

Karatsuba Multiplication for Integers

Jakob Schulz, Emin Karayel

April 18, 2024

Abstract

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

Contents

1	Preliminaries	3
2	Auxiliary Sum Lemmas	7
2.1	<i>semiring-1</i> Sums	9
2.1.1	Power Sums	10
2.2	<i>nat</i> Sums	11
3	Sums in Monoids	12
3.1	Kronecker delta	16
3.2	Power sums	16
3.2.1	Algebraic operations	17
3.3	<i>monoid-sum-list</i> in the context <i>residues</i>	18
4	The <i>estimation</i> tactic	18
5	Some Automation for <i>Root-Balanced-Tree.Time-Monad</i>	19
6	Running Time Formalization for some functions available in <i>Main</i>	20
6.1	Functions on <i>bool</i>	20
6.1.1	Not	20
6.1.2	disj / conj	20
6.1.3	equal	21
6.2	Functions involving pairs	21
6.2.1	<i>fst</i> / <i>snd</i>	21

6.3	Functions on <i>nat</i>	21
6.3.1	(+)	21
6.3.2	(*)	21
6.3.3	(\wedge)	22
6.3.4	(-)	22
6.3.5	(<) / (\leq)	23
6.3.6	(=)	23
6.3.7	<i>max</i>	24
6.3.8	(<i>div</i>) / (<i>mod</i>)	24
6.3.9	(<i>dvd</i>)	25
6.3.10	<i>even</i> / <i>odd</i>	25
6.4	List functions	26
6.4.1	<i>take</i>	26
6.4.2	<i>drop</i>	26
6.4.3	(@)	26
6.4.4	<i>fold</i>	27
6.4.5	<i>rev</i>	27
6.4.6	<i>replicate</i>	27
6.4.7	<i>length</i>	27
6.4.8	<i>List.null</i>	28
6.4.9	<i>butlast</i>	28
6.4.10	<i>map</i>	29
6.4.11	<i>foldl</i>	29
6.4.12	<i>concat</i>	29
6.4.13	(!)	30
6.4.14	<i>zip</i>	30
6.4.15	<i>map2</i>	30
6.4.16	<i>upt</i>	31
6.5	Syntactic sugar	31
7	Representations	33
7.1	Abstract Representations	33
7.2	Abstract Representations 2	34
8	Representing <i>nat</i> in LSBF	35
8.1	Type definition	36
8.2	Conversions	36
8.3	Truncating and filling	38
8.4	Right-shifts	41
8.5	Subdividing lists	41
8.5.1	Splitting a list in two blocks	41
8.5.2	Splitting a list in multiple blocks	41
8.6	The <i>bitsize</i> function	42
8.6.1	The <i>next-power-of-2</i> function	43

8.7	Addition	44
8.7.1	Increment operation	44
8.7.2	Addition with a carry bit	45
8.7.3	Addition	46
8.8	Comparison and subtraction	47
8.8.1	Comparison	47
8.8.2	Subtraction	48
8.9	(Grid) Multiplication	49
8.10	Syntax bundles	49
9	Running time of <i>Nat-LSBF</i>	51
9.1	Truncating and filling	51
9.2	Right-shifts	52
9.3	Subdividing lists	53
9.3.1	Splitting a list in two blocks	53
9.3.2	Splitting a list in multiple blocks	53
9.4	The <i>bitsize</i> function	54
9.4.1	The <i>is-power-of-2</i> function	54
9.5	Addition	55
9.6	Comparison and subtraction	57
9.7	(Grid) Multiplication	58
9.8	Syntax bundles	59
10	Representing <i>int</i> in LSBF	60
10.1	Type definition	60
10.2	Conversions	60
10.3	Addition	60
10.4	Grid Multiplication	61
11	Karatsuba Multiplication	61
12	Running Time of Karatsuba Multiplication	65
13	Code Generation	69

1 Preliminaries

Some general preliminaries.

theory *Karatsuba-Preliminaries*

imports *Main Expander-Graphs.Extra-Congruence-Method HOL-Number-Theory.Residues*
begin

lemma *prop-iffI*:

assumes $Q \implies P$ R

assumes $\neg Q \implies P$ S

shows P (if Q then R else S)
<proof>

lemma *let-prop-cong*:
assumes $T = T'$
assumes $P (f T) (f' T')$
shows $P (\text{let } x = T \text{ in } f x) (\text{let } x = T' \text{ in } f' x)$
<proof>

lemma *set-subseteqD*:
assumes $\text{set } xs \subseteq A$
shows $\bigwedge i. i < \text{length } xs \implies xs ! i \in A$
<proof>

lemma *set-subseteqI*:
assumes $\bigwedge i. i < \text{length } xs \implies xs ! i \in A$
shows $\text{set } xs \subseteq A$
<proof>

lemma *Nat-max-le-sum*: $\max (a :: \text{nat}) b \leq a + b$
<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*: $\text{map } (\lambda j. j + c) [a..<b] = [a + c..<b + c]$
<proof>

lemma *filter-even-upt-even*: $\text{filter even } [0..<2*n] = \text{map } ((* 2) [0..<n]$
<proof>

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

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

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

lemma *length-filter-even*: $\text{length } (\text{filter even } [0..<n]) = (\text{if even } n \text{ then } n \text{ div } 2 \text{ else } n \text{ div } 2 + 1)$
<proof>

lemma *length-filter-odd*: $\text{length } (\text{filter odd } [0..<n]) = n \text{ div } 2$
<proof>

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

<proof>

lemma *filter-odd-nth*:

assumes $i < \text{length } (\text{filter odd } [0..<n])$

shows $\text{filter odd } [0..<n] ! i = 2 * i + 1$

<proof>

fun *sublist* **where**

$\text{sublist } 0 \ n \ xs = \text{take } n \ xs$

$| \text{sublist } (\text{Suc } m) \ (\text{Suc } n) \ (a \ \# \ xs) = \text{sublist } m \ n \ xs$

$| \text{sublist } (\text{Suc } m) \ 0 \ xs = []$

$| \text{sublist } (\text{Suc } m) \ (\text{Suc } n) \ [] = []$

lemma *length-sublist[simp]*: $\text{length } (\text{sublist } m \ n \ xs) = \text{card } (\{m..<n\} \cap \{0..<\text{length } xs\})$

<proof>

lemma *length-sublist'*:

assumes $m \leq n$

assumes $n \leq \text{length } xs$

shows $\text{length } (\text{sublist } m \ n \ xs) = n - m$

<proof>

lemma *nth-sublist*:

assumes $m \leq n$

assumes $n \leq \text{length } xs$

assumes $i < n - m$

shows $\text{sublist } m \ n \ xs ! i = xs ! (m + i)$

<proof>

lemma *filter-map-map2*:

assumes $\text{length } b = m$

assumes $\text{length } c = m$

shows $[f \ (b!i) \ (c!i). \ i \leftarrow [0..<m]] = \text{map2 } f \ b \ c$

<proof>

fun *map3* **where**

$\text{map3 } f \ (x \ \# \ xs) \ (y \ \# \ ys) \ (z \ \# \ zs) = f \ x \ y \ z \ \# \ \text{map3 } f \ xs \ ys \ zs$

$| \text{map3 } f \ - \ - \ - = []$

lemma *map3-as-map*: $\text{map3 } f \ xs \ ys \ zs = \text{map } (\lambda((x, y), z). f \ x \ y \ z) \ (\text{zip } (\text{zip } xs \ ys) \ zs)$

<proof>

lemma *filter-map-map3*:

assumes $\text{length } b = m$

assumes $\text{length } c = m$

shows $[f \ (b!i) \ (c!i) \ i. \ i \leftarrow [0..<m]] = \text{map3 } f \ b \ c \ [0..<m]$

<proof>

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* ($\lambda((x,y),z),w). f\ x\ y\ z\ w$) (*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* ($\lambda x\ y. f\ x\ (g\ y)$) *xs* *ys*
<proof>

lemma *map2-of-map-l*: *map2* *f* (*map* *g* *xs*) *ys* = *map2* ($\lambda x\ y. f\ (g\ x)\ y$) *xs* *ys*
<proof>

lemma *map2-of-map2-r*: *map2* *f* *xs* (*map2* *g* *ys* *zs*) = *map3* ($\lambda x\ y\ z. f\ x\ (g\ y\ z)$) *xs* *ys* *zs*
<proof>

lemma *map-of-map3*: *map* *f* (*map3* *g* *xs* *ys* *zs*) = *map3* ($\lambda x\ y\ z. f\ (g\ x\ y\ z)$) *xs* *ys* *zs*
<proof>

lemma *cyclic-index-lemma*:

fixes *n* :: *nat*

assumes $\sigma < n \ \varrho < n \ i < n$
shows $(\sigma + \varrho) \bmod n = i \iff \varrho = (n + i - \sigma) \bmod n$
 $\langle \text{proof} \rangle$

lemma (*in residues*) *residues-minus-eq*: $x \ominus_R y = (x - y) \bmod m$
 $\langle \text{proof} \rangle$

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

context *ring*
begin

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

lemma *nat-embedding-closed[simp]*: *nat-embedding* $x \in \text{carrier } R$
 $\langle \text{proof} \rangle$

lemma *int-embedding-closed[simp]*: *int-embedding* $x \in \text{carrier } R$
 $\langle \text{proof} \rangle$

lemma *nat-embedding-a-hom*: *nat-embedding* $(x + y) = \text{nat-embedding } x \oplus \text{nat-embedding } y$
 $\langle \text{proof} \rangle$

lemma *nat-embedding-m-hom*: *nat-embedding* $(x * y) = \text{nat-embedding } x \otimes \text{nat-embedding } y$
 $\langle \text{proof} \rangle$

lemma *nat-embedding-exp-hom*: *nat-embedding* $(x \hat{^} y) = \text{nat-embedding } x [\hat{^}] y$
 $\langle \text{proof} \rangle$

lemma *int-embedding-neg-hom*: *int-embedding* $(- x) = \ominus \text{int-embedding } x$
 $\langle \text{proof} \rangle$

end

lemma *int-exp-hom*: *int* $x \hat{^} i = \text{int } (x \hat{^} i)$
 $\langle \text{proof} \rangle$

end

2 Auxiliary Sum Lemmas

theory *Karatsuba-Sum-Lemmas*

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

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

<proof>

lemma *sum-list-split-0*: $(\sum i \leftarrow [0..< \text{Suc } n]. f i) = f 0 + (\sum i \leftarrow [1..< \text{Suc } n]. f i)$

<proof>

lemma *sum-list-index-trafo*: $(\sum i \leftarrow xs. f (g i)) = (\sum i \leftarrow \text{map } g xs. f i)$

<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*: $n = j - k \implies (\sum i \leftarrow [k+1..< j+1]. f i) = (\sum i \leftarrow [k..< j]. f (i + 1))$

<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 $\text{int } (\sum i \leftarrow xs. g i) = (\sum i \leftarrow xs. \text{int } (g i))$

<proof>

lemma *sum-list-split-Suc*:

assumes $n = \text{Suc } n'$

shows $(\sum i \leftarrow [0..< n]. f i) = (\sum i \leftarrow [0..< n']. f i) + f n'$

<proof>

lemma *sum-list-estimation-leq*:

assumes $\bigwedge i. i \in \text{set } xs \implies f i \leq B$

shows $(\sum i \leftarrow xs. f i) \leq \text{length } xs * B$

<proof>

lemma *sum-list-estimation-le*:

assumes $\bigwedge i. i \in \text{set } xs \implies f i < B$

assumes $xs \neq []$

shows $(\sum i \leftarrow xs. f i) < \text{length } xs * B$

<proof>

2.1 *semiring-1* Sums

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

lemma (in *semiring-1-cancel*) *of-bool-disj*: $of\text{-}bool\ (x \vee y) = of\text{-}bool\ x + of\text{-}bool\ y$
 $- of\text{-}bool\ x * of\text{-}bool\ y$
 ⟨*proof*⟩

