Karatsuba Multiplication for Integers

Jakob Schulz, Emin Karayel

February 20, 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)
⟨*proof*⟩

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* $f'\ x$)
⟨*proof*⟩

lemma *set-subseteqD*:
assumes *set xs* $\subseteq A$
shows $\bigwedge i.\ i <$ *length xs* $\Longrightarrow$ *xs* ! $i \in A$
⟨*proof*⟩

lemma *set-subseteqI*:
assumes $\bigwedge i.\ i <$ *length xs* $\Longrightarrow$ *xs* ! $i \in A$
shows *set xs* $\subseteq A$
⟨*proof*⟩

lemma *Nat-max-le-sum*: *max* (a :: *nat*) $b \leq a + b$
⟨*proof*⟩

lemma *upt-add-eq-append'*:
assumes $a \leq b$ $b \leq c$
shows $[a..<c] = [a..<b]$ @ $[b..<c]$
⟨*proof*⟩

lemma *map-add-const-upt*: *map* ($\lambda j.\ j + c$) $[a..<b] = [a + c..<b + c]$
⟨*proof*⟩

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

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

lemma *length-filter-even*: *length* (*filter even* $[0..<n]$) = (*if even* n *then* n *div* 2 *else* n *div* $2 + 1$)
⟨*proof*⟩
lemma *length-filter-odd*: *length* (*filter odd* $[0..<n]$) = n *div* 2
⟨*proof*⟩

lemma *filter-even-nth*:
assumes $i <$ *length* (*filter even* $[0..<n]$)
shows *filter even* $[0..<n]$! $i = 2 * i$

$\langle proof \rangle$

lemma *filter-odd-nth*:
  assumes $i < length\ (filter\ odd\ [0..<n])$
  shows $filter\ odd\ [0..<n]\ !\ i = 2 * i + 1$
$\langle proof \rangle$

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*]: $length\ (sublist\ m\ n\ xs) = card\ (\{m..<n\} \cap \{0..<length\ xs\})$
  $\langle proof \rangle$

lemma *length-sublist'*:
  assumes $m \le n$
  assumes $n \le length\ xs$
  shows $length\ (sublist\ m\ n\ xs) = n - m$
  $\langle proof \rangle$

lemma *nth-sublist*:
  assumes $m \le n$
  assumes $n \le length\ xs$
  assumes $i < n - m$
  shows $sublist\ m\ n\ xs\ !\ i = xs\ !\ (m + i)$
  $\langle proof \rangle$

lemma *filter-map-map2*:
  assumes $length\ b = m$
  assumes $length\ c = m$
  shows $[f\ (b!i)\ (c!i).\ i \leftarrow [0..<m]] = map2\ f\ b\ c$
  $\langle proof \rangle$

fun *map3* **where**
$map3\ f\ (x\ \#\ xs)\ (y\ \#\ ys)\ (z\ \#\ zs) = f\ x\ y\ z\ \#\ map3\ f\ xs\ ys\ zs$
$|\ map3\ f\ \text{-}\ \text{-}\ \text{-} = []$

lemma *map3-as-map*: $map3\ f\ xs\ ys\ zs = map\ (\lambda((x,\ y),\ z).\ f\ x\ y\ z)\ (zip\ (zip\ xs\ ys)\ zs)$
  $\langle proof \rangle$

lemma *filter-map-map3*:
  assumes $length\ b = m$
  assumes $length\ c = m$
  shows $[f\ (b!i)\ (c!i)\ i.\ i \leftarrow [0..<m]] = map3\ f\ b\ c\ [0..<m]$
  $\langle proof \rangle$

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*)
  ⟨*proof*⟩

lemma *nth-map2*:
  assumes *i* < *length xs*
  assumes *i* < *length ys*
  shows *map2 f xs ys* ! *i* = *f* (*xs* ! *i*) (*ys* ! *i*)
  ⟨*proof*⟩
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*)
  ⟨*proof*⟩
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*)
  ⟨*proof*⟩
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*)
  ⟨*proof*⟩

lemma *map2-of-map-r*: *map2 f xs* (*map g ys*) = *map2* (λ*x y*. *f x* (*g y*)) *xs ys*
  ⟨*proof*⟩
lemma *map2-of-map-l*: *map2 f* (*map g xs*) *ys* = *map2* (λ*x y*. *f* (*g x*) *y*) *xs ys*
  ⟨*proof*⟩
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*
  ⟨*proof*⟩
lemma *map-of-map3*: *map f* (*map3 g xs ys zs*) = *map3* (λ*x y z*. *f* (*g x y z*)) *xs ys zs*
  ⟨*proof*⟩

lemma *cyclic-index-lemma*:
  fixes *n* :: *nat*

assumes $\sigma < n$ $\varrho < n$ $i < n$
shows $(\sigma + \varrho)$ *mod* $n = i \longleftrightarrow \varrho = (n + i - \sigma)$ *mod* n
⟨*proof*⟩

lemma (**in** *residues*) *residues-minus-eq*: $x \ominus_R y = (x - y)$ *mod* m
⟨*proof*⟩

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

context *ring*
begin

fun *nat-embedding* :: *nat* ⇒ *'a* **where**
nat-embedding 0 = **0**
| *nat-embedding* (*Suc n*) = *nat-embedding n* ⊕ **1**
fun *int-embedding* :: *int* ⇒ *'a* **where**
int-embedding n = (*if n* ≥ *0 then nat-embedding* (*nat n*) *else* ⊖ *nat-embedding* (*nat* (−*n*)))

lemma *nat-embedding-closed*[*simp*]: *nat-embedding x* ∈ *carrier R*
⟨*proof*⟩
lemma *int-embedding-closed*[*simp*]: *int-embedding x* ∈ *carrier R*
⟨*proof*⟩

lemma *nat-embedding-a-hom*: *nat-embedding* (*x* + *y*) = *nat-embedding x* ⊕ *nat-embedding y*
⟨*proof*⟩
lemma *nat-embedding-m-hom*: *nat-embedding* (*x* ∗ *y*) = *nat-embedding x* ⊗ *nat-embedding y*
⟨*proof*⟩
lemma *nat-embedding-exp-hom*: *nat-embedding* (*x* ^ *y*) = *nat-embedding x* [^] *y*
⟨*proof*⟩
lemma *int-embedding-neg-hom*: *int-embedding* (− *x*) = ⊖ *int-embedding x*
⟨*proof*⟩

end

lemma *int-exp-hom*: *int x* ^ *i* = *int* (*x* ^ *i*)
⟨*proof*⟩

end

2 Auxiliary Sum Lemmas

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

lemma *sum-list-eq*: $(\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = g\ x) \Longrightarrow sum\text{-}list\ (map\ f\ xs) = sum\text{-}list\ (map\ g\ xs)$
⟨*proof*⟩

lemma *sum-list-split-0*: $(\sum i \leftarrow [0..<Suc\ n].\ f\ i) = f\ 0 + (\sum i \leftarrow [1..<Suc\ n].\ f\ i)$
⟨*proof*⟩
lemma *sum-list-index-trafo*: $(\sum i \leftarrow xs.\ f\ (g\ i)) = (\sum i \leftarrow map\ g\ xs.\ f\ i)$
⟨*proof*⟩
lemma *sum-list-index-shift*: $(\sum i \leftarrow [a..<b].\ f\ (i + c)) = (\sum i \leftarrow [a+c..<b+c].\ f\ i)$
⟨*proof*⟩

lemma *list-sum-index-shift*: $n = j - k \Longrightarrow (\sum i \leftarrow [k+1..<j+1].\ f\ i) = (\sum i \leftarrow [k..<j].\ f\ (i + 1))$
⟨*proof*⟩

lemma *list-sum-index-shift'*: $(\sum i \leftarrow [0..<m].\ a\ (i + c)) = (\sum i \leftarrow [c..<m+c].\ a\ i)$
⟨*proof*⟩

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*⟩

lemma *sum-list-linear*:
assumes $\bigwedge a\ b.\ f\ (a + b) = f\ a + f\ b$
assumes $f\ 0 = 0$
shows $f\ (\sum i \leftarrow xs.\ g\ i) = (\sum i \leftarrow xs.\ f\ (g\ i))$
⟨*proof*⟩
lemma *sum-list-int*:
shows $int\ (\sum i \leftarrow xs.\ g\ i) = (\sum i \leftarrow xs.\ int\ (g\ i))$
⟨*proof*⟩

lemma *sum-list-split-Suc*:
assumes $n = Suc\ n'$
shows $(\sum i \leftarrow [0..<n].\ f\ i) = (\sum i \leftarrow [0..<n'].\ f\ i) + f\ n'$
⟨*proof*⟩

lemma *sum-list-estimation-leq*:
assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i \leq B$
shows $(\sum i \leftarrow xs.\ f\ i) \leq length\ xs * B$
⟨*proof*⟩

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*⟩

2.1 *semiring-1* Sums

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

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
⟨*proof*⟩
lemma (**in** *semiring-1*) *of-bool-disj-excl*: $\neg (x \wedge y) \Longrightarrow$ *of-bool* $(x \vee y) =$ *of-bool* $x +$ *of-bool* y
⟨*proof*⟩

lemma (**in** *semiring-1*) *of-bool-var-swap*:
$(\sum i \leftarrow xs.\ \textit{of-bool}\ (i = j) * f\ i) = (\sum i \leftarrow xs.\ \textit{of-bool}\ (i = j) * f\ j)$
⟨*proof*⟩
lemma $(\sum i \leftarrow xs.\ \textit{of-bool}\ (i = j) * f\ i) = \textit{count-list}\ xs\ j * f\ j$
⟨*proof*⟩
lemma (**in** *semiring-1*) *of-bool-distinct*:
distinct $xs \Longrightarrow (\sum i \leftarrow xs.\ \textit{of-bool}\ (i = j) * f\ i\ j) = \textit{of-bool}\ (j \in \textit{set}\ xs) * f\ j\ j$
⟨*proof*⟩
lemma (**in** *semiring-1*) *of-bool-distinct-in*:
distinct $xs \Longrightarrow j \in \textit{set}\ xs \Longrightarrow (\sum i \leftarrow xs.\ \textit{of-bool}\ (i = j) * f\ i\ j) = f\ j\ j$
⟨*proof*⟩

lemma (**in** *linordered-semiring-1*) *of-bool-sum-leq-1*:
assumes *distinct xs*
assumes $\bigwedge i\ j.\ i \in \textit{set}\ xs \Longrightarrow j \in \textit{set}\ xs \Longrightarrow P\ i \Longrightarrow P\ j \Longrightarrow i = j$
shows $(\sum l \leftarrow xs.\ \textit{of-bool}\ (P\ l)) \leq 1$
⟨*proof*⟩
instantiation *nat* :: *linordered-semiring-1*
begin
instance ⟨*proof*⟩
end

lemma (**in** *semiring-1*) *sum-list-mult-sum-list*: $(\sum i \leftarrow xs.\ f\ i) * (\sum j \leftarrow ys.\ g\ j) = (\sum i \leftarrow xs.\ \sum j \leftarrow ys.\ f\ i * g\ j)$
⟨*proof*⟩

lemma (**in** *semiring-1*) *semiring-1-sum-list-eq*:
$(\bigwedge i.\ i \in \textit{set}\ xs \Longrightarrow f\ i = g\ i) \Longrightarrow (\sum i \leftarrow xs.\ f\ i) = (\sum i \leftarrow xs.\ g\ i)$
⟨*proof*⟩

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*⟩

lemma (**in** *semiring-1*) *sum-append*:
$(\sum i \leftarrow (xs\ @\ ys).\ f\ i) = (\sum i \leftarrow xs.\ f\ i) + (\sum i \leftarrow ys.\ f\ i)$
⟨*proof*⟩

lemma (**in** *semiring-1*) *sum-append'*:
assumes $zs = xs @ ys$
shows $(\sum i \leftarrow zs.\ f\ i) = (\sum i \leftarrow xs.\ f\ i) + (\sum i \leftarrow ys.\ f\ i)$
⟨*proof*⟩

