* < *length xs*›
by (*metis append-eq-conv-conj le-add1 le-eq-less-or-eq list-isolate-nth trans-less-add1*)
qed

8.3 Truncating and filling

fun *truncate-reversed* :: *bool list* ⇒ *bool list* **where**
truncate-reversed [] = []
| *truncate-reversed* (*x*#*xs*) = (*if x then x*#*xs else truncate-reversed xs*)

```
definition truncate :: nat-lsbf ⇒ nat-lsbf where
truncate xs = rev (truncate-reversed (rev xs))

abbreviation truncated where truncated x ≡ truncate x = x

lemma truncate-reversed-eqI[simp]: xs = (replicate n False) @ ys ⟹ truncate-reversed
xs = truncate-reversed ys
  by (induction n arbitrary: xs ys) auto
corollary truncate-eqI[simp]: xs = ys @ (replicate n False) ⟹ truncate xs =
truncate ys
  by (simp add: truncate-def)

lemma replicate-truncate-reversed: ∃ n. (replicate n False) @ truncate-reversed xs
= xs
proof (induction xs)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  then obtain n where 1: replicate n False @ truncate-reversed xs = xs by blast
  hence a # xs = a # replicate n False @ truncate-reversed xs by simp
  show ?case
  proof (cases a)
    case True
    then have truncate-reversed (a # xs) = a # xs by simp
    also have ... = replicate 0 False @ a # xs by simp
    finally show ?thesis by simp
  next
    case False
    then have truncate-reversed (a # xs) = truncate-reversed xs by simp
    hence replicate (Suc n) False @ truncate-reversed (a # xs) = False # replicate
n False @ truncate-reversed xs
      by simp
    with 1 False have replicate (Suc n) False @ truncate-reversed (a # xs) = a #
xs by simp
    then show ?thesis by blast
  qed
qed
corollary truncate-replicate: ∃ n. truncate xs @ (replicate n False) = xs
proof -
  from replicate-truncate-reversed[of rev xs]
  obtain n where replicate n False @ truncate-reversed (rev xs) = rev xs by blast
  hence rev (truncate-reversed (rev xs)) @ rev (replicate n False) = xs
    using rev-append[symmetric, of truncate-reversed (rev xs) replicate n False]
    using rev-rev-ident[of xs]
    by simp
  hence truncate xs @ replicate n False = xs by (simp add: truncate-def)
  thus ?thesis by blast
```

qed
lemma *decompose-trailing-zeros*: *xs* = *truncate xs* @ (*replicate* (*length xs* − *length* (*truncate xs*)) *False*)
  using *truncate-replicate*[*of xs*]
  by (*metis add-diff-cancel-left′ length-append length-replicate*)

lemma *truncate-reversed-length-ineq*: *length* (*truncate-reversed xs*) ≤ *length xs*
  by (*induction xs*) *simp-all*
lemma *truncate-length-ineq*: *length* (*truncate xs*) ≤ *length xs*
  by (*metis Nat-LSBF.truncate-def length-rev truncate-reversed-length-ineq*)

lemma *truncate-reversed-fixed-point-iff*: *truncate-reversed x* = *x* ⟷ (*x* = [] ∨ *hd x* = *True*)
proof (*induction x*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a x*)
  then have (*a* # *x* = [] ∨ *hd* (*a* # *x*) = *True*) = *a* **by** *simp*
  moreover have *a* ⟹ *truncate-reversed* (*a* # *x*) = *a* # *x* **by** *simp*
  moreover have ¬ *a* ⟹ *truncate-reversed* (*a* # *x*) = *truncate-reversed x* **by** *simp*
  hence ¬ *a* ⟹ *length* (*truncate-reversed* (*a* # *x*)) ≤ *length x*
    using *truncate-reversed-length-ineq*[*of x*] **by** *simp*
  hence ¬ *a* ⟹ *truncate-reversed* (*a* # *x*) ≠ (*a* # *x*)
    using *neq-if-length-neq*[*of a*#*x x*] **by** *force*
  ultimately show *?case* **by** *simp*
qed

lemma *truncated-iff*: *truncated x* ⟷ (*x* = [] ∨ *last x* = *True*)
proof −
  have *truncated x* ⟷ *truncate-reversed* (*rev x*) = *rev x*
    by (*simp add*: *truncate-def rev-swap*)
  also have ... ⟷ *rev x* = [] ∨ *hd* (*rev x*) = *True*
    using *truncate-reversed-fixed-point-iff*[*of rev x*] **.**
  also have ... ⟷ *x* = [] ∨ *last x* = *True*
    by (*simp add*: *hd-rev*)
  finally show *?thesis* **.**
qed

lemma *hd-truncate-reversed*: *truncate-reversed xs* ≠ [] ⟹ *hd* (*truncate-reversed xs*) = *True*
proof (*induction xs*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  show *?case*
  proof (*rule ccontr*)

assume *1*: *hd* (*truncate-reversed* (*a* # *xs*)) ≠ *True*
then have *a* = *False* **by** *auto*
with *1* **have** *hd* (*truncate-reversed xs*) ≠ *True* **by** *simp*
hence *truncate-reversed xs* = [] **using** *Cons.IH* **by** *blast*
hence *truncate-reversed* (*a* # *xs*) = [] **using** ‹*a* = *False*› **by** *simp*
thus *False* **using** *Cons.prems* **by** *simp*
qed
qed

lemma *last-truncate*: *truncate xs* ≠ [] ⟹ *last* (*truncate xs*) = *True*
using *hd-truncate-reversed last-rev* **by** (*auto simp*: *truncate-def*)

lemma *truncate-truncate*[*simp*]: *truncate* (*truncate xs*) = *truncate xs*
using *truncated-iff*[*of truncate xs*] *last-truncate* **by** *auto*

lemma *truncate-reversed-Nil-iff*: *truncate-reversed xs* = [] ⟷ (∃ *n*. *xs* = *replicate n False*)
proof
show *truncate-reversed xs* = [] ⟹ ∃ *n*. *xs* = *replicate n False*
proof (*induction xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then have *a* = *False truncate-reversed* (*a*#*xs*) = *truncate-reversed xs*
by (*auto split*: *if-splits*)
then obtain *n* **where** *xs* = *replicate n False* **using** *Cons* **by** *auto*
hence *a* # *xs* = *replicate* (*Suc n*) *False* **using** ‹*a* = *False*› **by** *simp*
thus *?case* **by** *blast*
qed
next
show ∃ *n*. *xs* = *replicate n False* ⟹ *truncate-reversed xs* = []
proof (*induction xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case*
by (*metis Cons-replicate-eq truncate-reversed.simps*(*2*))
qed
qed

lemma *truncate-Nil-iff*: *truncate xs* = [] ⟷ (∃ *n*. *xs* = *replicate n False*)
using *truncate-reversed-Nil-iff*[*of rev xs*]
by (*auto simp*: *truncate-def*) (*metis rev-replicate rev-rev-ident*)

corollary *truncate-neq-Nil*: *truncate xs* ≠ [] ⟹ ∃ *ys zs*. *xs* = *ys* @ *True* # *zs*
using *truncate-Nil-iff*[*of xs*]

by (*metis* (*full-types*) *hd-Cons-tl hd-truncate-reversed replicate-truncate-reversed truncate-reversed-Nil-iff*)

lemma *truncate-Cons*: *truncate* (*a* # *xs*) = (*if* ¬*a* ∧ (*truncate xs* = []) *then* [] *else* *a* # *truncate xs*)
proof (*cases truncate xs* = [])
  case *True*
  then obtain *n* **where** *xs* = *replicate n False* **using** *truncate-Nil-iff* **by** *blast*
  then have *truncate* (*a* # *xs*) = *truncate* [*a*] **by** *simp*
  then show *?thesis* **using** *True* **by** (*simp add*: *truncate-def*)
next
  case *False*
  then obtain *ys n* **where** *xs* = *ys* @ *True* # (*replicate n False*)
    using *truncate-neq-Nil*[*of xs*] *bit-strong-decomp-2*[*of xs True*] **by** *auto*
  then have *truncate xs* = *ys* @ [*True*] **by** (*auto simp*: *truncate-def*)
  moreover have *truncate* (*a* # *xs*) = *a* # *ys* @ [*True*]
    using ‹*xs* = *ys* @ *True* # (*replicate n False*)› **by** (*auto simp*: *truncate-def*)
  ultimately show *?thesis* **by** *simp*
qed

lemma *truncate-eq-Cons*: *truncate xs* = *truncate ys* ⟹ *truncate* (*a* # *xs*) = *truncate* (*a* # *ys*)
  using *truncate-Cons* **by** *simp*

lemma *truncate-as-take*: ⋀*xs*. ∃ *n*. *truncate xs* = *take n xs*
  using *truncate-replicate append-eq-conv-conj* **by** *blast*

lemma *to-nat-zero-iff*: *to-nat xs* = *0* ⟷ *truncate xs* = []
proof (*induction xs*)
  case *Nil*
  then show *?case* **by** (*simp add*: *truncate-def*)
next
  case (*Cons a xs*)
  have *to-nat* (*a* # *xs*) = *0* ⟷ (*eval-bool a* = *0* ∧ *to-nat xs* = *0*) **by** *simp*
  also have ... ⟷ (*a* = *False* ∧ *to-nat xs* = *0*) **using** *eval-bool-inj*[*of a False*] **by** *auto*
  also have ... ⟷ (*a* = *False* ∧ *truncate xs* = []) **using** *Cons.IH* **by** *simp*
  also have ... ⟷ (*truncate* (*a* # *xs*) = []) **using** *truncate-Cons* **by** *simp*
  finally show *?case* **.**
qed

lemma *to-nat-eq-imp-truncate-eq*: *to-nat xs* = *to-nat ys* ⟹ *truncate xs* = *truncate ys*
proof (*induction xs arbitrary*: *ys*)
  case *Nil*
  then show *?case* **using** *to-nat-zero-iff* **by** (*simp add*: *truncate-def*)
next
  case (*Cons a xs*)
  show *?case*

```
  proof (cases ys = [])
    case True
    then have to-nat ys = 0 by simp
    hence to-nat (a # xs) = 0 using Cons.prems by simp
    with ‹to-nat ys = 0› show truncate (a # xs) = truncate ys
      using to-nat-zero-iff[of a # xs] to-nat-zero-iff[of ys] by simp
  next
    case False
    then obtain b zs where ys = b # zs by (meson neq-Nil-conv)
    then have to-nat (a # xs) = to-nat (b # zs) using Cons.prems by simp
    then have 1: eval-bool a + 2 ∗ to-nat xs = eval-bool b + 2 ∗ to-nat zs by simp
    then have eval-bool a = eval-bool b
     by (metis add-cancel-right-left double-not-eq-Suc-double eval-bool.elims plus-1-eq-Suc)
    hence a = b using eval-bool-inj by simp
    from 1 have to-nat xs = to-nat zs
      using ‹eval-bool a = eval-bool b› by auto
    hence truncate xs = truncate zs using Cons.IH by simp
    hence truncate (a # xs) = truncate (b # zs) using ‹a = b›
      using truncate-eq-Cons[of xs zs a] by simp
    thus ?thesis using ‹ys = b # zs› by simp
  qed
qed

lemma truncate-from-nat[simp]: truncate (from-nat x) = from-nat x
  unfolding truncated-iff
  by (induction x rule: from-nat.induct) auto

lemma truncate-and-length-eq-imp-eq:
  assumes truncate xs = truncate ys length xs = length ys
  shows xs = ys
proof −
  obtain n where 1: xs = truncate xs @ replicate n False
    by (metis truncate-replicate)
  then have 2: length xs = length (truncate xs) + n
    by (metis length-append length-replicate)
  obtain m where 3: ys = truncate ys @ replicate m False
    by (metis truncate-replicate)
  then have length ys = length (truncate ys) + m
    by (metis length-append length-replicate)
  with 2 assms have n = m by simp
  with 1 3 assms show ?thesis by algebra
qed

lemma nat-lsbf-eqI:
  assumes to-nat xs = to-nat ys
  assumes length xs = length ys
  shows xs = ys
  using assms
  using to-nat-eq-imp-truncate-eq truncate-and-length-eq-imp-eq by blast
```

interpretation *nat-lsbf*: *abstract-representation from-nat to-nat truncate*
proof
  show *to-nat* ∘ *from-nat* = *id*
    using *to-nat-from-nat comp-apply* **by** *fastforce*
next
  show *from-nat* ∘ *to-nat* = *truncate*
    using *from-to-f-criterion*[*of to-nat from-nat truncate*]
    using *to-nat-from-nat truncate-from-nat to-nat-eq-imp-truncate-eq*
    using *comp-apply*
    by *fastforce*
qed

lemma *truncated-Cons-imp-truncated-tl*: *truncated* (*x* # *xs*) ⟹ *truncated xs*
  using *truncated-iff* **by** *fastforce*

definition *fill* **where** *fill n xs* = *xs* @ *replicate* (*n* − *length xs*) *False*

lemma *to-nat-fill*[*simp*]: *to-nat* (*fill n xs*) = *to-nat xs*
  by (*simp add*: *fill-def*)

lemma *length-fill*[*intro*]: *length xs* ≤ *n* ⟹ *length* (*fill n xs*) = *n*
  by (*simp add*: *fill-def*)

lemma *take-id*: *length xs* = *k* ⟹ *take k xs* = *xs*
  by *simp*
lemma *fill-id*: *length xs* ≥ *k* ⟹ *fill k xs* = *xs*
  unfolding *fill-def* **by** *simp*

lemma *length-fill'*: *length* (*fill n xs*) = *max n* (*length xs*)
  by (*simp add*: *fill-def*)

lemma *length-fill-max*[*simp*]:
  length (*fill* (*max* (*length xs*) (*length ys*)) *xs*) = *max* (*length xs*) (*length ys*)
  length (*fill* (*max* (*length xs*) (*length ys*)) *ys*) = *max* (*length xs*) (*length ys*)
  by (*intro length-fill*, *simp*)+

lemma *truncate-fill*: *truncate* (*fill k xs*) = *truncate xs*
  by (*simp add*: *fill-def*)

lemma *fill-truncate*: *length xs* ≤ *k* ⟹ *fill k* (*truncate xs*) = *fill k xs*
proof −
  assume *length xs* ≤ *k*
  obtain *n* **where** *n-def*: *xs* = *truncate xs* @ *replicate n False*
    using *truncate-replicate* **by** *metis*
  then have *length xs* = *length* (*truncate xs*) + *n* **by** (*metis length-append length-replicate*)
  then have *length* (*truncate xs*) + *n* ≤ *k* **using** ‹*length xs* ≤ *k*› **by** *simp*

from *n-def* **have** *fill k xs* = (*truncate xs* @ *replicate n False*) @ *replicate* (*k* − *length* (*truncate xs* @ *replicate n False*)) *False*
  using *fill-def* **by** *presburger*
 also have ... = *truncate xs* @ *replicate* (*n* + (*k* − *length* (*truncate xs* @ *replicate n False*))) *False*
  by (*simp add*: *replicate-add*)
 also have ... = *truncate xs* @ *replicate* (*n* + (*k* − (*length* (*truncate xs*) + *n*))) *False*
  by *simp*
 also have ... = *truncate xs* @ *replicate* (*k* − (*length* (*truncate xs*))) *False*
  using ‹*length* (*truncate xs*) + *n* ≤ *k*› **by** *simp*
 also have ... = *fill k* (*truncate xs*) **by** (*simp add*: *fill-def*)
 finally show *?thesis* **by** *simp*
qed

lemma *fill-take-com*: *fill k* (*take k xs*) = *take k* (*fill k xs*)
 using *fill-def* **by** *fastforce*

lemma *to-nat-length-lower-bound-truncated*: *xs* ≠ [] ⟹ *truncated xs* ⟹ *to-nat xs* ≥ *2* ^ (*length xs* − *1*)
proof −
 assume *xs* ≠ [] *truncated xs*
 then obtain *xs'* **where** *xs* = *xs'* @ [*True*]
  by (*metis*(*full-types*) *append-butlast-last-id last-truncate*)
 then show *?thesis* **using** *to-nat-length-lower-bound*[*of xs'*] **by** *simp*
qed

lemma *to-nat-length-bound-truncated*: *truncated xs* ⟹ *to-nat xs* < *2* ^ *n* ⟹ *length xs* ≤ *n*
proof (*rule ccontr*)
 assume *truncated xs to-nat xs* < *2* ^ *n* ¬ *length xs* ≤ *n*
 show *False*
 proof (*cases xs* = [])
  case *True*
  then show *?thesis* **using** ‹¬ *length xs* ≤ *n*› **by** *simp*
 next
  case *False*
  have *length xs* ≥ *n* + *1* **using** ‹¬ *length xs* ≤ *n*› **by** *simp*
  then have *to-nat xs* ≥ *2* ^ *n*
   using *to-nat-length-lower-bound-truncated*[*of xs*]
   using *False* ‹*truncated xs*›
   by (*meson add-le-imp-le-diff dual-order.trans one-le-numeral power-increasing*)
  then show *?thesis* **using** ‹*to-nat xs* < *2* ^ *n*› **by** *simp*
 qed
qed

8.4 Right-shifts

definition *shift-right* :: *nat* ⇒ *nat-lsbf* ⇒ *nat-lsbf* **where**

shift-right n xs = (*replicate n False*) @ *xs*

lemma *to-nat-shift-right*[*simp*]: *to-nat* (*shift-right n xs*) = *2* ^ *n* ∗ *to-nat xs*
unfolding *shift-right-def* **using** *to-nat-app* **by** *simp*

lemma *length-shift-right*[*simp*]: *length* (*shift-right n xs*) = *n* + *length xs*
unfolding *shift-right-def* **by** *simp*

8.5 Subdividing lists

8.5.1 Splitting a list in two blocks

fun *split-at* :: *nat* ⇒ *'a list* ⇒ *'a list* × *'a list* **where**
split-at m xs = (*take m xs*, *drop m xs*)

definition *split* :: *nat-lsbf* ⇒ *nat-lsbf* × *nat-lsbf* **where**
split xs = (*let n* = *length xs div* (*2*::*nat*) *in split-at n xs*)

lemma *app-split*: *split xs* = (*x0*, *x1*) ⟹ *xs* = *x0* @ *x1*
unfolding *split-def Let-def* **using** *append-take-drop-id*[*of length xs div 2 xs*] **by** *simp*

lemma *length-split*: *length xs mod 2* = *0* ⟹ *split xs* = (*x0*, *x1*) ⟹ *length x0* = *length xs div 2* ∧ *length x1* = *length xs div 2*
unfolding *split-def* **by** *fastforce*

lemma *length-split-le*:
assumes *split xs* = (*x0*, *x1*)
shows *length x0* ≤ *length xs* **and** *length x1* ≤ *length xs*
using *app-split*[*OF assms*] **by** *simp-all*

8.5.2 Splitting a list in multiple blocks

subdivide n xs divides the list *xs* into blocks of size *n*.

fun *subdivide* :: *nat* ⇒ *'a list* ⇒ *'a list list* **where**
subdivide 0 xs = *undefined*
| *subdivide n* [] = []
| *subdivide n xs* = *take n xs* # *subdivide n* (*drop n xs*)

value *concat* [[*0*..<*2*], [*4*..<*7*], [*1*..<*5*]]

value *subdivide 2* [*0*..<*6*]
value *subdivide 3* [*0*..<*6*]
value *subdivide* (*2* ^ *2*) [*0*..<*2* ^ *6*]

lemma *concat-subdivide*: *n* > *0* ⟹ *concat* (*subdivide n xs*) = *xs*
by (*induction n xs rule*: *subdivide.induct*) *simp-all*

lemma *subdivide-step*:

assumes *n* > *0*
assumes *xs* ≠ []
assumes *length xs* = *n* ∗ *k*
obtains *ys zs* **where** *xs* = *ys* @ *zs length ys* = *n length zs* = *n* ∗ (*k* − *1*)
subdivide n xs = *ys* # *subdivide n zs*
proof −
from *assms* **obtain** *a xs'* **where** *xs* = *a* # *xs'* **using** *list.exhaust* **by** *blast*
from *assms* **have** *k* > *0*
using *zero-less-iff-neq-zero* **by** *fastforce*
then obtain *k'* **where** *k* = *Suc k'* **using** *gr0-implies-Suc* **by** *auto*
then have *length xs* = *n* + *n* ∗ *k'* **using** *assms*(*3*) **by** *simp*
define *ys zs* **where** *ys* = *take n xs zs* = *drop n xs*
with ‹*length xs* = *n* + *n* ∗ *k'*› **have** *xs* = *ys* @ *zs length ys* = *n length zs* = *n* ∗ *k'* **by** *simp-all*
moreover have *subdivide n xs* = *ys* # *subdivide n zs* **using** *ys-zs-def assms*(*1*) *assms*(*2*) *Suc-diff-1 subdivide.simps*(*3*)
‹*xs* = *a* # *xs'*› **by** *metis*
ultimately show (⋀*ys zs*.
xs = *ys* @ *zs* ⟹
length ys = *n* ⟹
length zs = *n* ∗ (*k* − *1*) ⟹
subdivide n xs = *ys* # *subdivide n zs* ⟹ *thesis*) ⟹
thesis
by (*simp add*: ‹*k* = *Suc k'*›)
qed

lemma *subdivide-step'*:
assumes *n* > *0*
assumes *xs* ≠ []
shows *subdivide n xs* = (*take n xs*) # *subdivide n* (*drop n xs*)
using *assms*
by (*cases n*; *cases xs*; *simp-all*)

lemma *subdivide-correct*:
assumes *n* > *0*
assumes *length xs* = *n* ∗ *k*
shows *length* (*subdivide n xs*) = *k* ∧ (*x* ∈ *set* (*subdivide n xs*) ⟶ *length x* = *n*)
using *assms*
proof (*induction k arbitrary*: *xs n x*)
case *0*
then have *subdivide n xs* = [] **using** *0 gr0-conv-Suc* **by** *force*
then show *?case* **by** *simp*
next
case (*Suc k*)
then have *xs* ≠ [] **by** *force*
from *subdivide-step*[*OF* ‹*n* > *0*› *this* ‹*length xs* = *n* ∗ *Suc k*›] **obtain** *ys zs* **where** *ys-zs*:
xs = *ys* @ *zs*
length ys = *n*

```
    length zs = n * (Suc k - 1)
    subdivide n xs = ys # subdivide n zs
    by blast
  then have length zs = n * k by simp
  note IH = Suc.IH[OF ‹n > 0› this]
  from IH show ?case using ys-zs by simp
qed

lemma nth-nth-subdivide:
  assumes n > 0
  assumes length xs = n * k
  assumes i < k j < n
  shows subdivide n xs ! i ! j = xs ! (i * n + j)
  using assms
proof (induction k arbitrary: xs i)
  case 0
  then show ?case by simp
next
  case (Suc k)
  then have xs ≠ [] by auto
  with Suc subdivide-step obtain ys zs where xs = ys @ zs length ys = n length
zs = n * (Suc k - 1)
    subdivide n xs = ys # subdivide n zs by blast
  then have length zs = n * k by simp
  show ?case
  proof (cases i)
    case 0
    then have subdivide n xs ! i ! j = ys ! (i * n + j) using ‹subdivide n xs = ys
# subdivide n zs› by simp
    then show ?thesis using ‹xs = ys @ zs› 0 ‹j < n› ‹length ys = n›
      by (simp add: nth-append)
  next
    case (Suc i')
    then have subdivide n xs ! i ! j = subdivide n zs ! i' ! j
      using ‹subdivide n xs = ys # subdivide n zs› by simp
    also have ... = zs ! (i' * n + j)
      apply (intro Suc.IH[of zs i'])
      subgoal using ‹n > 0› .
      subgoal using ‹length zs = n * k› .
      subgoal using ‹i < Suc k› ‹i = Suc i'› by simp
      subgoal using ‹j < n› .
      done
    also have ... = xs ! (i * n + j)
      using ‹i = Suc i'› ‹xs = ys @ zs› ‹length ys = n›
      by (metis ab-semigroup-add-class.add-ac(1) mult-Suc nth-append-length-plus)
    finally show ?thesis .
  qed
qed
```