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

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

lemma $(\sum i \leftarrow xs.\ of\text{-}bool\ (i = j) * f\ i) = count\text{-}list\ xs\ j * f\ j$
 ⟨*proof*⟩

lemma (in *semiring-1*) *of-bool-distinct*:
 $distinct\ xs \implies (\sum i \leftarrow xs.\ of\text{-}bool\ (i = j) * f\ i\ j) = of\text{-}bool\ (j \in set\ xs) * f\ j\ j$
 ⟨*proof*⟩

lemma (in *semiring-1*) *of-bool-distinct-in*:
 $distinct\ xs \implies j \in set\ xs \implies (\sum i \leftarrow xs.\ of\text{-}bool\ (i = j) * f\ i\ j) = f\ j\ j$
 ⟨*proof*⟩

lemma (in *linordered-semiring-1*) *of-bool-sum-leq-1*:
 assumes *distinct xs*
 assumes $\bigwedge i\ j.\ i \in set\ xs \implies j \in set\ xs \implies P\ i \implies P\ j \implies i = j$
 shows $(\sum l \leftarrow xs.\ of\text{-}bool\ (P\ l)) \leq 1$
 ⟨*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*:
 $(\bigwedge i.\ i \in set\ xs \implies f\ i = g\ i) \implies (\sum i \leftarrow xs.\ f\ i) = (\sum i \leftarrow xs.\ g\ i)$
 ⟨*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*: $(\sum i \leftarrow xs. \text{of-bool } (P i) * f i) =$
 $(\sum i \leftarrow \text{filter } P xs. f i)$
<proof>

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

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

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

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

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

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

lemma (in *comm-semiring-1*) *power-sum-product*:

assumes $m > 0$

assumes $n \geq m$

shows

$(\sum i \leftarrow [0..<n]. f i * x ^ i) * (\sum j \leftarrow [0..<m]. g j * x ^ j) =$
 $(\sum k \leftarrow [0..<m]. (\sum i \leftarrow [0..<\text{Suc } k]. f i * g (k - i)) * x ^ k) +$
 $(\sum k \leftarrow [m..<n]. (\sum i \leftarrow [\text{Suc } k - m..<\text{Suc } k]. f i * g (k - i)) * x ^ k) +$
 $(\sum k \leftarrow [n..<n + m - 1]. (\sum i \leftarrow [\text{Suc } k - m..<n]. f i * g (k - i)) * x ^ k)$
<proof>

lemma (in *comm-semiring-1*) *power-sum-product-same-length*:

assumes $n > 0$

shows $(\sum i \leftarrow [0..<n]. f i * x ^ i) * (\sum j \leftarrow [0..<n]. g j * x ^ j) =$

$$\begin{aligned}
& (\sum k \leftarrow [0..<n]. (\sum i \leftarrow [0..<Suc\ k]. f\ i * g\ (k - i)) * x^{\wedge} k) + \\
& (\sum k \leftarrow [n..<2 * n - 1]. (\sum i \leftarrow [Suc\ k - n..<n]. f\ i * g\ (k - i)) * x^{\wedge} k) \\
& \langle proof \rangle
\end{aligned}$$

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

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

2.2 nat Sums

lemma *geo-sum-nat*:
assumes $(q :: nat) > 1$
shows $(q - 1) * (\sum i \leftarrow [0..<n]. q^{\wedge} i) = q^{\wedge} n - 1$
 $\langle proof \rangle$

lemma *geo-sum-bound*:
assumes $(q :: nat) > 1$
assumes $\bigwedge i. i < n \implies f\ i < q$
shows $(\sum i \leftarrow [0..<n]. f\ i * q^{\wedge} i) < q^{\wedge} n$
 $\langle proof \rangle$

lemma *power-sum-nat-split-div-mod*:
assumes $x > 1$
assumes $c > 0$
assumes $\bigwedge i. i < n \implies (f\ i :: nat) < x^{\wedge} c$
assumes $j \leq n$
shows $(\sum i \leftarrow [0..<n]. f\ i * x^{\wedge} (i * c)) \mathit{div}\ x^{\wedge} (j * c)$
 $= (\sum i \leftarrow [0..<n - j]. f\ (j + i) * x^{\wedge} (i * c))$
 $(\sum i \leftarrow [0..<n]. f\ i * x^{\wedge} (i * c)) \mathit{mod}\ x^{\wedge} (j * c)$
 $= (\sum i \leftarrow [0..<j]. f\ i * x^{\wedge} (i * c))$
 $\langle proof \rangle$

lemma *power-sum-nat-extract-coefficient*:
assumes $x > 1$
assumes $c > 0$
assumes $\bigwedge i. i < n \implies (f\ i :: nat) < x^{\wedge} c$
assumes $j < n$
shows $((\sum i \leftarrow [0..<n]. f\ i * x^{\wedge} (i * c)) \mathit{div}\ x^{\wedge} (j * c)) \mathit{mod}\ x^{\wedge} c = f\ j$
 $\langle proof \rangle$

```

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

end

```

3 Sums in Monoids

theory *Monoid-Sums*

imports *HOL-Algebra.Ring Expander-Graphs.Extra-Congruence-Method Karat-suba-Preliminaries HOL-Library.Multiset HOL-Number-Theory.Residues Karat-suba-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 [] = 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) 0
  <proof>

```

end

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

syntax

```

-monoid-sum-list ::  $index \Rightarrow idt \Rightarrow 'c\ list \Rightarrow 'c \Rightarrow 'a$ 
  (( $\exists \oplus \dashv\leftarrow \cdot$ ) [1000, 0, 51, 10] 10)

```

translations