2.1.1 Power Sums

lemma (**in** *semiring-1*) *sum-list-of-bool-filter*: $(\sum i \leftarrow xs.\ \mathit{of\text{-}bool}\ (P\ i) * f\ i) = (\sum i \leftarrow \mathit{filter}\ P\ xs.\ f\ i)$
⟨*proof*⟩

lemma *upt-filter-less*: $\mathit{filter}\ (\lambda i.\ i < c)\ [a..<b] = [a..<\mathit{min}\ b\ c]$
⟨*proof*⟩

lemma *upt-filter-geq*: $\mathit{filter}\ (\lambda i.\ i \geq c)\ [a..<b] = [\mathit{max}\ a\ c..<b]$
⟨*proof*⟩

lemma (**in** *semiring-1*) *sum-list-of-bool-less*: $(\sum i \leftarrow [a..<b].\ \mathit{of\text{-}bool}\ (i < c) * f\ i) = (\sum i \leftarrow [a..<\mathit{min}\ b\ c].\ f\ i)$
⟨*proof*⟩

lemma (**in** *semiring-1*) *sum-list-of-bool-geq*: $(\sum i \leftarrow [a..<b].\ \mathit{of\text{-}bool}\ (i \geq c) * f\ i) = (\sum i \leftarrow [\mathit{max}\ a\ c..<b].\ f\ i)$
⟨*proof*⟩

lemma (**in** *semiring-1*) *sum-list-of-bool-range*: $(\sum i \leftarrow [a..<b].\ \mathit{of\text{-}bool}\ (i \in \mathit{set}\ [c..<d]) * f\ i) =$
$(\sum i \leftarrow [\mathit{max}\ a\ c..<\mathit{min}\ b\ d].\ f\ i)$
⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *cauchy-product*:
$(\sum i \leftarrow [0..<n].\ f\ i) * (\sum j \leftarrow [0..<m].\ g\ j) =$
$(\sum k \leftarrow [0..<n + m - 1].\ \sum l \leftarrow [k + 1 - m..<\mathit{min}\ (k + 1)\ n].\ f\ l * g\ (k - l))$
⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *power-sum-product*:
assumes $m > 0$
assumes $n \geq m$
shows
$(\sum i\leftarrow[0..<n].\ f\ i * x\ \hat{}\ i) * (\sum j\leftarrow[0..<m].\ g\ j * x\ \hat{}\ j) =$
$(\sum k\leftarrow[0..<m].\ (\sum i\leftarrow[0..<\mathit{Suc}\ k].\ f\ i * g\ (k - i)) * x\ \hat{}\ k) +$
$(\sum k\leftarrow[m..<n].\ (\sum i\leftarrow[\mathit{Suc}\ k - m..<\mathit{Suc}\ k].\ f\ i * g\ (k - i)) * x\ \hat{}\ k) +$
$(\sum k\leftarrow[n..<n + m - 1].\ (\sum i\leftarrow[\mathit{Suc}\ k - m..<n].\ f\ i * g\ (k - i)) * x\ \hat{}\ k)$
⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *power-sum-product-same-length*:
assumes $n > 0$
shows $(\sum i\leftarrow[0..<n].\ f\ i * x\ \hat{}\ i) * (\sum j\leftarrow[0..<n].\ g\ j * x\ \hat{}\ j) =$

$(\sum k \leftarrow [0..<n]. (\sum i \leftarrow [0..<Suc\ k]. f\ i * g\ (k - i)) * x \,\hat{}\, k) +$
$(\sum k \leftarrow [n..<2 * n - 1]. (\sum i \leftarrow [Suc\ k - n..<n]. f\ i * g\ (k - i)) * x \,\hat{}\, k)$
⟨*proof*⟩

lemma (**in** *semiring-1*) *sum-index-transformation*:
shows $(\sum i \leftarrow xs.\ f\ (g\ i)) = (\sum j \leftarrow map\ g\ xs.\ f\ j)$
⟨*proof*⟩

lemma (**in** *comm-semiring-1*) *power-sum-split*:
fixes $f :: nat \Rightarrow 'a$
fixes $x :: 'a$
fixes $c :: nat$
assumes $j \leq n$
shows $(\sum i \leftarrow [0..<n].\ f\ i * x \,\hat{}\, (i * c)) =$
$(\sum i \leftarrow [0..<j].\ f\ i * x \,\hat{}\, (i * c)) +$
$x \,\hat{}\, (j * c) * (\sum i \leftarrow [0..<n - j].\ f\ (j + i) * x \,\hat{}\, (i * c))$
⟨*proof*⟩

2.2 *nat* Sums

lemma *geo-sum-nat*:
assumes $(q :: nat) > 1$
shows $(q - 1) * (\sum i \leftarrow [0..<n].\ q \,\hat{}\, i) = q \,\hat{}\, n - 1$
⟨*proof*⟩

lemma *geo-sum-bound*:
assumes $(q :: nat) > 1$
assumes $\bigwedge i.\ i < n \Longrightarrow f\ i < q$
shows $(\sum i \leftarrow [0..<n].\ f\ i * q \,\hat{}\, i) < q \,\hat{}\, n$
⟨*proof*⟩

lemma *power-sum-nat-split-div-mod*:
assumes $x > 1$
assumes $c > 0$
assumes $\bigwedge i.\ i < n \Longrightarrow (f\ i :: nat) < x \,\hat{}\, c$
assumes $j \leq n$
shows $(\sum i \leftarrow [0..<n].\ f\ i * x \,\hat{}\, (i * c))\ div\ x \,\hat{}\, (j * c)$
$= (\sum i \leftarrow [0..<n - j].\ f\ (j + i) * x \,\hat{}\, (i * c))$
$(\sum i \leftarrow [0..<n].\ f\ i * x \,\hat{}\, (i * c))\ mod\ x \,\hat{}\, (j * c)$
$= (\sum i \leftarrow [0..<j].\ f\ i * x \,\hat{}\, (i * c))$
⟨*proof*⟩

lemma *power-sum-nat-extract-coefficient*:
assumes $x > 1$
assumes $c > 0$
assumes $\bigwedge i.\ i < n \Longrightarrow (f\ i :: nat) < x \,\hat{}\, c$
assumes $j < n$
shows $((\sum i \leftarrow [0..<n].\ f\ i * x \,\hat{}\, (i * c))\ div\ x \,\hat{}\, (j * c))\ mod\ x \,\hat{}\, c = f\ j$
⟨*proof*⟩

lemma *power-sum-nat-eq*:
assumes $x > 1$
assumes $c > 0$
assumes $\bigwedge i.\ i < n \Longrightarrow (f\ i :: nat) < x \mathbin{\hat{}} c$
assumes $\bigwedge i.\ i < n \Longrightarrow g\ i < x \mathbin{\hat{}} c$
assumes $(\sum i \leftarrow [0..<n].\ f\ i * x \mathbin{\hat{}} (i * c)) = (\sum i \leftarrow [0..<n].\ g\ i * x \mathbin{\hat{}} (i * c))$
(**is** *?sumf* = *?sumg*)
shows $\bigwedge i.\ i < n \Longrightarrow f\ i = g\ i$
⟨*proof*⟩

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* :: $['c \Rightarrow 'a, 'c\ list] \Rightarrow 'a$ **where**
monoid-sum-list f $[] = \mathbf{0}$
| *monoid-sum-list f* $(x \# xs) = f\ x \oplus$ *monoid-sum-list f xs*

lemma *monoid-sum-list f xs* = *foldr* $(\oplus)$ (*map f xs*) $\mathbf{0}$
⟨*proof*⟩

end

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

syntax
-monoid-sum-list :: *index* ⇒ *idt* ⇒ *'c list* ⇒ *'c* ⇒ *'a*
((*3*$\bigoplus$--←-. -) [*1000*, *0*, *51*, *10*] *10*)
translations
$\bigoplus_G i{\leftarrow}xs.\ b \rightleftharpoons$ *CONST abelian-monoid.monoid-sum-list G* (λ*i. b*) *xs*

context *abelian-monoid*
begin

lemma *monoid-sum-list-finsum*:

assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i \in carrier\ G$
assumes *distinct xs*
shows $(\bigoplus i \leftarrow xs.\ f\ i) = (\bigoplus i \in set\ xs.\ f\ i)$
$\langle proof \rangle$

lemma *monoid-sum-list-cong*:
assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i = g\ i$
shows $(\bigoplus i \leftarrow xs.\ f\ i) = (\bigoplus i \leftarrow xs.\ g\ i)$
$\langle proof \rangle$

lemma *monoid-sum-list-closed*[*simp*]:
assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i \in carrier\ G$
shows $(\bigoplus i \leftarrow xs.\ f\ i) \in carrier\ G$
$\langle proof \rangle$

lemma *monoid-sum-list-add-in*:
assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i \in carrier\ G$
assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow g\ i \in carrier\ G$
shows $(\bigoplus i \leftarrow xs.\ f\ i) \oplus (\bigoplus i \leftarrow xs.\ g\ i) =$
$(\bigoplus i \leftarrow xs.\ f\ i \oplus g\ i)$
$\langle proof \rangle$

lemma *monoid-sum-list-0*[*simp*]: $(\bigoplus i \leftarrow xs.\ \mathbf{0}) = \mathbf{0}$
$\langle proof \rangle$

lemma *monoid-sum-list-swap*:
assumes[*simp*]: $\bigwedge i\ j.\ i \in set\ xs \Longrightarrow j \in set\ ys \Longrightarrow f\ i\ j \in carrier\ G$
shows $(\bigoplus i \leftarrow xs.\ (\bigoplus j \leftarrow ys.\ f\ i\ j)) =$
$(\bigoplus j \leftarrow ys.\ (\bigoplus i \leftarrow xs.\ f\ i\ j))$
$\langle proof \rangle$

lemma *monoid-sum-list-index-transformation*:
$(\bigoplus i \leftarrow (map\ g\ xs).\ f\ i) = (\bigoplus i \leftarrow xs.\ f\ (g\ i))$
$\langle proof \rangle$

lemma *monoid-sum-list-index-shift-0*:
$(\bigoplus i \leftarrow [c..<c+n].\ f\ i) = (\bigoplus i \leftarrow [0..<n].\ f\ (c + i))$
$\langle proof \rangle$

lemma *monoid-sum-list-index-shift*:
$(\bigoplus l \leftarrow [a..<b].\ f\ (l+c)) = (\bigoplus l \leftarrow [(a+c)..<(b+c)].\ f\ l)$
$\langle proof \rangle$

lemma *monoid-sum-list-app*:
assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i \in carrier\ G$
assumes $\bigwedge i.\ i \in set\ ys \Longrightarrow f\ i \in carrier\ G$
shows $(\bigoplus i \leftarrow xs\ @\ ys.\ f\ i) = (\bigoplus i \leftarrow xs.\ f\ i) \oplus (\bigoplus i \leftarrow ys.\ f\ i)$
$\langle proof \rangle$

lemma *monoid-sum-list-app'*:
  assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i \in carrier\ G$
  assumes $\bigwedge i.\ i \in set\ ys \Longrightarrow f\ i \in carrier\ G$
  assumes $xs\ @\ ys = zs$
  shows $(\bigoplus i \leftarrow zs.\ f\ i) = (\bigoplus i \leftarrow xs.\ f\ i) \oplus (\bigoplus i \leftarrow ys.\ f\ i)$
  ⟨*proof*⟩

lemma *monoid-sum-list-extract*:
  assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i \in carrier\ G$
  assumes $\bigwedge i.\ i \in set\ ys \Longrightarrow f\ i \in carrier\ G$
  assumes $f\ x \in carrier\ G$
  shows $(\bigoplus i \leftarrow xs\ @\ x\ \#\ ys.\ f\ i) = f\ x \oplus (\bigoplus i \leftarrow (xs\ @\ ys).\ f\ i)$
⟨*proof*⟩

lemma *monoid-sum-list-Suc*:
  assumes $\bigwedge i.\ i < Suc\ r \Longrightarrow f\ i \in carrier\ G$
  shows $(\bigoplus i \leftarrow [0..<Suc\ r].\ f\ i) = (\bigoplus i \leftarrow [0..<r].\ f\ i) \oplus f\ r$
  ⟨*proof*⟩

lemma *bij-betw-diff-singleton*: $a \in A \Longrightarrow b \in B \Longrightarrow bij\text{-}betw\ f\ A\ B \Longrightarrow f\ a = b$ $\Longrightarrow bij\text{-}betw\ f\ (A - \{a\})\ (B - \{b\})$
  ⟨*proof*⟩

lemma $a \in A \Longrightarrow bij\text{-}betw\ f\ A\ B \Longrightarrow bij\text{-}betw\ f\ (A - \{a\})\ (B - \{f\ a\})$
  ⟨*proof*⟩