lemma *subdivide-concat*:
assumes *n* > *0*
assumes ⋀*i*. *i* < *length xs* ⟹ *length* (*xs* ! *i*) = *n*
shows *subdivide n* (*concat xs*) = *xs*
proof (*intro iffD1*[*OF concat-eq-concat-iff*])
show *concat* (*subdivide n* (*concat xs*)) = *concat xs*
using *concat-subdivide*[*OF* ‹*n* > *0*›] **.**
have *map length xs* = *replicate* (*length xs*) *n*
apply (*intro replicate-eqI*)
subgoal by *simp*
subgoal using *assms* **by** (*metis in-set-conv-nth length-map nth-map*)
done
then have *length* (*concat xs*) = *length xs* ∗ *n*
by (*simp add*: *length-concat sum-list-replicate*)
then show *length* (*subdivide n* (*concat xs*)) = *length xs*
apply (*intro conjunct1*[*OF subdivide-correct*] ‹*n* > *0*›) **by** *simp*
show ∀(*x*, *y*)∈*set* (*zip* (*subdivide n* (*concat xs*)) *xs*). *length x* = *length y*
proof
fix *z*
assume *a*: *z* ∈ *set* (*zip* (*subdivide n* (*concat xs*)) *xs*)
then obtain *x y* **where** *z* = (*x*, *y*) **by** *fastforce*
from *a* **obtain** *i* **where** *i* < *length xs z* = *zip* (*subdivide n* (*concat xs*)) *xs* ! *i*
using ‹*length* (*subdivide n* (*concat xs*)) = *length xs*›
by (*metis* (*no-types*, *lifting*) *gen-length-def in-set-conv-nth length-code length-zip min-0R min-add-distrib-left*)
then have *subdivide n* (*concat xs*) ! *i* = *x xs* ! *i* = *y*
using ‹*z* = (*x*, *y*)› ‹*length* (*subdivide n* (*concat xs*)) = *length xs*› **by** *simp-all*
then have *length x* = *n* **using** ‹*i* < *length xs*› ‹*length* (*subdivide n* (*concat xs*)) = *length xs*›
using ‹*length* (*concat xs*) = *length xs* ∗ *n*›
‹*n* > *0*› *mult.commute*[*of n length xs*]
by (*metis nth-mem subdivide-correct*)
moreover from ‹*xs* ! *i* = *y*› ‹*i* < *length xs*› **have** *length y* = *n* **using** *assms* **by** *blast*
ultimately show *case z of* (*x*, *y*) ⇒ *length x* = *length y* **using** ‹*z* = (*x*, *y*)› **by** *simp*
qed
qed

lemma *to-nat-subdivide*:
assumes *n* > *0*
assumes *length xs* = *n* ∗ *k*
shows *to-nat xs* = (∑ *i* ← [*0*..<*k*]. *to-nat* (*subdivide n xs* ! *i*) ∗ *2* ^ (*i* ∗ *n*))
using *assms*
proof (*induction k arbitrary*: *xs*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc k*)

then have *length* (*take n xs*) = *n length* (*drop n xs*) = *n* ∗ *k* **by** *simp-all*
from *Suc* **have** *xs* ≠ [] **by** *auto*
have (∑ *i* ← [*0*..<*Suc k*]. *to-nat* (*subdivide n xs* ! *i*) ∗ *2* ^ (*i* ∗ *n*))
= *to-nat* (*subdivide n xs* ! *0*) ∗ *2* ^ (*0* ∗ *n*) + (∑ *i* ← [*1*..<*Suc k*]. *to-nat* (*subdivide n xs* ! *i*) ∗ *2* ^ (*i* ∗ *n*))
by (*intro sum-list-split-0*)
also have *subdivide n xs* ! *0* = *take n xs*
using *Suc* ‹*xs* ≠ []› *subdivide-step*′[*OF* ‹*0* < *n*› ‹*xs* ≠ []›] **by** *simp*
also have (∑ *i* ← [*1*..<*Suc k*]. *to-nat* (*subdivide n xs* ! *i*) ∗ *2* ^ (*i* ∗ *n*))
= (∑ *i* ← [*0*..<*k*]. *to-nat* (*subdivide n xs* ! (*i* + *1*)) ∗ *2* ^ ((*i* + *1*) ∗ *n*))
using *sum-list-index-shift*[*of* λ*i*. *to-nat* (*subdivide n xs* ! *i*) ∗ *2* ^ (*i* ∗ *n*) *1 0 k*]
by *simp*
also have ... = (∑ *i* ← [*0*..<*k*]. *to-nat* (*subdivide n* (*drop n xs*) ! *i*) ∗ *2* ^ ((*i* + *1*) ∗ *n*))
using *subdivide-step*′[*OF* ‹*0* < *n*› ‹*xs* ≠ []›] **by** *simp*
also have ... = (∑ *i* ← [*0*..<*k*]. (*to-nat* (*subdivide n* (*drop n xs*) ! *i*) ∗ (*2* ^ *n* ∗ *2* ^ (*i* ∗ *n*))))
by (*simp add*: *power-add*)
also have ... = (∑ *i* ← [*0*..<*k*]. *2* ^ *n* ∗ (*to-nat* (*subdivide n* (*drop n xs*) ! *i*) ∗ *2* ^ (*i* ∗ *n*)))
by (*simp add*: *mult.left-commute*)
also have ... = *2* ^ *n* ∗ (∑ *i* ← [*0*..<*k*]. *to-nat* (*subdivide n* (*drop n xs*) ! *i*) ∗ *2* ^ (*i* ∗ *n*))
by (*simp add*: *sum-list-const-mult*)
also have ... = *2* ^ *n* ∗ *to-nat* (*drop n xs*)
using *Suc.IH*[*OF* ‹*0* < *n*› ‹*length* (*drop n xs*) = *n* ∗ *k*›] **by** *argo*
finally have (∑ *i* ← [*0*..<*Suc k*]. *to-nat* (*subdivide n xs* ! *i*) ∗ *2* ^ (*i* ∗ *n*))
= *to-nat* (*take n xs*) + *2* ^ *n* ∗ *to-nat* (*drop n xs*)
by *simp*
also have ... = *to-nat* (*take n xs* @ *drop n xs*)
by (*simp only*: *to-nat-app* ‹*length* (*take n xs*) = *n*›)
also have ... = *to-nat xs* **by** *simp*
finally show *to-nat xs* = (∑ *i* ← [*0*..<*Suc k*]. *to-nat* (*subdivide n xs* ! *i*) ∗ *2* ^ (*i* ∗ *n*))
by *simp*
qed

8.6 The *bitsize* function

bitsize n calculates how many bits are needed in the LSBF encoding of *n*.

fun *bitsize* :: *nat* ⇒ *nat* **where**
bitsize 0 = *0*
| *bitsize n* = *1* + *bitsize* (*n div 2*)

lemma *bitsize-is-floorlog*: *bitsize* = *floorlog 2*
apply (*intro ext*)
subgoal for *n*
apply (*induction n rule*: *bitsize.induct*)
by (*auto simp add*: *floorlog-eq-zero-iff compute-floorlog*)

```
  done

corollary bitsize-bitlen: int (bitsize n) = bitlen (int n)
  unfolding bitsize-is-floorlog bitlen-def by simp

lemma bitsize-eq: bitsize n = length (from-nat n)
proof (induction n rule: less-induct)
  case (less n)
  then show ?case
  proof (cases n = 0)
    case True
    then show ?thesis by simp
  next
    case False
    then have 1: bitsize n = 1 + bitsize (n div 2)
      by (metis bitsize.elims)
    from False have length (from-nat n) = length ((if n mod 2 = 0 then False else
True) # from-nat (n div 2))
      by (metis from-nat.elims)
    also have ... = 1 + bitsize (n div 2) using less[of n div 2] False by simp
    finally show bitsize n = length (from-nat n) using 1 by simp
  qed
qed

lemma bitsize-zero-iff: bitsize n = 0 ⟷ n = 0
  by (simp add: bitsize-is-floorlog floorlog-eq-zero-iff)

lemma truncated-iff': truncated x ⟷ length x = bitsize (to-nat x)
proof
  assume truncated x
  then have x = from-nat (to-nat x) unfolding nat-lsbf.f-fixed-point-iff' .
  then show length x = bitsize (to-nat x) unfolding bitsize-eq by simp
next
  assume length x = bitsize (to-nat x)
  then have length x = length (from-nat (to-nat x)) unfolding bitsize-eq .
  moreover have to-nat x = to-nat (from-nat (to-nat x)) by simp
  ultimately show truncated x unfolding nat-lsbf.f-fixed-point-iff'
    by (intro nat-lsbf-eqI; argo)
qed

lemma bitsize-length: bitsize n ≤ k ⟷ n < 2 ^ k
  unfolding bitsize-is-floorlog floorlog-le-iff by simp

lemma two-pow-bitsize-pos-bound: n > 0 ⟹ 2 ^ bitsize n ≤ 2 * n
proof -
  assume n > 0
  then have 2 ^(bitsize n - 1) ≤ n
    using bitsize-length[of n bitsize n - 1] by fastforce
  then have 2 ^(bitsize n - 1 + 1) ≤ 2 * n by simp
```

also have *bitsize n* − *1* + *1* = *bitsize n* **using** *bitsize-zero-iff* [*of n*] ‹*n* > *0*› **by** *simp*
finally show *?thesis* **.**
qed

lemma *two-pow-bitsize-bound*: *2* ^ *bitsize n* ≤ *2* ∗ *n* + *1*
using *two-pow-bitsize-pos-bound*[*of n*] **by** (*cases n*) *simp-all*

lemma *bitsize-mono*: *n1* ≤ *n2* ⟹ *bitsize n1* ≤ *bitsize n2*
unfolding *bitsize-is-floorlog* **by** (*rule floorlog-mono*)

8.6.1 The *next-power-of-2* function

lemma *power-of-2-recursion*: (∃ *k*. (*n*::*nat*) = *2* ^ *k*) ⟷ (*n* = *1* ∨ (*n mod 2* = *0* ∧ (∃ *k*. *n div 2* = *2* ^ *k*)))
proof
assume ∃ *k*. *n* = *2* ^ *k*
then obtain *k* **where** *k-def*: *n* = *2* ^ *k* **by** *blast*
show *n* = *1* ∨ (*n mod 2* = *0* ∧ (∃ *k*. *n div 2* = *2* ^ *k*))
using *k-def* **by** (*cases k*) *simp-all*
next
assume *n* = *1* ∨ (*n mod 2* = *0* ∧ (∃ *k*. *n div 2* = *2* ^ *k*))
then consider *n* = *1* | *n mod 2* = *0* ∧ (∃ *k*. *n div 2* = *2* ^ *k*) **by** *argo*
then show ∃ *k*. *n* = *2* ^ *k*
proof *cases*
case *1*
then have *n* = *2* ^ *0* **by** *simp*
then show *?thesis* **by** *blast*
next
case *2*
then obtain *k* **where** *n div 2* = *2* ^ *k* **by** *blast*
with *2* **have** *n* = *2* ^ *Suc k* **by** *auto*
then show *?thesis* **by** *blast*
qed
qed

fun *is-power-of-2* :: *nat* ⇒ *bool* **where**
is-power-of-2 0 = *False*
| *is-power-of-2* (*Suc 0*) = *True*
| *is-power-of-2 n* = ((*n mod 2* = *0*) ∧ *is-power-of-2* (*n div 2*))

lemma *is-power-of-2-correct*: *is-power-of-2 n* ⟷ (∃ *k*. *n* = *2* ^ *k*)
proof (*induction n rule*: *is-power-of-2.induct*)
case *1*
then show *?case* **by** *simp*
next
case *2*
then show *?case* **by** (*metis is-power-of-2.simps*(*2*) *nat-power-eq-Suc-0-iff*)
next

```
  case (3 va)
  let ?n = Suc (Suc va)
  have is-power-of-2 ?n = ((?n mod 2 = 0) ∧ is-power-of-2 (?n div 2))
    by simp
  also have ... = ((?n mod 2 = 0) ∧ (∃ k. (?n div 2) = 2 ^ k))
    using 3 by argo
  also have ... = (∃ k. ?n = 2 ^ k)
    using power-of-2-recursion[of ?n] by simp
  finally show ?case .
qed


fun next-power-of-2 :: nat ⇒ nat where
next-power-of-2 n = (if is-power-of-2 n then n else 2 ^ (bitsize n))

lemma next-power-of-2-lower-bound: next-power-of-2 k ≥ k
  apply (cases is-power-of-2 k)
  subgoal by simp
  subgoal premises prems
  proof −
    from prems have next-power-of-2 k − 1 = 2 ^ bitsize k − 1 by simp
    also have ... = 2 ^ (length (from-nat k)) − 1 using bitsize-eq by simp
    also have ... ≥ k using to-nat-length-upper-bound[of from-nat k] by simp
    finally show ?thesis by simp
  qed
  done

lemma next-power-of-2-upper-bound:
  assumes k ≠ 0
  shows next-power-of-2 k ≤ 2 ∗ k
  apply (cases is-power-of-2 k)
  subgoal by simp
  subgoal premises prems
  proof −
    have 2 ^ (length (from-nat k) − 1) ≤ to-nat (from-nat k)
      apply (intro to-nat-length-lower-bound-truncated)
      subgoal using assms by (cases k; simp)
      subgoal by simp
      done
    then have 2 ^ length (from-nat k) ≤ 2 ∗ to-nat (from-nat k)
      using assms by (cases k; simp)
    also have ... = 2 ∗ k by simp
    also have 2 ^ length (from-nat k) = next-power-of-2 k
      using prems bitsize-eq by simp
    finally show ?thesis .
  qed
  done

lemma next-power-of-2-upper-bound': next-power-of-2 k ≤ 2 ∗ k + 1
```

apply (*cases k*)
subgoal by *simp*
subgoal using *next-power-of-2-upper-bound*[*of k*] **by** *simp*
done

lemma *next-power-of-2-is-power-of-2*: $\exists k.$ *next-power-of-2 n* = *2* ^ *k*
using *is-power-of-2-correct* **by** *simp*

8.7 Addition

fun *bit-add-carry* :: *bool* ⇒ *bool* ⇒ *bool* ⇒ *bool* × *bool* **where**
bit-add-carry False False False = (*False*, *False*)
| *bit-add-carry False False True* = (*True*, *False*)
| *bit-add-carry False True False* = (*True*, *False*)
| *bit-add-carry False True True* = (*False*, *True*)
| *bit-add-carry True False False* = (*True*, *False*)
| *bit-add-carry True False True* = (*False*, *True*)
| *bit-add-carry True True False* = (*False*, *True*)
| *bit-add-carry True True True* = (*True*, *True*)

lemma *bit-add-carry-correct*: *bit-add-carry c x y* = (*a*, *b*) ⟹ *eval-bool c* + *eval-bool x* + *eval-bool y* = *eval-bool a* + *2* ∗ *eval-bool b*
by (*cases c*; *cases x*; *cases y*) *auto*

8.7.1 Increment operation

fun *inc-nat* :: *nat-lsbf* ⇒ *nat-lsbf* **where**
inc-nat [] = [*True*]
| *inc-nat* (*False* # *xs*) = *True* # *xs*
| *inc-nat* (*True* # *xs*) = *False* # (*inc-nat xs*)

lemma *length-inc-nat'*: *length* (*inc-nat xs*) = *length xs* + *of-bool* (*to-nat xs* + *1* ≥ *2* ^ *length xs*)
proof (*induction xs rule*: *inc-nat.induct*)
case *1*
then show *?case* **by** *simp*
next
case (*2 xs*)
then show *?case* **using** *to-nat-length-bound*[*of xs*] **by** *simp*
next
case (*3 xs*)
then show *?case* **by** *simp*
qed

lemma *length-inc-nat-lower*: *length* (*inc-nat xs*) ≥ *length xs*
unfolding *length-inc-nat'* **by** *simp*

lemma *length-inc-nat-upper*: *length* (*inc-nat xs*) ≤ *length xs* + *1*
unfolding *length-inc-nat'* **by** *simp*

lemma *inc-nat-nonempty*: *inc-nat xs* ≠ []
  by (*induction xs rule*: *inc-nat.induct*) *simp-all*

lemma *inc-nat-replicate-True*: *inc-nat* (*replicate m True*) = *replicate m False* @ [*True*]
  by (*induction m*) *simp-all*

lemma *inc-nat-replicate-True-2*: *inc-nat* (*replicate m True* @ *False* # *ys*) = *replicate m False* @ *True* # *ys*
  by (*induction m*) *simp-all*

lemma *length-inc-nat-iff*: *length* (*inc-nat xs*) = *length xs* ⟷ (∃ *ys zs*. *xs* = *ys* @ *False* # *zs*)
proof (*intro iffI*, *rule ccontr*)
  assume ∄ *ys zs*. *xs* = *ys* @ *False* # *zs*
  then have ∀ *i* ∈ {*0*..<*length xs*}. *xs*!*i* = *True*
    by (*metis*(*full-types*) *atLeastLessThan-iff in-set-conv-nth split-list*)
  then have *xs* = *replicate* (*length xs*) *True*
    by (*simp only*: *list-is-replicate-iff*)
  then show *length* (*inc-nat xs*) = *length xs* ⟹ *False*
    using *inc-nat-replicate-True*
    by (*metis length-append-singleton length-replicate n-not-Suc-n*)
next
  assume ∃ *ys zs*. *xs* = *ys* @ *False* # *zs*
  then have ∃ *n zs′*. *xs* = *replicate n True* @ *False* # *zs′*
    using *bit-strong-decomp-1* **by** *fastforce*
  then show *length* (*inc-nat xs*) = *length xs*
    using *inc-nat-replicate-True-2* **by** *fastforce*
qed

lemma *inc-nat-last-bit-True*: *length* (*inc-nat xs*) = *Suc* (*length xs*) ⟹ ∃ *zs*. *inc-nat xs* = *zs* @ [*True*]
  by (*induction xs rule*: *inc-nat.induct*) *auto*

lemma *inc-nat-truncated*: *truncated xs* ⟹ *truncated* (*inc-nat xs*)
proof (*induction xs rule*: *inc-nat.induct*)
  case *1*
  then show *?case* **using** *truncate-def* **by** *simp*
next
  case (*2 xs*)
  then show *?case* **by** (*simp add*: *truncated-iff*)
next
  case (*3 xs*)
  then show *?case* **by** (*simp add*: *truncated-iff inc-nat-nonempty split*: *if-splits*)
qed

lemma *inc-nat-correct*: *to-nat* (*inc-nat xs*) = *to-nat xs* + *1*
  by (*induction xs rule*: *inc-nat.induct*) *simp-all*

lemma *length-inc-nat*: *length* (*inc-nat xs*) = *max* (*length xs*) (*floorlog 2* (*to-nat xs* + *1*))
proof (*induction xs rule*: *inc-nat.induct*)
  case *1*
  then show *?case* **by** (*simp add*: *compute-floorlog*)
next
  case (*2 xs*)
  then show *?case* **using** *to-nat-length-bound*[*of False # xs*]
    by (*simp add*: *floorlog-leI*)
next
  case (*3 xs*)
  then have *length* (*inc-nat* (*True # xs*)) = *Suc* (*max* (*length xs*) (*floorlog 2* (*Suc* (*to-nat xs*))))
    by *simp*
  also have ... = *max* (*length* (*True # xs*)) (*Suc* (*floorlog 2* (*Suc* (*to-nat xs*))))
    by *simp*
  also have ... = *max* (*length* (*True # xs*)) (*floorlog 2* (*2* ∗ *Suc* (*to-nat xs*)))
    apply (*intro arg-cong2*[**where** *f* = *max*] *refl*)
    by (*simp add*: *compute-floorlog*)
  finally show *?case* **by** *simp*
qed

8.7.2 Addition with a carry bit

fun *add-carry* :: *bool* ⇒ *nat-lsbf* ⇒ *nat-lsbf* ⇒ *nat-lsbf* **where**
add-carry False [] *y* = *y*
| *add-carry False x* [] = *x*
| *add-carry True* [] *y* = *inc-nat y*
| *add-carry True x* [] = *inc-nat x*
| *add-carry c* (*x#xs*) (*y#ys*) = (*let* (*a*, *b*) = *bit-add-carry c x y in a#*(*add-carry b xs ys*))

lemma *add-carry-correct*: *to-nat* (*add-carry c x y*) = *eval-bool c* + *to-nat x* + *to-nat y*
proof (*induction c x y rule*: *add-carry.induct*)
  case (*1 y*)
  then show *?case* **by** *simp*
next
  case (*2 v va*)
  then show *?case* **by** *simp*
next
  case (*3 y*)
  then show *?case* **using** *inc-nat-correct* **by** *simp*
next
  case (*4 v va*)
  then show *?case* **using** *inc-nat-correct* **by** *simp*
next
  case (*5 c x xs y ys*)
  define *a b* **where** *a* = *fst* (*bit-add-carry c x y*) *b* = *snd* (*bit-add-carry c x y*)

```
  then have to-nat (add-carry c (x#xs) (y#ys)) = to-nat (a # add-carry b xs ys)
    by (simp add: case-prod-beta' Let-def)
  also have ... = eval-bool a + 2 * to-nat (add-carry b xs ys) by simp
  also have ... = eval-bool a + 2 * (eval-bool b + to-nat xs + to-nat ys)
    using 5 a-b-def prod.collapse[of bit-add-carry c x y] by algebra
  also have ... = eval-bool c + eval-bool x + eval-bool y + 2 * (to-nat xs + to-nat
ys)
    using bit-add-carry-correct a-b-def by (simp add: prod-eq-iff)
  also have ... = eval-bool c + to-nat (x#xs) + to-nat (y#ys) by simp
  finally show ?case .
qed

lemma length-add-carry': length (add-carry c xs ys) = max (length xs) (length ys)
+ of-bool (to-nat xs + to-nat ys + of-bool c ≥ 2 ^ max (length xs) (length ys))
proof (induction c xs ys rule: add-carry.induct)
  case (1 y)
  then show ?case using to-nat-length-bound[of y] by simp
next
  case (2 v va)
  then show ?case
    using to-nat-length-bound[of va] by simp
next
  case (3 y)
  then show ?case by (simp add: length-inc-nat')
next
  case (4 v va)
  then show ?case by (simp add: length-inc-nat')
next
  case (5 c x xs y ys)

  have l: 2 ^ Suc a ≤ 2 * b + 1 ⟷ 2 ^ Suc a ≤ 2 * b for a b :: nat
    by fastforce

  obtain a b where bit-add-carry c x y = (a, b) by fastforce
  then have add-carry c (x # xs) (y # ys) = a # (add-carry b xs ys) by simp
  then have length (add-carry c (x # xs) (y # ys)) = 1 + max (length xs) (length
ys) + of-bool (2 ^ max (length xs) (length ys) ≤ to-nat xs + to-nat ys + of-bool b)
    using 5.IH[OF ‹bit-add-carry c x y = (a, b)›[symmetric] refl] by (simp only:
length-Cons)
  also have ... = max (length (x # xs)) (length (y # ys)) + of-bool (2 ^ max
(length xs) (length ys) ≤ to-nat xs + to-nat ys + of-bool b)
    by simp
  also have ... = max (length (x # xs)) (length (y # ys)) + of-bool (2 ^ max
(length (x # xs)) (length (y # ys)) ≤ to-nat (x # xs) + to-nat (y # ys) + of-bool
c)
  proof (intro arg-cong2[where f = (+)] refl arg-cong[where f = of-bool])
    have to-nat (x # xs) + to-nat (y # ys) + of-bool c =
        2 * to-nat xs + 2 * to-nat ys + of-bool x + of-bool y + of-bool c
      by simp
```

also have ... = *2* ∗ *to-nat xs* + *2* ∗ *to-nat ys* + *of-bool a* + *2* ∗ *of-bool b*
using *bit-add-carry-correct*[*OF* ‹*bit-add-carry c x y* = (*a*, *b*)›] **by** *simp*
finally have *r*: *to-nat* (*x* # *xs*) + *to-nat* (*y* # *ys*) + *of-bool c* = ... **.**
show (*2* ^ *max* (*length xs*) (*length ys*) ≤ *to-nat xs* + *to-nat ys* + *of-bool b*) =
(*2* ^ *max* (*length* (*x* # *xs*)) (*length* (*y* # *ys*)) ≤ *to-nat* (*x* # *xs*) + *to-nat* (*y* # *ys*) + *of-bool c*)
unfolding *r* **using** *l*[*of max* (*length xs*) (*length ys*) *to-nat xs* + *to-nat ys* + *of-bool b*]
by *auto*
qed
finally show *?case* **.**
qed

lemma *length-add-carry*: *length* (*add-carry c xs ys*) = *max* (*max* (*length xs*) (*length ys*)) (*floorlog 2* (*of-bool c* + *to-nat xs* + *to-nat ys*))
proof (*induction c xs ys rule*: *add-carry.induct*)
case (*1 y*)
then show *?case* **using** *to-nat-length-bound*[*of y*]
by (*simp add*: *floorlog-leI*)
next
case (*2 v va*)
then show *?case* **using** *to-nat-length-bound*[*of v* # *va*]
by (*simp add*: *floorlog-leI*)
next
case (*3 y*)
then show *?case* **by** (*simp add*: *length-inc-nat*)
next
case (*4 v va*)
then show *?case* **by** (*simp add*: *length-inc-nat*)
next
case (*5 c x xs y ys*)
obtain *a b* **where** *bit-add-carry c x y* = (*a*, *b*) **by** *fastforce*
then have *add-carry c* (*x* # *xs*) (*y* # *ys*) = *a* # (*add-carry b xs ys*) **by** *simp*
then have *length* (*add-carry c* (*x* # *xs*) (*y* # *ys*)) = *Suc* (*max* (*max* (*length xs*) (*length ys*)) (*floorlog 2* (*of-bool b* + *to-nat xs* + *to-nat ys*)))
using *5* ‹*bit-add-carry c x y* = (*a*, *b*)› **by** (*simp only*: *length-Cons*)
also have ... = *max* (*max* (*length* (*x* # *xs*)) (*length* (*y* # *ys*))) (*1* + *floorlog 2* (*of-bool b* + *to-nat xs* + *to-nat ys*))
by *simp*
also have ... = *max* (*max* (*length* (*x* # *xs*)) (*length* (*y* # *ys*))) (*floorlog 2* (*of-bool c* + *to-nat* (*x* # *xs*) + *to-nat* (*y* # *ys*)))
proof (*cases of-bool a* + *2* ∗ (*of-bool b* + *to-nat xs* + *to-nat ys*) > *0*)
case *True*
then show *?thesis*
proof (*intro arg-cong2*[**where** *f* = *max*] *refl*)
have *floorlog 2* (*of-bool c* + *to-nat* (*x* # *xs*) + *to-nat* (*y* # *ys*)) =
floorlog 2 ((*of-bool c* + *of-bool x* + *of-bool y*) + *2* ∗ (*to-nat xs* + *to-nat ys*))
by *simp*