```

 $\oplus_{G^{i \leftarrow xs}}$  b  $\equiv$  CONST abelian-monoid.monoid-sum-list G ( $\lambda i. b$ ) xs

```

context *abelian-monoid*

begin

lemma *monoid-sum-list-finsum*:

assumes $\bigwedge i. i \in \text{set } xs \implies f i \in \text{carrier } G$
assumes *distinct xs*
shows $(\bigoplus i \leftarrow xs. f i) = (\bigoplus i \in \text{set } xs. f i)$
<proof>

lemma *monoid-sum-list-cong*:

assumes $\bigwedge i. i \in \text{set } xs \implies f i = g i$
shows $(\bigoplus i \leftarrow xs. f i) = (\bigoplus i \leftarrow xs. g i)$
<proof>

lemma *monoid-sum-list-closed[simp]*:

assumes $\bigwedge i. i \in \text{set } xs \implies f i \in \text{carrier } G$
shows $(\bigoplus i \leftarrow xs. f i) \in \text{carrier } G$
<proof>

lemma *monoid-sum-list-add-in*:

assumes $\bigwedge i. i \in \text{set } xs \implies f i \in \text{carrier } G$
assumes $\bigwedge i. i \in \text{set } xs \implies g i \in \text{carrier } G$
shows $(\bigoplus i \leftarrow xs. f i) \oplus (\bigoplus i \leftarrow xs. g i) =$
 $(\bigoplus i \leftarrow xs. f i \oplus g i)$
<proof>

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

<proof>

lemma *monoid-sum-list-swap*:

assumes[simp]: $\bigwedge i j. i \in \text{set } xs \implies j \in \text{set } ys \implies f i j \in \text{carrier } G$
shows $(\bigoplus i \leftarrow xs. (\bigoplus j \leftarrow ys. f i j)) =$
 $(\bigoplus j \leftarrow ys. (\bigoplus i \leftarrow xs. f i j))$
<proof>

lemma *monoid-sum-list-index-transformation*:

$(\bigoplus i \leftarrow (\text{map } g \text{ } xs). f i) = (\bigoplus i \leftarrow xs. f (g i))$
<proof>

lemma *monoid-sum-list-index-shift-0*:

$(\bigoplus i \leftarrow [c..<c+n]. f i) = (\bigoplus i \leftarrow [0..<n]. f (c + i))$
<proof>

lemma *monoid-sum-list-index-shift*:

$(\bigoplus l \leftarrow [a..<b]. f (l+c)) = (\bigoplus l \leftarrow [(a+c)..<(b+c)]. f l)$
<proof>

lemma *monoid-sum-list-app*:

assumes $\bigwedge i. i \in \text{set } xs \implies f i \in \text{carrier } G$
assumes $\bigwedge i. i \in \text{set } ys \implies f i \in \text{carrier } G$
shows $(\bigoplus i \leftarrow xs @ ys. f i) = (\bigoplus i \leftarrow xs. f i) \oplus (\bigoplus i \leftarrow ys. f i)$
<proof>

lemma monoid-sum-list-app':

assumes $\bigwedge i. i \in \text{set } xs \implies f i \in \text{carrier } G$
assumes $\bigwedge i. i \in \text{set } ys \implies f i \in \text{carrier } G$
assumes $xs @ ys = zs$
shows $(\bigoplus i \leftarrow zs. f i) = (\bigoplus i \leftarrow xs. f i) \oplus (\bigoplus i \leftarrow ys. f i)$
 $\langle \text{proof} \rangle$

lemma monoid-sum-list-extract:

assumes $\bigwedge i. i \in \text{set } xs \implies f i \in \text{carrier } G$
assumes $\bigwedge i. i \in \text{set } ys \implies f i \in \text{carrier } G$
assumes $f x \in \text{carrier } G$
shows $(\bigoplus i \leftarrow xs @ x \# ys. f i) = f x \oplus (\bigoplus i \leftarrow (xs @ ys). f i)$
 $\langle \text{proof} \rangle$

lemma monoid-sum-list-Suc:

assumes $\bigwedge i. i < \text{Suc } r \implies f i \in \text{carrier } G$
shows $(\bigoplus i \leftarrow [0..<\text{Suc } r]. f i) = (\bigoplus i \leftarrow [0..<r]. f i) \oplus f r$
 $\langle \text{proof} \rangle$

lemma bij-betw-diff-singleton: $a \in A \implies b \in B \implies \text{bij-betw } f A B \implies f a = b$
 $\implies \text{bij-betw } f (A - \{a\}) (B - \{b\})$

$\langle \text{proof} \rangle$

lemma $a \in A \implies \text{bij-betw } f A B \implies \text{bij-betw } f (A - \{a\}) (B - \{f a\})$

$\langle \text{proof} \rangle$

lemma monoid-sum-list-multiset-eq:

assumes $\text{mset } xs = \text{mset } ys$
assumes $\bigwedge i. i \in \text{set } xs \implies f i \in \text{carrier } G$
shows $(\bigoplus i \leftarrow xs. f i) = (\bigoplus i \leftarrow ys. f i)$
 $\langle \text{proof} \rangle$

lemma monoid-sum-list-index-permutation:

assumes $\text{distinct } xs$
assumes $\text{distinct } ys \vee \text{length } xs = \text{length } ys$
assumes $\text{bij-betw } f (\text{set } xs) (\text{set } ys)$
assumes $\bigwedge i. i \in \text{set } ys \implies g i \in \text{carrier } G$
shows $(\bigoplus i \leftarrow ys. g i) = (\bigoplus i \leftarrow xs. g (f i))$
 $\langle \text{proof} \rangle$

lemma monoid-sum-list-split:

assumes $[\text{simp}]$: $\bigwedge i. i < b + c \implies f i \in \text{carrier } G$
shows $(\bigoplus l \leftarrow [0..<b]. f l) \oplus (\bigoplus l \leftarrow [b..<b+c]. f l) = (\bigoplus l \leftarrow [0..<b+c]. f l)$
 $\langle \text{proof} \rangle$

lemma monoid-sum-list-splice:

assumes $[\text{simp}]$: $\bigwedge i. i < 2 * n \implies f i \in \text{carrier } G$
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 $\bigwedge i. i < n \implies f i \in \text{carrier } G$

shows $(\bigoplus i \leftarrow [0..<n]. f i) = (\bigoplus i \leftarrow [0..<n \text{ div } 2]. f (2*i)) \oplus (\bigoplus i \leftarrow [0..<n \text{ div } 2]. f (2*i+1))$

<proof>

end

context *abelian-group*

begin

lemma *monoid-sum-list-minus-in:*

assumes $\bigwedge i. i \in \text{set } xs \implies f i \in \text{carrier } G$

shows $\ominus (\bigoplus i \leftarrow xs. f i) = (\bigoplus i \leftarrow xs. \ominus f i)$

<proof>

lemma *monoid-sum-list-diff-in:*

assumes*[simp]*: $\bigwedge i. i \in \text{set } xs \implies f i \in \text{carrier } G$

assumes*[simp]*: $\bigwedge i. i \in \text{set } xs \implies g i \in \text{carrier } G$

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]*: $c \in \text{carrier } R$

shows $(\bigoplus i \leftarrow xs. c) = (\text{nat-embedding } (\text{length } xs)) \otimes c$

<proof>

lemma *monoid-sum-list-in-right:*

assumes $y \in \text{carrier } R$

assumes $\bigwedge i. i \in \text{set } xs \implies f i \in \text{carrier } R$

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 $y \in \text{carrier } R$

assumes $\bigwedge i. i \in \text{set } xs \implies f i \in \text{carrier } R$

shows $(\bigoplus i \leftarrow xs. y \otimes f i) = y \otimes (\bigoplus i \leftarrow xs. f i)$

<proof>

lemma *monoid-sum-list-prod:*

assumes $\bigwedge i. i \in \text{set } xs \implies f i \in \text{carrier } R$
assumes $\bigwedge i. i \in \text{set } ys \implies g i \in \text{carrier } R$
shows $(\bigoplus i \leftarrow xs. f i) \otimes (\bigoplus j \leftarrow ys. g j) = (\bigoplus i \leftarrow xs. (\bigoplus j \leftarrow ys. f i \otimes g j))$
 <proof>

3.1 Kronecker delta

definition *delta where*

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

lemma *delta-closed[simp]*: $\text{delta } i j \in \text{carrier } R$
 <proof>

lemma *delta-sym*: $\text{delta } i j = \text{delta } j i$
 <proof>

lemma *delta-refl[simp]*: $\text{delta } i i = \mathbf{1}$
 <proof>

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

lemma *monoid-sum-list-only-delta[simp]*:
 $j < n \implies (\bigoplus i \leftarrow [0..<n]. \text{delta } i j) = \mathbf{1}$
 <proof>

3.2 Power sums

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

rewrite $?x \in \text{carrier } R \implies (?x [\uparrow] ?n) [\uparrow] ?m = ?x [\uparrow] (?n * ?m)$ and $?a * ?b = ?b * ?a$ inside power sum

lemma *monoid-pow-sum-nat-pow-pow*:
assumes $x \in \text{carrier } R$
shows $(\bigoplus i \leftarrow xs. f i \otimes x [\uparrow] ((g i :: \text{nat}) * h i)) = (\bigoplus i \leftarrow xs. f i \otimes (x [\uparrow] h i) [\uparrow] g i)$
 <proof>

end

context *cring*
begin

Split a power sum at some term

lemma *monoid-pow-sum-list-split*:

assumes $l + k = n$

assumes $\bigwedge i. i < n \implies f i \in \text{carrier } R$

assumes $x \in \text{carrier } R$

shows $(\bigoplus i \leftarrow [0..<n]. f i \otimes x [\ulcorner] i) =$

$(\bigoplus i \leftarrow [0..<l]. f i \otimes x [\ulcorner] i) \oplus$

$x [\ulcorner] l \otimes (\bigoplus i \leftarrow [0..<k]. f (l + i) \otimes x [\ulcorner] i)$

<proof>

split power sum at term, more general

lemma *monoid-pow-sum-split*:

assumes $l + k = n$

assumes $\bigwedge i. i < n \implies f i \in \text{carrier } R$

assumes $x \in \text{carrier } R$

shows $(\bigoplus i \leftarrow [0..<n]. f i \otimes x [\ulcorner] (i * c)) =$

$(\bigoplus i \leftarrow [0..<l]. f i \otimes x [\ulcorner] (i * c)) \oplus$

$x [\ulcorner] (l * c) \otimes (\bigoplus i \leftarrow [0..<k]. f (l + i) \otimes x [\ulcorner] (i * c))$

<proof>

3.2.1 Algebraic operations

addition

lemma *monoid-pow-sum-add*:

assumes $\bigwedge i. i \in \text{set } xs \implies f i \in \text{carrier } R$

assumes $\bigwedge i. i \in \text{set } xs \implies g i \in \text{carrier } R$

assumes $x \in \text{carrier } R$

shows $(\bigoplus i \leftarrow xs. f i \otimes x [\ulcorner] (i :: \text{nat})) \oplus (\bigoplus i \leftarrow xs. g i \otimes x [\ulcorner] i) = (\bigoplus i \leftarrow xs. (f i \oplus g i) \otimes x [\ulcorner] i)$

<proof>

lemma *monoid-pow-sum-add'*:

assumes $\bigwedge i. i \in \text{set } xs \implies f i \in \text{carrier } R$

assumes $\bigwedge i. i \in \text{set } xs \implies g i \in \text{carrier } R$

assumes $x \in \text{carrier } R$

shows $(\bigoplus i \leftarrow xs. f i \otimes x [\ulcorner] ((i :: \text{nat}) * c)) \oplus (\bigoplus i \leftarrow xs. g i \otimes x [\ulcorner] (i * c)) = (\bigoplus i \leftarrow xs. (f i \oplus g i) \otimes x [\ulcorner] (i * c))$

<proof>

unary minus

lemma *monoid-pow-sum-minus*:

assumes $\bigwedge i. i \in \text{set } xs \implies f i \in \text{carrier } R$

assumes $x \in \text{carrier } R$

shows $\ominus (\bigoplus i \leftarrow xs. f i \otimes x [\ulcorner] (i :: \text{nat})) = (\bigoplus i \leftarrow xs. (\ominus f i) \otimes x [\ulcorner] i)$

<proof>

minus

lemma *monoid-pow-sum-diff*:

assumes $\bigwedge i. i \in \text{set } xs \implies f i \in \text{carrier } R$

assumes $\bigwedge i. i \in \text{set } xs \implies g \ i \in \text{carrier } R$
assumes $x \in \text{carrier } R$
shows $(\bigoplus i \leftarrow xs. f \ i \otimes x \ [\uparrow] (i::nat)) \ominus (\bigoplus i \leftarrow xs. g \ i \otimes x \ [\uparrow] (i::nat)) =$
 $(\bigoplus i \leftarrow xs. (f \ i \ominus g \ i) \otimes x \ [\uparrow] i)$
 $\langle \text{proof} \rangle$

lemma *monoid-pow-sum-diff'*:

assumes $\bigwedge i. i \in \text{set } xs \implies f \ i \in \text{carrier } R$
assumes $\bigwedge i. i \in \text{set } xs \implies g \ i \in \text{carrier } R$
assumes $x \in \text{carrier } R$
shows $(\bigoplus i \leftarrow xs. f \ i \otimes x \ [\uparrow] ((i::nat) * c)) \ominus (\bigoplus i \leftarrow xs. g \ i \otimes x \ [\uparrow] (i * c)) =$
 $(\bigoplus i \leftarrow xs. (f \ i \ominus g \ i) \otimes x \ [\uparrow] (i * c))$
 $\langle \text{proof} \rangle$

end

3.3 monoid-sum-list in the context residues

context *residues*

begin

lemma *monoid-sum-list-eq-sum-list*:

$(\bigoplus_R i \leftarrow xs. f \ i) = (\sum i \leftarrow xs. f \ i) \text{ mod } m$
 $\langle \text{proof} \rangle$

lemma *monoid-sum-list-mod-in*:

$(\bigoplus_R i \leftarrow xs. f \ i) = (\bigoplus_R i \leftarrow xs. (f \ i) \text{ mod } m)$
 $\langle \text{proof} \rangle$

lemma *monoid-sum-list-eq-sum-list'*:

$(\bigoplus_R i \leftarrow xs. f \ i \text{ mod } m) = (\sum i \leftarrow xs. f \ i) \text{ mod } m$
 $\langle \text{proof} \rangle$

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 $C \implies P \ A \ A'$

assumes $\neg C \implies P \ B \ B'$

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

⟨proof⟩

lemma *if-leqI*:

assumes $C \implies A \leq t$
assumes $\neg C \implies B \leq t$
shows (if C then A else B) $\leq t$
⟨proof⟩

lemma *if-le-max*:

(if C then ($t1 :: 'a :: \text{linorder}$) else $t2$) $\leq \max t1 t2$
⟨proof⟩

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

method *itrans* **for** $step :: 'a :: \text{order} =$

(*match conclusion in* $s \leq t$ **for** $s t :: 'a \Rightarrow \langle \text{rule } \text{order.trans}[of s step t] \rangle$)

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** $f h \Rightarrow \langle$
match conclusion in $g f \leq t$ **for** g and $t :: \text{nat} \Rightarrow$
 $\langle \text{rule } \text{order.trans}[of g f g h t], \text{intro } \text{mono-intros refl estimate} \rangle \rangle$

| $x \leq y$ **for** $x y \Rightarrow \langle$
match conclusion in $g x \leq t$ **for** g and $t :: \text{nat} \Rightarrow$
 $\langle \text{rule } \text{order.trans}[of g x g y t], \text{intro } \text{mono-intros refl estimate} \rangle \rangle$

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*: $\text{time } (s \ggg f) = \text{time } s + \text{time } (f \text{ (val } s))$
⟨proof⟩

lemma *time-tick*: $\text{time } (\text{tick } s) = 1$
⟨proof⟩

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 $f1 \ggg g1 = f2 \ggg g2$
 $\langle proof \rangle$

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* \Rightarrow *bool tm* **where**
Not-tm True = 1 return False
 $|$ *Not-tm False = 1 return True*

lemma *val-Not-tm*[*simp, val-simp*]: $val (Not\text{-}tm\ x) = Not\ x$
 $\langle proof \rangle$

lemma *time-Not-tm*[*simp*]: $time (Not\text{-}tm\ x) = 1$
 $\langle proof \rangle$

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*]: $val (disj\text{-}tm\ x\ y) = (x \vee y)$
 $\langle proof \rangle$

lemma *time-disj-tm*[*simp*]: $time (disj\text{-}tm\ x\ y) = 1$
 $\langle proof \rangle$

lemma *val-conj-tm*[*simp, val-simp*]: $val (conj\text{-}tm\ x\ y) = (x \wedge y)$
 $\langle proof \rangle$

lemma *time-conj-tm*[*simp*]: $time (conj\text{-}tm\ x\ y) = 1$

<proof>

6.1.3 equal

fun *equal-bool-tm* :: *bool* \Rightarrow *bool* \Rightarrow *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) \leq 2*
<proof>

6.2 Functions involving pairs

6.2.1 fst / snd

fun *fst-tm* :: '*a* \times '*b* \Rightarrow '*a tm* **where**
fst-tm (*x*, *y*) =1 *return x*
fun *snd-tm* :: '*a* \times '*b* \Rightarrow '*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* \Rightarrow *nat* \Rightarrow *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* \Rightarrow *nat* \Rightarrow *nat tm* **where**
times-nat-tm 0 *n* =1 *return 0*
| *times-nat-tm* (*Suc m*) *n* =1 *do {*

```

    r ← times-nat-tm m n;
    plus-nat-tm n r
  }

```

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

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

6.3.3 (\wedge)