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 $(\bigoplus i \leftarrow xs.\ f\ i) = (\bigoplus i \leftarrow ys.\ f\ i)$
  ⟨*proof*⟩
lemma *monoid-sum-list-index-permutation*:
  assumes $distinct\ xs$
  assumes $distinct\ ys \lor length\ xs = length\ ys$
  assumes $bij\text{-}betw\ f\ (set\ xs)\ (set\ ys)$
  assumes $\bigwedge i.\ i \in set\ ys \Longrightarrow g\ i \in carrier\ G$
  shows $(\bigoplus i \leftarrow ys.\ g\ i) = (\bigoplus i \leftarrow xs.\ g\ (f\ i))$
  ⟨*proof*⟩

lemma *monoid-sum-list-split*:
  assumes[*simp*]: $\bigwedge i.\ i < b + c \Longrightarrow f\ i \in carrier\ G$
  shows $(\bigoplus l \leftarrow [0..<b].\ f\ l) \oplus (\bigoplus l \leftarrow [b..< b + c].\ f\ l) = (\bigoplus l \leftarrow [0..< b + c].$ $f\ l)$
  ⟨*proof*⟩

lemma *monoid-sum-list-splice*:
  assumes[*simp*]: $\bigwedge i.\ i < 2 * n \Longrightarrow f\ i \in carrier\ G$
  shows $(\bigoplus i \leftarrow [0..< 2 * n].\ f\ i) = (\bigoplus i \leftarrow [0..<n].\ f\ (2*i)) \oplus (\bigoplus i \leftarrow [0..<n].$ $f\ (2*i+1))$

⟨*proof*⟩

lemma *monoid-sum-list-even-odd-split*:
  assumes *even* (*n*::*nat*)
  assumes $\bigwedge i.\ i < n \Longrightarrow f\ i \in carrier\ G$
  shows $(\bigoplus i \leftarrow [0..<n].\ f\ i) = (\bigoplus i \leftarrow [0..< n\ div\ 2].\ f\ (2*i)) \oplus (\bigoplus i \leftarrow [0..< n\ div\ 2].\ f\ (2*i+1))$
  ⟨*proof*⟩

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 $\ominus\ (\bigoplus i \leftarrow xs.\ f\ i) = (\bigoplus i \leftarrow xs.\ \ominus\ f\ i)$
  ⟨*proof*⟩

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*⟩

end

context *ring*
begin

lemma *monoid-sum-list-const*:
  assumes[*simp*]: $c \in carrier\ R$
  shows $(\bigoplus i \leftarrow xs.\ c) = (nat\text{-}embedding\ (length\ xs)) \otimes c$
  ⟨*proof*⟩

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 $(\bigoplus i \leftarrow xs.\ f\ i \otimes y) = (\bigoplus i \leftarrow xs.\ f\ i) \otimes y$
  ⟨*proof*⟩

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 $(\bigoplus i \leftarrow xs.\ y \otimes f\ i) = y \otimes (\bigoplus i \leftarrow xs.\ f\ i)$
  ⟨*proof*⟩

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 $(\bigoplus i \leftarrow xs.\ f\ i) \otimes (\bigoplus j \leftarrow ys.\ g\ j) = (\bigoplus i \leftarrow xs.\ (\bigoplus j \leftarrow ys.\ f\ i \otimes g\ j))$
$\langle proof \rangle$

3.1 Kronecker delta

definition *delta* **where**
$delta\ i\ j = (if\ i = j\ then\ \mathbf{1}\ else\ \mathbf{0})$

lemma *delta-closed*[*simp*]: $delta\ i\ j \in carrier\ R$
$\langle proof \rangle$

lemma *delta-sym*: $delta\ i\ j = delta\ j\ i$
$\langle proof \rangle$

lemma *delta-refl*[*simp*]: $delta\ i\ i = \mathbf{1}$
$\langle proof \rangle$

lemma *monoid-sum-list-delta*[*simp*]:
assumes[*simp*]: $\bigwedge i.\ i < n \Longrightarrow f\ i \in carrier\ R$
assumes[*simp*]: $j < n$
shows $(\bigoplus i \leftarrow [0..<n].\ delta\ i\ j \otimes f\ i) = f\ j$
$\langle proof \rangle$

lemma *mononid-sum-list-only-delta*[*simp*]:
$j < n \Longrightarrow (\bigoplus i \leftarrow [0..<n].\ delta\ i\ j) = \mathbf{1}$
$\langle proof \rangle$

3.2 Power sums

lemma *geo-monoid-list-sum*:
assumes[*simp*]: $x \in carrier\ R$
shows $(\mathbf{1} \ominus x) \otimes (\bigoplus l \leftarrow [0..<r].\ x\ [\urcorner]\ l) = (\mathbf{1} \ominus x\ [\urcorner]\ r)$
$\langle proof \rangle$

rewrite $?x \in carrier\ R \Longrightarrow (?x\ [\urcorner]\ ?n)\ [\urcorner]\ ?m = ?x\ [\urcorner]\ (?n * ?m)$ and $?a * ?b = ?b * ?a$ inside power sum

lemma *monoid-pow-sum-nat-pow-pow*:
assumes $x \in carrier\ R$
shows $(\bigoplus i \leftarrow xs.\ f\ i \otimes x\ [\urcorner]\ ((g\ i :: nat) * h\ i)) = (\bigoplus i \leftarrow xs.\ f\ i \otimes (x\ [\urcorner]\ h\ i)\ [\urcorner]\ g\ i)$
$\langle proof \rangle$

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 $(\bigoplus i \leftarrow [0..<n].\ f\ i \otimes x\ [\hat{}]\ i) =$
    $(\bigoplus i \leftarrow [0..<l].\ f\ i \otimes x\ [\hat{}]\ i) \oplus$
    $x\ [\hat{}]\ l \otimes (\bigoplus i \leftarrow [0..<k].\ f\ (l + i) \otimes x\ [\hat{}]\ i)$
$\langle proof \rangle$

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 $(\bigoplus i \leftarrow [0..<n].\ f\ i \otimes x\ [\hat{}]\ (i * c)) =$
    $(\bigoplus i \leftarrow [0..<l].\ f\ i \otimes x\ [\hat{}]\ (i * c)) \oplus$
    $x\ [\hat{}]\ (l * c) \otimes (\bigoplus i \leftarrow [0..<k].\ f\ (l + i) \otimes x\ [\hat{}]\ (i * c))$
$\langle proof \rangle$

3.2.1 Algebraic operations

addition

lemma *monoid-pow-sum-add*:
  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 $(\bigoplus i \leftarrow xs.\ f\ i \otimes x\ [\hat{}]\ (i::nat)) \oplus (\bigoplus i \leftarrow xs.\ g\ i \otimes x\ [\hat{}]\ i) = (\bigoplus i \leftarrow xs.\ (f\ i \oplus g\ i) \otimes x\ [\hat{}]\ i)$
$\langle proof \rangle$

lemma *monoid-pow-sum-add'*:
  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 $(\bigoplus i \leftarrow xs.\ f\ i \otimes x\ [\hat{}]\ ((i::nat) * c)) \oplus (\bigoplus i \leftarrow xs.\ g\ i \otimes x\ [\hat{}]\ (i * c)) = (\bigoplus i \leftarrow xs.\ (f\ i \oplus g\ i) \otimes x\ [\hat{}]\ (i * c))$
$\langle proof \rangle$

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 $\ominus (\bigoplus i \leftarrow xs.\ f\ i \otimes x\ [\hat{}]\ (i :: nat)) = (\bigoplus i \leftarrow xs.\ (\ominus f\ i) \otimes x\ [\hat{}]\ i)$
$\langle proof \rangle$

minus

lemma *monoid-pow-sum-diff*:
  assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow f\ i \in carrier\ R$

assumes $\bigwedge i.\ i \in \mathit{set}\ \mathit{xs} \Longrightarrow g\ i \in \mathit{carrier}\ R$
assumes $x \in \mathit{carrier}\ R$
shows $(\bigoplus i \leftarrow \mathit{xs}.\ f\ i \otimes x\ [\urcorner]\ (i{::}\mathit{nat})) \ominus (\bigoplus i \leftarrow \mathit{xs}.\ g\ i \otimes x\ [\urcorner]\ (i :: \mathit{nat})) =$
$(\bigoplus i \leftarrow \mathit{xs}.\ (f\ i \ominus g\ i) \otimes x\ [\urcorner]\ i)$
⟨*proof*⟩

lemma *monoid-pow-sum-diff'*:
assumes $\bigwedge i.\ i \in \mathit{set}\ \mathit{xs} \Longrightarrow f\ i \in \mathit{carrier}\ R$
assumes $\bigwedge i.\ i \in \mathit{set}\ \mathit{xs} \Longrightarrow g\ i \in \mathit{carrier}\ R$
assumes $x \in \mathit{carrier}\ R$
shows $(\bigoplus i \leftarrow \mathit{xs}.\ f\ i \otimes x\ [\urcorner]\ ((i{::}\mathit{nat}) * c)) \ominus (\bigoplus i \leftarrow \mathit{xs}.\ g\ i \otimes x\ [\urcorner]\ (i * c)) =$
$(\bigoplus i \leftarrow \mathit{xs}.\ (f\ i \ominus g\ i) \otimes x\ [\urcorner]\ (i * c))$
⟨*proof*⟩

end

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

context *residues*
begin

lemma *monoid-sum-list-eq-sum-list*:
$(\bigoplus_R\ i \leftarrow \mathit{xs}.\ f\ i) = (\sum i \leftarrow \mathit{xs}.\ f\ i)\ \mathit{mod}\ m$
⟨*proof*⟩

lemma *monoid-sum-list-mod-in*:
$(\bigoplus_R\ i \leftarrow \mathit{xs}.\ f\ i) = (\bigoplus_R\ i \leftarrow \mathit{xs}.\ (f\ i)\ \mathit{mod}\ m)$
⟨*proof*⟩

lemma *monoid-sum-list-eq-sum-list'*:
$(\bigoplus_R\ i \leftarrow \mathit{xs}.\ f\ i\ \mathit{mod}\ m) = (\sum i \leftarrow \mathit{xs}.\ f\ i)\ \mathit{mod}\ m$
⟨*proof*⟩

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 $P\ (\mathit{if}\ C\ \mathit{then}\ A\ \mathit{else}\ B)\ (\mathit{if}\ C'\ \mathit{then}\ A'\ \mathit{else}\ B')$

$\langle proof \rangle$

lemma *if-leqI*:
  assumes $C \Longrightarrow A \leq t$
  assumes $\neg\ C \Longrightarrow B \leq t$
  shows $(\textit{if } C \textit{ then } A \textit{ else } B) \leq t$
  $\langle proof \rangle$

lemma *if-le-max*:
  $(\textit{if } C \textit{ then } (t1 :: 'a :: linorder) \textit{ else } t2) \leq \textit{max } t1\ t2$
  $\langle proof \rangle$

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

method *itrans* **for** *step* :: *'a* :: *order* =
  (*match* **conclusion in** $s \leq t$ **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** $P = (\leq)$] *Nat.le0 one-le-numeral*