also have ... = *floorlog 2* ((*of-bool a* + *2* ∗ *of-bool b*) + *2* ∗ (*to-nat xs* + *to-nat ys*))
using *bit-add-carry-correct*[*OF* ‹*bit-add-carry c x y* = (*a*, *b*)›] **by** *simp*
also have ... = *floorlog 2* (*of-bool a* + *2* ∗ (*of-bool b* + *to-nat xs* + *to-nat ys*))
by *simp*
also have ... = *1* + *floorlog 2* (*of-bool b* + *to-nat xs* + *to-nat ys*)
using *compute-floorlog*[*of 2 of-bool a* + *2* ∗ (*of-bool b* + *to-nat xs* + *to-nat ys*)] *True*
by *simp*
finally show ... = *floorlog 2* (*of-bool c* + *to-nat* (*x* # *xs*) + *to-nat* (*y* # *ys*)) **by** *simp*
qed
next
case *False*
then have *01*: *of-bool a* = *0* *of-bool b* = *0* *to-nat xs* = *0* *to-nat ys* = *0* **by** *simp-all*
then have *02*: *of-bool c* = *0* *of-bool x* = *0* *of-bool y* = *0*
using *bit-add-carry-correct*[*OF* ‹*bit-add-carry c x y* = (*a*, *b*)›] **by** *simp-all*
from *01 02* **show** *?thesis* **by** (*simp add*: *floorlog-def*)
qed
finally show *?case* **.**
qed

lemma *length-add-carry-lower*: *length* (*add-carry c xs ys*) ≥ *max* (*length xs*) (*length ys*)
unfolding *length-add-carry′* **by** *simp*

lemma *length-add-carry-upper*: *length* (*add-carry c xs ys*) ≤ *max* (*length xs*) (*length ys*) + *1*
unfolding *length-add-carry′* **by** *simp*

lemma *add-carry-last-bit-True*: *length* (*add-carry c xs ys*) = *max* (*length xs*) (*length ys*) + *1* ⟹ ∃ *zs*. *add-carry c xs ys* = *zs* @ [*True*]
proof (*induction c xs ys rule*: *add-carry.induct*)
case (*1 y*)
then show *?case* **by** *simp*
next
case (*2 v va*)
then show *?case* **by** *simp*
next
case (*3 y*)
then show *?case* **by** (*simp add*: *inc-nat-last-bit-True*)
next
case (*4 v va*)
then show *?case* **by** (*simp add*: *inc-nat-last-bit-True*)
next
case (*5 c x xs y ys*)
obtain *a b* **where** *bit-add-carry c x y* = (*a*, *b*) **by** *fastforce*
then have *1*: *add-carry c* (*x* # *xs*) (*y* # *ys*) = *a* # (*add-carry b xs ys*)

by *simp*
from *5* **have** *length* (*add-carry b xs ys*) = *max* (*length* (*x* # *xs*)) (*length* (*y* # *ys*))
using ‹*bit-add-carry c x y* = (*a*, *b*)› **by** *auto*
also have ... = *max* (*length xs*) (*length ys*) + *1* **by** *simp*
finally obtain *zs* **where** *add-carry b xs ys* = *zs* @ [*True*] **using** *5* ‹*bit-add-carry c x y* = (*a*, *b*)›
by *presburger*
then show *?case* **using** *1* **by** *simp*
qed

lemma *add-carry-com*: *add-carry c xs ys* = *add-carry c ys xs*
apply (*intro nat-lsbf-eqI*)
subgoal by (*simp add*: *add-carry-correct*)
subgoal by (*simp only*: *length-add-carry′ max.commute add.commute*)
done

lemma *add-carry-rNil*[*simp*]: *add-carry True y* [] = *inc-nat y*
by (*cases y*; *simp*)
lemma *add-carry-rNil-nocarry*[*simp*]: *add-carry False y* [] = *y*
by (*cases y*; *simp*)

lemma *add-carry-True-inc-nat*:
add-carry True xs ys = *inc-nat* (*add-carry False xs ys*) ∧
add-carry True xs ys = *add-carry False* (*inc-nat xs*) *ys* ∧
add-carry True xs ys = *add-carry False xs* (*inc-nat ys*)
proof (*induction xs arbitrary*: *ys*)
case *Nil*
then show *?case*
apply (*intro conjI*)
subgoal by *simp*
subgoal
apply (*cases ys*)
subgoal by *simp*
subgoal for *a ys′*
by (*cases a*) *simp-all*
done
subgoal by *simp*
done
next
case (*Cons a xs*)
then show *?case*
apply (*cases a*; *cases ys*)
subgoal by *simp*
subgoal for *b ys′*
apply (*cases b*)
subgoal by *fastforce*
subgoal by *simp*

```
      done
    subgoal by (simp add: add-carry-com)
    subgoal for b ys'
      apply (cases b)
      subgoal by fastforce
      subgoal by simp
      done
    done
qed

lemma inc-nat-add-carry:
  inc-nat (add-carry c xs ys) = add-carry c (inc-nat xs) ys ∧
   inc-nat (add-carry c xs ys) = add-carry c xs (inc-nat ys)
proof (cases c)
  case True
  then have
    add-carry c (inc-nat xs) ys = inc-nat (add-carry False (inc-nat xs) ys)
    add-carry c xs (inc-nat ys) = inc-nat (add-carry False xs (inc-nat ys))
    using add-carry-True-inc-nat by simp-all
  moreover have
    add-carry False (inc-nat xs) ys = inc-nat (add-carry False xs ys)
    using add-carry-True-inc-nat[of xs ys] by argo
   moreover have add-carry False xs (inc-nat ys) = inc-nat (add-carry False xs
ys)
    using add-carry-True-inc-nat[of xs ys] by argo
  ultimately show ?thesis using add-carry-True-inc-nat True by simp
next
  case False
  then show ?thesis using add-carry-True-inc-nat[of xs ys] by auto
qed

lemma add-carry-inc-nat-simps:
  add-carry True xs ys = inc-nat (add-carry False xs ys)
  add-carry False (inc-nat xs) ys = inc-nat (add-carry False xs ys)
  add-carry False xs (inc-nat ys) = inc-nat (add-carry False xs ys)
  using inc-nat-add-carry[of - xs ys] add-carry-True-inc-nat[of xs ys]
  by argo+

lemma add-carry-assoc: add-carry c2 (add-carry c1 xs ys) zs = add-carry c1 xs
(add-carry c2 ys zs)
  apply (intro nat-lsbf-eqI)
  subgoal by (simp add: add-carry-correct)
  subgoal
  proof -
    let ?t1 = of-bool c1 + to-nat xs + to-nat ys
    let ?t2 = of-bool c2 + to-nat ys + to-nat zs
    let ?t3 = of-bool c1 + of-bool c2 + to-nat xs + to-nat ys + to-nat zs

     have length (add-carry c2 (add-carry c1 xs ys) zs) = max (max (max (max
```

(*length xs*) (*length ys*)) (*floorlog 2 ?t1*)) (*length zs*))
(*floorlog 2 ?t3*)
unfolding *length-add-carry add-carry-correct eval-bool-is-of-bool*
by (*intro arg-cong2*[**where** *f* = *max*] *refl arg-cong2*[**where** *f* = *floorlog*]) *simp*
also have ... = *max* (*max* (*max* (*max* (*floorlog 2 ?t1*) (*floorlog 2 ?t3*)) (*length xs*)) (*length ys*)) (*length zs*)
using *max.commute max.assoc* **by** *presburger*
also have ... = *max* (*max* (*max* (*floorlog 2 ?t3*) (*length xs*)) (*length ys*)) (*length zs*) (**is** ... = *?t4*)
by (*intro arg-cong2*[**where** *f* = *max*] *refl max.absorb2 floorlog-mono*) *simp*
finally have *1*: *length* (*add-carry c2* (*add-carry c1 xs ys*) *zs*) = *?t4* **.**

have *length* (*add-carry c1 xs* (*add-carry c2 ys zs*)) = *max* (*max* (*length xs*) (*max* (*max* (*length ys*) (*length zs*)) (*floorlog 2 ?t2*)))
(*floorlog 2 ?t3*)
unfolding *length-add-carry add-carry-correct eval-bool-is-of-bool*
by (*intro arg-cong2*[**where** *f* = *max*] *refl arg-cong2*[**where** *f* = *floorlog*]) *simp*
also have ... = *max* (*max* (*max* (*max* (*floorlog 2 ?t2*) (*floorlog 2 ?t3*)) (*length xs*)) (*length ys*)) (*length zs*)
using *max.commute max.assoc* **by** *presburger*
also have ... = *max* (*max* (*max* (*floorlog 2 ?t3*) (*length xs*)) (*length ys*)) (*length zs*)
by (*intro arg-cong2*[**where** *f* = *max*] *refl max.absorb2 floorlog-mono*) *simp*
finally have *2*: *length* (*add-carry c1 xs* (*add-carry c2 ys zs*)) = *?t4* **.**

show *?thesis* **unfolding** *1 2* **by** (*rule refl*)
qed
done

lemma *truncated-add-carry*:
assumes *truncated xs truncated ys*
shows *truncated* (*add-carry c xs ys*)
proof −
have *length* (*add-carry c xs ys*) =
max (*max* (*length xs*) (*length ys*)) (*bitsize* (*of-bool c* + *to-nat xs* + *to-nat ys*))
unfolding *length-add-carry bitsize-is-floorlog* **by** *argo*
also have ... = *max* (*max* (*bitsize* (*to-nat xs*)) (*bitsize* (*to-nat ys*))) (*bitsize* (*of-bool c* + *to-nat xs* + *to-nat ys*))
using *truncated-iff' assms* **by** *algebra*
also have ... = *bitsize* (*of-bool c* + *to-nat xs* + *to-nat ys*)
using *bitsize-mono* **by** *simp*
also have ... = *bitsize* (*to-nat* (*add-carry c xs ys*))
by (*simp add*: *add-carry-correct*)
finally show *?thesis* **unfolding** *truncated-iff'* **.**
qed

8.7.3 Addition

definition *add-nat* :: *nat-lsbf* ⇒ *nat-lsbf* ⇒ *nat-lsbf* **where**

add-nat x y = *add-carry False x y*

corollary *length-add-nat-lower*: *length* (*add-nat xs ys*) ≥ *max* (*length xs*) (*length ys*)
unfolding *add-nat-def* **by** (*simp only*: *length-add-carry-lower*)

corollary *length-add-nat-upper*: *length* (*add-nat xs ys*) ≤ *max* (*length xs*) (*length ys*) + *1*
unfolding *add-nat-def* **using** *length-add-carry-upper*[*of False xs ys*] **by** *simp*

corollary *add-nat-last-bit-True*: *length* (*add-nat xs ys*) = *max* (*length xs*) (*length ys*) + *1* ⟹ ∃ *zs*. *add-nat xs ys* = *zs* @ [*True*]
unfolding *add-nat-def* **by** (*simp add*: *add-carry-last-bit-True*)

lemma *add-nat-correct*: *to-nat* (*add-nat x y*) = *to-nat x* + *to-nat y*
unfolding *add-nat-def* **using** *add-carry-correct* **by** *simp*

corollary *add-nat-com*: *add-nat xs ys* = *add-nat ys xs*
unfolding *add-nat-def* **by** (*simp add*: *add-carry-com*)

corollary *add-nat-assoc*: *add-nat xs* (*add-nat ys zs*) = *add-nat* (*add-nat xs ys*) *zs*
unfolding *add-nat-def* **using** *add-carry-assoc* **by** *simp*

corollary *truncated-add-nat*:
assumes *truncated xs truncated ys*
shows *truncated* (*add-nat xs ys*)
unfolding *add-nat-def*
by (*intro truncated-add-carry assms*)

8.8 Comparison and subtraction

8.8.1 Comparison

fun *compare-nat-same-length-reversed* :: *bool list* ⇒ *bool list* ⇒ *bool* **where**
compare-nat-same-length-reversed [] [] = *True*
| *compare-nat-same-length-reversed* (*False*#*xs*) (*False*#*ys*) = *compare-nat-same-length-reversed xs ys*
| *compare-nat-same-length-reversed* (*True*#*xs*) (*False*#*ys*) = *False*
| *compare-nat-same-length-reversed* (*False*#*xs*) (*True*#*ys*) = *True*
| *compare-nat-same-length-reversed* (*True*#*xs*) (*True*#*ys*) = *compare-nat-same-length-reversed xs ys*
| *compare-nat-same-length-reversed* - - = *undefined*

lemma *compare-nat-same-length-reversed-correct*:
length xs = *length ys* ⟹ *compare-nat-same-length-reversed xs ys* ⟷ *to-nat* (*rev xs*) ≤ *to-nat* (*rev ys*)
proof (*induction xs ys rule*: *compare-nat-same-length-reversed.induct*)
case *1*
then show *?case* **by** *simp*
next

case (*2 xs ys*)
have *to-nat* (*rev* (*False* # *xs*)) = *to-nat* (*rev xs*) *to-nat* (*rev* (*False* # *ys*)) = *to-nat* (*rev ys*)
using *to-nat-app* **by** *simp-all*
then have *to-nat* (*rev* (*False* # *xs*)) ≤ *to-nat* (*rev* (*False* # *ys*)) ⟷ *to-nat* (*rev xs*) ≤ *to-nat* (*rev ys*)
by *simp*
then show *?case* **using** *2* **by** *simp*
next
case (*3 xs ys*)
have *to-nat* (*rev* (*True* # *xs*)) = *2* ^ (*length xs*) + *to-nat* (*rev xs*)
using *to-nat-app* **by** *simp*
also have ... > *to-nat* (*rev ys*)
using *3 to-nat-length-upper-bound*[*of rev ys*] *leI le-add-diff-inverse2* **by** *fastforce*
also have *to-nat* (*rev ys*) = *to-nat* (*rev* (*False* # *ys*))
using *to-nat-app* **by** *simp*
finally have *to-nat* (*rev* (*True* # *xs*)) > *to-nat* (*rev* (*False* # *ys*)) **.**
thus *?case* **using** *3* **by** *simp*
next
case (*4 xs ys*)
have *to-nat* (*rev* (*False* # *xs*)) = *to-nat* (*rev xs*)
using *to-nat-app* **by** *simp*
also have ... ≤ *2* ^ (*length xs*)
using *to-nat-length-upper-bound*[*of rev xs*] **by** *simp*
also have ... ≤ *to-nat* (*rev* (*True* # *ys*))
using *to-nat-app 4* **by** *simp*
finally have *to-nat* (*rev* (*False* # *xs*)) ≤ *to-nat* (*rev* (*True* # *ys*)) **.**
thus *?case* **using** *4* **by** *simp*
next
case (*5 xs ys*)
have *to-nat* (*rev* (*True* # *xs*)) = *2* ^ (*length xs*) + *to-nat* (*rev xs*) *to-nat* (*rev* (*True* # *ys*)) = *2* ^ (*length ys*) + *to-nat* (*rev ys*)
using *to-nat-app* **by** *simp-all*
then have *to-nat* (*rev* (*True* # *xs*)) ≤ *to-nat* (*rev* (*True* # *ys*)) ⟷ *to-nat* (*rev xs*) ≤ *to-nat* (*rev ys*)
using *5* **by** *simp*
then show *?case* **using** *5* **by** *simp*
next
case (*6-1 va*)
then show *?case* **by** *simp*
next
case (*6-2 v va*)
then show *?case* **by** *simp*
next
case (*6-3 v va*)
then show *?case* **by** *simp*
next
case (*6-4 va*)
then show *?case* **by** *simp*

qed

fun *compare-nat-same-length* :: *nat-lsbf* ⇒ *nat-lsbf* ⇒ *bool* **where**
compare-nat-same-length xs ys = *compare-nat-same-length-reversed* (*rev xs*) (*rev ys*)

lemma *compare-nat-same-length-correct*:
length xs = *length ys* ⟹ *compare-nat-same-length xs ys* = (*to-nat xs* ≤ *to-nat ys*)
using *compare-nat-same-length-reversed-correct* **by** *simp*

definition *make-same-length* :: *nat-lsbf* ⇒ *nat-lsbf* ⇒ *nat-lsbf* × *nat-lsbf* **where**
make-same-length xs ys = (*let n* = *max* (*length xs*) (*length ys*) *in* ((*fill n xs*), (*fill n ys*)))

lemma *make-same-length-correct*:
assumes (*fill-xs*, *fill-ys*) = *make-same-length xs ys*
shows *length fill-ys* = *length fill-xs*
length fill-xs = *max* (*length xs*) (*length ys*)
to-nat fill-xs = *to-nat xs*
to-nat fill-ys = *to-nat ys*
using *assms* **by** (*simp-all add*: *Let-def make-same-length-def*)

definition *compare-nat* :: *nat-lsbf* ⇒ *nat-lsbf* ⇒ *bool* **where**
compare-nat xs ys = (*let* (*fill-xs*, *fill-ys*) = *make-same-length xs ys in compare-nat-same-length fill-xs fill-ys*)

lemma *compare-nat-correct*: *compare-nat xs ys* = (*to-nat xs* ≤ *to-nat ys*)
proof −
obtain *fill-xs fill-ys* **where** *fills-def*: *make-same-length xs ys* = (*fill-xs*, *fill-ys*)
by *fastforce*
then show *?thesis* **unfolding** *compare-nat-def Let-def*
using *make-same-length-correct*[*OF fills-def*[*symmetric*]]
using *compare-nat-same-length-reversed-correct*[*of rev fill-xs rev fill-ys*]
by *simp*
qed

8.8.2 Subtraction

definition *subtract-nat* :: *nat-lsbf* ⇒ *nat-lsbf* ⇒ *nat-lsbf* **where**
subtract-nat xs ys = (*if compare-nat xs ys then* [] *else*
let (*fill-xs*, *fill-ys*) = *make-same-length xs ys in*
butlast (*add-carry True fill-xs* (*map Not fill-ys*)))

lemma *add-complement*: *add-nat xs* (*map Not xs*) = *replicate* (*length xs*) *True*
proof (*induction xs*)
case *Nil*
then show *?case* **unfolding** *add-nat-def* **by** *simp*
next

case (*Cons a xs*)
have *add-nat* (*a* # *xs*) (*map Not* (*a* # *xs*)) = *True* # (*add-carry False xs* (*map Not xs*))
unfolding *add-nat-def* **by** (*cases a*) *simp-all*
also have ... = *True* # (*replicate* (*length xs*) *True*)
using *Cons.IH* **by** (*simp add*: *add-nat-def*)
finally show *?case* **by** *simp*
qed

lemma *to-nat-complement*: *to-nat* (*map Not xs*) = *2* ^ (*length xs*) − *1* − *to-nat xs*
using *add-complement*[*of xs*] *to-nat-replicate-true*[*of length xs*] *add-nat-correct*[*of xs map Not xs*]
by *simp*

lemma *to-nat-butlast*: *zs* = *xs* @ [*True*] ⟹ *to-nat* (*butlast zs*) = *to-nat zs* − *2* ^ *length xs*
using *to-nat-app*[*of xs* [*True*]] **by** *simp*

lemma *inc-nat-true-prefix*[*simp*]: *inc-nat* (*replicate n True* @ [*False*] @ *ys*) = *replicate n False* @ [*True*] @ *ys*
by (*induction n arbitrary*: *ys*) *simp-all*

lemma *length-inc-nat-aux*: *zs* = *replicate n True* @ [*False*] @ *ys* ⟹ *length* (*inc-nat zs*) = *length zs*
using *inc-nat-true-prefix*[*of n ys*] **by** *simp*

lemma *length-inc-nat-aux-2*: *length* (*inc-nat* (*xs* @ [*False*] @ *ys*)) = *length* (*xs* @ [*False*] @ *ys*)
proof −
define *zs* **where** *zs* = *xs* @ [*False*] @ *ys*
with *bit-strong-decomp-1*[*of zs False*] **obtain** *ys′ n* **where** *zs* = *replicate n True* @ [*False*] @ *ys′*
by *auto*
then show *?thesis* **using** *length-inc-nat-aux zs-def* **by** *simp*
qed

lemma *subtract-nat-aux*: *to-nat* (*subtract-nat xs ys*) = (*to-nat xs*) − (*to-nat ys*) ∧ *length* (*subtract-nat xs ys*) ≤ *max* (*length xs*) (*length ys*)
proof (*cases compare-nat xs ys*)
case *True*
then show *?thesis* **using** *compare-nat-correct* **unfolding** *subtract-nat-def* **by** *simp*
next
case *False*

obtain *fill-xs fill-ys* **where** *fills-def*: *make-same-length xs ys* = (*fill-xs*, *fill-ys*)
by *fastforce*
note *fills-props* = *make-same-length-correct*[*OF fills-def*[*symmetric*]]

define *n* **where** *n* = *max* (*length xs*) (*length ys*)
then have *length fill-xs* = *n length fill-ys* = *n* **using** *fills-props* **by** *auto*

from *False* **have** *to-nat fill-xs* > *to-nat fill-ys*
using *fills-props compare-nat-correct* **by** *simp*
then have *n* > *0* **using** ‹*length fill-xs* = *n*› **by** *auto*

let *?add* = *add-carry True fill-xs* (*map Not fill-ys*)

have *subtract-nat-xs-ys*: *subtract-nat xs ys* = *butlast ?add*
unfolding *subtract-nat-def* **using** *False fills-def* **by** *simp*

have *to-nat fill-ys* ≤ *2* ^ *n* − *1 to-nat fill-xs* ≤ *2* ^ *n* − *1 to-nat* (*map Not fill-ys*) ≤ *2* ^ *n* − *1*
subgoal using *to-nat-length-upper-bound*[*of fill-ys*] ‹*length fill-ys* = *n*› **by** *argo*
subgoal using *to-nat-length-upper-bound*[*of fill-xs*] ‹*length fill-xs* = *n*› **by** *argo*
subgoal using *to-nat-length-upper-bound*[*of map Not fill-ys*] ‹*length fill-ys* = *n*› **by** *simp*
done
then have *to-nat ?add* ≤ (*2* ^ *n* − *1*) + (*2* ^ *n* − *1*) + *1* **unfolding** *add-carry-correct* **by** *simp*
also have ... = *2* ^ (*n* + *1*) − *2* + *1* **by** *simp*
also have ... = *2* ^ (*n* + *1*) − *1*
using *Nat.diff-diff-right*[*of 1 2 2* ^ (*n* + *1*)] *Nat.diff-add-assoc2*[*of 2 2* ^ (*n* + *1*) *1*]
by *simp*
finally have *to-nat ?add* ≤ ... **.**

from ‹*to-nat fill-xs* > *to-nat fill-ys*› **have** *to-nat fill-xs* ≥ *to-nat fill-ys* + *1* **by** *simp*
then have *to-nat fill-xs* + *2* ^ *n* ≥ *2* ^ *n* + *to-nat fill-ys* + *1* **by** *simp*
then have *to-nat fill-xs* + (*2* ^ *n* − *1* − *to-nat fill-ys*) ≥ *2* ^ *n* **by** *simp*
then have *to-nat fill-xs* + *to-nat* (*map Not fill-ys*) ≥ *2* ^ *n*
using *to-nat-complement*[*of fill-ys*] ‹*length fill-ys* = *n*› **by** *simp*
then have *to-nat ?add* ≥ *2* ^ *n*
using *add-carry-correct fills-props* **by** *simp*
then have *length ?add* ≥ *n* + *1*
using *to-nat-bound-to-length-bound* **by** *simp*
then have *length ?add* = *n* + *1*
using *length-add-carry-upper*[*of True fill-xs map Not fill-ys*] ‹*length fill-xs* = *n*› ‹*length fill-ys* = *n*›
by *simp*

then obtain *zs* **where** *?add* = *zs* @ [*True*] *length zs* = *n*
using *add-carry-last-bit-True*[*of True fill-xs map Not fill-ys*] ‹*length fill-xs* = *n*› ‹*length fill-ys* = *n*›
by *auto*
then have *1*: *to-nat* (*butlast ?add*) = *to-nat fill-xs* + *to-nat* (*map Not fill-ys*) + *1* − *2* ^ *n*

unfolding *to-nat-butlast*[*OF* ‹*?add* = *zs* @ [*True*]›]
using *add-carry-correct* **by** (*metis Suc-eq-plus1 add.assoc eval-bool.simps*(*1*) *plus-1-eq-Suc*)
also have ... = *to-nat fill-xs* + (*2* ^ *n* − *1* − *to-nat fill-ys*) + *1* − *2* ^ *n*
unfolding *to-nat-complement*[*of fill-ys*] ‹*length fill-ys* = *n*› **by** (*rule refl*)
also have ... = *to-nat fill-xs* + (*2* ^ *n* − *1*) − *to-nat fill-ys* + *1* − *2* ^ *n*
using *le-add-diff-inverse*[*OF* ‹*to-nat fill-ys* ≤ *2* ^ *n* − *1*›] **by** *linarith*
also have ... = *to-nat fill-xs* − *to-nat fill-ys* + (*2* ^ *n* − *1*) − (*2* ^ *n* − *1*)
using ‹*to-nat fill-xs* > *to-nat fill-ys*› **by** *simp*
also have ... = *to-nat fill-xs* − *to-nat fill-ys* **by** *simp*
finally have *2*: *to-nat* (*subtract-nat xs ys*) = *to-nat xs* − *to-nat ys*
unfolding *subtract-nat-xs-ys fills-props* **.**

have *3*: *length* (*butlast ?add*) = *n*
using ‹*length ?add* = *n* + *1*› **by** *simp*

show *?thesis*
apply (*intro conjI*)
subgoal by (*fact 2*)
subgoal using *3* **unfolding** *subtract-nat-xs-ys n-def*[*symmetric*] **by** *simp*
done
qed

corollary *subtract-nat-correct*: *to-nat* (*subtract-nat xs ys*) = (*to-nat xs*) − (*to-nat ys*)
using *subtract-nat-aux* **by** *simp*

corollary *length-subtract-nat-le*: *length* (*subtract-nat xs ys*) ≤ *max* (*length xs*) (*length ys*)
using *subtract-nat-aux* **by** *simp*