```

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

```

lemma *val-power-nat-tm[simp, val-simp]*: *val (power-nat-tm a n) = a ^ n*
 ⟨*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 $m \geq 2$
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]*: $\text{time } (\text{minus-nat-tm } m \ n) = \min m \ n + 1$
 ⟨proof⟩

6.3.5 ($<$) / (\leq)

fun *less-eq-nat-tm* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$ *tm* **and** *less-nat-tm* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$
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]*: $\text{val } (\text{less-eq-nat-tm } n \ m) = (n \leq m)$
and *val-less-nat-tm[simp, val-simp]*: $\text{val } (\text{less-nat-tm } m \ n) = (m < n)$
 ⟨proof⟩

lemma *time-less-eq-nat-tm-aux*: $\text{time } (\text{less-eq-nat-tm } (m + k) \ (n + k)) = 2 * k + \text{time } (\text{less-eq-nat-tm } m \ n)$
 ⟨proof⟩

lemma *time-less-nat-tm-aux*: $\text{time } (\text{less-nat-tm } (m + k) \ (n + k)) = 2 * k + \text{time } (\text{less-nat-tm } m \ n)$
 ⟨proof⟩

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

lemma *time-less-nat-tm*: $\text{time } (\text{less-nat-tm } m \ n) = 2 * \min m \ n + 1 + \text{of-bool } (m < n)$
 ⟨proof⟩

lemma *time-less-eq-nat-tm-le*: $\text{time } (\text{less-eq-nat-tm } n \ m) \leq 2 * \min n \ m + 2$
 ⟨proof⟩

lemma *time-less-nat-tm-le*: $\text{time } (\text{less-nat-tm } m \ n) \leq 2 * \min m \ n + 2$
 ⟨proof⟩

6.3.6 (=)

fun *equal-nat-tm* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$ *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]*: $\text{val } (\text{equal-nat-tm } x \ y) = (x = y)$
 ⟨proof⟩

lemma *time-equal-nat-tm*: $\text{time } (\text{equal-nat-tm } x \ y) = \min x \ y + 1$
 ⟨proof⟩

6.3.7 *max*

fun *max-nat-tm* :: *nat* \Rightarrow *nat* \Rightarrow *nat tm* **where**

```
max-nat-tm x y =1 do {  
  b  $\leftarrow$  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*) \leq 2 * *min* *x y* + 3
(*proof*)

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

fun *divmod-nat-tm* :: *nat* \Rightarrow *nat* \Rightarrow (*nat* \times *nat*) *tm* **where**

```
divmod-nat-tm m n =1 do {  
  n0  $\leftarrow$  equal-nat-tm n 0;  
  m-lt-n  $\leftarrow$  less-nat-tm m n;  
  b  $\leftarrow$  disj-tm n0 m-lt-n;  
  if b then return (0, m) else do {  
    m-minus-n  $\leftarrow$  minus-nat-tm m n;  
    (q, r)  $\leftarrow$  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*) \leq 8 * *m* + 2 * *n* + 5
(*proof*)

definition *divide-nat-tm* :: *nat* \Rightarrow *nat* \Rightarrow *nat tm* **where**

divide-nat-tm *m n* =1 *divmod-nat-tm* *m n* $\gg=$ *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]*: $\text{time} (\text{append-tm } xs \ ys) = \text{length } xs + 1$
<proof>

6.4.4 fold

fun *fold-tm* **where**
fold-tm f [] $s = 1$ *return* s
| *fold-tm* f ($x \# xs$) $s = 1$ *do* {
 $r \leftarrow f \ x \ s$;
 fold-tm f xs r
}

lemma *val-fold-tm[simp, val-simp]*: $\text{val} (\text{fold-tm } f \ xs \ s) = \text{fold} (\lambda x \ y. \text{val} (f \ x \ y))$
 $xs \ s$
<proof>

lemma *time-fold-tm-Cons*: $\text{time} (\text{fold-tm} (\lambda x \ y. \text{return } (x \# y)) \ xs \ s) = \text{length } xs$
 $+ 1$
<proof>

6.4.5 rev

definition *rev-tm* **where** *rev-tm* $xs = 1$ *fold-tm* ($\lambda x \ y. \text{return } (x \# y)$) xs []

lemma *val-rev-tm[simp, val-simp]*: $\text{val} (\text{rev-tm } xs) = \text{rev } xs$
<proof>

lemma *time-rev-tm-le[simp]*: $\text{time} (\text{rev-tm } xs) = \text{length } xs + 2$
<proof>

6.4.6 replicate

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

lemma *val-replicate-tm[simp, val-simp]*: $\text{val} (\text{replicate-tm } n \ x) = \text{replicate } n \ x$
<proof>

lemma *time-replicate-tm*: $\text{time} (\text{replicate-tm } n \ x) = n + 1$
<proof>

6.4.7 length

fun *gen-length-tm* :: $\text{nat} \Rightarrow 'a \text{ list} \Rightarrow \text{nat } \text{tm}$ **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 \Rightarrow 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 \Rightarrow 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 \Rightarrow 'a list tm **where**
butlast-tm $[] = 1$ return $[]$
 | *butlast-tm* $(x \# xs) = 1$ do {
 $b \leftarrow$ *null-tm* xs ;
 if b then return $[]$ else do {
 $r \leftarrow$ *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) = $2 * (\text{length } xs - 1) + 1 + \text{of_bool}$ (*length* $xs \geq 1$)
 ⟨*proof*⟩

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

6.4.10 map

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

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

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

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

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

6.4.11 foldl

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

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

6.4.12 concat

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

}

lemma *val-concat-tm*[simp, val-simp]: $val (concat\text{-}tm\ xs) = concat\ xs$
(proof)

lemma *time-concat-tm*[simp]: $time (concat\text{-}tm\ xs) = 1 + 2 * length\ xs + length (concat\ xs)$
(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 $val (nth\text{-}tm\ xs\ i) = xs\ !\ i$
(proof)

lemma *time-nth-tm*[simp]:
assumes $i < length\ xs$
shows $time (nth\text{-}tm\ xs\ i) = i + 1$
(proof)

6.4.14 zip

fun *zip-tm* :: 'a list \Rightarrow 'b list \Rightarrow ('a \times 'b) list **tm** **where**
zip-tm xs [] =1 return []
| *zip-tm* [] ys =1 return []
| *zip-tm* (x # xs) (y # ys) =1 do { rs \leftarrow *zip-tm* xs ys; return ((x, y) # rs) }

lemma *val-zip-tm*[simp, val-simp]: $val (zip\text{-}tm\ xs\ ys) = zip\ xs\ ys$
(proof)

lemma *time-zip-tm*[simp]: $time (zip\text{-}tm\ xs\ ys) = min (length\ xs) (length\ ys) + 1$
(proof)

6.4.15 map2

definition *map2-tm* **where**

map2-tm f xs ys =1 do {
 xys \leftarrow *zip-tm* xs ys;
 map-tm ($\lambda(x,y). f\ x\ y$) xys
}

lemma *val-map2-tm*[simp, val-simp]: $val (map2\text{-}tm\ f\ xs\ ys) = map2 (\lambda x\ y. val (f\ x\ y))\ xs\ ys$
(proof)

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

6.4.16 *upt*

function *upt-tm* **where**

```
upt-tm i j = 1 do {
  b  $\leftarrow$  less-nat-tm i j;
  (if b then do {
    rs  $\leftarrow$  upt-tm (Suc i) j;
    return (i # rs)
  } else return [] )
}
```

$\langle \text{proof} \rangle$

termination $\langle \text{proof} \rangle$

declare *upt-tm.simps*[*simp del*]

lemma *val-upt-tm*[*simp, val-simp*]: $\text{val } (\text{upt-tm } i j) = [i..<j]$

$\langle \text{proof} \rangle$

lemma *time-upt-tm-le*: $\text{time } (\text{upt-tm } i j) \leq (j - i) * (2 * j + 3) + 2 * j + 2$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

6.5 Syntactic sugar

consts *equal-tm* :: $'a \Rightarrow 'a \Rightarrow \text{bool tm}$

adhoc-overloading *equal-tm* *equal-nat-tm*

adhoc-overloading *equal-tm* *equal-bool-tm*

consts *plus-tm* :: $'a \Rightarrow 'a \Rightarrow 'a \text{ tm}$

adhoc-overloading *plus-tm* *plus-nat-tm*

consts *times-tm* :: $'a \Rightarrow 'a \Rightarrow 'a \text{ tm}$

adhoc-overloading *times-tm* *times-nat-tm*

consts *power-tm* :: $'a \Rightarrow \text{nat} \Rightarrow 'a \text{ tm}$

adhoc-overloading *power-tm* *power-nat-tm*

consts *minus-tm* :: $'a \Rightarrow 'a \Rightarrow 'a \text{ tm}$

adhoc-overloading *minus-tm* *minus-nat-tm*

consts *less-tm* :: $'a \Rightarrow 'a \Rightarrow \text{bool tm}$

adhoc-overloading *less-tm* *less-nat-tm*

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

```
consts divide-tm :: 'a ⇒ 'a ⇒ 'a tm
adhoc-overloading divide-tm divide-nat-tm
```

```
consts mod-tm :: 'a ⇒ 'a ⇒ 'a tm
adhoc-overloading mod-tm mod-nat-tm
```

```
bundle main-tm-syntax
```

```
begin
```

```
notation equal-tm (infixl =t 51)
notation Not-tm (¬t - [40] 40)
notation conj-tm (infixr ∧t 35)
notation disj-tm (infixr ∨t 30)
notation append-tm (infixr @t 65)
notation plus-tm (infixl +t 65)
notation times-tm (infixl *t 70)
notation power-tm (infixr ^t 80)
notation minus-tm (infixl -t 65)
notation less-tm (infix <t 50)
notation less-eq-tm (infix ≤t 50)
notation mod-tm (infixl modt 70)
notation divide-tm (infixl divt 70)
notation dvd-tm (infix dvdt 50)
```

```
end
```

```
bundle no-main-tm-syntax
```

```
begin
```

```
no-notation equal-tm (infixl =t 51)
no-notation Not-tm (¬t - [40] 40)
no-notation conj-tm (infixr ∧t 35)
no-notation disj-tm (infixr ∨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 ^t 80)
no-notation minus-tm (infixl -t 65)
no-notation less-tm (infix <t 50)
no-notation less-eq-tm (infix ≤t 50)
no-notation mod-tm (infixl modt 70)
no-notation divide-tm (infixl divt 70)
no-notation dvd-tm (infix dvdt 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:  $to\text{-type} \circ from\text{-type} = id$ 
  assumes from-to:  $from\text{-type} \circ to\text{-type} = f$ 
begin
```

```
lemma to-from-elim[simp]:  $to\text{-type} (from\text{-type} x) = x$ 
   $\langle proof \rangle$ 
```

```
lemma from-to-elim:  $from\text{-type} (to\text{-type} x) = f x$ 
   $\langle proof \rangle$ 
```

```
lemma f-idem:  $f \circ f = f$ 
   $\langle proof \rangle$ 
```

```
corollary f-idem-elim[simp]:  $f (f x) = f x$ 
   $\langle proof \rangle$ 
```

```
lemma f-from:  $f \circ from\text{-type} = from\text{-type}$ 
   $\langle proof \rangle$ 
```

```
corollary f-from-elim[simp]:  $f (from\text{-type} x) = from\text{-type} x$ 
   $\langle proof \rangle$ 
```

```
lemma to-f:  $to\text{-type} \circ f = to\text{-type}$ 
   $\langle proof \rangle$ 
```

```
corollary to-f-elim[simp]:  $to\text{-type} (f x) = to\text{-type} x$ 
   $\langle proof \rangle$ 
```

```
lemma f-fixed-point-iff:  $f x = x \iff (\exists y. x = from\text{-type} y)$ 
   $\langle proof \rangle$ 
```

```
lemma f-fixed-point-iff':  $f x = x \iff x = from\text{-type} (to\text{-type} x)$ 
   $\langle proof \rangle$ 
```