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

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

  | $x \leq y$ **for** *x y* ⇒ ‹
    match conclusion in $g\ x \leq t$ *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*: $time\ (s \gg= f) = time\ s + time\ (f\ (val\ s))$
  $\langle proof \rangle$

lemma *time-tick*: $time\ (tick\ s) = 1$
  $\langle proof \rangle$

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-Not-tm*[*simp*]: *time* (*Not-tm x*) = *1*
⟨*proof*⟩

6.1.2 disj / conj

definition *disj-tm* **where** *disj-tm x y =1 return* ($x \vee y$)
definition *conj-tm* **where** *conj-tm x y =1 return* ($x \wedge y$)

lemma *val-disj-tm*[*simp*, *val-simp*]: *val* (*disj-tm x y*) = ($x \vee y$)
⟨*proof*⟩
lemma *time-disj-tm*[*simp*]: *time* (*disj-tm x y*) = *1*
⟨*proof*⟩
lemma *val-conj-tm*[*simp*, *val-simp*]: *val* (*conj-tm x y*) = ($x \wedge y$)
⟨*proof*⟩
lemma *time-conj-tm*[*simp*]: *time* (*conj-tm x y*) = *1*

⟨*proof*⟩

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*)
⟨*proof*⟩

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

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*
⟨*proof*⟩
lemma *time-fst-tm*[*simp*]: *time* (*fst-tm p*) = *1*
⟨*proof*⟩
lemma *val-snd-tm*[*simp*, *val-simp*]: *val* (*snd-tm p*) = *snd p*
⟨*proof*⟩
lemma *time-snd-tm*[*simp*]: *time* (*snd-tm p*) = *1*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-plus-nat-tm*[*simp*]: *time* (*plus-nat-tm m n*) = *m* + *1*
⟨*proof*⟩

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* {

$r \leftarrow$ *times-nat-tm m n*;
plus-nat-tm n r
}

lemma *val-times-nat-tm*[*simp*]: *val* (*times-nat-tm m n*) = *m* ∗ *n*
⟨*proof*⟩

lemma *time-times-nat-tm*[*simp*]: *time* (*times-nat-tm m n*) = *m* ∗ (*n* + *2*) + *1*
⟨*proof*⟩

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* {
$r \leftarrow$ *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*
⟨*proof*⟩

lemma *time-power-nat-tm-aux0*: *time* (*power-nat-tm 0 n*) = *2* ∗ *n* + *1*
⟨*proof*⟩

lemma *time-power-nat-tm-aux1*: *time* (*power-nat-tm 1 n*) = *5* ∗ *n* + *1*
⟨*proof*⟩

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*⟩

lemma *time-power-nat-tm-le*: *time* (*power-nat-tm m n*) ≤ *3* ∗ *m* ^ *Suc n* + *5* ∗ *n* + *1*
⟨*proof*⟩

lemma *time-power-nat-tm-2-le*: *time* (*power-nat-tm 2 n*) ≤ *12* ∗ *2* ^ *n*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-minus-nat-tm*[*simp*]: *time* (*minus-nat-tm m n*) = *min m n* + *1*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩
lemma *time-less-nat-tm-aux*: *time* (*less-nat-tm* (*m* + *k*) (*n* + *k*)) = *2* ∗ *k* + *time* (*less-nat-tm m n*)
⟨*proof*⟩

lemma *time-less-eq-nat-tm*: *time* (*less-eq-nat-tm n m*) = *2* ∗ *min n m* + *1* + *of-bool* (*m* < *n*)
⟨*proof*⟩
lemma *time-less-nat-tm*: *time* (*less-nat-tm m n*) = *2* ∗ *min m n* + *1* + *of-bool* (*m* < *n*)
⟨*proof*⟩

lemma *time-less-eq-nat-tm-le*: *time* (*less-eq-nat-tm n m*) ≤ *2* ∗ *min n m* + *2*
⟨*proof*⟩
lemma *time-less-nat-tm-le*: *time* (*less-nat-tm m n*) ≤ *2* ∗ *min m n* + *2*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *time-equal-nat-tm*: *time* (*equal-nat-tm x y*) = *min x y* + *1*
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *time-max-nat-tm*: *time* (*max-nat-tm x y*) = *2* ∗ *min x y* + *2* + *of-bool* (*y* < *x*)
  ⟨*proof*⟩

lemma *time-max-nat-tm-le*: *time* (*max-nat-tm x y*) ≤ *2* ∗ *min x y* + *3*
  ⟨*proof*⟩

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*⟩

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*)
  ⟨*proof*⟩

lemma *time-divmod-nat-tm-le*: *time* (*divmod-nat-tm m n*) ≤ *8* ∗ *m* + *2* ∗ *n* + *5*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-divide-nat-tm-le*: *time* (*divide-nat-tm m n*) ≤ *8* ∗ *m* + *2* ∗ *n* + *7*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-mod-nat-tm-le*: *time* (*mod-nat-tm m n*) ≤ *8* ∗ *m* + *2* ∗ *n* + *7*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *time-dvd-tm-le*: *time* (*dvd-tm a b*) ≤ *8* ∗ *b* + *2* ∗ *a* + *9*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-even-tm-le*: *time* (*even-tm a*) ≤ *8* ∗ *a* + *13*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-odd-tm-le*: *time* (*odd-tm a*) ≤ *8* ∗ *a* + *14*
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *time-take-tm*: *time* (*take-tm n xs*) = *min n* (*length xs*) + *1*
  ⟨*proof*⟩

lemma *time-take-tm-le*: *time* (*take-tm n xs*) ≤ *n* + *1*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *time-drop-tm*: *time* (*drop-tm n xs*) = *min n* (*length xs*) + *1*
  ⟨*proof*⟩

lemma *time-drop-tm-le*: *time* (*drop-tm n xs*) ≤ *n* + *1*
  ⟨*proof*⟩

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*

⟨*proof*⟩

lemma *time-append-tm*[*simp*]: *time* (*append-tm xs ys*) = *length xs* + *1*
⟨*proof*⟩

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* (λ*x y*. *val* (*f x y*)) *xs s*
⟨*proof*⟩

lemma *time-fold-tm-Cons*: *time* (*fold-tm* (λ*x y*. *return* (*x* # *y*)) *xs s*) = *length xs* + *1*
⟨*proof*⟩

6.4.5 *rev*

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

lemma *val-rev-tm*[*simp*, *val-simp*]: *val* (*rev-tm xs*) = *rev xs*
⟨*proof*⟩

lemma *time-rev-tm-le*[*simp*]: *time* (*rev-tm xs*) = *length xs* + *2*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-replicate-tm*: *time* (*replicate-tm n x*) = *n* + *1*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-gen-length-tm*[*simp*]: *time* (*gen-length-tm n xs*) = *length xs* + *1*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-length-tm*[*simp*]: *time* (*length-tm xs*) = *length xs* + *1*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-null-tm*[*simp*]: *time* (*null-tm xs*) = *1*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-butlast-tm*: *time* (*butlast-tm xs*) = *2* ∗ (*length xs* − *1*) + *1* + *of-bool* (*length xs* ≥ *1*)
⟨*proof*⟩

lemma *time-butlast-tm-le*: $time\ (butlast\text{-}tm\ xs) \le 2 * length\ xs + 1$
⟨*proof*⟩

6.4.10 *map*

fun $map\text{-}tm :: ('a \Rightarrow 'b\ tm) \Rightarrow 'a\ list \Rightarrow 'b\ list\ tm$ **where**
$map\text{-}tm\ f\ []\ =1\ return\ []$
$|\ map\text{-}tm\ f\ (x\ \#\ xs)\ =1\ do\ \{$
$\quad r \leftarrow f\ x;$
$\quad rs \leftarrow map\text{-}tm\ f\ xs;$
$\quad return\ (r\ \#\ rs)$
$\}$

lemma *val-map-tm*[*simp*, *val-simp*]: $val\ (map\text{-}tm\ f\ xs) = map\ (\lambda x.\ val\ (f\ x))\ xs$
⟨*proof*⟩

lemma *time-map-tm*: $time\ (map\text{-}tm\ f\ xs) = (\sum i \leftarrow xs.\ time\ (f\ i)) + length\ xs + 1$
⟨*proof*⟩

lemma *time-map-tm-constant*:
assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow time\ (f\ i) = c$
shows $time\ (map\text{-}tm\ f\ xs) = (c + 1) * length\ xs + 1$
⟨*proof*⟩

lemma *time-map-tm-bounded*:
assumes $\bigwedge i.\ i \in set\ xs \Longrightarrow time\ (f\ i) \le c$
shows $time\ (map\text{-}tm\ f\ xs) \le (c + 1) * length\ xs + 1$
⟨*proof*⟩

6.4.11 *foldl*

fun $foldl\text{-}tm :: ('a \Rightarrow 'b \Rightarrow 'a\ tm) \Rightarrow 'a \Rightarrow 'b\ list \Rightarrow 'a\ tm$ **where**
$foldl\text{-}tm\ f\ a\ []\ =1\ return\ a$
$|\ foldl\text{-}tm\ f\ a\ (x\ \#\ xs)\ =1\ do\ \{$
$\quad r \leftarrow f\ a\ x;$
$\quad foldl\text{-}tm\ f\ r\ xs$
$\}$

lemma *val-foldl-tm*[*simp*, *val-simp*]: $val\ (foldl\text{-}tm\ f\ a\ xs) = foldl\ (\lambda x\ y.\ val\ (f\ x\ y))\ a\ xs$
⟨*proof*⟩

6.4.12 *concat*

fun *concat-tm* **where**
$concat\text{-}tm\ []\ =1\ return\ []$
$|\ concat\text{-}tm\ (x\ \#\ xs)\ =1\ do\ \{$
$\quad r \leftarrow concat\text{-}tm\ xs;$
$\quad append\text{-}tm\ x\ r$

}

lemma *val-concat-tm*[*simp*, *val-simp*]: *val* (*concat-tm xs*) = *concat xs*
⟨*proof*⟩

lemma *time-concat-tm*[*simp*]: *time* (*concat-tm xs*) = *1* + *2* ∗ *length xs* + *length* (*concat xs*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-nth-tm*[*simp*]:
assumes *i* < *length xs*
shows *time* (*nth-tm xs i*) = *i* + *1*
⟨*proof*⟩

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*
⟨*proof*⟩

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

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*
⟨*proof*⟩

lemma *time-map2-tm-bounded*:
  assumes *length xs* = *length ys*
  assumes $\bigwedge x\ y.\ x \in set\ xs \Longrightarrow y \in set\ ys \Longrightarrow time\ (f\ x\ y) \leq c$
  shows $time\ (map2\text{-}tm\ f\ xs\ ys) \leq (c + 2) * length\ xs + 3$
⟨*proof*⟩

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* [])
}
  ⟨*proof*⟩
termination ⟨*proof*⟩
declare *upt-tm.simps*[*simp del*]

lemma *val-upt-tm*[*simp*, *val-simp*]: *val* (*upt-tm i j*) = [*i*..<*j*]
  ⟨*proof*⟩
lemma *time-upt-tm-le*: $time\ (upt\text{-}tm\ i\ j) \leq (j - i) * (2 * j + 3) + 2 * j + 2$
⟨*proof*⟩

lemma *time-upt-tm-le'*: $time\ (upt\text{-}tm\ i\ j) \leq 2 * j * j + 5 * j + 2$
  ⟨*proof*⟩