8.9 (Grid) Multiplication

fun *grid-mul-nat* :: *nat-lsbf* ⇒ *nat-lsbf* ⇒ *nat-lsbf* **where**
grid-mul-nat [] - = []
| *grid-mul-nat* (*False#xs*) *y* = *False* # (*grid-mul-nat xs y*)
| *grid-mul-nat* (*True#xs*) *y* = *add-nat* (*False* # (*grid-mul-nat xs y*)) *y*

lemma *grid-mul-nat-correct*: *to-nat* (*grid-mul-nat x y*) = *to-nat x* ∗ *to-nat y*
by (*induction x y rule*: *grid-mul-nat.induct*) (*simp-all add*: *add-nat-correct*)

lemma *length-grid-mul-nat*: *length* (*grid-mul-nat xs ys*) ≤ *length xs* + *length ys*
proof (*induction xs ys rule*: *grid-mul-nat.induct*)
case (*1 uu*)
then show *?case* **by** *simp*
next
case (*2 xs y*)
then show *?case* **by** *simp*
next

```
  case (3 xs y)
  show ?case
  proof (rule ccontr)
    assume ¬ length (grid-mul-nat (True # xs) y) ≤ length (True # xs) + length
y
    then have l: length (grid-mul-nat (True # xs) y) = length xs + length y + 2
      using length-add-nat-upper[of False # grid-mul-nat xs y y] 3 by simp

   then have length (add-nat (False # grid-mul-nat xs y) y) = max (length (False
# grid-mul-nat xs y)) (length y) + 1
      using length-add-nat-upper[of False # grid-mul-nat xs y y] 3 by simp
    then obtain as where add-nat (False # grid-mul-nat xs y) y = as @ [True]
      using add-nat-last-bit-True[of False # grid-mul-nat xs y y] by auto
    then have as-def: grid-mul-nat (True # xs) y = as @ [True] by simp
    then have length-as: length as = length xs + length y + 1 using l by simp

    from as-def have m: to-nat (True # xs) ∗ to-nat y = to-nat (as @ [True])
      using grid-mul-nat-correct by metis
    also have to-nat (as @ [True]) ≥ 2 ^ length as
      using to-nat-length-lower-bound by simp
    also have 2 ^ length as = 2 ^ (length xs + length y + 1) using length-as by
simp
    also have to-nat (True # xs) ∗ to-nat y < 2 ^ (length xs + 1) ∗ 2 ^ length y
      apply (intro mult-less-le-imp-less)
      subgoal using to-nat-length-upper-bound[of True # xs] by simp
      subgoal using to-nat-length-upper-bound[of y] by simp
      subgoal by simp
      subgoal
        apply (rule ccontr)
        using m to-nat-length-lower-bound[of as] by simp
      done
    finally show False by (simp add: power-add)
  qed
qed
```

8.10 Syntax bundles

```
abbreviation shift-right-flip xs n ≡ shift-right n xs
bundle nat-lsbf-syntax
begin
  notation add-nat (infixl +_n 65)
  notation compare-nat (infixl ≤_n 50)
  notation subtract-nat (infixl −_n 65)
  notation grid-mul-nat (infixl ∗_n 70)
  notation shift-right-flip (infixl >>_n 55)
end

bundle no-nat-lsbf-syntax
begin
```

```
  no-notation add-nat (infixl +n 65)
  no-notation compare-nat (infixl ≤n 50)
  no-notation subtract-nat (infixl −n 65)
  no-notation grid-mul-nat (infixl ∗n 70)
  no-notation shift-right-flip (infixl >>n 55)
end

unbundle nat-lsbf-syntax

end
theory Karatsuba-Runtime-Lemmas
  imports Complex-Main Akra-Bazzi.Akra-Bazzi-Method
begin
```

An explicit bound for a specific class of recursive functions.

```
context
  fixes a b c d :: nat
  fixes f :: nat ⇒ nat
  assumes small-bounds: f 0 ≤ a f (Suc 0) ≤ a
  assumes recursive-bound: ⋀n. n > 1 ⟹ f n ≤ c ∗ n + d + f (n div 2)
begin

private fun g where
g 0 = a
| g (Suc 0) = a
| g n = c ∗ n + d + g (n div 2)

private lemma f-g-bound: f n ≤ g n
  apply (induction n rule: g.induct)
  subgoal using small-bounds by simp
  subgoal using small-bounds by simp
  subgoal for x using recursive-bound[of Suc (Suc x)] by auto
  done

private lemma g-mono-aux: a ≤ g n
  by (induction n rule: g.induct) simp-all

private lemma g-mono: m ≤ n ⟹ g m ≤ g n
proof (induction m arbitrary: n rule: g.induct)
  case 1
  then show ?case using g-mono-aux by simp
next
  case 2
  then show ?case using g-mono-aux by simp
next
  case (3 x)
  then obtain y where n = Suc (Suc y) using Suc-le-D by blast
  have g (Suc (Suc x)) = c ∗ Suc (Suc x) + d + g (Suc (Suc x) div 2)
    by simp
```

also have ... ≤ *c* ∗ *n* + *d* + *g* (*n div 2*)
using *3*
by (*metis add-mono add-mono-thms-linordered-semiring*(*3*) *div-le-mono nat-mult-le-cancel-disj*)
finally show *?case* using ‹*n* = *Suc* (*Suc y*)› by *simp*
qed

private lemma *g-powers-of-2*: *g* (*2* ^ *n*) = *d* ∗ *n* + *c* ∗ (*2* ^ (*n* + *1*) − *2*) + *a*
proof (*induction n*)
case (*Suc n*)
then obtain *n*′ where *2* ^ *Suc n* = *Suc* (*Suc n*′)
by (*metis g.cases less-exp not-less-eq zero-less-Suc*)
then have *g* (*2* ^ *Suc n*) = *c* ∗ *2* ^ *Suc n* + *d* + *g* (*2* ^ *n*)
by (*metis g.simps*(*3*) *nonzero-mult-div-cancel-right power-Suc2 zero-neq-numeral*)
also have ... = *c* ∗ *2* ^ *Suc n* + *d* + *d* ∗ *n* + *c* ∗ (*2* ^ (*n* + *1*) − *2*) + *a*
using *Suc* by *simp*
also have ... = *d* ∗ *Suc n* + *c* ∗ (*2* ^ *Suc n* + (*2* ^ (*n* + *1*) − *2*)) + *a*
using *add-mult-distrib2*[*symmetric*, *of c*] by *simp*
finally show *?case* by *simp*
qed *simp*

private lemma *pow-ineq*:
assumes *m* ≥ (*1* :: *nat*)
assumes *p* ≥ *2*
shows *p* ^ *m* > *m*
using *assms*
apply (*induction m*)
subgoal by *simp*
subgoal for *m*
by (*cases m*) (*simp-all add*: *less-trans-Suc*)
done

private lemma *next-power-of-2*:
assumes *m* ≥ (*1* :: *nat*)
shows ∃ *n k*. *m* = *2* ^ *n* + *k* ∧ *k* < *2* ^ *n*
proof −
from *ex-power-ivl1*[*OF order.refl assms*] obtain *n* where *2* ^ *n* ≤ *m m* < *2* ^
(*n* + *1*)
by *auto*
then have *m* = *2* ^ *n* + (*m* − *2* ^ *n*) *m* − *2* ^ *n* < *2* ^ *n* by *simp-all*
then show *?thesis* by *blast*
qed

lemma *div-2-recursion-linear*: *f n* ≤ (*2* ∗ *d* + *4* ∗ *c*) ∗ *n* + *a*
proof (*cases n* ≥ *1*)
case *True*
then obtain *m k* where *n* = *2* ^ *m* + *k k* < *2* ^ *m* using *next-power-of-2* by
blast
have *f n* ≤ *g n* using *f-g-bound* by *simp*
also have ... ≤ *g* (*2* ^ *m* + *2* ^ *m*) using ‹*n* = *2* ^ *m* + *k*› ‹*k* < *2* ^ *m*› *g-mono*

by *simp*
also have ... = $d * Suc\ m + c * (2\ \hat{}\ (Suc\ m + 1) - 2) + a$
using *g-powers-of-2*[*of Suc m*]
apply (*subst mult-2*[*symmetric*])
apply (*subst power-Suc*[*symmetric*])
.
also have ... ≤ $d * Suc\ m + c * 2\ \hat{}\ (Suc\ m + 1) + a$ **by** *simp*
also have ... ≤ $d * 2\ \hat{}\ Suc\ m + c * 2\ \hat{}\ (Suc\ m + 1) + a$ **using** *less-exp*[*of Suc m*]
by (*meson add-le-mono less-or-eq-imp-le mult-le-mono*)
also have ... = $(2 * d + 4 * c) * 2\ \hat{}\ m + a$ **using** *mult.assoc add-mult-distrib* **by** *simp*
also have ... ≤ $(2 * d + 4 * c) * n + a$
using ‹$n = 2\ \hat{}\ m + k$› *power-increasing*[*of m n*] **by** *simp*
finally show *?thesis* **.**
next
case *False*
then have $n = 0$ **by** *simp*
then show *?thesis* **using** *small-bounds* **by** *simp*
qed

end

General Lemmas for Landau notation.

lemma *landau-o-plus-aux'*:
fixes *f g*
assumes $f \in o[F](g)$
shows $O[F](g) = O[F](\lambda x.\ f\ x + g\ x)$
apply (*intro equalityI subsetI*)
subgoal using *landau-o.big.trans*[*OF - landau-o.plus-aux*[*OF assms*]] **by** *simp*
subgoal for *h*
using *assms* **by** *simp*
done

lemma *powr-bigo-linear-index-transformation*:
fixes *fl* :: *nat* ⇒ *nat*
fixes *f* :: *nat* ⇒ *real*
assumes $(\lambda x.\ real\ (fl\ x)) \in O(\lambda n.\ real\ n)$
assumes $f \in O(\lambda n.\ real\ n\ powr\ p)$
assumes $p > 0$
shows $f \circ fl \in O(\lambda n.\ real\ n\ powr\ p)$
proof −
obtain *c1* **where** $c1 > 0\ \forall_F\ x\ in\ sequentially.\ norm\ (real\ (fl\ x)) \le c1 * norm\ (real\ x)$
using *landau-o.bigE*[*OF assms(1)*] **by** *auto*
then obtain *N1* **where** *fl-bound*: $\forall x.\ x \ge N1 \longrightarrow norm\ (real\ (fl\ x)) \le c1 * norm\ (real\ x)$
unfolding *eventually-at-top-linorder* **by** *blast*
obtain *c2* **where** $c2 > 0\ \forall_F\ x\ in\ sequentially.\ norm\ (f\ x) \le c2 * norm\ (real\ x$

```
powr p)
    using landau-o.bigE[OF assms(2)] by auto
  then obtain N2 where f-bound: ∀x. x ≥ N2 ⟶ norm (f x) ≤ c2 ∗ norm (real
x powr p)
    unfolding eventually-at-top-linorder by blast

  define cf :: real where cf = Max {norm (f y) | y. y ≤ N2}
  then have cf ≥ 0 using Max-in[of {norm (f y) | y. y ≤ N2}] norm-ge-zero by
fastforce
  define c where c = c2 ∗ c1 powr p
  then have c > 0 using ‹c1 > 0› ‹c2 > 0› by simp

  have ∀x. x ≥ N1 ⟶ norm (f (fl x)) ≤ cf + c ∗ norm (real x) powr p
  proof (intro allI impI)
    fix x
    assume x ≥ N1
    show norm (f (fl x)) ≤ cf + c ∗ norm (real x) powr p
    proof (cases fl x ≥ N2)
      case True
      then have norm (f (fl x)) ≤ c2 ∗ norm (real (fl x) powr p)
        using f-bound by simp
      also have ... = c2 ∗ norm (real (fl x)) powr p
        by simp
      also have ... ≤ c2 ∗ (c1 ∗ norm (real x)) powr p
        apply (intro mult-mono order.refl powr-mono2 norm-ge-zero)
        subgoal using ‹p > 0› by simp
        subgoal using fl-bound ‹x ≥ N1› by simp
        subgoal using ‹c2 > 0› by simp
        subgoal by simp
        done
      also have ... = c2 ∗ (c1 powr p ∗ norm (real x) powr p)
        apply (intro arg-cong[where f = (∗) c2] powr-mult norm-ge-zero)
        using ‹c1 > 0› by simp
      also have ... = c ∗ norm (real x) powr p unfolding c-def by simp
      also have ... ≤ cf + c ∗ norm (real x) powr p using ‹cf ≥ 0› by simp
      finally show ?thesis .
    next
      case False
      then have norm (f (fl x)) ≤ cf unfolding cf-def
        by (intro Max-ge) auto
      also have ... ≤ cf + c ∗ norm (real x) powr p
        using ‹c > 0› by simp
      finally show ?thesis .
    qed
  qed
  then have f ∘ fl ∈ O(λx. cf + c ∗ (real x) powr p)
    apply (intro landau-o.big-mono)
    unfolding eventually-at-top-linorder comp-apply by fastforce
  also have ... = O(λx. c ∗ (real x) powr p)
```

proof (*intro landau-o-plus-aux'*[*symmetric*])
have ($\lambda x.\ cf$) ∈ *O*($\lambda x.$ *real x powr 0*) **by** *simp*
moreover have ($\lambda x.$ *real x powr 0*) ∈ *o*($\lambda x.$ *real x powr p*)
using *iffD2*[*OF powr-smallo-iff*, *OF filterlim-real-sequentially sequentially-bot* ‹$p > 0$›] **.**
ultimately have ($\lambda x.\ cf$) ∈ *o*($\lambda x.$ *real x powr p*)
by (*rule landau-o.big-small-trans*)
also have ... = *o*($\lambda x.\ c *$ (*real x*) *powr p*)
using *landau-o.small.cmult* ‹$c > 0$› **by** *simp*
finally show ($\lambda x.\ cf$) ∈ ... **.**
qed
also have ... = *O*($\lambda x.$ (*real x*) *powr p*) **using** *landau-o.big.cmult* ‹$c > 0$› **by** *simp*
finally show *?thesis* **.**
qed

lemma *real-mono*: ($a \leq b$) = (*real a* ≤ *real b*)
by *simp*

lemma *real-linear*: *real* ($a + b$) = *real a* + *real b*
by *simp*

lemma *real-multiplicative*: *real* ($a * b$) = *real a* ∗ *real b*
by *simp*

lemma (**in** *landau-pair*) *big-1-mult-left*:
fixes *f g h*
assumes $f \in L\ F\ (g)$ $h \in L\ F\ (\lambda\text{-}.\ 1)$
shows ($\lambda x.\ h\ x * f\ x$) ∈ $L\ F\ (g)$
proof −
have ($\lambda x.\ f\ x * h\ x$) ∈ $L\ F\ (g)$ **using** *assms* **by** (*rule big-1-mult*)
also have ($\lambda x.\ f\ x * h\ x$) = ($\lambda x.\ h\ x * f\ x$) **by** *auto*
finally show *?thesis* **.**
qed

lemma *norm-nonneg*: $x \geq 0 \Longrightarrow$ *norm* $x = x$ **by** *simp*

lemma *landau-mono-always*:
fixes *f g*
assumes $\bigwedge x.\ f\ x \geq (0 :: real)$ $\bigwedge x.\ g\ x \geq 0$
assumes $\bigwedge x.\ f\ x \leq g\ x$
shows $f \in O[F](g)$
apply (*intro landau-o.bigI*[*of 1*])
using *assms* **by** *simp-all*

end

9 Running time of *Nat-LSBF*

theory *Nat-LSBF-TM*

imports *Nat-LSBF ../Karatsuba-Runtime-Lemmas ../Main-TM ../Estimation-Method*
begin

9.1 Truncating and filling

fun *truncate-reversed-tm* :: *nat-lsbf* ⇒ *nat-lsbf tm* **where**
truncate-reversed-tm [] =*1 return* []
| *truncate-reversed-tm* (*x* # *xs*) =*1* (*if x then return* (*x* # *xs*) *else truncate-reversed-tm xs*)

lemma *val-truncate-reversed-tm*[*simp*, *val-simp*]: *val* (*truncate-reversed-tm xs*) = *truncate-reversed xs*
by (*induction xs rule*: *truncate-reversed-tm.induct*) *simp-all*

lemma *time-truncate-reversed-tm-le*: *time* (*truncate-reversed-tm xs*) ≤ *length xs* + *1*
by (*induction xs rule*: *truncate-reversed-tm.induct*) *simp-all*

definition *truncate-tm* :: *nat-lsbf* ⇒ *nat-lsbf tm* **where**
truncate-tm xs =*1 do* {
rev-xs ← *rev-tm xs*;
truncate-rev-xs ← *truncate-reversed-tm rev-xs*;
rev-tm truncate-rev-xs
}

lemma *val-truncate-tm*[*simp*, *val-simp*]: *val* (*truncate-tm xs*) = *truncate xs*
by (*simp add*: *truncate-tm-def Nat-LSBF.truncate-def*)

lemma *time-truncate-tm-le*: *time* (*truncate-tm xs*) ≤ *3* ∗ *length xs* + *6*
using *add-mono*[*OF time-truncate-reversed-tm-le*[*of rev xs*] *truncate-reversed-length-ineq*[*of rev xs*]]
by (*simp add*: *truncate-tm-def*)

definition *fill-tm* :: *nat* ⇒ *nat-lsbf* ⇒ *nat-lsbf tm* **where**
fill-tm n xs =*1 do* {
k ← *length-tm xs*;
l ← $n -_t k$;
zeros ← *replicate-tm l False*;
xs $@_t$ *zeros*
}

lemma *val-fill-tm*[*simp*, *val-simp*]: *val* (*fill-tm n xs*) = *fill n xs*
by (*simp add*: *fill-tm-def fill-def*)

lemma *com-f-of-min-max*: *f a b* = *f b a* ⟹ *f* (*min a b*) (*max a b*) = *f a b*
by (*cases a* ≤ *b*; *simp add*: *max-def min-def*)
lemma *add-min-max*: *min* (*a*::′*a*:: *ordered-ab-semigroup-add*) *b* + *max a b* = *a* + *b*
by (*intro com-f-of-min-max add.commute*)

lemma *time-fill-tm*: *time* (*fill-tm n xs*) = *2* ∗ *length xs* + *n* + *5*
by (*simp add*: *fill-tm-def time-replicate-tm add-min-max*)

lemma *time-fill-tm-le*: *time* (*fill-tm n xs*) ≤ *3* ∗ *max n* (*length xs*) + *5*
unfolding *time-fill-tm* **by** *simp*

9.2 Right-shifts

definition *shift-right-tm* :: *nat* ⇒ *nat-lsbf* ⇒ *nat-lsbf tm* **where**
shift-right-tm n xs =*1 do* {
r ← *replicate-tm n False*;
r $@_t$ *xs*
}

lemma *val-shift-right-tm*[*simp*, *val-simp*]: *val* (*shift-right-tm n xs*) = *xs* $>>_n$ *n*
by (*simp add*: *shift-right-tm-def shift-right-def*)

lemma *time-shift-right-tm*[*simp*]: *time* (*shift-right-tm n xs*) = *2* ∗ *n* + *3*
by (*simp add*: *shift-right-tm-def time-replicate-tm*)

9.3 Subdividing lists

9.3.1 Splitting a list in two blocks

definition *split-at-tm* :: *nat* ⇒ *'a list* ⇒ (*'a list* × *'a list*) *tm* **where**
split-at-tm k xs =*1 do* {
xs1 ← *take-tm k xs*;
xs2 ← *drop-tm k xs*;
return (*xs1*, *xs2*)
}

lemma *val-split-at-tm*[*simp*, *val-simp*]: *val* (*split-at-tm k xs*) = *split-at k xs*
unfolding *split-at-tm-def* **by** *simp*

lemma *time-split-at-tm*: *time* (*split-at-tm k xs*) = *2* ∗ *min k* (*length xs*) + *3*
unfolding *split-at-tm-def tm-time-simps time-take-tm time-drop-tm* **by** *simp*

definition *split-tm* :: *nat-lsbf* ⇒ (*nat-lsbf* × *nat-lsbf*) *tm* **where**
split-tm xs =*1 do* {
n ← *length-tm xs*;
n-div-2 ← *n* div_t *2*;
split-at-tm n-div-2 xs
}

lemma *val-split-tm*[*simp*, *val-simp*]: *val* (*split-tm xs*) = *split xs*
by (*simp add*: *split-tm-def split-def Let-def*)

lemma *time-split-tm-le*: *time* (*split-tm xs*) ≤ *10* ∗ *length xs* + *16*
using *time-divide-nat-tm-le*[*of length xs 2*]

by (*simp add*: *split-tm-def time-split-at-tm*)

9.3.2 Splitting a list in multiple blocks

fun *subdivide-tm* :: *nat* ⇒ *'a list* ⇒ *'a list list tm* **where**
subdivide-tm 0 xs =1 undefined
| *subdivide-tm n* [] *=1 return* []
| *subdivide-tm n xs =1 do* {
  r ← *take-tm n xs*;
  s ← *drop-tm n xs*;
  rs ← *subdivide-tm n s*;
  return (*r* # *rs*)
}

lemma *val-subdivide-tm*[*simp*, *val-simp*]: $n > 0 \Longrightarrow$ *val* (*subdivide-tm n xs*) = *subdivide n xs*
by (*induction n xs rule*: *subdivide.induct*) *simp-all*

lemma *time-subdivide-tm-le-aux*:
assumes $n > 0$
shows *time* (*subdivide-tm n xs*) $\leq k * (2 * n + 3) +$ *time* (*subdivide-tm n* (*drop* $(k * n)$ *xs*))
proof (*induction k arbitrary*: *xs*)
case (*Suc k*)
show *?case*
proof (*cases xs*)
case *Nil*
then show *?thesis* **by** *simp*
next
case (*Cons a l*)
then have *time* (*subdivide-tm n* (*a* # *l*)) $\leq 2 * n + 3 +$ *time* (*subdivide-tm n* (*drop n* (*a* # *l*)))
using *gr0-implies-Suc*[*OF assms*] **by** (*auto simp*: *time-take-tm time-drop-tm*)
also have ... $\leq 2 * n + 3 + (k * (2 * n + 3) +$ *time* (*subdivide-tm n* (*drop* $(k * n)$ (*drop n* (*a* # *l*)))))
by (*intro add-mono order.refl Suc*)
also have ... = *Suc k* $* (2 * n + 3) +$ *time* (*subdivide-tm n* (*drop* (*Suc k* $* n$) (*a* # *l*)))
by (*simp add*: *add.commute*)
finally show *?thesis* **using** *Cons* **by** *simp*
qed
qed *simp*

lemma *time-subdivide-tm-le*:
fixes *xs* :: *'a list*
assumes $n > 0$
shows *time* (*subdivide-tm n xs*) $\leq 5 *$ *length xs* $+ 2 * n + 4$
proof −
define *k* **where** *k* = *length xs div n* + *1*

then have $k * n \geq$ *length xs* **using** *assms*
by (*meson div-less-iff-less-mult less-add-one order-less-imp-le*)
then have *drop-Nil*: *drop* ($k * n$) *xs* = [] **by** *simp*
have *time* (*subdivide-tm n xs*) $\leq k * (2 * n + 3) +$ *time* (*subdivide-tm n* ([] :: 'a list))
using *time-subdivide-tm-le-aux*[*OF assms*, *of xs k*] **unfolding** *drop-Nil* **.**
also have ... $= k * (2 * n + 3) + 1$ **using** *gr0-implies-Suc*[*OF assms*] **by** *auto*
also have ... $= (2 * n * ($*length xs div n*$) + 2 * n) + 3 * ($*length xs div n*$) + 4$
unfolding *k-def* **by** (*simp add*: *add-mult-distrib2*)
also have ... $\leq 5 *$ *length xs* $+ 2 * n + 4$
using *times-div-less-eq-dividend*[*of n length xs*] *div-le-dividend*[*of length xs n*] **by** *linarith*
finally show *?thesis* **.**
qed