```
lemma range-f-range-from:  $range f = range from\text{-type}$ 
   $\langle proof \rangle$ 
```

lemma *to-eq-iff-f-eq*: $to\text{-}type\ x = to\text{-}type\ y \iff f\ x = f\ y$
<proof>

lemma *from-inj*: $inj\ from\text{-}type$
<proof>

end

lemma *from-to-f-criterion*:
 assumes $to\text{-}type \circ from\text{-}type = id$
 assumes $f \circ from\text{-}type = from\text{-}type$
 assumes $\bigwedge x\ y. to\text{-}type\ x = to\text{-}type\ y \implies f\ x = f\ y$
 shows $from\text{-}type \circ to\text{-}type = f$
<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 $from\text{-}type :: 'a \Rightarrow 'b$
 fixes $to\text{-}type :: 'b \Rightarrow 'a$
 fixes $represented\text{-}set :: 'a\ set$
 assumes $to\text{-}from: \bigwedge x. x \in represented\text{-}set \implies to\text{-}type\ (from\text{-}type\ x) = x$
 assumes $to\text{-}type\text{-}in\text{-}represented\text{-}set: \bigwedge y. to\text{-}type\ y \in represented\text{-}set$
begin

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

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

lemma *reduce-reduce[simp]*: $reduced\ (reduce\ x)$
<proof>

definition *representations* **where**
 $representations \equiv from\text{-}type\ 'represented\text{-}set$

lemma *range-reduce*: $representations = range\ reduce$
<proof>

corollary *reduced-from-type[simp]*: $x \in represented\text{-}set \implies reduced\ (from\text{-}type\ x)$

<proof>

lemma *to-type-reduce*: $to\text{-type} (reduce\ x) = to\text{-type}\ x$
<proof>

lemma *reduced-iff*: $reduced\ x \longleftrightarrow (\exists y \in represented\text{-set}. x = from\text{-type}\ y)$
<proof>

lemma *to-eq-iff-f-eq*: $to\text{-type}\ x = to\text{-type}\ y \longleftrightarrow reduce\ x = reduce\ y$
<proof>

lemma *from-inj*: *inj-on from-type represented-set*
<proof>

corollary *from-bij-betw*: *bij-betw from-type represented-set representations*
<proof>

lemma *correctness-to-from*:

fixes $h :: 'a \Rightarrow 'a \Rightarrow 'a$

fixes $g :: 'b \Rightarrow 'b \Rightarrow 'b$

assumes $\bigwedge x\ y. to\text{-type}\ (g\ x\ y) = h\ (to\text{-type}\ x)\ (to\text{-type}\ y)$

shows $\bigwedge x\ y. x \in represented\text{-set} \Longrightarrow y \in represented\text{-set} \Longrightarrow reduce\ (g\ (from\text{-type}\ x)\ (from\text{-type}\ y)) = from\text{-type}\ (h\ x\ y)$
<proof>

end

lemma *from-to-f-criterion*:

assumes $\bigwedge x. x \in represented\text{-set} \Longrightarrow to\text{-type}\ (from\text{-type}\ x) = x$

assumes $\bigwedge x. x \in represented\text{-set} \Longrightarrow f\ (from\text{-type}\ x) = from\text{-type}\ x$

assumes $\bigwedge x\ y. to\text{-type}\ x = to\text{-type}\ y \Longrightarrow f\ x = f\ y$

assumes $\bigwedge y. to\text{-type}\ y \in represented\text{-set}$

shows $\bigwedge x. from\text{-type}\ (to\text{-type}\ x) = f\ x$

<proof>

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*: $i < length\ xs \Longrightarrow \exists xs1\ xs2. xs = xs1\ @\ (xs\ !\ i)\ \#\ xs2 \wedge length\ xs1 = i$
<proof>

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

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

lemma *set-bool-list*: $\text{set } xs \subseteq \{\text{True}, \text{False}\}$
 <proof>

lemma *bool-list-is-replicate-if*:
 assumes $a \notin \text{set } xs$ shows $xs = \text{replicate } (\text{length } xs) \ (\neg a)$
 <proof>

lemma *bit-strong-decomp-2*: $\exists ys \ zs. xs = ys @ a \# zs \implies \exists ys' \ n. xs = ys' @ a \# (\text{replicate } n \ (\neg a))$
 <proof>

lemma *bit-strong-decomp-1*: $\exists ys \ zs. xs = ys @ a \# zs \implies \exists ys' \ n. xs = (\text{replicate } n \ (\neg a) @ a \# ys')$
 <proof>

8.1 Type definition

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

8.2 Conversions

fun *eval-bool* :: *bool* \Rightarrow *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-bool* *a* \leq 1
 <proof>

lemma *eval-bool-inj*: *eval-bool* *a* = *eval-bool* *b* \implies *a* = *b*
 <proof>

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

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

value *from-nat 103*

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

lemma *to-nat-from-nat[simp]: to-nat (from-nat x) = x*
<proof>

lemma *to-nat-explicitly: to-nat xs = (∑ i ← [0..<length xs]. eval-bool (xs ! i) * 2ⁱ)*
<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]) = 2ⁿ*
<proof>

lemma *to-nat-replicate-true[simp]: to-nat (replicate n True) = 2ⁿ - 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 ≥ 2ⁿ ⇒ 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*: $to\text{-}nat\ (take\ k\ xs) = to\text{-}nat\ xs\ mod\ 2^{\wedge}k$
<proof>

lemma *to-nat-drop*: $to\text{-}nat\ (drop\ k\ xs) = to\text{-}nat\ xs\ div\ 2^{\wedge}k$
<proof>

lemma *to-nat-nth-True-bound*:

assumes $i < length\ xs$

assumes $xs\ !\ i = True$

shows $to\text{-}nat\ xs \geq 2^{\wedge}i$

<proof>

8.3 Truncating and filling

fun *truncate-reversed* :: $bool\ list \Rightarrow bool\ list$ **where**

truncate-reversed [] = []

| *truncate-reversed* (x#xs) = (if x then x#xs else *truncate-reversed* xs)

definition *truncate* :: $nat\ \text{lsbf} \Rightarrow nat\ \text{lsbf}$ **where**

truncate xs = rev (*truncate-reversed* (rev xs))

abbreviation *truncated* **where** *truncated* x \equiv *truncate* x = x

lemma *truncate-reversed-eqI[simp]*: $xs = (replicate\ n\ False) @ ys \Longrightarrow$ *truncate-reversed*
xs = truncate-reversed ys

<proof>

corollary *truncate-eqI[simp]*: $xs = ys @ (replicate\ n\ False) \Longrightarrow$ *truncate* xs =
truncate ys

<proof>

lemma *replicate-truncate-reversed*: $\exists n. (replicate\ n\ False) @$ *truncate-reversed* xs
= xs

<proof>

corollary *truncate-replicate*: $\exists n. truncate\ xs @ (replicate\ n\ False) = xs$

<proof>

lemma *decompose-trailing-zeros*: $xs = truncate\ xs @ (replicate\ (length\ xs - length$
 $(truncate\ xs))\ False)$

<proof>

lemma *truncate-reversed-length-ineq*: $length\ (truncate\text{-}reversed\ xs) \leq length\ xs$

<proof>

lemma *truncate-length-ineq*: $length\ (truncate\ xs) \leq length\ xs$

<proof>

lemma *truncate-reversed-fixed-point-iff*: $truncate\text{-}reversed\ x = x \longleftrightarrow (x = [] \vee hd$
 $x = True)$

<proof>

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

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

lemma *last-truncate*: $\text{truncate } xs \neq [] \implies \text{last } (\text{truncate } xs) = \text{True}$
(proof)

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

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

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

corollary *truncate-neq-Nil*: $\text{truncate } xs \neq [] \implies \exists ys zs. xs = ys @ \text{True} \# zs$
(proof)

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

lemma *truncate-eq-Cons*: $\text{truncate } xs = \text{truncate } ys \implies \text{truncate } (a \# xs) = \text{truncate } (a \# ys)$
(proof)

lemma *truncate-as-take*: $\bigwedge xs. \exists n. \text{truncate } xs = \text{take } n \text{ } xs$
(proof)

lemma *to-nat-zero-iff*: $\text{to-nat } xs = 0 \longleftrightarrow \text{truncate } xs = []$
(proof)

lemma *to-nat-eq-imp-truncate-eq*: $\text{to-nat } xs = \text{to-nat } ys \implies \text{truncate } xs = \text{truncate } ys$
(proof)

lemma *truncate-from-nat[simp]*: $\text{truncate } (\text{from-nat } x) = \text{from-nat } x$
(proof)

lemma *truncate-and-length-eq-imp-eq*:
 assumes $\text{truncate } xs = \text{truncate } ys$ $\text{length } xs = \text{length } ys$
 shows $xs = ys$
(proof)

lemma *nat-lsbf-eqI*:

assumes $to\text{-}nat\ x = to\text{-}nat\ y$

assumes $length\ x = length\ y$

shows $x = y$

<proof>

interpretation *nat-lsbf*: *abstract-representation from-nat to-nat truncate*

<proof>

lemma *truncated-Cons-imp-truncated-tl*: $truncated\ (x \# xs) \implies truncated\ xs$

<proof>

definition *fill where* $fill\ n\ xs = xs @ replicate\ (n - length\ xs)\ False$

lemma *to-nat-fill[simp]*: $to\text{-}nat\ (fill\ n\ xs) = to\text{-}nat\ xs$

<proof>

lemma *length-fill[intro]*: $length\ xs \leq n \implies length\ (fill\ n\ xs) = n$

<proof>

lemma *take-id*: $length\ xs = k \implies take\ k\ xs = xs$

<proof>

lemma *fill-id*: $length\ xs \geq k \implies fill\ k\ xs = xs$

<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*: $length\ xs \leq k \implies fill\ k\ (truncate\ xs) = fill\ k\ xs$

<proof>

lemma *fill-take-com*: $fill\ k\ (take\ k\ xs) = take\ k\ (fill\ k\ xs)$

<proof>

lemma *to-nat-length-lower-bound-truncated*: $xs \neq [] \implies truncated\ xs \implies to\text{-}nat\ xs \geq 2^{\wedge}(length\ xs - 1)$

<proof>

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

8.4 Right-shifts

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

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

lemma *length-shift-right[simp]*: $\text{length } (\text{shift-right } n \text{ } xs) = n + \text{length } xs$
 ⟨proof⟩

8.5 Subdividing lists

8.5.1 Splitting a list in two blocks

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

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

lemma *app-split*: $\text{split } xs = (x0, x1) \implies xs = x0 @ x1$
 ⟨proof⟩

lemma *length-split*: $\text{length } xs \bmod 2 = 0 \implies \text{split } xs = (x0, x1) \implies \text{length } x0 = \text{length } xs \text{ div } 2 \wedge \text{length } x1 = \text{length } xs \text{ div } 2$
 ⟨proof⟩

lemma *length-split-le*:

assumes $\text{split } xs = (x0, x1)$
shows $\text{length } x0 \leq \text{length } xs$ **and** $\text{length } x1 \leq \text{length } xs$
 ⟨proof⟩

8.5.2 Splitting a list in multiple blocks

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

fun *subdivide* :: $\text{nat} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list list}$ **where**
subdivide 0 $xs = \text{undefined}$
 | *subdivide* n $[] = []$
 | *subdivide* n $xs = \text{take } n \text{ } xs \# \text{subdivide } n \text{ } (\text{drop } n \text{ } xs)$

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

value *subdivide* 2 $[0..<6]$

value *subdivide* 3 $[0..<6]$

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

lemma *concat-subdivide*: $n > 0 \implies \text{concat } (\text{subdivide } n \text{ } xs) = xs$
<proof>

lemma *subdivide-step*:

assumes $n > 0$

assumes $xs \neq []$

assumes $\text{length } xs = n * k$

obtains $ys \ zs$ **where** $xs = ys @ zs$ $\text{length } ys = n$ $\text{length } zs = n * (k - 1)$
 $\text{subdivide } n \ xs = ys \# \text{subdivide } n \ zs$

<proof>

lemma *subdivide-step'*:

assumes $n > 0$

assumes $xs \neq []$

shows $\text{subdivide } n \ xs = (\text{take } n \ xs) \# \text{subdivide } n \ (\text{drop } n \ xs)$

<proof>

lemma *subdivide-correct*:

assumes $n > 0$

assumes $\text{length } xs = n * k$

shows $\text{length } (\text{subdivide } n \ xs) = k \wedge (x \in \text{set } (\text{subdivide } n \ xs) \implies \text{length } x = n)$

<proof>

lemma *nth-nth-subdivide*:

assumes $n > 0$

assumes $\text{length } xs = n * k$

assumes $i < k$ $j < n$

shows $\text{subdivide } n \ xs ! i ! j = xs ! (i * n + j)$

<proof>

lemma *subdivide-concat*:

assumes $n > 0$

assumes $\bigwedge i. i < \text{length } xs \implies \text{length } (xs ! i) = n$