6.5 Syntactic sugar

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

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

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

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

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

consts *less-tm* :: $'a \Rightarrow 'a \Rightarrow bool\ tm$
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* $\circ$ *from-type* = *id*
  assumes *from-to*: *from-type* $\circ$ *to-type* = f
begin

lemma *to-from-elem*[*simp*]: *to-type* (*from-type* x) = x
  ⟨*proof*⟩
lemma *from-to-elem*: *from-type* (*to-type* x) = $f\ x$
  ⟨*proof*⟩

lemma *f-idem*: $f \circ f = f$
⟨*proof*⟩

corollary *f-idem-elem*[*simp*]: $f\ (f\ x) = f\ x$
  ⟨*proof*⟩

lemma *f-from*: $f \circ$ *from-type* = *from-type*
⟨*proof*⟩

corollary *f-from-elem*[*simp*]: f (*from-type* x) = *from-type* x
  ⟨*proof*⟩

lemma *to-f*: *to-type* $\circ f$ = *to-type*
⟨*proof*⟩

corollary *to-f-elem*[*simp*]: *to-type* ($f\ x$) = *to-type* x
  ⟨*proof*⟩

lemma *f-fixed-point-iff*: $f\ x = x \longleftrightarrow (\exists\, y.\ x =$ *from-type* $y)$
⟨*proof*⟩

lemma *f-fixed-point-iff'*: $f\ x = x \longleftrightarrow x =$ *from-type* (*to-type* x)
  ⟨*proof*⟩

lemma *range-f-range-from*: *range* f = *range from-type*
⟨*proof*⟩

lemma *to-eq-iff-f-eq*: *to-type* x = *to-type* $y \longleftrightarrow f\ x = f\ y$
⟨*proof*⟩

lemma *from-inj*: *inj from-type*
⟨*proof*⟩

end

lemma *from-to-f-criterion*:
assumes *to-type* ∘ *from-type* = *id*
assumes f ∘ *from-type* = *from-type*
assumes $\bigwedge x\ y.$ *to-type* x = *to-type* $y \Longrightarrow f\ x = f\ y$
shows *from-type* ∘ *to-type* = f
⟨*proof*⟩

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* :: $'a \Rightarrow 'b$
fixes *to-type* :: $'b \Rightarrow 'a$
fixes *represented-set* :: $'a$ *set*
assumes *to-from*: $\bigwedge x.\ x \in$ *represented-set* $\Longrightarrow$ *to-type* (*from-type* x) = x
assumes *to-type-in-represented-set*: $\bigwedge y.$ *to-type* $y \in$ *represented-set*
begin

definition *reduce* **where**
reduce $x \equiv$ *from-type* (*to-type* x)

abbreviation *reduced* **where**
reduced $x \equiv$ *reduce* $x = x$

lemma *reduce-reduce*[*simp*]: *reduced* (*reduce* x)
⟨*proof*⟩

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

lemma *range-reduce*: *representations* = *range reduce*
⟨*proof*⟩

corollary *reduced-from-type*[*simp*]: $x \in$ *represented-set* $\Longrightarrow$ *reduced* (*from-type* x)

$\langle proof \rangle$

lemma *to-type-reduce*: *to-type* (*reduce x*) = *to-type x*
$\langle proof \rangle$

lemma *reduced-iff*: *reduced x* $\longleftrightarrow$ ($\exists$ *y*$\in$*represented-set*. *x* = *from-type y*)
$\langle proof \rangle$

lemma *to-eq-iff-f-eq*: *to-type x* = *to-type y* $\longleftrightarrow$ *reduce x* = *reduce y*
$\langle proof \rangle$

lemma *from-inj*: *inj-on from-type represented-set*
$\langle proof \rangle$

corollary *from-bij-betw*: *bij-betw from-type represented-set representations*
$\langle proof \rangle$

lemma *correctness-to-from*:
fixes *h* :: $'a \Rightarrow 'a \Rightarrow 'a$
fixes *g* :: $'b \Rightarrow 'b \Rightarrow 'b$
assumes $\bigwedge$*x y*. *to-type* (*g x y*) = *h* (*to-type x*) (*to-type y*)
shows $\bigwedge$*x y*. *x* $\in$ *represented-set* $\Longrightarrow$ *y* $\in$ *represented-set* $\Longrightarrow$ *reduce* (*g* (*from-type x*) (*from-type y*)) = *from-type* (*h x y*)
$\langle proof \rangle$

end

lemma *from-to-f-criterion*:
assumes $\bigwedge$*x*. *x* $\in$ *represented-set* $\Longrightarrow$ *to-type* (*from-type x*) = *x*
assumes $\bigwedge$*x*. *x* $\in$ *represented-set* $\Longrightarrow$ *f* (*from-type x*) = *from-type x*
assumes $\bigwedge$*x y*. *to-type x* = *to-type y* $\Longrightarrow$ *f x* = *f y*
assumes $\bigwedge$*y*. *to-type y* $\in$ *represented-set*
shows $\bigwedge$*x*. *from-type* (*to-type x*) = *f x*
$\langle proof \rangle$

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* $\Longrightarrow$ $\exists$ *xs1 xs2*. *xs* = *xs1* @ (*xs* ! *i*) # *xs2* $\wedge$ *length xs1* = *i*
$\langle proof \rangle$

lemma *list-is-replicate-iff*: $xs = replicate\ (length\ xs)\ x \longleftrightarrow (\forall\, i \in \{0..<length\ xs\}.\ xs\ !\ i = x)$
⟨*proof*⟩

lemma *list-is-replicate-iff2*: $xs = replicate\ (length\ xs)\ x \longleftrightarrow set\ xs = \{x\} \vee xs = []$
⟨*proof*⟩

lemma *set-bool-list*: $set\ xs \subseteq \{True,\ False\}$
⟨*proof*⟩
lemma *bool-list-is-replicate-if*:
assumes $a \notin set\ xs$ **shows** $xs = replicate\ (length\ xs)\ (\neg\ a)$
⟨*proof*⟩

lemma *bit-strong-decomp-2*: $\exists\, ys\ zs.\ xs = ys\ @\ a\ \#\ zs \Longrightarrow \exists\, ys'\ n.\ xs = ys'\ @\ a\ \#\ (replicate\ n\ (\neg\ a))$
⟨*proof*⟩

lemma *bit-strong-decomp-1*: $\exists\, ys\ zs.\ xs = ys\ @\ a\ \#\ zs \Longrightarrow \exists\, ys'\ n.\ xs = (replicate\ n\ (\neg\ a)\ @\ a\ \#\ ys')$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *eval-bool-leq-1*: $eval\text{-}bool\ a \leq 1$
⟨*proof*⟩

lemma *eval-bool-inj*: $eval\text{-}bool\ a = eval\text{-}bool\ b \Longrightarrow a = b$
⟨*proof*⟩

fun *to-nat* :: *nat-lsbf* ⇒ *nat* **where**
to-nat [] = *0*
| $to\text{-}nat\ (x\#xs) = (eval\text{-}bool\ x) + 2 * to\text{-}nat\ xs$

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

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

lemma *to-nat-from-nat*[*simp*]: *to-nat* (*from-nat x*) = *x*
⟨*proof*⟩

lemma *to-nat-explicitly*: *to-nat xs* = ($\sum$ *i* ← [*0..<length xs*]. *eval-bool* (*xs* ! *i*) ∗ *2* ^ *i*)
⟨*proof*⟩

lemma *to-nat-app*: *to-nat* (*xs* @ *ys*) = *to-nat xs* + (*2* ^ *length xs*) ∗ *to-nat ys*
⟨*proof*⟩

lemma *to-nat-length-upper-bound*: *to-nat xs* ≤ *2* ^ (*length xs*) − *1*
⟨*proof*⟩
lemma *to-nat-length-bound*: *to-nat xs* < *2* ^ *length xs*
⟨*proof*⟩
lemma *to-nat-length-lower-bound*: *to-nat* (*xs* @ [*True*]) ≥ *2* ^ *length xs*
⟨*proof*⟩

lemma *to-nat-replicate-false*[*simp*]: *to-nat* (*replicate n False*) = *0*
⟨*proof*⟩

lemma *to-nat-one-bit*[*simp*]: *to-nat* (*replicate n False* @ [*True*]) = *2* ^ *n*
⟨*proof*⟩

lemma *to-nat-replicate-true*[*simp*]: *to-nat* (*replicate n True*) = *2* ^ *n* − *1*
⟨*proof*⟩

lemma *to-nat xs* = *0* ⟷ (∃ *n*. *xs* = *replicate n False*)
⟨*proof*⟩

lemma *to-nat-app-replicate*[*simp*]: *to-nat* (*xs* @ *replicate n False*) = *to-nat xs*
⟨*proof*⟩

lemma *change-bit-ineq*: *length xs* = *length ys* ⟹ *to-nat* (*xs* @ *False* # *zs*) < *to-nat* (*ys* @ *True* # *zs*)
⟨*proof*⟩

lemma *to-nat-ineq-imp-False-bit*: *to-nat xs* < *2* ^ *length xs* − *1* ⟹ ∃ *ys zs*. *xs* = *ys* @ *False* # *zs*
⟨*proof*⟩

lemma *to-nat-bound-to-length-bound*: *to-nat xs* ≥ *2* ^ *n* ⟹ *length xs* ≥ *n* + *1*
⟨*proof*⟩

lemma *to-nat-drop-take*: *to-nat xs* = *to-nat* (*take k xs*) + *2* ^ *k* ∗ *to-nat* (*drop k xs*)
⟨*proof*⟩

lemma *to-nat-take*: *to-nat* (*take k xs*) = *to-nat xs mod 2* ^ *k*
⟨*proof*⟩

lemma *to-nat-drop*: *to-nat* (*drop k xs*) = *to-nat xs div 2* ^ *k*
⟨*proof*⟩

lemma *to-nat-nth-True-bound*:
  assumes *i* < *length xs*
  assumes *xs* ! *i* = *True*
  shows *to-nat xs* ≥ *2* ^ *i*
⟨*proof*⟩

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*
  ⟨*proof*⟩
corollary *truncate-eqI*[*simp*]: *xs* = *ys* @ (*replicate n False*) ⟹ *truncate xs* = *truncate ys*
  ⟨*proof*⟩

lemma *replicate-truncate-reversed*: ∃ *n*. (*replicate n False*) @ *truncate-reversed xs* = *xs*
⟨*proof*⟩
corollary *truncate-replicate*: ∃ *n*. *truncate xs* @ (*replicate n False*) = *xs*
⟨*proof*⟩
lemma *decompose-trailing-zeros*: *xs* = *truncate xs* @ (*replicate* (*length xs* − *length* (*truncate xs*)) *False*)
  ⟨*proof*⟩

lemma *truncate-reversed-length-ineq*: *length* (*truncate-reversed xs*) ≤ *length xs*
  ⟨*proof*⟩
lemma *truncate-length-ineq*: *length* (*truncate xs*) ≤ *length xs*
  ⟨*proof*⟩

lemma *truncate-reversed-fixed-point-iff*: *truncate-reversed x* = *x* ⟷ (*x* = [] ∨ *hd x* = *True*)
⟨*proof*⟩

lemma *truncated-iff*: *truncated* $x \longleftrightarrow (x = [] \vee \mathit{last}\ x = \mathit{True})$
⟨*proof*⟩

lemma *hd-truncate-reversed*: *truncate-reversed* $xs \neq [] \Longrightarrow \mathit{hd}\ (\mathit{truncate\text{-}reversed}\ xs) = \mathit{True}$
⟨*proof*⟩

lemma *last-truncate*: *truncate* $xs \neq [] \Longrightarrow \mathit{last}\ (\mathit{truncate}\ xs) = \mathit{True}$
⟨*proof*⟩

lemma *truncate-truncate*[*simp*]: $\mathit{truncate}\ (\mathit{truncate}\ xs) = \mathit{truncate}\ xs$
⟨*proof*⟩

lemma *truncate-reversed-Nil-iff*: *truncate-reversed* $xs = [] \longleftrightarrow (\exists\, n.\ xs = \mathit{replicate}\ n\ \mathit{False})$
⟨*proof*⟩

lemma *truncate-Nil-iff*: *truncate* $xs = [] \longleftrightarrow (\exists\, n.\ xs = \mathit{replicate}\ n\ \mathit{False})$
⟨*proof*⟩