9.4 The *bitsize* function

fun *bitsize-tm* :: *nat* ⇒ *nat tm* **where**
bitsize-tm 0 =*1 return 0*
| *bitsize-tm n* =*1 do* {
n-div-2 ← *n* div_t *2*;
r ← *bitsize-tm n-div-2*;
1 $+_t$ *r*
}

lemma *val-bitsize-tm*[*simp*, *val-simp*]: *val* (*bitsize-tm n*) = *bitsize n*
by (*induction n rule*: *bitsize-tm.induct*) *simp-all*

fun *time-bitsize-tm-bound* :: *nat* ⇒ *nat* **where**
time-bitsize-tm-bound 0 = *1*
| *time-bitsize-tm-bound n* = $14 + 8 * n +$ *time-bitsize-tm-bound* (*n div 2*)

lemma *time-bitsize-tm-aux*:
time (*bitsize-tm n*) ≤ *time-bitsize-tm-bound n*
apply (*induction n rule*: *bitsize-tm.induct*)
subgoal by *simp*
subgoal for *n* **using** *time-divide-nat-tm-le*[*of Suc n 2*] **by** *simp*
done

lemma *time-bitsize-tm-aux2*: *time-bitsize-tm-bound n* $\leq (2 * 8 + 4 * 14) * n + 23$
apply (*intro div-2-recursion-linear*)
using *less-iff-Suc-add* **by** *auto*

lemma *time-bitsize-tm-le*: *time* (*bitsize-tm n*) $\leq 72 * n + 23$
using *order.trans*[*OF time-bitsize-tm-aux time-bitsize-tm-aux2*] **by** *simp*

9.4.1 The *is-power-of-2* function

fun *is-power-of-2-tm* :: *nat* ⇒ *bool tm* **where**

```
is-power-of-2-tm 0 =1 return False
| is-power-of-2-tm (Suc 0) =1 return True
| is-power-of-2-tm n =1 do {
    n-mod-2 ← n mod_t 2;
    n-div-2 ← n div_t 2;
    c1 ← n-mod-2 =_t 0;
    c2 ← is-power-of-2-tm n-div-2;
    c1 ∧_t c2
  }

lemma val-is-power-of-2-tm[simp, val-simp]: val (is-power-of-2-tm n) = is-power-of-2
n
  by (induction n rule: is-power-of-2-tm.induct) simp-all

lemma time-is-power-of-2-tm-le: time (is-power-of-2-tm n) ≤ 114 ∗ n + 1
proof −
  have time (is-power-of-2-tm n) ≤ (2 ∗ 25 + 4 ∗ 16) ∗ n + 1
    apply (intro div-2-recursion-linear)
    subgoal by simp
    subgoal by simp
    subgoal premises prems for n
    proof −
      from prems obtain n′ where n = Suc (Suc n′)
        by (metis Suc-diff-1 Suc-diff-Suc order-less-trans zero-less-one)

      then have time (is-power-of-2-tm n) =
          time (n mod_t 2) +
          time (n div_t 2) +
          time (is-power-of-2-tm (n div 2)) + 3
        by (simp add: time-equal-nat-tm)
      also have ... ≤ 16 ∗ n + time (is-power-of-2-tm (n div 2)) + 25
        apply (estimation estimate: time-mod-nat-tm-le)
        apply (estimation estimate: time-divide-nat-tm-le)
        apply simp
        done
      finally show ?thesis by simp
    qed
    done
  then show ?thesis by simp
qed

definition next-power-of-2-tm :: nat ⇒ nat tm where
next-power-of-2-tm n =1 do {
  b ← is-power-of-2-tm n;
  if b then return n else do {
    r ← bitsize-tm n;
    2 ^_t r
  }
}
```

lemma *val-next-power-of-2-tm*[*simp*, *val-simp*]: *val* (*next-power-of-2-tm n*) = *next-power-of-2 n*
  by (*simp add*: *next-power-of-2-tm-def*)

lemma *time-next-power-of-2-tm-le*: *time* (*next-power-of-2-tm n*) ≤ *208* ∗ *n* + *37*
proof (*cases is-power-of-2 n*)
  case *True*
  then show *?thesis*
    using *time-is-power-of-2-tm-le*[*of n*]
    by (*simp add*: *next-power-of-2-tm-def*)
next
  case *False*
  then have *time* (*next-power-of-2-tm n*) =
      time (*is-power-of-2-tm n*) +
      time (*bitsize-tm n*) +
      time (*power-nat-tm 2* (*bitsize n*)) + *1*
    by (*simp add*: *next-power-of-2-tm-def*)
  also have ... ≤ *186* ∗ *n* + *6* ∗ *2* ^ (*bitsize n*) + *5* ∗ *bitsize n* + *26*
    apply (*estimation estimate*: *time-is-power-of-2-tm-le*)
    apply (*estimation estimate*: *time-bitsize-tm-le*)
    apply (*estimation estimate*: *time-power-nat-tm-le*)
    by *simp*
  also have ... ≤ *186* ∗ *n* + *11* ∗ *2* ^ (*bitsize n*) + *26*
    by *simp*
  also have ... ≤ *208* ∗ *n* + *37*
    by (*estimation estimate*: *two-pow-bitsize-bound*) *simp*
  finally show *?thesis* **.**
qed

9.5 Addition

fun *bit-add-carry-tm* :: *bool* ⇒ *bool* ⇒ *bool* ⇒ (*bool* × *bool*) *tm* **where**
bit-add-carry-tm False False False =*1* *return* (*False*, *False*)
| *bit-add-carry-tm False False True* =*1* *return* (*True*, *False*)
| *bit-add-carry-tm False True False* =*1* *return* (*True*, *False*)
| *bit-add-carry-tm False True True* =*1* *return* (*False*, *True*)
| *bit-add-carry-tm True False False* =*1* *return* (*True*, *False*)
| *bit-add-carry-tm True False True* =*1* *return* (*False*, *True*)
| *bit-add-carry-tm True True False* =*1* *return* (*False*, *True*)
| *bit-add-carry-tm True True True* =*1* *return* (*True*, *True*)

lemma *val-bit-add-carry-tm*[*simp*, *val-simp*]: *val* (*bit-add-carry-tm x y z*) = *bit-add-carry x y z*
  by (*induction x y z rule*: *bit-add-carry-tm.induct*; *simp*)
lemma *time-bit-add-carry-tm*[*simp*]: *time* (*bit-add-carry-tm x y z*) = *1*
  by (*induction x y z rule*: *bit-add-carry-tm.induct*; *simp*)

fun *inc-nat-tm* :: *nat-lsbf* ⇒ *nat-lsbf tm* **where**

inc-nat-tm [] =1 *return* [*True*]
| *inc-nat-tm* (*False* # *xs*) =1 *return* (*True* # *xs*)
| *inc-nat-tm* (*True* # *xs*) =1 *do* {
    r ← *inc-nat-tm xs*;
    return (*False* # *r*)
  }

lemma *val-inc-nat-tm*[*simp*, *val-simp*]: *val* (*inc-nat-tm xs*) = *inc-nat xs*
  by (*induction xs rule*: *inc-nat-tm.induct*) *simp-all*

lemma *time-inc-nat-tm-le*: *time* (*inc-nat-tm xs*) ≤ *length xs* + *1*
  by (*induction xs rule*: *inc-nat-tm.induct*) *simp-all*

fun *add-carry-tm* :: *bool* ⇒ *nat-lsbf* ⇒ *nat-lsbf* ⇒ *nat-lsbf tm* **where**
add-carry-tm False [] *y* =1 *return y*
| *add-carry-tm False* (*x* # *xs*) [] =1 *return* (*x* # *xs*)
| *add-carry-tm True* [] *y* =1 *do* {
    r ← *inc-nat-tm y*;
    return r
  }
| *add-carry-tm True* (*x* # *xs*) [] =1 *do* {
    r ← *inc-nat-tm* (*x* # *xs*);
    return r
  }
| *add-carry-tm c* (*x* # *xs*) (*y* # *ys*) =1 *do* {
    (*a*, *b*) ← *bit-add-carry-tm c x y*;
    r ← *add-carry-tm b xs ys*;
    return (*a* # *r*)
  }

lemma *val-add-carry-tm*[*simp*, *val-simp*]: *val* (*add-carry-tm c xs ys*) = *add-carry c xs ys*
  by (*induction c xs ys rule*: *add-carry-tm.induct*) (*simp-all split*: *prod.splits*)

lemma *time-add-carry-tm-le*: *time* (*add-carry-tm c xs ys*) ≤ *2* ∗ *max* (*length xs*) (*length ys*) + *2*
proof (*induction c xs ys rule*: *add-carry-tm.induct*)
  case (*3 y*)
  then show *?case* **using** *time-inc-nat-tm-le*[*of y*] **by** *simp*
next
  case (*4 x xs*)
  then show *?case* **using** *time-inc-nat-tm-le*[*of x* # *xs*] **by** *simp*
qed (*simp-all split*: *prod.splits*)

definition *add-nat-tm* :: *nat-lsbf* ⇒ *nat-lsbf* ⇒ *nat-lsbf tm* **where**
add-nat-tm xs ys =1 *do* {
  r ← *add-carry-tm False xs ys*;
  return r
}

lemma *val-add-nat-tm*[*simp*, *val-simp*]: *val* (*add-nat-tm xs ys*) = *xs* $+_n$ *ys*
by (*simp add*: *add-nat-tm-def add-nat-def*)

lemma *time-add-nat-tm-le*: *time* (*add-nat-tm xs ys*) ≤ *2* ∗ *max* (*length xs*) (*length ys*) + *3*
using *time-add-carry-tm-le*[*of - xs ys*] **by** (*simp add*: *add-nat-tm-def*)

9.6 Comparison and subtraction

fun *compare-nat-same-length-reversed-tm* :: *bool list* ⇒ *bool list* ⇒ *bool tm* **where**
compare-nat-same-length-reversed-tm [] [] =*1 return True*
| *compare-nat-same-length-reversed-tm* (*False* # *xs*) (*False* # *ys*) =*1 compare-nat-same-length-reversed-tm xs ys*
| *compare-nat-same-length-reversed-tm* (*True* # *xs*) (*False* # *ys*) =*1 return False*
| *compare-nat-same-length-reversed-tm* (*False* # *xs*) (*True* # *ys*) =*1 return True*
| *compare-nat-same-length-reversed-tm* (*True* # *xs*) (*True* # *ys*) =*1 compare-nat-same-length-reversed-tm xs ys*
| *compare-nat-same-length-reversed-tm - -* =*1 undefined*

lemma *val-compare-nat-same-length-reversed-tm*[*simp*, *val-simp*]:
assumes *length xs* = *length ys*
shows *val* (*compare-nat-same-length-reversed-tm xs ys*) = *compare-nat-same-length-reversed xs ys*
using *assms* **by** (*induction xs ys rule*: *compare-nat-same-length-reversed-tm.induct*) *simp-all*

lemma *time-compare-nat-same-length-reversed-tm-le*:
length xs = *length ys* ⟹ *time* (*compare-nat-same-length-reversed-tm xs ys*) ≤ *length xs* + *1*
by (*induction xs ys rule*: *compare-nat-same-length-reversed-tm.induct*) *simp-all*

fun *compare-nat-same-length-tm* :: *nat-lsbf* ⇒ *nat-lsbf* ⇒ *bool tm* **where**
compare-nat-same-length-tm xs ys =*1 do* {
rev-xs ← *rev-tm xs*;
rev-ys ← *rev-tm ys*;
compare-nat-same-length-reversed-tm rev-xs rev-ys
}

lemma *val-compare-nat-same-length-tm*[*simp*, *val-simp*]:
assumes *length xs* = *length ys*
shows *val* (*compare-nat-same-length-tm xs ys*) = *compare-nat-same-length xs ys*
using *assms* **by** *simp*

lemma *time-compare-nat-same-length-tm-le*:
length xs = *length ys* ⟹ *time* (*compare-nat-same-length-tm xs ys*) ≤ *3* ∗ *length xs* + *6*
using *time-compare-nat-same-length-reversed-tm-le*[*of rev xs rev ys*]
by *simp*

definition *make-same-length-tm* :: *nat-lsbf* ⇒ *nat-lsbf* ⇒ (*nat-lsbf* × *nat-lsbf*) *tm*
where
make-same-length-tm xs ys =*1 do* {
  len-xs ← *length-tm xs*;
  len-ys ← *length-tm ys*;
  n ← *max-nat-tm len-xs len-ys*;
  fill-xs ← *fill-tm n xs*;
  fill-ys ← *fill-tm n ys*;
  return (*fill-xs*, *fill-ys*)
}

lemma *val-make-same-length-tm*[*simp*, *val-simp*]: *val* (*make-same-length-tm xs ys*)
= *make-same-length xs ys*
 by (*simp add*: *make-same-length-tm-def make-same-length-def del*: *max-nat-tm.simps*)

lemma *time-make-same-length-tm-le*: *time* (*make-same-length-tm xs ys*) ≤ *10* ∗
max (*length xs*) (*length ys*) + *16*
proof −
  have *time* (*make-same-length-tm xs ys*) = *13* + *3* ∗ *length xs* + *3* ∗ *length ys* +
    (*time* (*max-nat-tm* (*length xs*) (*length ys*)) + *2* ∗ *max* (*length xs*) (*length ys*))
    by (*simp add*: *make-same-length-tm-def time-fill-tm del*: *max-nat-tm.simps*)
  also have ... ≤ *10* ∗ *max* (*length xs*) (*length ys*) + *16*
    using *time-max-nat-tm-le*[*of length xs length ys*] **by** *simp*
  finally show *?thesis* **.**
qed

definition *compare-nat-tm* :: *nat-lsbf* ⇒ *nat-lsbf* ⇒ *bool tm* **where**
compare-nat-tm xs ys =*1 do* {
  (*fill-xs*, *fill-ys*) ← *make-same-length-tm xs ys*;
  compare-nat-same-length-tm fill-xs fill-ys
}

lemma *val-compare-nat-tm*[*simp*, *val-simp*]: *val* (*compare-nat-tm xs ys*) = (*xs* $\leq_n$
ys)
  using *make-same-length-correct*[**where** *xs* = *xs* **and** *ys* = *ys*]
 by (*simp add*: *compare-nat-tm-def compare-nat-def del*: *compare-nat-same-length-tm.simps*
compare-nat-same-length.simps split: *prod.splits*)

lemma *time-compare-nat-tm-le*: *time* (*compare-nat-tm xs ys*) ≤ *13* ∗ *max* (*length*
xs) (*length ys*) + *23*
proof −
  obtain *fill-xs fill-ys* **where** *fills-defs*: *make-same-length xs ys* = (*fill-xs*, *fill-ys*)
by *fastforce*
  then have *time* (*compare-nat-tm xs ys*) = *time* (*make-same-length-tm xs ys*) +
      time (*compare-nat-same-length-tm fill-xs fill-ys*) + *1*
    by (*simp add*: *compare-nat-tm-def del*: *compare-nat-same-length-tm.simps*)
  also have ... ≤ (*10* ∗ *max* (*length xs*) (*length ys*) + *16*) +
      (*3* ∗ *max* (*length xs*) (*length ys*) + *6*) + *1*

```
    apply (intro add-mono order.refl time-make-same-length-tm-le)
    using time-compare-nat-same-length-tm-le[of fill-xs fill-ys]
    using make-same-length-correct[OF fills-defs[symmetric]] by argo
  finally show ?thesis by simp
qed

definition subtract-nat-tm :: nat-lsbf ⇒ nat-lsbf ⇒ nat-lsbf tm where
subtract-nat-tm xs ys =1 do {
  b ← compare-nat-tm xs ys;
  if b then return [] else do {
    (fill-xs, fill-ys) ← make-same-length-tm xs ys;
    fill-ys-comp ← map-tm Not-tm fill-ys;
    a ← add-carry-tm True fill-xs fill-ys-comp;
    butlast-tm a
  }
}

lemma val-subtract-nat-tm[simp, val-simp]: val (subtract-nat-tm xs ys) = xs −n ys
  by (simp add: subtract-nat-tm-def subtract-nat-def Let-def split: prod.splits)

lemma time-map-tm-Not-tm: time (map-tm Not-tm xs) = 2 ∗ length xs + 1
  using time-map-tm-constant[of xs Not-tm 1] by simp

lemma time-subtract-nat-tm-le: time (subtract-nat-tm xs ys) ≤ 30 ∗ max (length
xs) (length ys) + 48
proof −
  obtain x1 x2 where x12: make-same-length xs ys = (x1, x2) by fastforce
  note x12-simps = make-same-length-correct[OF x12[symmetric]]
  then have max12: max (length x1) (length x2) = max (length xs) (length ys)
    by simp
  show ?thesis
  proof (cases compare-nat xs ys)
    case True
    then show ?thesis
      using time-compare-nat-tm-le[of xs ys]
      by (simp add: subtract-nat-tm-def)
  next
    case False
    then have time (subtract-nat-tm xs ys) =
        Suc (time (compare-nat-tm xs ys) +
             (time (make-same-length-tm xs ys) +
              (time (map-tm Not-tm x2) +
               (time (add-carry-tm True x1 (map Not x2)) +
                (time (butlast-tm (add-carry True x1 (map Not x2)))))))
      by (simp add: subtract-nat-tm-def x12)
    also have ... ≤ 30 ∗ max (length xs) (length ys) + 48
      apply (subst Suc-eq-plus1)
      apply (estimation estimate: time-compare-nat-tm-le)
      apply (estimation estimate: time-make-same-length-tm-le)
```

apply (*subst time-map-tm-Not-tm*)
apply (*estimation estimate*: *time-add-carry-tm-le*)
apply (*estimation estimate*: *time-butlast-tm-le*)
apply (*estimation estimate*: *time-inc-nat-tm-le*)
apply (*estimation estimate*: *length-add-carry-upper*)
apply (*subst length-map*)+
apply (*subst max12*)+
apply (*subst x12-simps*)+
apply *simp*
done
finally show *?thesis* **.**
qed
qed

9.7 (Grid) Multiplication

fun *grid-mul-nat-tm* :: *nat-lsbf* ⇒ *nat-lsbf* ⇒ *nat-lsbf tm* **where**
grid-mul-nat-tm [] *ys* =*1 return* []
| *grid-mul-nat-tm* (*False* # *xs*) *ys* =*1 do* {
r ← *grid-mul-nat-tm xs ys*;
return (*False* # *r*)
}
| *grid-mul-nat-tm* (*True* # *xs*) *ys* =*1 do* {
r ← *grid-mul-nat-tm xs ys*;
add-nat-tm (*False* # *r*) *ys*
}

lemma *val-grid-mul-nat-tm*[*simp*, *val-simp*]: *val* (*grid-mul-nat-tm xs ys*) = *xs* $*_n$ *ys*
by (*induction xs ys rule*: *grid-mul-nat-tm.induct*) *simp-all*

lemma *euler-sum-bound*: $\sum$ {..(*n*::*nat*)} ≤ *n* ∗ *n*
by (*induction n*) *simp-all*

lemma *time-grid-mul-nat-tm-le*:
time (*grid-mul-nat-tm xs ys*) ≤ *8* ∗ *length xs* ∗ *max* (*length xs*) (*length ys*) + *1*
proof −
have *time* (*grid-mul-nat-tm xs ys*) ≤ *2* ∗ ($\sum$ {..*length xs*}) + *length xs* ∗ (*2* ∗ *length ys* + *4*) + *1*
proof (*induction xs ys rule*: *grid-mul-nat-tm.induct*)
case (*1 ys*)
then show *?case* **by** *simp*
next
case (*2 xs ys*)
then show *?case* **by** *simp*
next
case (*3 xs ys*)
then have *time* (*grid-mul-nat-tm* (*True* # *xs*) *ys*) ≤
time (*grid-mul-nat-tm xs ys*) +

time (*add-nat-tm* (*False* # *grid-mul-nat xs ys*) *ys*) + *1* (**is** *?l* ≤ *?i* + - + *1*)
by *simp*
also have ... ≤ *?i* + *2* ∗ *max* (*1* + *length* (*grid-mul-nat xs ys*)) (*length ys*) + *4*
by (*estimation estimate*: *time-add-nat-tm-le*) *simp*
also have ... ≤ *?i* + *2* ∗ (*length xs* + *length ys* + *1*) + *4*
apply (*estimation estimate*: *length-grid-mul-nat*[*of xs ys*])
by (*simp-all add*: *length-grid-mul-nat*)
also have ... = *?i* + *2* ∗ (*length* (*True* # *xs*)) + *2* ∗ *length ys* + *4*
by *simp*
also have ... ≤ *2* ∗ (∑ {..*length* (*True* # *xs*)}) + *length* (*True* # *xs*) ∗ (*2* ∗ *length ys* + *4*) + *1*
using *3* **by** *simp*
finally show *?case* **.**
qed
also have ... ≤ *2* ∗ *length xs* ∗ *length xs* + *2* ∗ *length xs* ∗ *length ys* + *4* ∗ *length xs* + *1*
by (*estimation estimate*: *euler-sum-bound*) (*simp add*: *distrib-left*)
also have ... ≤ *6* ∗ *length xs* ∗ *length xs* + *2* ∗ *length xs* ∗ *length ys* + *1*
by (*simp add*: *leI*)
also have ... ≤ *8* ∗ *length xs* ∗ *max* (*length xs*) (*length ys*) + *1*
by (*simp add*: *add.commute add-mult-distrib nat-mult-max-right*)
finally show *?thesis* **.**
qed

9.8 Syntax bundles

abbreviation *shift-right-tm-flip* **where** *shift-right-tm-flip xs n* ≡ *shift-right-tm n xs*

bundle *nat-lsbf-tm-syntax*
begin
notation *add-nat-tm* (**infixl** $+_{nt}$ *65*)
notation *compare-nat-tm* (**infixl** $\leq_{nt}$ *50*)
notation *subtract-nat-tm* (**infixl** $-_{nt}$ *65*)
notation *grid-mul-nat-tm* (**infixl** $*_{nt}$ *70*)
notation *shift-right-tm-flip* (**infixl** $>>_{nt}$ *55*)
end

bundle *no-nat-lsbf-tm-syntax*
begin
no-notation *add-nat-tm* (**infixl** $+_{nt}$ *65*)
no-notation *compare-nat-tm* (**infixl** $\leq_{nt}$ *50*)
no-notation *subtract-nat-tm* (**infixl** $-_{nt}$ *65*)
no-notation *grid-mul-nat-tm* (**infixl** $*_{nt}$ *70*)
no-notation *shift-right-tm-flip* (**infixl** $>>_{nt}$ *55*)
end

unbundle *nat-lsbf-tm-syntax*

end
theory *Int-LSBF*
  imports *Nat-LSBF HOL−Algebra.IntRing*
begin

10 Representing *int* in LSBF

10.1 Type definition

datatype *sign* = *Positive* | *Negative*
type-synonym *int-lsbf* = *sign* × *nat-lsbf*