shows $\text{subdivide } n \ (\text{concat } xs) = xs$

<proof>

lemma *to-nat-subdivide*:

assumes $n > 0$

assumes $\text{length } xs = n * k$

shows $\text{to-nat } xs = (\sum i \leftarrow [0.. k]. \text{to-nat } (\text{subdivide } n \ xs ! i) * 2 ^ (i * n))$

<proof>

8.6 The *bitsize* function

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

fun *bitsize* :: $\text{nat} \Rightarrow \text{nat}$ **where**

bitsize 0 = 0

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

lemma *bitsize-is-floorlog*: $\text{bitsize } = \text{floorlog } 2$
<proof>

corollary *bitsize-bitlen*: $\text{int } (\text{bitsize } n) = \text{bitlen } (\text{int } n)$
<proof>

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

lemma *bitsize-zero-iff*: $\text{bitsize } n = 0 \longleftrightarrow n = 0$
<proof>

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

lemma *bitsize-length*: $\text{bitsize } n \leq k \longleftrightarrow n < 2 \wedge k$
<proof>

lemma *two-pow-bitsize-pos-bound*: $n > 0 \implies 2 \wedge \text{bitsize } n \leq 2 * n$
<proof>

lemma *two-pow-bitsize-bound*: $2 \wedge \text{bitsize } n \leq 2 * n + 1$
<proof>

lemma *bitsize-mono*: $n1 \leq n2 \implies \text{bitsize } n1 \leq \text{bitsize } n2$
<proof>

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

lemma *power-of-2-recursion*: $(\exists k. (n::\text{nat}) = 2 \wedge k) \longleftrightarrow (n = 1 \vee (n \text{ mod } 2 = 0 \wedge (\exists k. n \text{ div } 2 = 2 \wedge k)))$
<proof>

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

lemma *is-power-of-2-correct*: $\text{is-power-of-2 } n \longleftrightarrow (\exists k. n = 2 \wedge k)$
<proof>

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

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

lemma *next-power-of-2-upper-bound*:

assumes $k \neq 0$

shows $\text{next-power-of-2 } k \leq 2 * k$

<proof>

lemma *next-power-of-2-upper-bound'*: $\text{next-power-of-2 } k \leq 2 * k + 1$

<proof>

lemma *next-power-of-2-is-power-of-2*: $\exists k. \text{next-power-of-2 } n = 2 \wedge k$

<proof>

8.7 Addition

fun *bit-add-carry* :: $\text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool} \times \text{bool}$ **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*: $\text{bit-add-carry } c \ x \ y = (a, b) \implies \text{eval-bool } c + \text{eval-bool } x + \text{eval-bool } y = \text{eval-bool } a + 2 * \text{eval-bool } b$

<proof>

8.7.1 Increment operation

fun *inc-nat* :: $\text{nat-lsbf} \Rightarrow \text{nat-lsbf}$ **where**

inc-nat [] = [*True*]

| *inc-nat* (*False* # *xs*) = *True* # *xs*

| *inc-nat* (*True* # *xs*) = *False* # (*inc-nat* *xs*)

lemma *length-inc-nat'*: $\text{length } (\text{inc-nat } xs) = \text{length } xs + \text{of-bool } (\text{to-nat } xs + 1 \geq 2 \wedge \text{length } xs)$

<proof>

lemma *length-inc-nat-lower*: $\text{length } (\text{inc-nat } xs) \geq \text{length } xs$

<proof>

lemma *length-inc-nat-upper*: $\text{length } (\text{inc-nat } xs) \leq \text{length } xs + 1$

<proof>

lemma *inc-nat-nonempty*: $\text{inc-nat } xs \neq []$

<proof>

lemma *inc-nat-replicate-True*: $\text{inc-nat } (\text{replicate } m \ \text{True}) = \text{replicate } m \ \text{False} @ [\text{True}]$

<proof>

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

<proof>

lemma *length-inc-nat-iff*: $\text{length } (\text{inc-nat } xs) = \text{length } xs \iff (\exists \text{ ys zs. } xs = \text{ys } @ \text{ False } \# \text{ zs})$

<proof>

lemma *inc-nat-last-bit-True*: $\text{length } (\text{inc-nat } xs) = \text{Suc } (\text{length } xs) \implies \exists \text{ zs. } \text{inc-nat } xs = \text{zs } @ [\text{True}]$

<proof>

lemma *inc-nat-truncated*: $\text{truncated } xs \implies \text{truncated } (\text{inc-nat } xs)$

<proof>

lemma *inc-nat-correct*: $\text{to-nat } (\text{inc-nat } xs) = \text{to-nat } xs + 1$

<proof>

lemma *length-inc-nat*: $\text{length } (\text{inc-nat } xs) = \max (\text{length } xs) (\text{floorlog } 2 (\text{to-nat } xs + 1))$

<proof>

8.7.2 Addition with a carry bit

fun *add-carry* :: $\text{bool} \Rightarrow \text{nat-lsbf} \Rightarrow \text{nat-lsbf} \Rightarrow \text{nat-lsbf}$ **where**

add-carry *False* [] *y* = *y*

| *add-carry* *False* *x* [] = *x*

| *add-carry* *True* [] *y* = *inc-nat* *y*

| *add-carry* *True* *x* [] = *inc-nat* *x*

| *add-carry* *c* (*x*#*xs*) (*y*#*ys*) = (let (*a*, *b*) = *bit-add-carry* *c* *x* *y* in *a*#(*add-carry* *b* *xs* *ys*))

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

<proof>

lemma *length-add-carry'*: $\text{length } (\text{add-carry } c \text{ } xs \text{ } ys) = \max (\text{length } xs) (\text{length } ys) + \text{of-bool } (\text{to-nat } xs + \text{to-nat } ys + \text{of-bool } c \geq 2 \wedge \max (\text{length } xs) (\text{length } ys))$

<proof>

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

<proof>

lemma *length-add-carry-lower*: $\text{length } (\text{add-carry } c \text{ } xs \text{ } ys) \geq \max (\text{length } xs) (\text{length } ys)$

<proof>

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

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

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

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

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

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

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

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

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

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

8.7.3 Addition

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

corollary *length-add-nat-lower*: $\text{length } (\text{add-nat } xs \ ys) \geq \max (\text{length } xs) (\text{length } ys)$

<proof>

corollary *length-add-nat-upper*: $\text{length } (\text{add-nat } xs \ ys) \leq \max (\text{length } xs) (\text{length } ys) + 1$

<proof>

corollary *add-nat-last-bit-True*: $\text{length } (\text{add-nat } xs \ ys) = \max (\text{length } xs) (\text{length } ys) + 1 \implies \exists zs. \text{add-nat } xs \ ys = zs \text{ @ } [True]$

<proof>

lemma *add-nat-correct*: $\text{to-nat } (\text{add-nat } x \ y) = \text{to-nat } x + \text{to-nat } y$

<proof>

corollary *add-nat-com*: $\text{add-nat } xs \ ys = \text{add-nat } ys \ xs$

<proof>

corollary *add-nat-assoc*: $\text{add-nat } xs \ (\text{add-nat } ys \ zs) = \text{add-nat } (\text{add-nat } xs \ ys) \ zs$

<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* \Rightarrow *bool list* \Rightarrow *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*:

$\text{length } xs = \text{length } ys \implies \text{compare-nat-same-length-reversed } xs \ ys \longleftrightarrow \text{to-nat } (\text{rev } xs) \leq \text{to-nat } (\text{rev } ys)$

<proof>

fun *compare-nat-same-length* :: *nat-lsbf* \Rightarrow *nat-lsbf* \Rightarrow *bool* **where**

compare-nat-same-length xs ys = *compare-nat-same-length-reversed (rev xs) (rev ys)*

lemma *compare-nat-same-length-correct*:

$length\ xs = length\ ys \implies compare\text{-}nat\text{-}same\text{-}length\ xs\ ys = (to\text{-}nat\ xs \leq to\text{-}nat\ ys)$
 ⟨proof⟩

definition $make\text{-}same\text{-}length :: nat\text{-}lsbf \Rightarrow nat\text{-}lsbf \Rightarrow nat\text{-}lsbf \times nat\text{-}lsbf$ **where**
 $make\text{-}same\text{-}length\ xs\ ys = (let\ n = max\ (length\ xs)\ (length\ ys)\ in\ ((fill\ n\ xs), (fill\ n\ ys)))$

lemma $make\text{-}same\text{-}length\text{-}correct$:
assumes $(fill\ xs, fill\ ys) = make\text{-}same\text{-}length\ xs\ ys$
shows $length\ fill\ ys = length\ fill\ xs$
 $length\ fill\ xs = max\ (length\ xs)\ (length\ ys)$
 $to\text{-}nat\ fill\ xs = to\text{-}nat\ xs$
 $to\text{-}nat\ fill\ ys = to\text{-}nat\ ys$
 ⟨proof⟩

definition $compare\text{-}nat :: nat\text{-}lsbf \Rightarrow nat\text{-}lsbf \Rightarrow bool$ **where**
 $compare\text{-}nat\ xs\ ys = (let\ (fill\ xs, fill\ ys) = make\text{-}same\text{-}length\ xs\ ys\ in\ compare\text{-}nat\text{-}same\text{-}length\ fill\ xs\ fill\ ys)$

lemma $compare\text{-}nat\text{-}correct$: $compare\text{-}nat\ xs\ ys = (to\text{-}nat\ xs \leq to\text{-}nat\ ys)$
 ⟨proof⟩

8.8.2 Subtraction

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

lemma $add\text{-}complement$: $add\text{-}nat\ xs\ (map\ Not\ xs) = replicate\ (length\ xs)\ True$
 ⟨proof⟩

lemma $to\text{-}nat\text{-}complement$: $to\text{-}nat\ (map\ Not\ xs) = 2^{length\ xs} - 1 - to\text{-}nat\ xs$
 ⟨proof⟩

lemma $to\text{-}nat\text{-}butlast$: $zs = xs @ [True] \implies to\text{-}nat\ (butlast\ zs) = to\text{-}nat\ zs - 2^{length\ xs}$
 ⟨proof⟩

lemma $inc\text{-}nat\text{-}true\text{-}prefix[simp]$: $inc\text{-}nat\ (replicate\ n\ True @ [False] @ ys) = replicate\ n\ False @ [True] @ ys$
 ⟨proof⟩

lemma $length\text{-}inc\text{-}nat\text{-}aux$: $zs = replicate\ n\ True @ [False] @ ys \implies length\ (inc\text{-}nat\ zs) = length\ zs$
 ⟨proof⟩

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

lemma *subtract-nat-aux*: $\text{to-nat } (\text{subtract-nat } xs \ ys) = (\text{to-nat } xs) - (\text{to-nat } ys) \wedge \text{length } (\text{subtract-nat } xs \ ys) \leq \max (\text{length } xs) (\text{length } ys)$
 ⟨proof⟩

corollary *subtract-nat-correct*: $\text{to-nat } (\text{subtract-nat } xs \ ys) = (\text{to-nat } xs) - (\text{to-nat } ys)$
 ⟨proof⟩

corollary *length-subtract-nat-le*: $\text{length } (\text{subtract-nat } xs \ ys) \leq \max (\text{length } xs) (\text{length } ys)$
 ⟨proof⟩

8.9 (Grid) Multiplication

fun *grid-mul-nat* :: $\text{nat-lsbf} \Rightarrow \text{nat-lsbf} \Rightarrow \text{nat-lsbf}$ **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*: $\text{to-nat } (\text{grid-mul-nat } x \ y) = \text{to-nat } x * \text{to-nat } y$
 ⟨proof⟩

lemma *length-grid-mul-nat*: $\text{length } (\text{grid-mul-nat } xs \ ys) \leq \text{length } xs + \text{length } ys$
 ⟨proof⟩

8.10 Syntax bundles

abbreviation *shift-right-flip* $xs \ n \equiv \text{shift-right } n \ xs$

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 $a\ b\ c\ d :: \text{nat}$

fixes $f :: \text{nat} \Rightarrow \text{nat}$

assumes *small-bounds*: $f\ 0 \leq a\ f\ (\text{Suc}\ 0) \leq a$

assumes *recursive-bound*: $\bigwedge n. n > 1 \implies f\ n \leq c * n + d + f\ (n\ \text{div}\ 2)$

begin

private fun g **where**

$g\ 0 = a$

| $g\ (\text{Suc}\ 0) = a$

| $g\ n = c * n + d + g\ (n\ \text{div}\ 2)$

private lemma *f-g-bound*: $f\ n \leq g\ n$

<proof> **lemma** *g-mono-aux*: $a \leq g\ n$

<proof> **lemma** *g-mono*: $m \leq n \implies g\ m \leq g\ n$

<proof> **lemma** *g-powers-of-2*: $g\ (2^n) = d * n + c * (2^{n+1} - 2) + a$

<proof> **lemma** *pow-ineq*:

assumes $m \geq (1 :: \text{nat})$

assumes $p \geq 2$

shows $p^m > m$

<proof> **lemma** *next-power-of-2*:

assumes $m \geq (1 :: \text{nat})$

shows $\exists n\ k. m = 2^n + k \wedge k < 2^n$

<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 $fl :: \text{nat} \Rightarrow \text{nat}$

fixes $f :: \text{nat} \Rightarrow \text{real}$

assumes $(\lambda x. \text{real}\ (fl\ x)) \in O(\lambda n. \text{real}\ n)$