corollary *truncate-neq-Nil*: *truncate* $xs \neq [] \Longrightarrow \exists\, ys\ zs.\ xs = ys\ @\ \mathit{True}\ \#\ zs$
⟨*proof*⟩

lemma *truncate-Cons*: $\mathit{truncate}\ (a\ \#\ xs) = (\mathit{if}\ \neg a \wedge (\mathit{truncate}\ xs = [])\ \mathit{then}\ []\ \mathit{else}\ a\ \#\ \mathit{truncate}\ xs)$
⟨*proof*⟩

lemma *truncate-eq-Cons*: $\mathit{truncate}\ xs = \mathit{truncate}\ ys \Longrightarrow \mathit{truncate}\ (a\ \#\ xs) = \mathit{truncate}\ (a\ \#\ ys)$
⟨*proof*⟩

lemma *truncate-as-take*: $\bigwedge xs.\ \exists\, n.\ \mathit{truncate}\ xs = \mathit{take}\ n\ xs$
⟨*proof*⟩

lemma *to-nat-zero-iff*: *to-nat* $xs = 0 \longleftrightarrow \mathit{truncate}\ xs = []$
⟨*proof*⟩

lemma *to-nat-eq-imp-truncate-eq*: *to-nat* xs = *to-nat* $ys \Longrightarrow \mathit{truncate}\ xs = \mathit{truncate}\ ys$
⟨*proof*⟩

lemma *truncate-from-nat*[*simp*]: *truncate* (*from-nat* x) = *from-nat* x
⟨*proof*⟩

lemma *truncate-and-length-eq-imp-eq*:
assumes $\mathit{truncate}\ xs = \mathit{truncate}\ ys\ \mathit{length}\ xs = \mathit{length}\ ys$
shows $xs = ys$
⟨*proof*⟩

lemma *nat-lsbf-eqI*:
assumes *to-nat xs* = *to-nat ys*
assumes *length xs* = *length ys*
shows *xs* = *ys*
⟨*proof*⟩

interpretation *nat-lsbf*: *abstract-representation from-nat to-nat truncate*
⟨*proof*⟩

lemma *truncated-Cons-imp-truncated-tl*: *truncated* (*x* # *xs*) $\Longrightarrow$ *truncated xs*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *length-fill*[*intro*]: *length xs* ≤ *n* $\Longrightarrow$ *length* (*fill n xs*) = *n*
⟨*proof*⟩

lemma *take-id*: *length xs* = *k* $\Longrightarrow$ *take k xs* = *xs*
⟨*proof*⟩
lemma *fill-id*: *length xs* ≥ *k* $\Longrightarrow$ *fill k xs* = *xs*
⟨*proof*⟩

lemma *length-fill'*: *length* (*fill n xs*) = *max n* (*length xs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *truncate-fill*: *truncate* (*fill k xs*) = *truncate xs*
⟨*proof*⟩

lemma *fill-truncate*: *length xs* ≤ *k* $\Longrightarrow$ *fill k* (*truncate xs*) = *fill k xs*
⟨*proof*⟩

lemma *fill-take-com*: *fill k* (*take k xs*) = *take k* (*fill k xs*)
⟨*proof*⟩

lemma *to-nat-length-lower-bound-truncated*: *xs* ≠ [] $\Longrightarrow$ *truncated xs* $\Longrightarrow$ *to-nat xs* ≥ *2* ^ (*length xs* − *1*)
⟨*proof*⟩

lemma *to-nat-length-bound-truncated*: *truncated xs* $\Longrightarrow$ *to-nat xs* $<$ *2* $\hat{}$ *n* $\Longrightarrow$ *length xs* $\leq$ *n*
⟨*proof*⟩

8.4 Right-shifts

definition *shift-right* :: *nat* $\Rightarrow$ *nat-lsbf* $\Rightarrow$ *nat-lsbf* **where**
shift-right n xs = (*replicate n False*) @ *xs*

lemma *to-nat-shift-right*[*simp*]: *to-nat* (*shift-right n xs*) = *2* $\hat{}$ *n* $*$ *to-nat xs*
⟨*proof*⟩

lemma *length-shift-right*[*simp*]: *length* (*shift-right n xs*) = *n* + *length xs*
⟨*proof*⟩

8.5 Subdividing lists

8.5.1 Splitting a list in two blocks

fun *split-at* :: *nat* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list* $\times$ *'a list* **where**
split-at m xs = (*take m xs*, *drop m xs*)

definition *split* :: *nat-lsbf* $\Rightarrow$ *nat-lsbf* $\times$ *nat-lsbf* **where**
split xs = (*let n* = *length xs div* (*2*::*nat*) *in split-at n xs*)

lemma *app-split*: *split xs* = (*x0*, *x1*) $\Longrightarrow$ *xs* = *x0* @ *x1*
⟨*proof*⟩

lemma *length-split*: *length xs mod 2* = *0* $\Longrightarrow$ *split xs* = (*x0*, *x1*) $\Longrightarrow$ *length x0* = *length xs div 2* $\wedge$ *length x1* = *length xs div 2*
⟨*proof*⟩

lemma *length-split-le*:
assumes *split xs* = (*x0*, *x1*)
shows *length x0* $\leq$ *length xs* **and** *length x1* $\leq$ *length xs*
⟨*proof*⟩

8.5.2 Splitting a list in multiple blocks

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

fun *subdivide* :: *nat* $\Rightarrow$ *'a list* $\Rightarrow$ *'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* $(2 \mathbin{\char`\^} 2)$ $[0..{<}2 \mathbin{\char`\^} 6]$

lemma *concat-subdivide*: $n > 0 \Longrightarrow concat\ (subdivide\ n\ xs) = xs$
 ⟨*proof*⟩

lemma *subdivide-step*:
 assumes $n > 0$
 assumes $xs \neq []$
 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*⟩

lemma *subdivide-step'*:
 assumes $n > 0$
 assumes $xs \neq []$
 shows $subdivide\ n\ xs = (take\ n\ xs)\ \#\ subdivide\ n\ (drop\ n\ xs)$
 ⟨*proof*⟩

lemma *subdivide-correct*:
 assumes $n > 0$
 assumes $length\ xs = n * k$
 shows $length\ (subdivide\ n\ xs) = k \wedge (x \in set\ (subdivide\ n\ xs) \longrightarrow length\ x = n)$
 ⟨*proof*⟩

lemma *nth-nth-subdivide*:
 assumes $n > 0$
 assumes $length\ xs = n * k$
 assumes $i < k$ $j < n$
 shows $subdivide\ n\ xs\ !\ i\ !\ j = xs\ !\ (i * n + j)$
 ⟨*proof*⟩

lemma *subdivide-concat*:
 assumes $n > 0$
 assumes $\bigwedge i.\ i < length\ xs \Longrightarrow length\ (xs\ !\ i) = n$
 shows $subdivide\ n\ (concat\ xs) = xs$
⟨*proof*⟩

lemma *to-nat-subdivide*:
 assumes $n > 0$
 assumes $length\ xs = n * k$
 shows $to\text{-}nat\ xs = (\sum i \leftarrow [0..{<}k].\ to\text{-}nat\ (subdivide\ n\ xs\ !\ i) * 2 \mathbin{\char`\^} (i * n))$
 ⟨*proof*⟩

8.6 The *bitsize* function

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

fun *bitsize* :: $nat \Rightarrow nat$ **where**
$bitsize\ 0 = 0$

| $bitsize\ n = 1 + bitsize\ (n\ div\ 2)$

lemma *bitsize-is-floorlog*: $bitsize = floorlog\ 2$
⟨*proof*⟩

corollary *bitsize-bitlen*: $int\ (bitsize\ n) = bitlen\ (int\ n)$
⟨*proof*⟩

lemma *bitsize-eq*: $bitsize\ n = length\ (from\text{-}nat\ n)$
⟨*proof*⟩

lemma *bitsize-zero-iff*: $bitsize\ n = 0 \longleftrightarrow n = 0$
⟨*proof*⟩

lemma *truncated-iff'*: $truncated\ x \longleftrightarrow length\ x = bitsize\ (to\text{-}nat\ x)$
⟨*proof*⟩

lemma *bitsize-length*: $bitsize\ n \leq k \longleftrightarrow n < 2\ \hat{}\ k$
⟨*proof*⟩

lemma *two-pow-bitsize-pos-bound*: $n > 0 \Longrightarrow 2\ \hat{}\ bitsize\ n \leq 2 * n$
⟨*proof*⟩

lemma *two-pow-bitsize-bound*: $2\ \hat{}\ bitsize\ n \leq 2 * n + 1$
⟨*proof*⟩

lemma *bitsize-mono*: $n1 \leq n2 \Longrightarrow bitsize\ n1 \leq bitsize\ n2$
⟨*proof*⟩

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

lemma *power-of-2-recursion*: $(\exists k.\ (n::nat) = 2\ \hat{}\ k) \longleftrightarrow (n = 1 \lor (n\ mod\ 2 = 0 \land (\exists k.\ n\ div\ 2 = 2\ \hat{}\ k)))$
⟨*proof*⟩

fun *is-power-of-2* :: $nat \Rightarrow bool$ **where**
$is\text{-}power\text{-}of\text{-}2\ 0 = False$
| $is\text{-}power\text{-}of\text{-}2\ (Suc\ 0) = True$
| $is\text{-}power\text{-}of\text{-}2\ n = ((n\ mod\ 2 = 0) \land is\text{-}power\text{-}of\text{-}2\ (n\ div\ 2))$

lemma *is-power-of-2-correct*: $is\text{-}power\text{-}of\text{-}2\ n \longleftrightarrow (\exists k.\ n = 2\ \hat{}\ k)$
⟨*proof*⟩

fun *next-power-of-2* :: $nat \Rightarrow nat$ **where**
$next\text{-}power\text{-}of\text{-}2\ n = (if\ is\text{-}power\text{-}of\text{-}2\ n\ then\ n\ else\ 2\ \hat{}\ (bitsize\ n))$

lemma *next-power-of-2-lower-bound*: $next\text{-}power\text{-}of\text{-}2\ k \geq k$
⟨*proof*⟩

lemma *next-power-of-2-upper-bound*:
  assumes $k \neq 0$
  shows $\textit{next-power-of-2}\ k \leq 2 * k$
  ⟨*proof*⟩

lemma *next-power-of-2-upper-bound'*: $\textit{next-power-of-2}\ k \leq 2 * k + 1$
  ⟨*proof*⟩

lemma *next-power-of-2-is-power-of-2*: $\exists k.\ \textit{next-power-of-2}\ n = 2\ \widehat{}\ k$
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*⟩

lemma *length-inc-nat-lower*: *length* (*inc-nat xs*) ≥ *length xs*
  ⟨*proof*⟩

lemma *length-inc-nat-upper*: *length* (*inc-nat xs*) ≤ *length xs* + *1*
  ⟨*proof*⟩

lemma *inc-nat-nonempty*: *inc-nat xs* ≠ []
  ⟨*proof*⟩

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

$\langle proof\rangle$

lemma *inc-nat-replicate-True-2*: $\textit{inc-nat}\ (\textit{replicate}\ m\ \textit{True}\ @\ \textit{False}\ \#\ ys) = \textit{replicate}\ m\ \textit{False}\ @\ \textit{True}\ \#\ ys$
$\langle proof\rangle$

lemma *length-inc-nat-iff*: $\textit{length}\ (\textit{inc-nat}\ xs) = \textit{length}\ xs \longleftrightarrow (\exists\, ys\ zs.\ xs = ys\ @\ \textit{False}\ \#\ zs)$
$\langle proof\rangle$

lemma *inc-nat-last-bit-True*: $\textit{length}\ (\textit{inc-nat}\ xs) = \textit{Suc}\ (\textit{length}\ xs) \Longrightarrow \exists\, zs.\ \textit{inc-nat}\ xs = zs\ @\ [\textit{True}]$
$\langle proof\rangle$

lemma *inc-nat-truncated*: $\textit{truncated}\ xs \Longrightarrow \textit{truncated}\ (\textit{inc-nat}\ xs)$
$\langle proof\rangle$

lemma *inc-nat-correct*: $\textit{to-nat}\ (\textit{inc-nat}\ xs) = \textit{to-nat}\ xs + \mathit{1}$
$\langle proof\rangle$

lemma *length-inc-nat*: $\textit{length}\ (\textit{inc-nat}\ xs) = \textit{max}\ (\textit{length}\ xs)\ (\textit{floorlog}\ \mathit{2}\ (\textit{to-nat}\ xs + \mathit{1}))$
$\langle proof\rangle$