10.2 Conversions

fun *from-int* :: *int* ⇒ *int-lsbf* **where**
from-int x = (*if x* ≥ *0 then* (*Positive*, *from-nat* (*nat x*)) *else* (*Negative*, *from-nat* (*nat* (−*x*))))
fun *to-int* :: *int-lsbf* ⇒ *int* **where**
to-int (*Positive*, *xs*) = *int* (*to-nat xs*)
| *to-int* (*Negative*, *xs*) = − *int* (*to-nat xs*)

lemma *to-int-from-int*[*simp*]: *to-int* (*from-int x*) = *x*
  by (*cases x* ≥ *0*) *simp-all*

fun *truncate-int* :: *int-lsbf* ⇒ *int-lsbf* **where**
truncate-int (*Positive*, *xs*) = (*Positive*, *truncate xs*)
| *truncate-int* (*Negative*, *xs*) = (*let ys* = *truncate xs in if ys* = [] *then* (*Positive*, []) *else* (*Negative*, *ys*))

lemma *to-int-truncate*[*simp*]: *to-int* (*truncate-int xs*) = *to-int xs*
  by (*induction xs rule*: *truncate-int.induct*) (*simp-all add*: *Let-def to-nat-zero-iff*)

lemma *truncate-from-int*[*simp*]: *truncate-int* (*from-int x*) = *from-int x*
  apply (*cases x* ≥ *0*)
  subgoal by *simp*
  subgoal unfolding *Let-def*
  proof −
    assume ¬ *x* ≥ *0*
    then have *to-nat* (*from-nat* (*nat* (− *x*))) > *0* **by** *simp*
    then have *truncate* (*from-nat* (*nat* (− *x*))) ≠ [] **using** *to-nat-zero-iff nless-le* **by** *blast*
    then show *?thesis* **by** *simp*
  qed
  done

lemma *pos-and-neg-imp-zero*:
  assumes *to-int* (*Positive*, *x*) = *to-int* (*Negative*, *y*)
  shows *to-nat x* = *0* ∧ *to-nat y* = *0*
proof −

have *to-int* (*Positive*, *x*) ≥ *0* *to-int* (*Negative*, *y*) ≤ *0* **by** *simp-all*
with *assms* **have** *to-int* (*Positive*, *x*) = *0* *to-int* (*Negative*, *y*) = *0* **by** *simp-all*
thus *?thesis* **by** *simp-all*
qed

lemma *to-int-eq-imp-truncate-int-eq*: *to-int* (*a*, *x*) = *to-int* (*b*, *y*) ⟹ *truncate-int* (*a*, *x*) = *truncate-int* (*b*, *y*)
apply (*cases a*; *cases b*)
subgoal by (*simp add*: *to-nat-eq-imp-truncate-eq*[*of x y*])
subgoal
using *pos-and-neg-imp-zero*[*of x y*] *to-nat-zero-iff*
by *fastforce*
subgoal using *to-nat-zero-iff* **by** (*simp add*: *Let-def*)
subgoal by (*simp add*: *to-nat-eq-imp-truncate-eq*[*of x y*])
done

lemma *from-int-to-int*: *from-int* ∘ *to-int* = *truncate-int*
proof −
have (⋀*x y*. *to-int x* = *to-int y* ⟹ *truncate-int x* = *truncate-int y*)
using *to-int-eq-imp-truncate-int-eq* **by** *auto*
thus *?thesis*
using *from-to-f-criterion*[*of to-int from-int truncate-int*]
using *truncate-from-int to-int-from-int*
using *comp-apply*
by *fastforce*
qed

interpretation *int-lsbf*: *abstract-representation from-int to-int truncate-int*
proof
show *to-int* ∘ *from-int* = *id*
using *to-int-from-int comp-apply* **by** *fastforce*
next
show *from-int* ∘ *to-int* = *truncate-int*
using *from-int-to-int comp-apply* **by** *fastforce*
qed

10.3 Addition

fun *add-int* :: *int-lsbf* ⇒ *int-lsbf* ⇒ *int-lsbf* **where**
add-int (*Negative*, *xs*) (*Negative*, *ys*) = (*Negative*, *add-nat xs ys*)
| *add-int* (*Positive*, *xs*) (*Positive*, *ys*) = (*Positive*, *add-nat xs ys*)
| *add-int* (*Positive*, *xs*) (*Negative*, *ys*) = (*if compare-nat xs ys then* (*Negative*, *subtract-nat ys xs*) *else* (*Positive*, *subtract-nat xs ys*))
| *add-int* (*Negative*, *xs*) (*Positive*, *ys*) = (*if compare-nat xs ys then* (*Positive*, *subtract-nat ys xs*) *else* (*Negative*, *subtract-nat xs ys*))

lemma *add-int-correct*: *to-int* (*add-int x y*) = *to-int x* + *to-int y*
apply (*induction x y rule*: *add-int.induct*)
subgoal by (*simp add*: *add-nat-correct*)

subgoal by (*simp add*: *add-nat-correct*)
apply (*auto simp only*: *add-int.simps compare-nat-correct subtract-nat-correct to-int.simps split*: *if-splits*)
done

fun *nat-mul-to-int-mul* :: (*nat-lsbf* ⇒ *nat-lsbf* ⇒ *nat-lsbf*) ⇒ *int-lsbf* ⇒ *int-lsbf* ⇒ *int-lsbf* **where**
nat-mul-to-int-mul f (*x*, *xs*) (*y*, *ys*) = ((*if x* = *y then Positive else Negative*), *f xs ys*)

lemma *nat-mul-to-int-mul-correct*:
assumes ⋀*x y*. *to-nat* (*f x y*) = *to-nat x* ∗ *to-nat y*
shows ⋀*x y xs ys*. *to-int* (*nat-mul-to-int-mul f* (*x*, *xs*) (*y*, *ys*)) = *to-int* (*x*, *xs*) ∗ *to-int* (*y*, *ys*)
subgoal for *x y xs ys*
by (*cases x*; *cases y*) (*simp-all add*: *assms*)
done

10.4 Grid Multiplication

fun *grid-mul-int* **where** *grid-mul-int x y* = *nat-mul-to-int-mul grid-mul-nat x y*

corollary *grid-mul-int-correct*: *to-int* (*grid-mul-int x y*) = *to-int x* ∗ *to-int y*
using *nat-mul-to-int-mul-correct*[*OF grid-mul-nat-correct*]
by (*metis grid-mul-int.elims surj-pair*)

end

11 Karatsuba Multiplication

theory *Karatsuba*
imports *../Binary-Representations/Nat-LSBF ../Binary-Representations/Int-LSBF ../Estimation-Method*
begin

This theory contains an implementation of the Karatsuba Multiplication on type *nat-lsbf*.

definition *abs-diff* :: *nat-lsbf* ⇒ *nat-lsbf* ⇒ *nat-lsbf* **where**
abs-diff x y = $(x -_n y) +_n (y -_n x)$

lemma *abs-diff-correct*: *int* (*to-nat* (*abs-diff x y*)) = *abs* (*int* (*to-nat x*) − *int* (*to-nat y*))
unfolding *abs-diff-def* **by** (*simp add*: *add-nat-correct subtract-nat-correct*)

lemma *abs-diff-length*: *length* (*abs-diff xs ys*) ≤ *max* (*length xs*) (*length ys*)
proof (*cases compare-nat xs ys*)
case *True*
then have $xs -_n ys$ = [] **by** (*simp add*: *subtract-nat-def*)
then have *abs-diff xs ys* = $ys -_n xs$ **by** (*simp add*: *abs-diff-def add-nat-def*)

then show *?thesis* **using** *length-subtract-nat-le*[*of ys xs*] **by** *simp*
next
case *False*
then have $ys \leq_n xs$ **by** (*simp only*: *compare-nat-correct*)
then have $ys -_n xs = []$ **by** (*simp add*: *subtract-nat-def*)
then have *abs-diff xs ys* $=$ $xs -_n ys$ **by** (*simp add*: *abs-diff-def add-nat-com add-nat-def*)
then show *?thesis* **using** *length-subtract-nat-le*[*of xs ys*] **by** *simp*
qed

For small inputs, implementations of Karatsuba Multiplication usually switch to grid multiplication. The threshold does not matter for the asymptotic running time, hence we will just arbitrarily choose *42*.

definition *karatsuba-lower-bound* :: *nat* **where**
karatsuba-lower-bound ≡ *42*

lemma *karatsuba-lower-bound-requirement*:
karatsuba-lower-bound ≥ *1*
unfolding *karatsuba-lower-bound-def* **by** *simp*

A first version of the algorithm assumes the input numbers have a length which is a power of 2. The function *karatsuba-on-power-of-2-length* takes the specified length as additional first argument.

fun *karatsuba-on-power-of-2-length* :: *nat* ⇒ *nat-lsbf* ⇒ *nat-lsbf* ⇒ *nat-lsbf* **where**

```
karatsuba-on-power-of-2-length k x y =
(if k ≤ karatsuba-lower-bound
then x *n y
else let
    (x0, x1) = split x;
    (y0, y1) = split y;
    k-div-2 = (k div 2);
    prod0 = karatsuba-on-power-of-2-length k-div-2 x0 y0;
    prod1 = karatsuba-on-power-of-2-length k-div-2 x1 y1;
    prod2 = karatsuba-on-power-of-2-length k-div-2
      (fill k-div-2 (abs-diff x0 x1))
      (fill k-div-2 (abs-diff y0 y1));
    add01 = prod0 +n prod1;
    r = (if (x1 ≤n x0) = (y1 ≤n y0)
        then add01 −n prod2
        else add01 +n prod2)
  in prod0 +n (r >>n k-div-2) +n (prod1 >>n k))
```

declare *karatsuba-on-power-of-2-length.simps*[*simp del*]

locale *karatsuba-context* =
fixes *k l* :: *nat*
fixes *x y* :: *nat-lsbf*
assumes *k-power-of-2*: *k* = *2* ^ *l*
assumes *length-x*: *length x* = *k*

assumes *length-y*: *length y* = *k*
assumes *recursion-condition*: ¬ *k* ≤ *karatsuba-lower-bound*
begin

definition *x0* **where** *x0* = *fst* (*split x*)
definition *x1* **where** *x1* = *snd* (*split x*)
definition *y0* **where** *y0* = *fst* (*split y*)
definition *y1* **where** *y1* = *snd* (*split y*)
definition *k-div-2* **where** *k-div-2* = *k div 2*
definition *prod0* **where** *prod0* = *karatsuba-on-power-of-2-length k-div-2 x0 y0*
definition *prod1* **where** *prod1* = *karatsuba-on-power-of-2-length k-div-2 x1 y1*
definition *prod2* **where** *prod2* = *karatsuba-on-power-of-2-length k-div-2*
(*fill k-div-2* (*abs-diff x0 x1*))
(*fill k-div-2* (*abs-diff y0 y1*))
definition *add01* **where** *add01* = *prod0* $+_n$ *prod1*
definition *r* **where** *r* = (*if* (*x1* $\leq_n$ *x0*) = (*y1* $\leq_n$ *y0*)
then add01 $-_n$ *prod2*
else add01 $+_n$ *prod2*)

lemma *split-x*: *split x* = (*x0*, *x1*) **using** *x0-def x1-def* **by** *simp*
lemma *split-y*: *split y* = (*y0*, *y1*) **using** *y0-def y1-def* **by** *simp*

lemmas *defs1* = *split-x split-y*
lemmas *defs2* = *prod0-def prod1-def prod2-def k-div-2-def add01-def r-def*

lemma *recursive*: *karatsuba-on-power-of-2-length k x y* =
prod0 $+_n$ (*r* $>>_n$ *k-div-2*) $+_n$ (*prod1* $>>_n$ *k*)
unfolding *karatsuba-on-power-of-2-length.simps*[*of k x y*]
using *defs1 defs2 recursion-condition*
by (*simp only*: *if-False Let-def case-prod-conv*)

lemma *l-ge-1*: *l* ≥ *1*
using *karatsuba-lower-bound-requirement recursion-condition k-power-of-2*
by (*cases l*; *simp*)

lemma *k-even*: *k mod 2* = *0*
using *k-power-of-2 l-ge-1* **by** *simp*

lemma *k-div-2*: *k-div-2* = *2* ^ (*l* − *1*)
unfolding *k-div-2-def* **using** *k-power-of-2 l-ge-1* **by** (*simp add*: *power-diff*)

lemma *k-div-2-less-k*: *k-div-2* < *k*
unfolding *k-div-2-def* **using** *k-power-of-2* **by** *simp*

lemma *length-x-split*: *length x0* = *k-div-2 length x1* = *k-div-2*
unfolding *k-div-2-def* **using** *k-even length-split*[*OF - split-x*] *length-x* **by** *argo*+

lemma *length-y-split*: *length y0* = *k-div-2 length y1* = *k-div-2*
unfolding *k-div-2-def* **using** *k-even length-split*[*OF - split-y*] *length-y* **by** *argo*+

lemma *length-abs-diff-x0-x1*: *length* (*abs-diff x0 x1*) ≤ *k-div-2*
  using *abs-diff-length*[*of x0 x1*] *length-x-split* **by** *simp*
lemma *length-fill-abs-diff-x0-x1*: *length* (*fill k-div-2* (*abs-diff x0 x1*)) = *k-div-2*
  by (*intro length-fill length-abs-diff-x0-x1*)

lemma *length-abs-diff-y0-y1*: *length* (*abs-diff y0 y1*) ≤ *k-div-2*
  using *abs-diff-length*[*of y0 y1*] *length-y-split* **by** *simp*
lemma *length-fill-abs-diff-y0-y1*: *length* (*fill k-div-2* (*abs-diff y0 y1*)) = *k-div-2*
  by (*intro length-fill length-abs-diff-y0-y1*)

lemmas *IH-prems1* = *recursion-condition split-x*[*symmetric*] *refl split-y*[*symmetric*]
refl k-div-2-def
    k-div-2 length-x-split(*1*) *length-y-split*(*1*)

lemmas *IH-prems2* = *recursion-condition split-x*[*symmetric*] *refl split-y*[*symmetric*]
refl k-div-2-def
    prod0-def k-div-2 length-x-split(*2*) *length-y-split*(*2*)

lemmas *IH-prems3* = *recursion-condition split-x*[*symmetric*] *refl split-y*[*symmetric*]
refl k-div-2-def
    prod0-def prod1-def k-div-2 length-fill-abs-diff-x0-x1 length-fill-abs-diff-y0-y1

end

lemma *karatsuba-on-power-of-2-length-correct*:
  assumes *k* = *2* ^ *l*
  assumes *length x* = *k length y* = *k*
  shows *to-nat* (*karatsuba-on-power-of-2-length k x y*) = *to-nat x* ∗ *to-nat y*
using *assms* **proof** (*induction k x y arbitrary*: *l rule*: *karatsuba-on-power-of-2-length.induct*)
  case (*1 k x y l*)
  show *?case*
  proof (*cases k* ≤ *karatsuba-lower-bound*)
    case *True*
    then show *?thesis*
      unfolding *karatsuba-on-power-of-2-length.simps*[*of k x y*]
      by (*simp add*: *grid-mul-nat-correct*)
  next
    case *False*
    then interpret *r*: *karatsuba-context k l x y* **using** *1.prems*
      by (*unfold-locales*; *simp*)
    from *r.l-ge-1* **obtain** *l′* **where** *l* = *Suc l′*
      by (*metis less-eqE plus-1-eq-Suc*)
    then have *k div 2* = *2* ^ *l′* **using** ‹*k* = *2* ^ *l*› **by** *simp*

    have *to-nat-x*: *to-nat x* = *to-nat r.x0* + *2* ^ (*k div 2*) ∗ *to-nat r.x1*
      unfolding *r.k-div-2-def*[*symmetric*]
     using *app-split*[*OF r.split-x*] *to-nat-app*[*of r.x0 r.x1*] *r.length-x-split* **by** *algebra*

have *to-nat-y*: *to-nat y* = *to-nat r.y0* + *2* ^ (*k div 2*) ∗ *to-nat r.y1*
unfolding *r.k-div-2-def*[*symmetric*]
using *app-split*[*OF r.split-y*] *to-nat-app*[*of r.y0 r.y1*] *r.length-y-split* **by** *algebra*

have *4*: *to-nat r.prod0* = *to-nat r.x0* ∗ *to-nat r.y0*
unfolding *r.prod0-def*
by (*intro 1*(*1*)[*OF r.IH-prems1*])
have *5*: *to-nat r.prod1* = *to-nat r.x1* ∗ *to-nat r.y1*
unfolding *r.prod1-def*
by (*intro 1*(*2*)[*OF r.IH-prems2*])
have *to-nat r.prod2* = *to-nat* (*fill r.k-div-2* (*abs-diff r.x0 r.x1*)) ∗ *to-nat* (*fill r.k-div-2* (*abs-diff r.y0 r.y1*))
unfolding *r.prod2-def*
by (*intro 1*(*3*)[*OF r.IH-prems3*])
hence *int* (*to-nat r.prod2*) = *abs* (*int* (*to-nat r.x0*) − *int* (*to-nat r.x1*)) ∗ *abs* (*int* (*to-nat r.y0*) − *int* (*to-nat r.y1*))
using *abs-diff-correct* **by** *simp*
then have *int* (*to-nat r.prod2*) = *abs* ((*int* (*to-nat r.x0*) − *int* (*to-nat r.x1*)) ∗ (*int* (*to-nat r.y0*) − *int* (*to-nat r.y1*)))
by (*subst abs-mult*, *assumption*)
then have *6*: (*if* (*compare-nat r.x1 r.x0*) = (*compare-nat r.y1 r.y0*) *then int* (*to-nat r.prod2*) *else* − *int* (*to-nat r.prod2*)) = (*int* (*to-nat r.x0*) − *int* (*to-nat r.x1*)) ∗ (*int* (*to-nat r.y0*) − *int* (*to-nat r.y1*))
apply (*cases to-nat r.x0* ≥ *to-nat r.x1*; *cases to-nat r.y0* ≥ *to-nat r.y1*)
by (*simp-all add*: *compare-nat-correct mult-nonneg-nonpos mult-nonneg-nonpos2 mult-nonpos-nonpos*)

have *7*: *int* (*to-nat r.r*) = *int* (*to-nat r.x0*) ∗ *int* (*to-nat r.y1*) + *int* (*to-nat r.x1*) ∗ *int* (*to-nat r.y0*)
proof (*cases* (*r.x1* $\leq_n$ *r.x0*) = (*r.y1* $\leq_n$ *r.y0*))
case *True*
then have *int-p*: *int* (*to-nat r.r*) = *int* (*to-nat r.prod0* + *to-nat r.prod1* − *to-nat r.prod2*)
unfolding *r.r-def r.add01-def*
by (*simp add*: *subtract-nat-correct add-nat-correct*)
have *int-prod2*: *int* (*to-nat r.prod2*) = (*int* (*to-nat r.x0*) − *int* (*to-nat r.x1*)) ∗ (*int* (*to-nat r.y0*) − *int* (*to-nat r.y1*))
using *6 True* **by** *simp*
have − (*int* (*to-nat r.x0*) ∗ *int* (*to-nat r.y1*)) ≤ *int* (*to-nat r.x1*) ∗ *int* (*to-nat r.y0*)
apply (*intro order.trans*[*of* − (*int* (*to-nat r.x0*) ∗ *int* (*to-nat r.y1*)) *0 int* (*to-nat r.x1*) ∗ *int* (*to-nat r.y0*)])
by *simp-all*
then have *to-nat r.prod0* + *to-nat r.prod1* ≥ *to-nat r.prod2*
apply (*intro iffD1*[*OF zle-int*])
by (*simp add*: *4 5 int-prod2 left-diff-distrib right-diff-distrib*)
then have *int* (*to-nat r.r*) = *int* (*to-nat r.prod0*) + *int* (*to-nat r.prod1*) − *int* (*to-nat r.prod2*)
using *int-p* **by** *simp*

then show *?thesis* **using** *int-prod2* **by** (*simp add*: *left-diff-distrib right-diff-distrib 4 5*)
next
case *False*
then have *int* (*to-nat r.r*) = *int* (*to-nat r.prod0*) + *int* (*to-nat r.prod1*) + *int* (*to-nat r.prod2*)
unfolding *r.r-def*
by (*simp add*: *add-nat-correct r.add01-def*)
moreover from *False 6* **have** − *int* (*to-nat r.prod2*) = (*int* (*to-nat r.x0*) − *int* (*to-nat r.x1*)) ∗ (*int* (*to-nat r.y0*) − *int* (*to-nat r.y1*))
by *simp*
then have *int* (*to-nat r.prod2*) = − (*int* (*to-nat r.x0*) − *int* (*to-nat r.x1*)) ∗ (*int* (*to-nat r.y0*) − *int* (*to-nat r.y1*))
by *linarith*
ultimately show *?thesis* **by** (*simp add*: *4 5 left-diff-distrib right-diff-distrib*)
qed

from *r.recursive* **have** *int* (*to-nat* (*karatsuba-on-power-of-2-length k x y*)) =
int (*to-nat* (*r.prod0* $+_n$ (*r.r* $>>_n$ *r.k-div-2*) $+_n$ (*r.prod1* $>>_n$ *k*))) **by** *simp*
also have ... = *int* (*to-nat r.prod0*) + *int* (*to-nat* (*shift-right r.k-div-2 r.r*)) + *int* (*to-nat* (*shift-right k r.prod1*))
by (*simp add*: *add-nat-correct*)
also have ... = *int* (*to-nat r.prod0*) + *int* (2 ^ (*k div 2*) ∗ *to-nat r.r*) + *int* (2 ^ *k* ∗ *to-nat r.prod1*)
by (*simp only*: *to-nat-shift-right r.k-div-2-def*)
also have ... = *int* (*to-nat r.prod0*) + 2 ^ (*k div 2*) ∗ *int* (*to-nat r.r*) + 2 ^ *k* ∗ *int* (*to-nat r.prod1*)
by *simp*
also have ... = *int* (*to-nat r.x0*) ∗ *int* (*to-nat r.y0*) + 2 ^ (*k div 2*) ∗ (*int* (*to-nat r.x0*) ∗ *int* (*to-nat r.y1*) + *int* (*to-nat r.x1*) ∗ *int* (*to-nat r.y0*)) + 2 ^ *k* ∗ *int* (*to-nat r.x1*) ∗ *int* (*to-nat r.y1*)
using *7 4 5*
by *simp*
also have ... = (*int* (*to-nat r.x0*) + 2 ^ (*k div 2*) ∗ (*int* (*to-nat r.x1*)))
∗ (*int* (*to-nat r.y0*) + 2 ^ (*k div 2*) ∗ (*int* (*to-nat r.y1*)))
proof −
have 2 ∗ (*k div 2*) = *k*
using *r.k-even* **by** *force*
have (*int* (*to-nat r.x0*) + 2 ^ (*k div 2*) ∗ (*int* (*to-nat r.x1*)))
∗ (*int* (*to-nat r.y0*) + 2 ^ (*k div 2*) ∗ (*int* (*to-nat r.y1*)))
= *int* (*to-nat r.x0*) ∗ *int* (*to-nat r.y0*)
+ (2::*int*) ^ (*k div 2*) ∗ (*int* (*to-nat r.x1*)) ∗ (*int* (*to-nat r.y0*))
+ (*int* (*to-nat r.x0*)) ∗ 2 ^ (*k div 2*) ∗ (*int* (*to-nat r.y1*))
+ (2::*int*) ^ (*k div 2*) ∗ (*int* (*to-nat r.x1*)) ∗ 2 ^ (*k div 2*) ∗ (*int* (*to-nat r.y1*))
using *distrib-left*[*of* (*int* (*to-nat r.x0*) + 2 ^ (*k div 2*) ∗ (*int* (*to-nat r.x1*))) *int* (*to-nat r.y0*) 2 ^ (*k div 2*) ∗ (*int* (*to-nat r.y1*))]
by (*simp add*: *ring-class.ring-distribs*(2))

also have ... = *int* (*to-nat r.x0*) ∗ *int* (*to-nat r.y0*)
+ (*2*::*int*) ^ (*k div 2*) ∗ (*int* (*to-nat r.x1*)) ∗ (*int* (*to-nat r.y0*))
+ (*int* (*to-nat r.x0*)) ∗ *2* ^ (*k div 2*) ∗ (*int* (*to-nat r.y1*))
+ ((*2*::*int*) ^ (*k div 2*) ∗ *2* ^ (*k div 2*)) ∗ (*int* (*to-nat r.x1*)) ∗ (*int* (*to-nat r.y1*))
by *simp*
also have (*2*::*int*) ^ (*k div 2*) ∗ *2* ^ (*k div 2*) = *2* ^ *k*
using *power-add*[*of 2*::*int k div 2 k div 2*, *symmetric*]
using ‹*2* ∗ (*k div 2*) = *k*›
by *simp*
finally have (*int* (*to-nat r.x0*) + *2* ^ (*k div 2*) ∗ (*int* (*to-nat r.x1*)))
∗ (*int* (*to-nat r.y0*) + *2* ^ (*k div 2*) ∗ (*int* (*to-nat r.y1*)))
= *int* (*to-nat r.x0*) ∗ *int* (*to-nat r.y0*)
+ *2* ^ (*k div 2*) ∗ (*int* (*to-nat r.x1*)) ∗ (*int* (*to-nat r.y0*))
+ (*int* (*to-nat r.x0*)) ∗ *2* ^ (*k div 2*) ∗ (*int* (*to-nat r.y1*))
+ (*2*::*int*) ^ *k* ∗ (*int* (*to-nat r.x1*)) ∗ (*int* (*to-nat r.y1*)) **by** *simp*
also have ... = *int* (*to-nat r.x0*) ∗ *int* (*to-nat r.y0*)
+ ((*2*::*int*) ^ (*k div 2*) ∗ (*int* (*to-nat r.x1*)) ∗ (*int* (*to-nat r.y0*))
+ (*2*::*int*) ^ (*k div 2*) ∗ (*int* (*to-nat r.x0*)) ∗ (*int* (*to-nat r.y1*)))
+ (*2*::*int*) ^ *k* ∗ (*int* (*to-nat r.x1*)) ∗ (*int* (*to-nat r.y1*))
by *simp*
also have ... = *int* (*to-nat r.x0*) ∗ *int* (*to-nat r.y0*)
+ (*2*::*int*) ^ (*k div 2*) ∗ (*int* (*to-nat r.x1*) ∗ *int* (*to-nat r.y0*) + *int* (*to-nat r.x0*) ∗ *int* (*to-nat r.y1*))
+ (*2*::*int*) ^ *k* ∗ (*int* (*to-nat r.x1*)) ∗ (*int* (*to-nat r.y1*))
using *distrib-left*[*of* (*2*::*int*) ^ (*k div 2*)] **by** *simp*
finally show *?thesis* **by** *simp*
qed
also have ... = *int* (*to-nat x*) ∗ *int* (*to-nat y*)
by (*simp add*: *to-nat-x to-nat-y*)
finally have *int* (*to-nat* (*karatsuba-on-power-of-2-length k x y*)) = *int* (*to-nat x* ∗ *to-nat y*)
by *simp*
thus *?thesis* **by** *presburger*
qed
qed