```

assumes  $f \in O(\lambda n. \text{real } n \text{ powr } p)$ 
assumes  $p > 0$ 
shows  $f \circ \text{fl} \in O(\lambda n. \text{real } n \text{ powr } p)$ 
⟨proof⟩

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

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

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

lemma (in landau-pair) big-1-mult-left:
  fixes  $f g h$ 
  assumes  $f \in L F (g) h \in L F (\lambda-. 1)$ 
  shows  $(\lambda x. h x * f x) \in L F (g)$ 
⟨proof⟩

lemma norm-nonneg:  $x \geq 0 \implies \text{norm } x = x$  ⟨proof⟩

lemma landau-mono-always:
  fixes  $f g$ 
  assumes  $\bigwedge x. f x \geq (0 :: \text{real}) \bigwedge x. g x \geq 0$ 
  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 ::  $\text{nat-lsbf} \Rightarrow \text{nat-lsbf } tm$  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]:  $\text{val } (\text{truncate-reversed-tm } xs) = \text{truncate-reversed } xs$ 
⟨proof⟩

```

```

lemma time-truncate-reversed-tm-le:  $\text{time } (\text{truncate-reversed-tm } xs) \leq \text{length } xs +$ 

```

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]*: $\text{time } (\text{shift-right-tm } n \text{ } xs) = 2 * n + 3$
 ⟨proof⟩

9.3 Subdividing lists

9.3.1 Splitting a list in two blocks

definition *split-at-tm* :: $\text{nat} \Rightarrow 'a \text{ list} \Rightarrow ('a \text{ list} \times 'a \text{ list}) \text{ tm}$ **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]*: $\text{val } (\text{split-at-tm } k \text{ } xs) = \text{split-at } k \text{ } xs$
 ⟨proof⟩

lemma *time-split-at-tm*: $\text{time } (\text{split-at-tm } k \text{ } xs) = 2 * \min k (\text{length } xs) + 3$
 ⟨proof⟩

definition *split-tm* :: $\text{nat-lsbf} \Rightarrow (\text{nat-lsbf} \times \text{nat-lsbf}) \text{ tm}$ **where**

```
split-tm xs =1 do {
  n ← length-tm xs;
  n-div-2 ← n divt 2;
  split-at-tm n-div-2 xs
}
```

lemma *val-split-tm[simp, val-simp]*: $\text{val } (\text{split-tm } xs) = \text{split } xs$
 ⟨proof⟩

lemma *time-split-tm-le*: $\text{time } (\text{split-tm } xs) \leq 10 * \text{length } xs + 16$
 ⟨proof⟩

9.3.2 Splitting a list in multiple blocks

fun *subdivide-tm* :: $\text{nat} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list list tm}$ **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]*: $n > 0 \implies \text{val } (\text{subdivide-tm } n \text{ } xs) = \text{subdivide } n \text{ } xs$
 ⟨proof⟩

lemma *time-subdivide-tm-le-aux*:

assumes $n > 0$

shows $\text{time } (\text{subdivide-tm } n \text{ } xs) \leq k * (2 * n + 3) + \text{time } (\text{subdivide-tm } n \text{ } (\text{drop } (k * n) \text{ } xs))$

<proof>

lemma *time-subdivide-tm-le*:

fixes $xs :: 'a \text{ list}$

assumes $n > 0$

shows $\text{time } (\text{subdivide-tm } n \text{ } xs) \leq 5 * \text{length } xs + 2 * n + 4$

<proof>

9.4 The *bitsize* function

fun *bitsize-tm* :: $\text{nat} \Rightarrow \text{nat tm}$ **where**

bitsize-tm 0 =1 return 0

| *bitsize-tm* n =1 do {

$n\text{-div-2} \leftarrow n \text{ div}_t 2$;

$r \leftarrow \text{bitsize-tm } n\text{-div-2}$;

$1 +_t r$

}

lemma *val-bitsize-tm[simp, val-simp]*: $\text{val } (\text{bitsize-tm } n) = \text{bitsize } n$

<proof>

fun *time-bitsize-tm-bound* :: $\text{nat} \Rightarrow \text{nat}$ **where**

time-bitsize-tm-bound 0 = 1

| *time-bitsize-tm-bound* n = $14 + 8 * n + \text{time-bitsize-tm-bound } (n \text{ div } 2)$

lemma *time-bitsize-tm-aux*:

$\text{time } (\text{bitsize-tm } n) \leq \text{time-bitsize-tm-bound } n$

<proof>

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

<proof>

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

<proof>

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

fun *is-power-of-2-tm* :: $\text{nat} \Rightarrow \text{bool tm}$ **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 {

$n\text{-mod-2} \leftarrow n \text{ mod}_t 2$;

$n\text{-div-2} \leftarrow n \text{ div}_t 2$;

$c1 \leftarrow n\text{-mod-2} =_t 0$;

$c2 \leftarrow \text{is-power-of-2-tm } n\text{-div-2}$;

```

    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) \leq 114 * n + 1*
 <proof>

definition *next-power-of-2-tm* :: *nat \Rightarrow nat tm* **where**
next-power-of-2-tm n = 1 do {
 b \leftarrow *is-power-of-2-tm n*;
 if b then return n else do {
 r \leftarrow *bitsize-tm n*;
 2 $\hat{\wedge}_t$ 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) \leq 208 * n + 37*
 <proof>

9.5 Addition

fun *bit-add-carry-tm* :: *bool \Rightarrow bool \Rightarrow bool \Rightarrow (bool \times 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 \Rightarrow 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 \leftarrow *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 $\text{length } xs = \text{length } ys$

shows $\text{val } (\text{compare-nat-same-length-reversed-tm } xs \text{ } ys) = \text{compare-nat-same-length-reversed}$
 $xs \text{ } ys$

<proof>

lemma *time-compare-nat-same-length-reversed-tm-le*:

$\text{length } xs = \text{length } ys \implies \text{time } (\text{compare-nat-same-length-reversed-tm } xs \text{ } ys) \leq$
 $\text{length } xs + 1$

<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 $\text{length } xs = \text{length } ys$

shows $\text{val } (\text{compare-nat-same-length-tm } xs \text{ } ys) = \text{compare-nat-same-length } xs \text{ } ys$

<proof>

lemma *time-compare-nat-same-length-tm-le*:

$\text{length } xs = \text{length } ys \implies \text{time } (\text{compare-nat-same-length-tm } xs \text{ } ys) \leq 3 * \text{length}$
 $xs + 6$