8.7.2 Addition with a carry bit

fun *add-carry* :: $\textit{bool} \Rightarrow \textit{nat-lsbf} \Rightarrow \textit{nat-lsbf} \Rightarrow \textit{nat-lsbf}$ **where**
$\textit{add-carry}\ \textit{False}\ []\ y = y$
$\mid \textit{add-carry}\ \textit{False}\ x\ [] = x$
$\mid \textit{add-carry}\ \textit{True}\ []\ y = \textit{inc-nat}\ y$
$\mid \textit{add-carry}\ \textit{True}\ x\ [] = \textit{inc-nat}\ x$
$\mid \textit{add-carry}\ c\ (x\#xs)\ (y\#ys) = (\textit{let}\ (a,\ b) = \textit{bit-add-carry}\ c\ x\ y\ \textit{in}\ a\#(\textit{add-carry}\ b\ xs\ ys))$

lemma *add-carry-correct*: $\textit{to-nat}\ (\textit{add-carry}\ c\ x\ y) = \textit{eval-bool}\ c + \textit{to-nat}\ x + \textit{to-nat}\ y$
$\langle proof\rangle$

lemma *length-add-carry'*: $\textit{length}\ (\textit{add-carry}\ c\ xs\ ys) = \textit{max}\ (\textit{length}\ xs)\ (\textit{length}\ ys) + \textit{of-bool}\ (\textit{to-nat}\ xs + \textit{to-nat}\ ys + \textit{of-bool}\ c \geq \mathit{2}\ \widehat{}\ \textit{max}\ (\textit{length}\ xs)\ (\textit{length}\ ys))$
$\langle proof\rangle$

lemma *length-add-carry*: $\textit{length}\ (\textit{add-carry}\ c\ xs\ ys) = \textit{max}\ (\textit{max}\ (\textit{length}\ xs)\ (\textit{length}\ ys))\ (\textit{floorlog}\ \mathit{2}\ (\textit{of-bool}\ c + \textit{to-nat}\ xs + \textit{to-nat}\ ys))$
$\langle proof\rangle$

lemma *length-add-carry-lower*: $\textit{length}\ (\textit{add-carry}\ c\ xs\ ys) \geq \textit{max}\ (\textit{length}\ xs)\ (\textit{length}\ ys)$
$\langle proof\rangle$

lemma *length-add-carry-upper*: $length\ (add\text{-}carry\ c\ xs\ ys) \leq max\ (length\ xs)\ (length\ ys) + 1$
⟨*proof*⟩

lemma *add-carry-last-bit-True*: $length\ (add\text{-}carry\ c\ xs\ ys) = max\ (length\ xs)\ (length\ ys) + 1 \Longrightarrow \exists\, zs.\ add\text{-}carry\ c\ xs\ ys = zs\ @\ [True]$
⟨*proof*⟩

lemma *add-carry-com*: $add\text{-}carry\ c\ xs\ ys = add\text{-}carry\ c\ ys\ xs$
⟨*proof*⟩

lemma *add-carry-rNil*[*simp*]: $add\text{-}carry\ True\ y\ [] = inc\text{-}nat\ y$
⟨*proof*⟩
lemma *add-carry-rNil-nocarry*[*simp*]: $add\text{-}carry\ False\ y\ [] = y$
⟨*proof*⟩

lemma *add-carry-True-inc-nat*:
$add\text{-}carry\ True\ xs\ ys = inc\text{-}nat\ (add\text{-}carry\ False\ xs\ ys) \wedge$
$add\text{-}carry\ True\ xs\ ys = add\text{-}carry\ False\ (inc\text{-}nat\ xs)\ ys \wedge$
$add\text{-}carry\ True\ xs\ ys = add\text{-}carry\ False\ xs\ (inc\text{-}nat\ ys)$
⟨*proof*⟩

lemma *inc-nat-add-carry*:
$inc\text{-}nat\ (add\text{-}carry\ c\ xs\ ys) = add\text{-}carry\ c\ (inc\text{-}nat\ xs)\ ys \wedge$
$inc\text{-}nat\ (add\text{-}carry\ c\ xs\ ys) = add\text{-}carry\ c\ xs\ (inc\text{-}nat\ ys)$
⟨*proof*⟩

lemma *add-carry-inc-nat-simps*:
$add\text{-}carry\ True\ xs\ ys = inc\text{-}nat\ (add\text{-}carry\ False\ xs\ ys)$
$add\text{-}carry\ False\ (inc\text{-}nat\ xs)\ ys = inc\text{-}nat\ (add\text{-}carry\ False\ xs\ ys)$
$add\text{-}carry\ False\ xs\ (inc\text{-}nat\ ys) = inc\text{-}nat\ (add\text{-}carry\ False\ xs\ ys)$
⟨*proof*⟩

lemma *add-carry-assoc*: $add\text{-}carry\ c2\ (add\text{-}carry\ c1\ xs\ ys)\ zs = add\text{-}carry\ c1\ xs\ (add\text{-}carry\ c2\ ys\ zs)$
⟨*proof*⟩

lemma *truncated-add-carry*:
assumes *truncated xs truncated ys*
shows *truncated* (*add-carry c xs ys*)
⟨*proof*⟩

8.7.3 Addition

definition $add\text{-}nat :: nat\text{-}lsbf \Rightarrow nat\text{-}lsbf \Rightarrow nat\text{-}lsbf$ **where**
$add\text{-}nat\ x\ y = add\text{-}carry\ False\ x\ y$

corollary *length-add-nat-lower*: *length* (*add-nat xs ys*) $\geq$ *max* (*length xs*) (*length ys*)
⟨*proof*⟩

corollary *length-add-nat-upper*: *length* (*add-nat xs ys*) $\leq$ *max* (*length xs*) (*length ys*) + *1*
⟨*proof*⟩

corollary *add-nat-last-bit-True*: *length* (*add-nat xs ys*) = *max* (*length xs*) (*length ys*) + *1* $\Longrightarrow$ $\exists$ *zs*. *add-nat xs ys* = *zs* @ [*True*]
⟨*proof*⟩

lemma *add-nat-correct*: *to-nat* (*add-nat x y*) = *to-nat x* + *to-nat y*
⟨*proof*⟩

corollary *add-nat-com*: *add-nat xs ys* = *add-nat ys xs*
⟨*proof*⟩

corollary *add-nat-assoc*: *add-nat xs* (*add-nat ys zs*) = *add-nat* (*add-nat xs ys*) *zs*
⟨*proof*⟩

corollary *truncated-add-nat*:
assumes *truncated xs truncated ys*
shows *truncated* (*add-nat xs ys*)
⟨*proof*⟩

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* $\Longrightarrow$ *compare-nat-same-length-reversed xs ys* $\longleftrightarrow$ *to-nat* (*rev xs*) $\leq$ *to-nat* (*rev ys*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

lemma *to-nat-complement*: *to-nat* (*map Not xs*) = *2* ^ (*length xs*) − *1* − *to-nat xs*
⟨*proof*⟩

lemma *to-nat-butlast*: *zs* = *xs* @ [*True*] ⟹ *to-nat* (*butlast zs*) = *to-nat zs* − *2* ^ *length xs*
⟨*proof*⟩

lemma *inc-nat-true-prefix*[*simp*]: *inc-nat* (*replicate n True* @ [*False*] @ *ys*) = *replicate n False* @ [*True*] @ *ys*
⟨*proof*⟩

lemma *length-inc-nat-aux*: *zs* = *replicate n True* @ [*False*] @ *ys* ⟹ *length* (*inc-nat zs*) = *length zs*
⟨*proof*⟩

lemma *length-inc-nat-aux-2*: *length* (*inc-nat* (*xs* @ [*False*] @ *ys*)) = *length* (*xs* @ [*False*] @ *ys*)
⟨*proof*⟩

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*⟩

corollary *subtract-nat-correct*: *to-nat* (*subtract-nat xs ys*) = (*to-nat xs*) − (*to-nat ys*)
⟨*proof*⟩

corollary *length-subtract-nat-le*: *length* (*subtract-nat xs ys*) ≤ *max* (*length xs*) (*length ys*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *length-grid-mul-nat*: *length* (*grid-mul-nat xs ys*) ≤ *length xs* + *length ys*
⟨*proof*⟩

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** $\leq_n$ *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** $\leq_n$ *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* $\Rightarrow$ *nat*
  assumes *small-bounds*: $f\ 0 \leq a\ f\ (Suc\ 0) \leq a$
  assumes *recursive-bound*: $\bigwedge n.\ n > 1 \Longrightarrow f\ n \leq c * n + d + f\ (n\ div\ 2)$
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*: $f\ n \leq g\ n$
  ⟨*proof*⟩ **lemma** *g-mono-aux*: $a \leq g\ n$
  ⟨*proof*⟩ **lemma** *g-mono*: $m \leq n \Longrightarrow g\ m \leq g\ n$
⟨*proof*⟩ **lemma** *g-powers-of-2*: $g\ (2\ \hat{}\ n) = d * n + c * (2\ \hat{}\ (n + 1) - 2) + a$
⟨*proof*⟩ **lemma** *pow-ineq*:
  assumes $m \geq (1 :: nat)$
  assumes $p \geq 2$
  shows $p\ \hat{}\ m > m$
  ⟨*proof*⟩ **lemma** *next-power-of-2*:
  assumes $m \geq (1 :: nat)$
  shows $\exists n\ k.\ m = 2\ \hat{}\ n + k \wedge k < 2\ \hat{}\ n$
⟨*proof*⟩

lemma *div-2-recursion-linear*: $f\ n \leq (2 * d + 4 * c) * n + a$
⟨*proof*⟩

end

General Lemmas for Landau notation.

lemma *landau-o-plus-aux'*:
  fixes $f\ g$
  assumes $f \in o[F](g)$
  shows $O[F](g) = O[F](\lambda x.\ f\ x + g\ x)$
  ⟨*proof*⟩

lemma *powr-bigo-linear-index-transformation*:
  fixes *fl* :: *nat* $\Rightarrow$ *nat*
  fixes f :: *nat* $\Rightarrow$ *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*⟩

lemma *real-mono*: $(a \leq b) = (real\ a \leq real\ b)$
⟨*proof*⟩

lemma *real-linear*: $real\ (a + b) = real\ a + real\ b$
⟨*proof*⟩

lemma *real-multiplicative*: $real\ (a * b) = real\ a * real\ b$
⟨*proof*⟩

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 $(\lambda x.\ h\ x * f\ x) \in L\ F\ (g)$
⟨*proof*⟩

lemma *norm-nonneg*: $x \geq 0 \Longrightarrow norm\ x = x$ ⟨*proof*⟩

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 $f \in O[F](g)$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-truncate-reversed-tm-le*: *time* (*truncate-reversed-tm xs*) ≤ *length xs* +

1
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *time-truncate-tm-le*: *time* (*truncate-tm xs*) ≤ *3* ∗ *length xs* + *6*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *com-f-of-min-max*: *f a b* = *f b a* ⟹ *f* (*min a b*) (*max a b*) = *f a b*
  ⟨*proof*⟩
lemma *add-min-max*: *min* (*a*::′*a*:: *ordered-ab-semigroup-add*) *b* + *max a b* = *a* + *b*
  ⟨*proof*⟩

lemma *time-fill-tm*: *time* (*fill-tm n xs*) = *2* ∗ *length xs* + *n* + *5*
  ⟨*proof*⟩

lemma *time-fill-tm-le*: *time* (*fill-tm n xs*) ≤ *3* ∗ *max n* (*length xs*) + *5*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *time-shift-right-tm*[*simp*]: *time* (*shift-right-tm n xs*) = *2* ∗ *n* + *3*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-split-at-tm*: *time* (*split-at-tm k xs*) = *2* ∗ *min k* (*length xs*) + *3*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-split-tm-le*: *time* (*split-tm xs*) ≤ *10* ∗ *length xs* + *16*
⟨*proof*⟩