function *len-kar-bound* **where**
len-kar-bound l = (*if 2* ^ *l* ≤ *karatsuba-lower-bound then 2* ∗ *karatsuba-lower-bound else 2* ^ *l* + *len-kar-bound* (*l* − *1*) + *4*)
by *pat-completeness auto*
termination
apply (*relation Wellfounded.measure* (λ*l*. *l*))
subgoal by *simp*
subgoal for *l*
using *karatsuba-lower-bound-requirement* **by** (*cases l*; *simp*)
done

declare *len-kar-bound.simps*[*simp del*]

lemma *length-karatsuba-on-power-of-2-aux*:
assumes $k = 2 \char`^ l$
assumes *length* $x = k$ *length* $y = k$
shows *length* (*karatsuba-on-power-of-2-length k x y*) ≤ *len-kar-bound l*
using *assms* **proof** (*induction k x y arbitrary*: *l rule*: *karatsuba-on-power-of-2-length.induct*)
case (*1 k x y*)
then show *?case*
proof (*cases k* ≤ *karatsuba-lower-bound*)
case *True*
then have *karatsuba-on-power-of-2-length k x y* = *grid-mul-nat x y*
unfolding *karatsuba-on-power-of-2-length.simps*[*of k x y*] **by** *argo*
also have *length ...* ≤ *length x* + *length y*
by (*rule length-grid-mul-nat*)
also have ... = *2* ∗ *k* **using** *1* **by** *linarith*
also have ... ≤ *len-kar-bound l*
unfolding *len-kar-bound.simps*[*of l*] **using** *1.prems True* **by** *simp*
finally show *?thesis* **.**
next
case *False*
then interpret *r*: *karatsuba-context k l x y* **using** *1.prems* **by** *unfold-locales simp-all*
from *r.recursive* **have** *length* (*karatsuba-on-power-of-2-length k x y*) =
length (*r.prod0* $+_n$ (*r.r* $>>_n$ *r.k-div-2*) $+_n$
(*r.prod1* $>>_n$ *k*))
by *argo*
also have ... ≤ *max* (*max* (*length r.prod0*)
(*2* ^ (*l* − *1*) +
max (*max* (*length r.prod0*) (*length r.prod1*) + *1*) (*length r.prod2*) + *1*) + *1*)
(*k* + *length r.prod1*) + *1*
unfolding *r.r-def r.add01-def*
apply (*estimation estimate*: *length-add-nat-upper*)
apply (*estimation estimate*: *length-add-nat-upper*)
unfolding *length-shift-right r.k-div-2 if-distrib*[*of length*]
apply (*estimation estimate*: *if-le-max*)
apply (*estimation estimate*: *length-add-nat-upper*)
apply (*estimation estimate*: *length-subtract-nat-le*)
apply (*estimation estimate*: *length-add-nat-upper*)
by *simp*
also have ... ≤ *max* (*max* (*len-kar-bound* (*l* − *1*))
(*2* ^ (*l* − *1*) +
max (*max* (*len-kar-bound* (*l* − *1*)) (*len-kar-bound* (*l* − *1*)) + *1*)
(*len-kar-bound* (*l* − *1*)) + *1*) + *1*)
(*k* + *len-kar-bound* (*l* − *1*)) + *1*
unfolding *r.prod0-def r.prod1-def r.prod2-def*
apply (*estimation estimate*: *1.IH*(*1*)[*OF r.IH-prems1*])
apply (*estimation estimate*: *1.IH*(*2*)[*OF r.IH-prems2*])
apply (*estimation estimate*: *1.IH*(*3*)[*OF r.IH-prems3*])

```
      by (rule order.refl)
    also have ... = max (2 ^ (l − 1) + len-kar-bound (l − 1) + 3)
     (2 ^ l + len-kar-bound (l − 1)) + 1
      unfolding max.idem r.k-power-of-2 by (simp del: One-nat-def)
    also have ... ≤ (2 ^ l + len-kar-bound (l − 1) + 3) + 1
      apply (intro add-mono order.refl)
      apply (intro max.boundedI)
      subgoal
        apply (intro add-mono order.refl) by simp
      subgoal by simp
      done
    also have ... = len-kar-bound l
      unfolding len-kar-bound.simps[of l] using False r.k-power-of-2 by simp
    finally show ?thesis .
  qed
qed

lemma len-kar-bound-le: len-kar-bound l ≤ 6 ∗ 2 ^ l + 2 ∗ karatsuba-lower-bound
proof (induction l rule: less-induct)
  case (less l)
  then show ?case
  proof (cases 2 ^ l ≤ karatsuba-lower-bound)
    case True
    then show ?thesis
      unfolding len-kar-bound.simps[of l] by simp
  next
    case False
     then have l − 1 < l using karatsuba-lower-bound-requirement by (cases l;
simp)
    then have l > 0 by simp
    from False have len-kar-bound l = 2 ^ l + len-kar-bound (l − 1) + 4
      unfolding len-kar-bound.simps[of l] by argo
    also have ... ≤ 2 ^ l + (6 ∗ 2 ^ (l − 1) + 2 ∗ karatsuba-lower-bound) + 4
      using less[OF ‹l − 1 < l›] by simp
   also have ... = 2 ∗ (2 ^ (l − 1)) + (6 ∗ 2 ^ (l − 1) + 2 ∗ karatsuba-lower-bound)
+ 4
      unfolding power-Suc[symmetric] Suc-diff-1[OF ‹l > 0›] by (rule refl)
    also have ... = 8 ∗ 2 ^ (l − 1) + 4 + 2 ∗ karatsuba-lower-bound by simp
    also have ... ≤ 8 ∗ 2 ^ (l − 1) + 4 ∗ 2 ^ (l − 1) + 2 ∗ karatsuba-lower-bound
by simp
    also have ... = 12 ∗ 2 ^ (l − 1) + 2 ∗ karatsuba-lower-bound by simp
    also have ... = 6 ∗ 2 ^ l + 2 ∗ karatsuba-lower-bound
      using Suc-diff-1[OF ‹l > 0›, symmetric] power-Suc[of 2::nat l − 1] by simp
    finally show ?thesis .
  qed
qed
```

The following is a pretty crude estimate for the length of the result of our Karatsuba implementation, but it suffices for our purposes.

lemma *length-karatsuba-on-power-of-2-length-le*:
assumes *k = 2 ^ l*
assumes *length x = k length y = k*
shows *length (karatsuba-on-power-of-2-length k x y) ≤ 6 ∗ k + 2 ∗ karatsuba-lower-bound*
using *order.trans[OF length-karatsuba-on-power-of-2-aux[OF assms] len-kar-bound-le]*
unfolding *assms* **.**

In order to multiply two integers of arbitrary length using Karatsuba multiplication, the input numbers can just be zero-padded.

fun *karatsuba-mul-nat* :: *nat-lsbf* ⇒ *nat-lsbf* ⇒ *nat-lsbf* **where**
karatsuba-mul-nat x y = (let k = next-power-of-2 (max (length x) (length y)) in
karatsuba-on-power-of-2-length k (fill k x) (fill k y))

We verify the correctness of Karatsuba multiplication:

theorem *karatsuba-mul-nat-correct*: *to-nat (karatsuba-mul-nat x y) = to-nat x ∗ to-nat y*
proof −
define *k* **where** *k = next-power-of-2 (max (length x) (length y))*
then obtain *l* **where** *k = 2 ^ l* **using** *next-power-of-2-is-power-of-2* **by** *blast*
have *1*: *to-nat (fill k x) = to-nat x to-nat (fill k y) = to-nat y* **by** *simp-all*
have *k ≥ length x k ≥ length y*
using *next-power-of-2-lower-bound[of max (length x) (length y)] k-def*
by *simp-all*
hence *length (fill k x) = k length (fill k y) = k* **using** *length-fill* **by** *simp-all*
show *?thesis* **unfolding** *k-def[symmetric] karatsuba-lower-bound-def*
using *karatsuba-on-power-of-2-length-correct[OF ‹k = 2 ^ l› ‹length (fill k x) = k› ‹length (fill k y) = k›]*
by *(simp only: karatsuba-mul-nat.simps Let-def k-def[symmetric] to-nat-fill)*
qed

lemma *length-karatsuba-mul-nat-le*: *length (karatsuba-mul-nat x y) ≤ 12 ∗ max (length x) (length y) + (6 + 2 ∗ karatsuba-lower-bound)*
proof −
let *?m = max (length x) (length y)*
define *k* **where** *k = next-power-of-2 ?m*
then obtain *l* **where** *k = 2 ^ l* **using** *next-power-of-2-is-power-of-2* **by** *auto*
from *k-def* **have** *?m ≤ k* **using** *next-power-of-2-lower-bound* **by** *simp*
from *k-def* **have** *karatsuba-mul-nat x y = karatsuba-on-power-of-2-length k (fill k x) (fill k y)*
unfolding *karatsuba-mul-nat.simps Let-def* **by** *argo*
also have *length ... ≤ 6 ∗ k + 2 ∗ karatsuba-lower-bound*
apply *(intro length-karatsuba-on-power-of-2-length-le[OF ‹k = 2 ^ l›] length-fill)*
subgoal using *‹?m ≤ k›* **by** *simp*
subgoal using *‹?m ≤ k›* **by** *simp*
done
also have *... ≤ 6 ∗ (2 ∗ ?m + 1) + 2 ∗ karatsuba-lower-bound*
apply *(intro add-mono mult-le-mono order.refl)*
unfolding *k-def* **by** *(rule next-power-of-2-upper-bound′)*

also have ... = *12* ∗ *?m* + (*6* + *2* ∗ *karatsuba-lower-bound*)
  by *simp*
finally show *?thesis* **.**
qed

Formally, we only implemented Karatsuba multiplication on natural numbers (not all integers). However, this does not really matter, as the multiplication can just be lifted to the integers. This lifting has already been done on other types, but for the sake of completeness we will just add it here as well:

fun *karatsuba-mul-int* **where**
karatsuba-mul-int x y = *nat-mul-to-int-mul karatsuba-mul-nat x y*

corollary *karatsuba-mul-int-correct*:
to-int (*karatsuba-mul-int x y*) = *to-int x* ∗ *to-int y*
  using *nat-mul-to-int-mul-correct*[*of karatsuba-mul-nat*] *karatsuba-mul-nat-correct*
  by (*metis karatsuba-mul-int.simps surj-pair*)

end

12 Running Time of Karatsuba Multiplication

theory *Karatsuba-TM*
  imports *Karatsuba ../Binary-Representations/Nat-LSBF-TM*
    ../Estimation-Method
begin

This theory contains a time monad version of Karatsuba multiplication, which is used to verify the asymptotic running time of $\mathcal{O}\left(n^{\log_2 3}\right)$.

definition *abs-diff-tm* :: *nat-lsbf* ⇒ *nat-lsbf* ⇒ *nat-lsbf tm* **where**
abs-diff-tm xs ys =*1 do* {
  r1 ← *xs* $-_{nt}$ *ys*;
  r2 ← *ys* $-_{nt}$ *xs*;
  r1 $+_{nt}$ *r2*
}

lemma *val-abs-diff-tm*[*simp*, *val-simp*]: *val* (*abs-diff-tm xs ys*) = *abs-diff xs ys*
  by (*simp add*: *abs-diff-tm-def abs-diff-def*)

lemma *time-abs-diff-tm-le*: *time* (*abs-diff-tm xs ys*) ≤ *62* ∗ *max* (*length xs*) (*length ys*) + *100*
proof −
  have *time* (*abs-diff-tm xs ys*) ≤ *time* (*xs* $-_{nt}$ *ys*) + *time* (*ys* $-_{nt}$ *xs*) +
      time ((*xs* $-_n$ *ys*) $+_{nt}$ (*ys* $-_n$ *xs*)) + *1*
    by (*simp add*: *abs-diff-tm-def*)
  also have ... ≤ *62* ∗ *max* (*length xs*) (*length ys*) + *100*
  apply (*estimation estimate*: *time-subtract-nat-tm-le*)
  apply (*estimation estimate*: *time-subtract-nat-tm-le*)

apply (*estimation estimate*: *time-add-nat-tm-le*)
using *length-subtract-nat-le*[*of xs ys*] *length-subtract-nat-le*[*of ys xs*]
by *linarith*
finally show *?thesis* **.**
qed

context *karatsuba-context*
begin

definition *fill-abs-diff-x* **where** *fill-abs-diff-x* = *fill k-div-2* (*abs-diff x0 x1*)
definition *fill-abs-diff-y* **where** *fill-abs-diff-y* = *fill k-div-2* (*abs-diff y0 y1*)
definition *sgnx* **where** *sgnx* = ($x1 \le_n x0$)
definition *sgny* **where** *sgny* = ($y1 \le_n y0$)
definition *sgnxy* **where** *sgnxy* = (*sgnx* = *sgny*)
definition r' **where** r' = (*if sgnxy then add01* $-_n$ *prod2 else add01* $+_n$ *prod2*)
definition *sr* **where** *sr* = *r* $>>_n$ *k-div-2*
definition *add0sr* **where** *add0sr* = *prod0* $+_n$ *sr*
definition *s1* **where** *s1* = *prod1* $>>_n$ *k*

lemma *r-r′*: $r = r'$
unfolding *r-def r′-def sgnxy-def sgnx-def sgny-def* **by** *argo*

lemmas *defs3* = *fill-abs-diff-x-def fill-abs-diff-y-def sgnx-def sgny-def sgnxy-def r-r′ r′-def sr-def add0sr-def s1-def*

end

lemma *add-nat-carry-aux*:
assumes *length* $x \le k$
assumes *length* $y \le k$
assumes *length* ($x +_n y$) = $k + 1$
shows *max* (*length x*) (*length y*) = *k Nat-LSBF.to-nat x* + *Nat-LSBF.to-nat y* ≥ 2 ^ *k*
proof −
have *length x* = *k* ∨ *length y* = *k*
proof (*rule ccontr*)
assume ¬ (*length x* = *k* ∨ *length y* = *k*)
then have *max* (*length x*) (*length y*) < *k* **using** *assms* **by** *simp*
then have *length* (*add-nat x y*) < *k* + *1* **using** *length-add-nat-upper*[*of x y*] **by** *linarith*
then show *False* **using** *assms* **by** *simp*
qed
then show *max* (*length x*) (*length y*) = *k* **using** *assms* **by** *linarith*
then obtain *z* **where** *add-nat x y* = *z* @ [*True*]
using *add-nat-last-bit-True assms* **by** *blast*
from *this*[*symmetric*] **have** *Nat-LSBF.to-nat x* + *Nat-LSBF.to-nat y* ≥ *2* ^ *length z*
using *add-nat-correct*[*of x y*] *to-nat-length-lower-bound*[*of z*] **by** *argo*
also have *2* ^ *length z* = *2* ^ *k* **using** ‹*add-nat x y* = *z* @ [*True*]› *assms* **by** *simp*

finally show *Nat-LSBF.to-nat x* + *Nat-LSBF.to-nat y* ≥ *2* ^ *k* **by** *simp*
qed

context begin

private fun *f* **where**
f k = (*if k* ≤ *karatsuba-lower-bound then 2* ∗ *k else f* (*k div 2*) + *k* + *4*)

declare *f.simps*[*simp del*]

private lemma *f-linear*: *f k* ≤ *6* ∗ *k*
  apply (*induction k rule*: *f.induct*)
  subgoal for *k*
    apply (*cases k* ≤ *karatsuba-lower-bound*)
    subgoal by (*simp add*: *f.simps*[*of k*])
    subgoal premises *prems*
    proof (*cases k* ≥ *5*)
      case *True*
      then show *?thesis* **using** *prems* **unfolding** *f.simps*[*of k*] **by** *simp*
    next
      case *False*
     then consider *k* = *2* | *k* = *3* | *k* = *4* **using** *prems karatsuba-lower-bound-requirement*
by *linarith*
      then show *?thesis* **using** *prems* **unfolding** *f.simps*[*of k*] **by** *fastforce*
    qed
    done
  done

private lemma *f-bound*:
  assumes *k* = *2* ^ *l*
  assumes *length x* = *k*
  assumes *length y* = *k*
  shows *length* (*karatsuba-on-power-of-2-length k x y*) ≤ *f k*
  using *assms*
proof (*induction k x y arbitrary*: *l rule*: *karatsuba-on-power-of-2-length.induct*)
  case (*1 k x y*)
  show *?case*
  proof (*cases k* ≤ *karatsuba-lower-bound*)
    case *True*
    then show *?thesis* **unfolding** *karatsuba-on-power-of-2-length.simps*[*of k x y*]
      using *length-grid-mul-nat*[*of x y*] *1.prems f.simps*[*of k*] **by** *simp*
  next
    case *False*
    then interpret *r* : *karatsuba-context k l x y*
      using *1.prems* **by** (*unfold-locales*; *simp*)
    have *len0*: *length r.prod0* ≤ *f* (*k div 2*)
      unfolding *r.prod0-def r.k-div-2-def*[*symmetric*]
      by (*intro 1*(*1*)[*OF r.IH-prems1*])
    have *len1*: *length r.prod1* ≤ *f* (*k div 2*)

```
      unfolding r.prod1-def r.k-div-2-def[symmetric]
      by (intro 1(2)[OF r.IH-prems2])
    have len2: length r.prod2 ≤ f (k div 2)
      unfolding r.prod2-def r.k-div-2-def[symmetric]
      by (intro 1(3)[OF r.IH-prems3])

    have len-p01: length (r.prod0 +n r.prod1) ≤ f (k div 2) + 1
      using length-add-nat-upper[of r.prod0 r.prod1] len0 len1 by linarith
    then have length (r.prod0 +n r.prod1 +n r.prod2) ≤ f (k div 2) + 2
      using length-add-nat-upper[of r.prod0 +n r.prod1 r.prod2] len2 by linarith
    moreover have length (r.prod0 +n r.prod1 −n r.prod2) ≤ f (k div 2) + 1
      using length-subtract-nat-le[of r.prod0 +n r.prod1 r.prod2] len-p01 len2
      by linarith
    ultimately have lenif: length (if r.sgnxy then r.prod0 +n r.prod1 −n r.prod2
          else r.prod0 +n r.prod1 +n r.prod2) ≤ f (k div 2) + 2 (is length ?if ≤
-)
      by simp

   have length (karatsuba-on-power-of-2-length k x y) ≤ max (r.k-div-2 + f (k div
2)) (k + f (k div 2)) + 4
      unfolding r.recursive
      apply (estimation estimate: length-add-nat-upper)
      apply (subst length-shift-right)
      apply (estimation estimate: length-add-nat-upper)
      apply (subst length-shift-right)
      unfolding r.r-def r.add01-def
      apply (subst if-distrib[of length])
      apply (estimation estimate: length-add-nat-upper)
      apply (estimation estimate: length-subtract-nat-le)
      apply (estimation estimate: length-add-nat-upper)
      apply (estimation estimate: len0)
      apply (estimation estimate: len1)
      apply (estimation estimate: len2)
      by auto
    also have ... = k + f (k div 2) + 4
      using r.k-div-2-less-k by simp
    finally show ?thesis unfolding f.simps[of k] using False by simp
  qed
qed

lemma length-karatsuba-on-power-of-2-length:
  assumes k = 2 ^ l
  assumes length x = k
  assumes length y = k
  shows length (karatsuba-on-power-of-2-length k x y) ≤ 6 ∗ k
  using f-bound[OF assms] f-linear[of k] by simp

end
```

```
function karatsuba-on-power-of-2-length-tm :: nat ⇒ nat-lsbf ⇒ nat-lsbf ⇒ nat-lsbf
tm where
karatsuba-on-power-of-2-length-tm k xs ys =1 do {
  b ← k ≤_t karatsuba-lower-bound;
  (if b then grid-mul-nat-tm xs ys else do {
    (x0, x1) ← split-tm xs;
    (y0, y1) ← split-tm ys;
    k-div-2 ← k div_t 2;
    prod0 ← karatsuba-on-power-of-2-length-tm k-div-2 x0 y0;
    prod1 ← karatsuba-on-power-of-2-length-tm k-div-2 x1 y1;
    abs-diff-x ← (abs-diff-tm x0 x1 ⋙= fill-tm k-div-2);
    abs-diff-y ← (abs-diff-tm y0 y1 ⋙= fill-tm k-div-2);
    prod2 ← karatsuba-on-power-of-2-length-tm k-div-2 abs-diff-x abs-diff-y;
    sgnx ← x1 ≤_nt x0;
    sgny ← y1 ≤_nt y0;
    sgnxy ← sgnx =_t sgny;
    — construct return value
    add01 ← prod0 +_nt prod1;
    r ← (if sgnxy then add01 −_nt prod2 else add01 +_nt prod2);
    sr ← r >>_nt k-div-2;
    add0sr ← prod0 +_nt sr;
    s1 ← prod1 >>_nt k;
    add0sr +_nt s1
  })
}
  by pat-completeness simp
termination
  by (relation Wellfounded.measure (λp. size (fst p))) simp-all

declare karatsuba-on-power-of-2-length-tm.simps[simp del]

lemma val-karatsuba-on-power-of-2-length-tm[simp, val-simp]:
  assumes k = 2 ^ l
  assumes length xs = k length ys = k
  shows val (karatsuba-on-power-of-2-length-tm k xs ys) = karatsuba-on-power-of-2-length
k xs ys
using assms proof (induction k arbitrary: l xs ys rule: less-induct)
  case (less k)
  show ?case
  proof (cases k ≤ karatsuba-lower-bound)
    case True
    then show ?thesis
      unfolding karatsuba-on-power-of-2-length-tm.simps[of k xs ys]
      karatsuba-on-power-of-2-length.simps[of k xs ys]
      val-bind-tm val-less-eq-nat-tm val-simps val-grid-mul-nat-tm
      by simp
  next
    case False
    interpret r: karatsuba-context k l xs ys
```

using *less False* **by** *unfold-locales simp-all*
have *val0*: *val* (*karatsuba-on-power-of-2-length-tm r.k-div-2 r.x0 r.y0*) = *r.prod0*
unfolding *r.prod0-def*
by (*intro less.IH*[*OF r.k-div-2-less-k r.k-div-2 r.length-x-split*(*1*) *r.length-y-split*(*1*)])
have *val1*: *val* (*karatsuba-on-power-of-2-length-tm r.k-div-2 r.x1 r.y1*) = *r.prod1*
unfolding *r.prod1-def*
by (*intro less.IH*[*OF r.k-div-2-less-k r.k-div-2 r.length-x-split*(*2*) *r.length-y-split*(*2*)])
have *val2*: *val* (*karatsuba-on-power-of-2-length-tm r.k-div-2 r.fill-abs-diff-x r.fill-abs-diff-y*) = *r.prod2*
unfolding *r.prod2-def r.fill-abs-diff-x-def*[*symmetric*] *r.fill-abs-diff-y-def*[*symmetric*]
apply (*intro less.IH*[*OF r.k-div-2-less-k r.k-div-2*])
subgoal unfolding *r.fill-abs-diff-x-def* **by** (*rule r.length-fill-abs-diff-x0-x1*)
subgoal unfolding *r.fill-abs-diff-y-def* **by** (*rule r.length-fill-abs-diff-y0-y1*)
done
have *val* (*karatsuba-on-power-of-2-length-tm k xs ys*) = *r.add0sr* $+_n$ *r.s1*
unfolding *karatsuba-on-power-of-2-length-tm.simps*[*of k xs ys*]
val-bind-tm val-less-eq-nat-tm val-simps val-split-tm r.split-x r.split-y
val-divide-nat-tm val-abs-diff-tm val-fill-tm r.k-div-2-def[*symmetric*]
val-compare-nat-tm val-subtract-nat-tm val-add-nat-tm val-equal-bool-tm val-shift-right-tm
Let-def Product-Type.prod.case r.defs2[*symmetric*] *r.defs3*[*symmetric*] *val0 val1 val2*
using *False* **by** *argo*
also have ... = *karatsuba-on-power-of-2-length k xs ys*
using *r.recursive*
unfolding *karatsuba-on-power-of-2-length.simps*[*of k xs ys*]
Let-def r.split-x r.split-y Product-Type.prod.case r.defs2[*symmetric*] *r.defs3*[*symmetric*]
by *argo*
finally show *?thesis* **.**
qed
qed