<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 ≤_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 ← grid-mul-nat-tm xs ys;
    return (False # r)
  }
| grid-mul-nat-tm (True # xs) ys = 1 do {
  r ← grid-mul-nat-tm xs ys;
  add-nat-tm (False # r) ys
}

```

lemma *val-grid-mul-nat-tm*[simp, val-simp]: *val* (grid-mul-nat-tm xs ys) = xs *_n ys
 ⟨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) ≤ 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 ≡ *shift-right-tm* n xs

bundle *nat-lsbf-tm-syntax*
begin
notation *add-nat-tm* (**infixl** +_{nt} 65)
notation *compare-nat-tm* (**infixl** ≤_{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** ≤_{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 $\bigwedge x y. \text{to-nat } (f \ x \ y) = \text{to-nat } x * \text{to-nat } y$
shows $\bigwedge x y \ xs \ ys. \text{to-int } (\text{nat-mul-to-int-mul } f \ (x, \ xs) \ (y, \ ys)) = \text{to-int } (x, \ xs) * \text{to-int } (y, \ ys)$
 ⟨*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 $\equiv 42$

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 \Rightarrow nat-lsbf \Rightarrow nat-lsbf \Rightarrow nat-lsbf **where**
karatsuba-on-power-of-2-length k x y =
 (if k \leq *karatsuba-lower-bound*
 then x \ast_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 \leq_n x0) = (y1 \leq_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 \wedge l
 assumes length-x: length x = k
 assumes length-y: length y = k
 assumes recursion-condition: \neg k \leq *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

$(\text{fill } k\text{-div-2 } (\text{abs-diff } x0 \ x1))$
 $(\text{fill } k\text{-div-2 } (\text{abs-diff } y0 \ y1))$
definition *add01* **where** $\text{add01} = \text{prod0} +_n \text{prod1}$
definition *r* **where** $r = (\text{if } (x1 \leq_n x0) = (y1 \leq_n y0)$
 $\text{then } \text{add01} -_n \text{prod2}$
 $\text{else } \text{add01} +_n \text{prod2})$

lemma *split-x*: $\text{split } x = (x0, x1) \langle \text{proof} \rangle$
lemma *split-y*: $\text{split } y = (y0, y1) \langle \text{proof} \rangle$

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* $k \ x \ y =$
 $\text{prod0} +_n (r \gg_n k\text{-div-2}) +_n (\text{prod1} \gg_n k)$
 $\langle \text{proof} \rangle$

lemma *l-ge-1*: $l \geq 1$
 $\langle \text{proof} \rangle$

lemma *k-even*: $k \bmod 2 = 0$
 $\langle \text{proof} \rangle$

lemma *k-div-2*: $k\text{-div-2} = 2^{\wedge} (l - 1)$
 $\langle \text{proof} \rangle$

lemma *k-div-2-less-k*: $k\text{-div-2} < k$
 $\langle \text{proof} \rangle$

lemma *length-x-split*: $\text{length } x0 = k\text{-div-2} \ \text{length } x1 = k\text{-div-2}$
 $\langle \text{proof} \rangle$

lemma *length-y-split*: $\text{length } y0 = k\text{-div-2} \ \text{length } y1 = k\text{-div-2}$
 $\langle \text{proof} \rangle$

lemma *length-abs-diff-x0-x1*: $\text{length } (\text{abs-diff } x0 \ x1) \leq k\text{-div-2}$
 $\langle \text{proof} \rangle$
lemma *length-fill-abs-diff-x0-x1*: $\text{length } (\text{fill } k\text{-div-2 } (\text{abs-diff } x0 \ x1)) = k\text{-div-2}$
 $\langle \text{proof} \rangle$

lemma *length-abs-diff-y0-y1*: $\text{length } (\text{abs-diff } y0 \ y1) \leq k\text{-div-2}$
 $\langle \text{proof} \rangle$
lemma *length-fill-abs-diff-y0-y1*: $\text{length } (\text{fill } k\text{-div-2 } (\text{abs-diff } y0 \ y1)) = k\text{-div-2}$
 $\langle \text{proof} \rangle$

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

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 $k = 2 \wedge l$

assumes $\text{length } x = k \text{ length } y = k$

shows $\text{to-nat } (\text{karatsuba-on-power-of-2-length } k \ x \ y) = \text{to-nat } x * \text{to-nat } y$

<proof>

function *len-kar-bound* **where**

len-kar-bound $l = (\text{if } 2 \wedge l \leq \text{karatsuba-lower-bound} \text{ then } 2 * \text{karatsuba-lower-bound}$
 else $2 \wedge l + \text{len-kar-bound } (l - 1) + 4$)

<proof>

termination

<proof>

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

lemma *length-karatsuba-on-power-of-2-aux*:

assumes $k = 2 \wedge l$

assumes $\text{length } x = k \text{ length } y = k$

shows $\text{length } (\text{karatsuba-on-power-of-2-length } k \ x \ y) \leq \text{len-kar-bound } l$

<proof>

lemma *len-kar-bound-le*: $\text{len-kar-bound } l \leq 6 * 2 \wedge l + 2 * \text{karatsuba-lower-bound}$
<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 $k = 2 \wedge l$

assumes $\text{length } x = k \text{ length } y = k$

shows $\text{length } (\text{karatsuba-on-power-of-2-length } k \ x \ y) \leq 6 * k + 2 * \text{karatsuba-lower-bound}$

<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* :: $\text{nat-lsbf} \Rightarrow \text{nat-lsbf} \Rightarrow \text{nat-lsbf}$ **where**

karatsuba-mul-nat $x \ y = (\text{let } k = \text{next-power-of-2 } (\max (\text{length } x) (\text{length } y)) \text{ in}$
 $\text{karatsuba-on-power-of-2-length } k \ (\text{fill } k \ x) \ (\text{fill } k \ y))$

We verify the correctness of Karatsuba multiplication:

theorem *karatsuba-mul-nat-correct*: $to\text{-}nat\ (karatsuba\text{-}mul\text{-}nat\ x\ y) = to\text{-}nat\ x * to\text{-}nat\ y$
 $\langle proof \rangle$

lemma *length-karatsuba-mul-nat-le*: $length\ (karatsuba\text{-}mul\text{-}nat\ x\ y) \leq 12 * max\ (length\ x)\ (length\ y) + (6 + 2 * karatsuba\text{-}lower\text{-}bound)$
 $\langle proof \rangle$

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 $x\ y = nat\text{-}mul\text{-}to\text{-}int\text{-}mul\ karatsuba\text{-}mul\text{-}nat\ x\ y$

corollary *karatsuba-mul-int-correct*:
 $to\text{-}int\ (karatsuba\text{-}mul\text{-}int\ x\ y) = to\text{-}int\ x * to\text{-}int\ y$
 $\langle proof \rangle$

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 $\mathcal{O}(n^{\log_2 3})$.

definition *abs-diff-tm* :: $nat\text{-}lsbf \Rightarrow nat\text{-}lsbf \Rightarrow nat\text{-}lsbf\ tm$ **where**
abs-diff-tm $xs\ ys = 1\ do\ \{$
 $\quad r1 \leftarrow xs\ \text{-}_{nt}\ ys;$
 $\quad r2 \leftarrow ys\ \text{-}_{nt}\ xs;$
 $\quad r1\ \text{+}_{nt}\ r2$
 $\}$

lemma *val-abs-diff-tm[simp, val-simp]*: $val\ (abs\text{-}diff\text{-}tm\ xs\ ys) = abs\text{-}diff\ xs\ ys$
 $\langle proof \rangle$

lemma *time-abs-diff-tm-le*: $time\ (abs\text{-}diff\text{-}tm\ xs\ ys) \leq 62 * max\ (length\ xs)\ (length\ ys) + 100$
 $\langle proof \rangle$

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* = $(x1 \leq_n x0)$

definition *sgny* **where** *sgny* = $(y1 \leq_n y0)$

definition *sgnxy* **where** *sgnxy* = $(sgnx = sgny)$

definition *r'* **where** *r'* = $(\text{if } sgnxy \text{ then } add01 \text{ } -_n \text{ } prod2 \text{ else } add01 \text{ } +_n \text{ } prod2)$

definition *sr* **where** *sr* = $r \gg_n k\text{-div-2}$

definition *add0sr* **where** *add0sr* = $prod0 +_n sr$

definition *s1* **where** *s1* = $prod1 \gg_n k$

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* $\leq k$

assumes *length y* $\leq k$

assumes *length (x +_n y)* = $k + 1$

shows $\max (\text{length } x) (\text{length } y) = k \text{ Nat-LSBF.to-nat } x + \text{Nat-LSBF.to-nat } y$
 $\geq 2 \wedge k$

<proof>

context begin

private fun *f* **where**

f k = $(\text{if } k \leq \text{karatsuba-lower-bound} \text{ then } 2 * k \text{ else } f (k \text{ div } 2) + k + 4)$

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

private lemma *f-linear*: $f k \leq 6 * k$

<proof> **lemma** *f-bound*:

assumes $k = 2 \wedge l$

assumes *length x* = k

assumes *length y* = k

shows *length (karatsuba-on-power-of-2-length k x y)* $\leq f k$

<proof>

lemma *length-karatsuba-on-power-of-2-length*:

assumes $k = 2 \wedge l$

assumes *length x* = k

assumes *length y* = k

shows *length (karatsuba-on-power-of-2-length k x y)* $\leq 6 * k$

<proof>

end

function *karatsuba-on-power-of-2-length-tm* :: *nat* \Rightarrow *nat-lsbf* \Rightarrow *nat-lsbf* \Rightarrow *nat-lsbf*
tm **where**

```
karatsuba-on-power-of-2-length-tm k xs ys = 1 do {
  b  $\leftarrow$  k  $\leq_t$  karatsuba-lower-bound;
  (if b then grid-mul-nat-tm xs ys else do {
    (x0, x1)  $\leftarrow$  split-tm xs;
    (y0, y1)  $\leftarrow$  split-tm ys;
    k-div-2  $\leftarrow$  k divt 2;
    prod0  $\leftarrow$  karatsuba-on-power-of-2-length-tm k-div-2 x0 y0;
    prod1  $\leftarrow$  karatsuba-on-power-of-2-length-tm k-div-2 x1 y1;
    abs-diff-x  $\leftarrow$  (abs-diff-tm x0 x1  $\gg$  fill-tm k-div-2);
    abs-diff-y  $\leftarrow$  (abs-diff-tm y0 y1  $\gg$  fill-tm k-div-2);
    prod2  $\leftarrow$  karatsuba-on-power-of-2-length-tm k-div-2 abs-diff-x abs-diff-y;
    sgnx  $\leftarrow$  x1  $\leq_{nt}$  x0;
    sgny  $\leftarrow$  y1  $\leq_{nt}$  y0;
    sgnxy  $\leftarrow$  sgnx  $=_t$  sgny;
    — construct return value
    add01  $\leftarrow$  prod0  $+_{nt}$  prod1;
    r  $\leftarrow$  (if sgnxy then add01  $-_{nt}$  prod2 else add01  $+_{nt}$  prod2);
    sr  $\leftarrow$  r  $\gg_{nt}$  k-div-2;
    add0sr  $\leftarrow$  prod0  $+_{nt}$  sr;
    s1  $\leftarrow$  prod1  $\gg_{nt}$  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  $\leq$  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 $\text{time}(\text{karatsuba-on-power-of-2-length-tm } k \text{ } xs \text{ } ys) \leq h \ k$
 ⟨proof⟩

lemma $n\text{-div-2}$: $n \text{ div } 2 = \text{nat } \lfloor \text{real } n / 2 \rfloor$
 ⟨proof⟩

function $h\text{-real} :: \text{nat} \Rightarrow \text{real}$ **where**
 $x \leq \text{karatsuba-lower-bound} \Rightarrow h\text{-real } x = 8 * x * x + 2 * x + 3$
 $| x > \text{karatsuba-lower-bound} \Rightarrow h\text{-real } x = 407 + 224 * x + 3 * h\text{-real } (\text{nat } (\lfloor \text{real } x / 2 \rfloor))$
 ⟨proof⟩

termination
 ⟨proof⟩

lemma $h\text{-h-real}$: $\text{real } (h \ k) = h\text{-real } k$
 ⟨proof⟩

lemma $h\text{-real-bigo}$: $h\text{-real} \in O(\lambda n. \text{real } n \text{ powr } \log 2 \ 3)$
 ⟨proof⟩

definition $\text{karatsuba-mul-nat-tm} :: \text{nat-lsbf} \Rightarrow \text{nat-lsbf} \Rightarrow \text{nat-lsbf } tm$ **where**
 $\text{karatsuba-mul-nat-tm } xs \text{ } ys = 1 \text{ do } \{$
 $\text{lenx} \leftarrow \text{length-tm } xs;$
 $\text{leny} \leftarrow \text{length-tm } ys;$
 $k \leftarrow \text{max-nat-tm } \text{lenx } \text{leny} \gg \text{next-power-of-2-tm};$
 $\text{fillx} \leftarrow \text{fill-tm } k \text{ } xs;$
 $\text{filly} \leftarrow \text{fill-tm } k \text{ } ys;$
 $\text{karatsuba-on-power-of-2-length-tm } k \text{ } \text{fillx } \text{filly}$
 $\}$

lemma $\text{val-karatsuba-mul-nat-tm}[\text{simp}, \text{val-simp}]$: $\text{val } (\text{karatsuba-mul-nat-tm } xs \text{ } ys)$
 $= \text{karatsuba-mul-nat } xs \text{ } ys$
 ⟨proof⟩

definition $\text{time-karatsuba-mul-nat-bound}$ **where**
 $\text{time-karatsuba-mul-nat-bound } m = 53 + 218 * (\text{next-power-of-2 } m) + h (\text{next-power-of-2 } m)$

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

theorem $\text{time-karatsuba-mul-nat-tm-le}$:
assumes $\text{max } (\text{length } xs) (\text{length } ys) = m$
shows $\text{time}(\text{karatsuba-mul-nat-tm } xs \text{ } ys) \leq \text{time-karatsuba-mul-nat-bound } m$
 ⟨proof⟩

theorem $\text{time-karatsuba-mul-nat-bound-bigo}$: $\text{time-karatsuba-mul-nat-bound} \in O(\lambda m. m \text{ powr } \log 2 \ 3)$
 ⟨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* \Rightarrow *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* \neq []
 \langle *proof* \rangle

lemma *from-numeral-truncated*: *truncated* (*from-numeral* *x*)
 \langle *proof* \rangle

lemma *to-nat-from-numeral-neq-zero*: *to-nat* (*from-numeral* *x*) \neq 0
 \langle *proof* \rangle

fun *to-numeral-of-truncated* :: *nat-lsbf* \Rightarrow *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*
 \langle *proof* \rangle

lemma *nat-of-num-to-numeral-of-truncated*:
 assumes *truncated* *xs*
 assumes *xs* \neq []
 shows *nat-of-num* (*to-numeral-of-truncated* *xs*) = *to-nat* *xs*
 \langle *proof* \rangle

definition *to-numeral* :: *nat-lsbf* \Rightarrow *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*
 \langle *proof* \rangle

lemma *nat-of-num-to-numeral*:
assumes $to\text{-}nat\ x \neq 0$
shows $nat\text{-}of\text{-}num\ (to\text{-}numeral\ x) = to\text{-}nat\ x$
 $\langle proof \rangle$

lemma *l0*:
assumes *truncated* x
shows $to\text{-}numeral\text{-}of\text{-}truncated\ x = num\text{-}of\text{-}nat\ (to\text{-}nat\ x)$
 $\langle proof \rangle$

lemma *l1*: $to\text{-}numeral\ x = num\text{-}of\text{-}nat\ (to\text{-}nat\ x)$
 $\langle proof \rangle$

lemma *l2*: $to\text{-}nat\ (from\text{-}numeral\ x) = nat\text{-}of\text{-}num\ x$
 $\langle proof \rangle$

lemma[*code*]:
 $(x::num) * y = to\text{-}numeral\ (karatsuba\text{-}mul\text{-}nat\ (from\text{-}numeral\ x)\ (from\text{-}numeral\ y))$
 $\langle proof \rangle$

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>.