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* ⟹ *val* (*subdivide-tm n xs*) = *subdivide n xs*
⟨*proof*⟩

lemma *time-subdivide-tm-le-aux*:
  assumes $n > 0$
  shows $time\ (subdivide\text{-}tm\ n\ xs) \le k * (2 * n + 3) + time\ (subdivide\text{-}tm\ n\ (drop\ (k * n)\ xs))$
⟨*proof*⟩

lemma *time-subdivide-tm-le*:
  fixes $xs :: 'a\ list$
  assumes $n > 0$
  shows $time\ (subdivide\text{-}tm\ n\ xs) \le 5 * length\ xs + 2 * n + 4$
⟨*proof*⟩

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*]: $val\ (bitsize\text{-}tm\ n) = bitsize\ n$
  ⟨*proof*⟩

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

lemma *time-bitsize-tm-aux*:
  $time\ (bitsize\text{-}tm\ n) \le time\text{-}bitsize\text{-}tm\text{-}bound\ n$
  ⟨*proof*⟩

lemma *time-bitsize-tm-aux2*: $time\text{-}bitsize\text{-}tm\text{-}bound\ n \le (2 * 8 + 4 * 14) * n + 23$
  ⟨*proof*⟩

lemma *time-bitsize-tm-le*: $time\ (bitsize\text{-}tm\ n) \le 72 * n + 23$
  ⟨*proof*⟩

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 ← $n\text{-}mod\text{-}2 =_t 0$;
    c2 ← *is-power-of-2-tm n-div-2*;

$c1 \wedge_t c2$
}

lemma *val-is-power-of-2-tm*[*simp*, *val-simp*]: *val* (*is-power-of-2-tm n*) = *is-power-of-2 n*
⟨*proof*⟩

lemma *time-is-power-of-2-tm-le*: *time* (*is-power-of-2-tm n*) ≤ *114* ∗ *n* + *1*
⟨*proof*⟩

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*;
    $2 \,\widehat{}_t\, r$
  }
}

lemma *val-next-power-of-2-tm*[*simp*, *val-simp*]: *val* (*next-power-of-2-tm n*) = *next-power-of-2 n*
⟨*proof*⟩

lemma *time-next-power-of-2-tm-le*: *time* (*next-power-of-2-tm n*) ≤ *208* ∗ *n* + *37*
⟨*proof*⟩

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*
⟨*proof*⟩
lemma *time-bit-add-carry-tm*[*simp*]: *time* (*bit-add-carry-tm x y z*) = *1*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-inc-nat-tm-le*: *time* (*inc-nat-tm xs*) ≤ *length xs* + *1*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-add-carry-tm-le*: *time* (*add-carry-tm c xs ys*) ≤ *2* ∗ *max* (*length xs*) (*length ys*) + *2*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-add-nat-tm-le*: *time* (*add-nat-tm xs ys*) ≤ *2* ∗ *max* (*length xs*) (*length ys*) + *3*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-compare-nat-same-length-tm-le*:
length xs = *length ys* ⟹ *time* (*compare-nat-same-length-tm xs ys*) ≤ *3* ∗ *length xs* + *6*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-make-same-length-tm-le*: *time* (*make-same-length-tm xs ys*) ≤ *10* ∗ *max* (*length xs*) (*length ys*) + *16*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *time-compare-nat-tm-le*: *time* (*compare-nat-tm xs ys*) ≤ *13* ∗ *max* (*length xs*) (*length ys*) + *23*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-map-tm-Not-tm*: *time* (*map-tm Not-tm xs*) = *2* ∗ *length xs* + *1*
⟨*proof*⟩

lemma *time-subtract-nat-tm-le*: *time* (*subtract-nat-tm xs ys*) ≤ *30* ∗ *max* (*length xs*) (*length ys*) + *48*
⟨*proof*⟩

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* {

  $r \leftarrow$ *grid-mul-nat-tm xs ys*;
  return (*False* # *r*)
}
| *grid-mul-nat-tm* (*True* # *xs*) *ys* =1 *do* {
  $r \leftarrow$ *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*
⟨*proof*⟩

lemma *euler-sum-bound*: $\sum$ {..(*n*::*nat*)} $\leq$ *n* $*$ *n*
⟨*proof*⟩

lemma *time-grid-mul-nat-tm-le*:
time (*grid-mul-nat-tm xs ys*) $\leq$ *8* $*$ *length xs* $*$ *max* (*length xs*) (*length ys*) + *1*
⟨*proof*⟩

9.8 Syntax bundles

abbreviation *shift-right-tm-flip* **where** *shift-right-tm-flip xs n* $\equiv$ *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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *truncate-from-int*[*simp*]: *truncate-int* (*from-int x*) = *from-int x*
⟨*proof*⟩

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*⟩

lemma *to-int-eq-imp-truncate-int-eq*: *to-int* (*a*, *x*) = *to-int* (*b*, *y*) ⟹ *truncate-int* (*a*, *x*) = *truncate-int* (*b*, *y*)
⟨*proof*⟩

lemma *from-int-to-int*: *from-int* ∘ *to-int* = *truncate-int*
⟨*proof*⟩

interpretation *int-lsbf*: *abstract-representation from-int to-int truncate-int*
⟨*proof*⟩

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*
⟨*proof*⟩

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 $\bigwedge$*x y*. *to-nat* (*f x y*) = *to-nat x* ∗ *to-nat y*
shows $\bigwedge$*x y xs ys*. *to-int* (*nat-mul-to-int-mul f* (*x*, *xs*) (*y*, *ys*)) = *to-int* (*x*, *xs*) ∗ *to-int* (*y*, *ys*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *abs-diff-length*: *length* (*abs-diff xs ys*) ≤ *max* (*length xs*) (*length ys*)
⟨*proof*⟩

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*
⟨*proof*⟩

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* $\le_n$ *x0*) = (*y1* $\le_n$ *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*) ⟨*proof*⟩
lemma *split-y*: *split y* = (*y0*, *y1*) ⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *l-ge-1*: *l* ≥ *1*
⟨*proof*⟩

lemma *k-even*: *k mod 2* = *0*
⟨*proof*⟩

lemma *k-div-2*: *k-div-2* = *2* ^ (*l* − *1*)
⟨*proof*⟩

lemma *k-div-2-less-k*: *k-div-2* < *k*
⟨*proof*⟩

lemma *length-x-split*: *length x0* = *k-div-2 length x1* = *k-div-2*
⟨*proof*⟩

lemma *length-y-split*: *length y0* = *k-div-2 length y1* = *k-div-2*
⟨*proof*⟩

lemma *length-abs-diff-x0-x1*: *length* (*abs-diff x0 x1*) ≤ *k-div-2*
⟨*proof*⟩
lemma *length-fill-abs-diff-x0-x1*: *length* (*fill k-div-2* (*abs-diff x0 x1*)) = *k-div-2*
⟨*proof*⟩

lemma *length-abs-diff-y0-y1*: *length* (*abs-diff y0 y1*) ≤ *k-div-2*
⟨*proof*⟩
lemma *length-fill-abs-diff-y0-y1*: *length* (*fill k-div-2* (*abs-diff y0 y1*)) = *k-div-2*
⟨*proof*⟩

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 $k = 2 \mathbin{\char`\^} l$
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*
⟨*proof*⟩

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*)
⟨*proof*⟩
termination
⟨*proof*⟩

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

lemma *length-karatsuba-on-power-of-2-aux*:
assumes *k* = *2* ^ *l*
assumes *length x* = *k length y* = *k*
shows *length* (*karatsuba-on-power-of-2-length k x y*) ≤ *len-kar-bound l*
⟨*proof*⟩

lemma *len-kar-bound-le*: *len-kar-bound l* ≤ *6* ∗ *2* ^ *l* + *2* ∗ *karatsuba-lower-bound*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *length-karatsuba-mul-nat-le*: *length* (*karatsuba-mul-nat x y*) ≤ *12* ∗ *max* (*length x*) (*length y*) + (*6* + *2* ∗ *karatsuba-lower-bound*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *time-abs-diff-tm-le*: *time* (*abs-diff-tm xs ys*) ≤ *62* ∗ *max* (*length xs*) (*length ys*) + *100*
⟨*proof*⟩

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'*: $r = r'$
⟨*proof*⟩

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* $\ge 2 \,\hat{}\, k$
⟨*proof*⟩

context begin

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

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

private lemma *f-linear*: *f k* $\le 6 * k$
⟨*proof*⟩ **lemma** *f-bound*:
assumes $k = 2 \,\hat{}\, l$
assumes *length* $x = k$
assumes *length* $y = k$
shows *length* (*karatsuba-on-power-of-2-length k x y*) $\le$ *f k*
⟨*proof*⟩

lemma *length-karatsuba-on-power-of-2-length*:
assumes $k = 2 \,\hat{}\, l$
assumes *length* $x = k$
assumes *length* $y = k$
shows *length* (*karatsuba-on-power-of-2-length k x y*) $\le 6 * k$
⟨*proof*⟩

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* {
  $b \leftarrow k \le_t$ *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* $\le_{nt}$ *x0*;
    sgny ← *y1* $\le_{nt}$ *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*
  })
}
  ⟨*proof*⟩
termination
  ⟨*proof*⟩

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*
⟨*proof*⟩

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*) $\leq$ *h k*
⟨*proof*⟩

lemma *n-div-2*: *n div 2* = *nat* ⌊*real n* / *2*⌋
⟨*proof*⟩

function *h-real* :: *nat* ⇒ *real* **where**
x $\leq$ *karatsuba-lower-bound* ⟹ *h-real x* = *8* ∗ *x* ∗ *x* + *2* ∗ *x* + *3*
| *x* > *karatsuba-lower-bound* ⟹ *h-real x* = *407* + *224* ∗ *x* + *3* ∗ *h-real* (*nat* (⌊*real x* / *2*⌋))
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *h-h-real*: *real* (*h k*) = *h-real k*
⟨*proof*⟩

lemma *h-real-bigo*: *h-real* ∈ *O*(λ*n*. *real n powr log 2 3*)
⟨*proof*⟩

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*⟩

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*) $\leq$ *time-karatsuba-mul-nat-bound m*
⟨*proof*⟩

theorem *time-karatsuba-mul-nat-bound-bigo*: *time-karatsuba-mul-nat-bound* ∈ *O*(λ*m*. *m powr log 2 3*)
⟨*proof*⟩

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* ≠ []
  ⟨*proof*⟩

lemma *from-numeral-truncated*: *truncated* (*from-numeral x*)
  ⟨*proof*⟩

lemma *to-nat-from-numeral-neq-zero*: *to-nat* (*from-numeral x*) ≠ *0*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *nat-of-num-to-numeral*:
  assumes *to-nat xs* $\neq$ *0*
  shows *nat-of-num* (*to-numeral xs*) = *to-nat xs*
  ⟨*proof*⟩

lemma *l0*:
  assumes *truncated xs*
  shows *to-numeral-of-truncated xs* = *num-of-nat* (*to-nat xs*)
  ⟨*proof*⟩

lemma *l1*: *to-numeral xs* = *num-of-nat* (*to-nat xs*)
  ⟨*proof*⟩

lemma *l2*: *to-nat* (*from-numeral x*) = *nat-of-num x*
  ⟨*proof*⟩

lemma[*code*]:
  (*x*::*num*) $*$ *y* = *to-numeral* (*karatsuba-mul-nat* (*from-numeral x*) (*from-numeral y*))
  ⟨*proof*⟩

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.