fun *h* **where**
h k = (*if k* ≤ *karatsuba-lower-bound then 2* ∗ *k* + *8* ∗ *k* ∗ *k* + *3*
else 407 + *224* ∗ *k* + *3* ∗ *h* (*k div 2*))
declare *h.simps*[*simp del*]

lemma *time-karatsuba-on-power-of-2-length-tm-le-h*:
assumes *k* = *2* ^ *l*
assumes *length xs* = *k length ys* = *k*
shows *time* (*karatsuba-on-power-of-2-length-tm k xs ys*) ≤ *h k*
using *assms* **proof** (*induction k arbitrary*: *xs ys l rule*: *less-induct*)
case (*less k*)
show *?case*
proof (*cases k* ≤ *karatsuba-lower-bound*)
case *True*
then have *time* (*karatsuba-on-power-of-2-length-tm k xs ys*) ≤
2 ∗ *k* + *8* ∗ *length xs* ∗ *max* (*length xs*) (*length ys*) + *3*
unfolding *karatsuba-on-power-of-2-length-tm.simps*[*of k xs ys*]
apply (*simp add*: *time-less-eq-nat-tm*)

apply (*subst Suc-eq-plus1*)+
apply (*estimation estimate*: *time-grid-mul-nat-tm-le*)
apply (*rule order.refl*)
done
also have ... = *2* ∗ *k* + *8* ∗ *k* ∗ *k* + *3* **unfolding** *less*(*3*) *less*(*4*) **by** *simp*
finally show *?thesis* **unfolding** *h.simps*[*of k*] **using** *True* **by** *simp*
next
case *False*
then interpret *r*: *karatsuba-context k l xs ys*
by (*unfold-locales*; *simp add*: *less*)
have *val0*: *val* (*karatsuba-on-power-of-2-length-tm r.k-div-2 r.x0 r.y0*) = *r.prod0*
unfolding *r.prod0-def*
by (*intro val-karatsuba-on-power-of-2-length-tm*[*OF r.k-div-2 r.length-x-split*(*1*) *r.length-y-split*(*1*)])
have *val1*: *val* (*karatsuba-on-power-of-2-length-tm r.k-div-2 r.x1 r.y1*) = *r.prod1*
unfolding *r.prod1-def*
by (*intro val-karatsuba-on-power-of-2-length-tm*[*OF r.k-div-2 r.length-x-split*(*2*) *r.length-y-split*(*2*)])
have *val2*: *val* (*karatsuba-on-power-of-2-length-tm r.k-div-2 r.fill-abs-diff-x r.fill-abs-diff-y*) = *r.prod2*
unfolding *r.prod2-def r.fill-abs-diff-x-def*[*symmetric*] *r.fill-abs-diff-y-def*[*symmetric*]
apply (*intro val-karatsuba-on-power-of-2-length-tm*[*OF r.k-div-2*])
subgoal unfolding *r.fill-abs-diff-x-def* **by** (*rule r.length-fill-abs-diff-x0-x1*)
subgoal unfolding *r.fill-abs-diff-y-def* **by** (*rule r.length-fill-abs-diff-y0-y1*)
done

have *len0*: *length* (*r.prod0*) ≤ *3* ∗ *k*
unfolding *r.prod0-def*
apply (*estimation estimate*: *length-karatsuba-on-power-of-2-length*[*OF r.k-div-2 r.length-x-split*(*1*) *r.length-y-split*(*1*)])
unfolding *r.k-div-2-def*
by *simp*
have *len1*: *length* (*r.prod1*) ≤ *3* ∗ *k*
unfolding *r.prod1-def*
apply (*estimation estimate*: *length-karatsuba-on-power-of-2-length*[*OF r.k-div-2 r.length-x-split*(*2*) *r.length-y-split*(*2*)])
unfolding *r.k-div-2-def*
by *simp*
have *len2*: *length* (*r.prod2*) ≤ *3* ∗ *k*
unfolding *r.prod2-def*
apply (*estimation estimate*: *length-karatsuba-on-power-of-2-length*[*OF r.k-div-2 r.length-fill-abs-diff-x0-x1 r.length-fill-abs-diff-y0-y1*])
unfolding *r.k-div-2-def*
by *simp*

have *len01*: *length r.add01* ≤ *3* ∗ *k* + *1*
unfolding *r.add01-def*
apply (*estimation estimate*: *length-add-nat-upper*)
apply (*estimation estimate*: *len0*)

apply (*estimation estimate*: *len1*)
by *simp*
have *lenr*: *length r.r* ≤ *3* ∗ *k* + *2*
unfolding *r.r-def if-distrib*[*of length*]
apply (*estimation estimate*: *length-subtract-nat-le*)
apply (*estimation estimate*: *length-add-nat-upper*)
apply (*estimation estimate*: *len01*)
apply (*estimation estimate*: *len2*)
by *simp*
have *lensr*: *length r.sr* ≤ *r.k-div-2* + *3* ∗ *k* + *2*
unfolding *r.sr-def*
apply (*subst length-shift-right*)
apply (*estimation estimate*: *lenr*)
by *simp*
have *len0sr*: *length r.add0sr* ≤ *r.k-div-2* + *3* ∗ *k* + *3*
unfolding *r.add0sr-def*
apply (*estimation estimate*: *length-add-nat-upper*)
apply (*estimation estimate*: *len0*)
apply (*estimation estimate*: *lensr*)
by *simp*
have *lens1*: *length r.s1* ≤ *4* ∗ *k*
unfolding *r.s1-def*
apply (*subst length-shift-right*)
apply (*estimation estimate*: *len1*)
by *simp*

have *time0*: *time* (*karatsuba-on-power-of-2-length-tm r.k-div-2 r.x0 r.y0*) ≤ *h r.k-div-2*
by (*intro less.IH*[*OF r.k-div-2-less-k r.k-div-2 r.length-x-split*(*1*) *r.length-y-split*(*1*)])
have *time1*: *time* (*karatsuba-on-power-of-2-length-tm r.k-div-2 r.x1 r.y1*) ≤ *h r.k-div-2*
by (*intro less.IH*[*OF r.k-div-2-less-k r.k-div-2 r.length-x-split*(*2*) *r.length-y-split*(*2*)])
have *time2*: *time* (*karatsuba-on-power-of-2-length-tm r.k-div-2 r.fill-abs-diff-x r.fill-abs-diff-y*) ≤ *h r.k-div-2*
apply (*intro less.IH*[*OF r.k-div-2-less-k r.k-div-2*])
subgoal unfolding *r.fill-abs-diff-x-def* **using** *r.length-fill-abs-diff-x0-x1* **by** *assumption*
subgoal unfolding *r.fill-abs-diff-y-def* **using** *r.length-fill-abs-diff-y0-y1* **by** *assumption*
done

have *time* (*karatsuba-on-power-of-2-length-tm k xs ys*) =
time ($k \le_t$ *karatsuba-lower-bound*) +
time (*split-tm xs*) +
time (*split-tm ys*) +
time ($k\ div_t\ 2$) +
time (*karatsuba-on-power-of-2-length-tm r.k-div-2 r.x0 r.y0*) +
time (*karatsuba-on-power-of-2-length-tm r.k-div-2 r.x1 r.y1*) +
time (*abs-diff-tm r.x0 r.x1*) + *time* (*fill-tm r.k-div-2* (*abs-diff r.x0 r.x1*)) +

time (*abs-diff-tm r.y0 r.y1*) + *time* (*fill-tm r.k-div-2* (*abs-diff r.y0 r.y1*)) +
time (*karatsuba-on-power-of-2-length-tm r.k-div-2 r.fill-abs-diff-x r.fill-abs-diff-y*)
+
time (*r.x1* $\leq_{nt}$ *r.x0*) +
time (*r.y1* $\leq_{nt}$ *r.y0*) +
time (*r.sgnx* $=_t$ *r.sgny*) +
time (*add-nat-tm r.prod0 r.prod1*) +
(*if r.sgnxy then time* (*r.add01* $-_{nt}$ *r.prod2*)
else time (*r.add01* $+_{nt}$ *r.prod2*)) +
time (*r.r* $>>_{nt}$ *r.k-div-2*) +
time (*r.prod0* $+_{nt}$ *r.sr*) +
time (*r.prod1* $>>_{nt}$ *k*) +
time (*r.add0sr* $+_{nt}$ *r.s1*) + *1*
unfolding *karatsuba-on-power-of-2-length-tm.simps*[*of k xs ys*]
tm-time-simps if-distrib[*of time*] *val-less-eq-nat-tm val-split-tm r.defs1*
Product-Type.prod.case val-divide-nat-tm r.defs2[*symmetric*] *r.defs3*[*symmetric*]
val-abs-diff-tm val-simps val-fill-tm val-karatsuba-on-power-of-2-length-tm
val-compare-nat-tm Let-def val0 val1 val2 val-add-nat-tm val-equal-bool-tm
val-subtract-nat-tm
by (*auto simp*: *False r.defs2*[*symmetric*] *r.defs3*[*symmetric*])
also have ... ≤ *2* ∗ *k* + *2* +
(*10* ∗ *k* + *16*) + (*10* ∗ *k* + *16*) +
(*8* ∗ *k* + *11*) +
h (*k div 2*) +
h (*k div 2*) +
(*31* ∗ *k* + *100*) +
(*2* ∗ *k* + *5*) +
(*31* ∗ *k* + *100*) +
(*2* ∗ *k* + *5*) +
h (*k div 2*) +
(*7* ∗ *k* + *23*) +
(*7* ∗ *k* + *23*) +
2 +
(*6* ∗ *k* + *3*) +
(*90* ∗ *k* + *78*) +
(*k* + *3*) +
(*7* ∗ *k* + *7*) +
(*2* ∗ *k* + *3*) +
(*8* ∗ *k* + *9*) +
1
apply (*intro add-mono*)
subgoal by (*estimation estimate*: *time-less-eq-nat-tm-le*) *simp*
subgoal by (*estimation estimate*: *time-split-tm-le*) (*simp add*: *less*)
subgoal by (*estimation estimate*: *time-split-tm-le*) (*simp add*: *less*)
subgoal by (*estimation estimate*: *time-divide-nat-tm-le*) *simp*
subgoal by (*estimation estimate*: *time0*) (*simp add*: *r.k-div-2-def*)
subgoal by (*estimation estimate*: *time1*) (*simp add*: *r.k-div-2-def*)
subgoal apply (*estimation estimate*: *time-abs-diff-tm-le*) **unfolding** *r.length-x-split*
r.k-div-2-def **by** *simp*

subgoal apply (*estimation estimate*: *time-fill-tm-le*) **using** *r.length-abs-diff-x0-x1 r.k-div-2-def* **by** *simp*
subgoal apply (*estimation estimate*: *time-abs-diff-tm-le*) **unfolding** *r.length-y-split r.k-div-2-def* **by** *simp*
subgoal apply (*estimation estimate*: *time-fill-tm-le*) **using** *r.length-abs-diff-y0-y1 r.k-div-2-def* **by** *simp*
subgoal by (*estimation estimate*: *time2*) (*simp add*: *r.k-div-2-def*)
subgoal apply (*estimation estimate*: *time-compare-nat-tm-le*) **using** *r.length-x-split r.k-div-2-def* **by** *simp*
subgoal apply (*estimation estimate*: *time-compare-nat-tm-le*) **using** *r.length-y-split r.k-div-2-def* **by** *simp*
subgoal using *time-equal-bool-tm-le* **by** *simp*
subgoal
apply (*estimation estimate*: *time-add-nat-tm-le*)
apply (*estimation estimate*: *len0*)
apply (*estimation estimate*: *len1*)
by *simp*
subgoal
apply (*estimation estimate*: *time-subtract-nat-tm-le*)
apply (*estimation estimate*: *time-add-nat-tm-le*)
apply (*estimation estimate*: *len01*)
apply (*estimation estimate*: *len2*)
by *simp*
subgoal using *r.k-div-2-def* **by** *simp*
subgoal
apply (*estimation estimate*: *time-add-nat-tm-le*)
apply (*estimation estimate*: *len0*)
apply (*estimation estimate*: *lensr*)
using *r.k-div-2-def* **by** *simp*
subgoal by *simp*
subgoal
apply (*estimation estimate*: *time-add-nat-tm-le*)
apply (*estimation estimate*: *len0sr*)
apply (*estimation estimate*: *lens1*)
using *r.k-div-2-less-k* **by** *presburger*
subgoal by *simp*
done
also have ... $= 407 + 224 * k + 3 * h\ (k\ div\ 2)$
by *simp*
finally show *?thesis* **unfolding** *h.simps*[*of k*] **using** *False* **by** *simp*
qed
qed

lemma *n-div-2*: $n\ div\ 2 = nat\ \lfloor real\ n\ /\ 2\rfloor$
by *linarith*

function *h-real* :: *nat* ⇒ *real* **where**
$x \le karatsuba\text{-}lower\text{-}bound \Longrightarrow h\text{-}real\ x = 8 * x * x + 2 * x + 3$
$\mid x > karatsuba\text{-}lower\text{-}bound \Longrightarrow h\text{-}real\ x = 407 + 224 * x + 3 * h\text{-}real\ (nat\ (\lfloor real$

x / *2*⌋))
  by *force simp-all*
termination
  by (*relation Wellfounded.measure* (*λx. x*)) (*simp-all add*: *n-div-2*[*symmetric*])

lemma *h-h-real*: *real* (*h k*) = *h-real k*
  apply (*induction k rule*: *h.induct*)
  subgoal for *k*
    apply (*cases k* ≤ *karatsuba-lower-bound*)
    by (*simp-all add*: *h-real.simps*[*of k*] *h.simps*[*of k*] *n-div-2 del*: *h-real.simps*)
  done

lemma *h-real-bigo*: *h-real* ∈ *O*(*λn. real n powr log 2 3*)
  by (*master-theorem 1 p'*: *1*) (*auto simp*: *powr-divide*)

definition *karatsuba-mul-nat-tm* :: *nat-lsbf* ⇒ *nat-lsbf* ⇒ *nat-lsbf tm* **where**
karatsuba-mul-nat-tm xs ys =*1 do* {
  lenx ← *length-tm xs*;
  leny ← *length-tm ys*;
  k ← *max-nat-tm lenx leny* ⋙= *next-power-of-2-tm*;
  fillx ← *fill-tm k xs*;
  filly ← *fill-tm k ys*;
  karatsuba-on-power-of-2-length-tm k fillx filly
}

lemma *val-karatsuba-mul-nat-tm*[*simp*, *val-simp*]: *val* (*karatsuba-mul-nat-tm xs ys*)
= *karatsuba-mul-nat xs ys*
proof −
  define *k* **where** *k* = *next-power-of-2* (*max* (*length xs*) (*length ys*))
  then obtain *l* **where** *k* = *2* ^ *l* **using** *next-power-of-2-is-power-of-2* **by** *auto*
  have *val* (*karatsuba-on-power-of-2-length-tm k* (*fill k xs*) (*fill k ys*)) =
      karatsuba-on-power-of-2-length k (*fill k xs*) (*fill k ys*)
    apply (*intro val-karatsuba-on-power-of-2-length-tm*[*OF* ‹*k* = *2* ^ *l*›])
    unfolding *k-def* **using** *next-power-of-2-lower-bound*[*of max* (*length xs*) (*length ys*)] **by** *auto*
  then show *?thesis*
    unfolding *karatsuba-mul-nat-tm-def karatsuba-mul-nat.simps val-simp Let-def k-def* **.**
qed

definition *time-karatsuba-mul-nat-bound* **where**
  time-karatsuba-mul-nat-bound m = *53* + *218* ∗ (*next-power-of-2 m*) + *h* (*next-power-of-2 m*)

The following two lemmas are one way to formally express the more informal statement "Karatsuba Multiplication needs $\mathcal{O}\left(n^{\log_2 3}\right)$ bit operations for input numbers of length n".

theorem *time-karatsuba-mul-nat-tm-le*:
  assumes *max* (*length xs*) (*length ys*) = *m*

shows *time* (*karatsuba-mul-nat-tm xs ys*) ≤ *time-karatsuba-mul-nat-bound m*
proof −
define *k* **where** *k* = *next-power-of-2 m*
then obtain *l* **where** *k* = *2* ^ *l* **using** *next-power-of-2-is-power-of-2* **by** *auto*
have *lens*: *length xs* ≤ *k length ys* ≤ *k*
using *assms next-power-of-2-lower-bound*[*of m*] *k-def* **by** *simp-all*
have *time* (*karatsuba-mul-nat-tm xs ys*) =
time (*length-tm xs*) +
time (*length-tm ys*) +
time (*max-nat-tm* (*length xs*) (*length ys*)) +
time (*next-power-of-2-tm* (*max* (*length xs*) (*length ys*))) +
time (*fill-tm k xs*) +
time (*fill-tm k ys*) +
time (*karatsuba-on-power-of-2-length-tm k* (*fill k xs*) (*fill k ys*)) + *1*
unfolding *karatsuba-mul-nat-tm-def tm-time-simps val-simp Let-def*
assms k-def[*symmetric*] **by** *presburger*
also have ... ≤
(*k* + *1*) + (*k* + *1*) + (*2* ∗ *k* + *3*) +
(*208* ∗ *k* + *37*) +
(*3* ∗ *k* + *5*) +
(*3* ∗ *k* + *5*) +
h k +
1
apply (*intro add-mono order.refl*)
subgoal by (*simp add*: *lens*)
subgoal by (*simp add*: *lens*)
subgoal apply (*estimation estimate*: *time-max-nat-tm-le*) **using** *lens* **by** *simp*
subgoal apply (*estimation estimate*: *time-next-power-of-2-tm-le*) **using** *lens* **by** *simp*
subgoal apply (*estimation estimate*: *time-fill-tm-le*) **using** *lens* **by** *simp*
subgoal apply (*estimation estimate*: *time-fill-tm-le*) **using** *lens* **by** *simp*
subgoal apply (*intro time-karatsuba-on-power-of-2-length-tm-le-h*[*OF* ‹*k* = *2* ^ *l*›]) **using** *lens*
by *auto*
done
also have ... = *53* + *218* ∗ *k* + *h k* **by** *simp*
finally show *?thesis* **unfolding** *k-def time-karatsuba-mul-nat-bound-def*[*symmetric*]
.
qed

theorem *time-karatsuba-mul-nat-bound-bigo*: *time-karatsuba-mul-nat-bound* ∈ *O*(λ*m*. *m powr log 2 3*)
proof −
define *t* **where** *t* = (λ*m*. *real* (*53* + *218* ∗ *m* + *h m*))
then have *time-karatsuba-mul-nat-bound* = *t* ∘ *next-power-of-2*
unfolding *time-karatsuba-mul-nat-bound-def* **by** *auto*
also have ... ∈ *O*(λ*m*. *m powr log 2 3*)
apply (*intro powr-bigo-linear-index-transformation*)
subgoal

proof −
  have (λ*x*. *real* (*next-power-of-2 x*)) ∈ *O*(λ*x*. *real* (*2* ∗ *x* + *1*))
    apply (*intro landau-mono-always*)
    using *next-power-of-2-upper-bound′ real-mono* **by** *simp-all*
  moreover have (λ*x*. *real* (*2* ∗ *x* + *1*)) ∈ *O*(*real*) **by** *auto*
  ultimately show (λ*x*. *real* (*next-power-of-2 x*)) ∈ *O*(*real*)
    using *landau-o.big.trans* **by** *blast*
qed
subgoal unfolding *t-def real-linear real-multiplicative h-h-real*
  apply (*intro sum-in-bigo*)
  subgoal by *auto*
  subgoal by *auto*
  subgoal using *h-real-bigo* **.**
  done
subgoal by *auto*
done
finally show *?thesis* **.**
qed

end

13 Code Generation

theory *Karatsuba-Code-Nat*
  imports *Main HOL−Library.Code-Binary-Nat Karatsuba*
begin

In this theory, the Karatsuba Multiplication implemented in *Karatsuba* is used for code generation. This is not really practical (except beginning at 3000 decimal digits), but merely a nice gimmick.

fun *from-numeral* :: *num* ⇒ *nat-lsbf* **where**
  from-numeral num.One = [*True*]
| *from-numeral* (*num.Bit0 x*) = *False* # *from-numeral x*
| *from-numeral* (*num.Bit1 x*) = *True* # *from-numeral x*

lemma *from-numeral-nonempty*: *from-numeral x* ≠ []
  by (*induction x rule*: *from-numeral.induct*; *simp*)

lemma *from-numeral-truncated*: *truncated* (*from-numeral x*)
  unfolding *truncated-iff*
  by (*induction x rule*: *from-numeral.induct*; *simp add*: *from-numeral-nonempty*)

lemma *to-nat-from-numeral-neq-zero*: *to-nat* (*from-numeral x*) ≠ *0*
  using *to-nat-zero-iff from-numeral-truncated from-numeral-nonempty* **by** *simp*

fun *to-numeral-of-truncated* :: *nat-lsbf* ⇒ *num* **where**
to-numeral-of-truncated [] = *num.One*
| *to-numeral-of-truncated* [*True*] = *num.One*

| *to-numeral-of-truncated* (*True* # *xs*) = *num.Bit1* (*to-numeral-of-truncated xs*)
| *to-numeral-of-truncated* (*False* # *xs*) = *num.Bit0* (*to-numeral-of-truncated xs*)

lemma *to-numeral-of-truncated-from-numeral*:
to-numeral-of-truncated (*from-numeral x*) = *x*
apply (*induction x*)
subgoal by *simp*
subgoal by *simp*
subgoal for *x* **by** (*cases from-numeral x*; *simp*)
done

lemma *nat-of-num-to-numeral-of-truncated*:
assumes *truncated xs*
assumes *xs* ≠ []
shows *nat-of-num* (*to-numeral-of-truncated xs*) = *to-nat xs*
using *assms* **proof** (*induction xs rule*: *to-numeral-of-truncated.induct*)
case *1*
then show *?case* **by** *blast*
next
case *2*
then show *?case* **by** *simp*
next
case (*3 v va*)
note *truncated-Cons-imp-truncated-tl*[*OF 3.prems*(*1*)]
from *3.IH*[*OF this*] **show** *?case* **by** *simp*
next
case (*4 xs*)
from *4.prems*(*1*) **have** *xs* ≠ []
apply (*intro ccontr*[*of xs* ≠ []])
by (*simp add*: *truncated-iff*)
note *truncated-Cons-imp-truncated-tl*[*OF 4.prems*(*1*)]
from *4.IH*[*OF this* ‹*xs* ≠ []›] **show** *?case* **by** *simp*
qed

definition *to-numeral* :: *nat-lsbf* ⇒ *num* **where**
to-numeral xs = (*let xs*′ = *Nat-LSBF.truncate xs in to-numeral-of-truncated xs*′)

lemma *to-numeral-from-numeral*: *to-numeral* (*from-numeral x*) = *x*
unfolding *to-numeral-def Let-def*
using *from-numeral-truncated to-numeral-of-truncated-from-numeral*
by *simp*

lemma *nat-of-num-to-numeral*:
assumes *to-nat xs* ≠ *0*
shows *nat-of-num* (*to-numeral xs*) = *to-nat xs*
unfolding *to-numeral-def Let-def*
using *assms nat-of-num-to-numeral-of-truncated*[*of truncate xs*, *OF truncate-truncate*]
unfolding *nat-lsbf.to-f-elem*
using *to-nat-zero-iff*

by *simp*

lemma *l0*:
assumes *truncated xs*
shows *to-numeral-of-truncated xs* = *num-of-nat* (*to-nat xs*)
using *assms*
by (*metis nat-of-num-inverse nat-of-num-to-numeral-of-truncated num-of-nat.simps*(*1*) *to-nat.simps*(*1*) *to-numeral-of-truncated.simps*(*1*))

lemma *l1*: *to-numeral xs* = *num-of-nat* (*to-nat xs*)
unfolding *to-numeral-def Let-def*
using *l0*[*of truncate xs*] *truncate-truncate*[*of xs*] *nat-lsbf.to-f-elem*
by *simp*

lemma *l2*: *to-nat* (*from-numeral x*) = *nat-of-num x*
by (*metis nat-of-num-to-numeral to-nat-from-numeral-neq-zero to-numeral-from-numeral*)

lemma[*code*]:
(*x*::*num*) ∗ *y* = *to-numeral* (*karatsuba-mul-nat* (*from-numeral x*) (*from-numeral y*))
unfolding *l1 karatsuba-mul-nat-correct l2 times-num-def* **by** (*rule refl*)

end

References

[1] A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. *Dokl. Akad. Nauk SSSR*, 145:293–294, 1962. http://mi.mathnet.ru/dan26729.

[2] T. Nipkow. Verified root-balanced trees. In B.-Y. E. Chang, editor, *Asian Symposium on Programming Languages and Systems, APLAS 2017*, volume 10695 of *LNCS*, pages 255–272. Springer, 2017. https://www21.in.tum.de/~nipkow/pubs/aplas17.pdf.