The Karatsuba Square Root Algorithm

Manuel Eberl

October 9, 2024

Abstract

This formalisation provides an executable version of Zimmerman's "Karatsuba Square Root" algorithm, which, given an integer $n \geq 0$, computes the integer square root $\lfloor \sqrt{n} \rfloor$ and the remainder $n - \lfloor \sqrt{n} \rfloor^2$. This is the algorithm used by the GNU Multiple Precision Arithmetic Library (GMP).

Similarly to Karatsuba multiplication, the algorithm is a divide-and-conquer algorithm that works by repeatedly splitting the input number n into four parts and recursively calls itself once on an input with roughly half as many bits as n, leading to a total running time of $O(M(n))$ (where $M(n)$ is the time required to multiply two n-bit numbers). This is significantly faster than the standard Heron method for large numbers (i.e. more than roughly 1000 bits).

As a simple application to interval arithmetic, an executable floating-point interval extension of the square-root operation is provided. For high-precision computations this is considerably more efficient than the interval extension method in the Isabelle distribution.

Contents

1 The function $\lceil \log_2 n \rceil$

```
theory Ceil_Log2
  imports Complex_Main
begin

definition ceillog2 :: "nat ⇒ nat" where
  "ceillog2 n = (if n = 0 then 0 else nat ⌈log 2 (real n)⌉)"

lemma ceillog2_0 [simp]: "ceillog2 0 = 0"
  and ceillog2_Suc_0 [simp]: "ceillog2 (Suc 0) = 0"
  and ceillog2_2 [simp]: "ceillog2 2 = 1"
  by (auto simp: ceillog2_def)

lemma ceillog2_2_power [simp]: "ceillog2 (2 ^ n) = n"
  by (auto simp: ceillog2_def)

lemma ceillog2_ge_log:
  assumes "n > 0"
  shows   "real (ceillog2 n) ≥ log 2 (real n)"
proof -
  have "real_of_int ⌈log 2 (real n)⌉ ≥ log 2 (real n)"
    by linarith
  thus ?thesis
    using assms unfolding ceillog2_def by auto
qed

lemma ceillog2_less_log:
  assumes "n > 0"
  shows   "real (ceillog2 n) < log 2 (real n) + 1"
proof -
  have "real_of_int ⌈log 2 (real n)⌉ < log 2 (real n) + 1"
    by linarith
  thus ?thesis
    using assms unfolding ceillog2_def by auto
qed

lemma ceillog2_le_iff:
  assumes "n > 0"
  shows   "ceillog2 n ≤ l ⟷ n ≤ 2 ^ l"
proof -
  have "ceillog2 n ≤ l ⟷ real n ≤ 2 ^ l"
    unfolding ceillog2_def using assms by (auto simp: log_le_iff powr_realpow)
  also have "2 ^ l = real (2 ^ l)"
    by simp
  also have "real n ≤ real (2 ^ l) ⟷ n ≤ 2 ^ l"
    by linarith
  finally show ?thesis .
qed
```

```
lemma ceillog2_ge_iff:
  assumes "n > 0"
  shows   "ceillog2 n ≥ l ⟷ 2 ^ l < 2 * n"
proof -
  have "-1 < (0 :: real)"
    by auto
  also have "... ≤ log 2 (real n)"
    using assms by auto
  finally have "ceillog2 n ≥ l ⟷ real l - 1 < log 2 (real n)"
    unfolding ceillog2_def using assms by (auto simp: le_nat_iff le_ceiling_iff)
  also have "... ⟷ real l < log 2 (real (2 * n))"
    using assms by (auto simp: log_mult)
  also have "... ⟷ 2 ^ l < real (2 * n)"
    using assms by (subst less_log_iff) (auto simp: powr_realpow)
  also have "2 ^ l = real (2 ^ l)"
    by simp
  also have "real (2 ^ l) < real (2 * n) ⟷ 2 ^ l < 2 * n"
    by linarith
  finally show ?thesis .
qed

lemma le_two_power_ceillog2: "n ≤ 2 ^ ceillog2 n"
proof (cases "n = 0")
  case False
  thus ?thesis
    using ceillog2_le_iff[of n "ceillog2 n"] by simp
qed auto

lemma two_power_ceillog2_gt:
  assumes "n > 0"
  shows   "2 * n > 2 ^ ceillog2 n"
  using ceillog2_ge_iff[of n "ceillog2 n"] assms by simp

lemma ceillog2_eqI:
  assumes "n ≤ 2 ^ l" "2 ^ l < 2 * n"
  shows   "ceillog2 n = l"
proof -
  from assms have "n > 0"
    by (intro Nat.gr0I) auto
  thus ?thesis using assms
    by (intro antisym[of _ l])
       (auto simp: ceillog2_le_iff ceillog2_ge_iff)
qed

lemma ceillog2_rec_even:
  assumes "k > 0"
  shows   "ceillog2 (2 * k) = Suc (ceillog2 k)"
  by (rule ceillog2_eqI) (auto simp: le_two_power_ceillog2 two_power_ceillog2_gt
```

```
assms)

lemma ceillog2_mono:
  assumes "m ≤ n"
  shows   "ceillog2 m ≤ ceillog2 n"
proof (cases "m = 0")
  case False
  have "⌈log 2 (real m)⌉ ≤ ⌈log 2 (real n)⌉"
    by (intro ceiling_mono) (use False assms in auto)
  hence "nat ⌈log 2 (real m)⌉ ≤ nat ⌈log 2 (real n)⌉"
    by linarith
  thus ?thesis using False assms
    unfolding ceillog2_def by simp
qed auto

lemma ceillog2_rec_odd:
  assumes "k > 0"
  shows   "ceillog2 (Suc (2 * k)) = Suc (ceillog2 (Suc k))"
proof -
  have "ceillog2 (2 * k + 2) ≤ ceillog2 (2 * k + 1)"
    using assms
    by (smt (verit, ccfv_threshold) One_nat_def add_diff_cancel_right'
add_gr_0 ceillog2_0
             ceillog2_le_iff dvd_triv_left le_less_Suc_eq le_two_power_ceillog2
linorder_not_less
             mult_pos_pos nat_arith.suc1 nat_power_eq_Suc_0_iff not_less_eq_eq
numeral_2_eq_2
             semiring_parity_class.even_mask_iff)
  moreover have "ceillog2 (2 * k + 2) ≥ ceillog2 (2 * k + 1)"
    by (rule ceillog2_mono) auto
  ultimately have "ceillog2 (2 * k + 2) = ceillog2 (2 * k + 1)"
    by (rule antisym)
  also have "2 * k + 2 = 2 * Suc k"
    by simp
  also have "ceillog2 (2 * Suc k) = Suc (ceillog2 (Suc k))"
    by (rule ceillog2_rec_even) auto
  finally show ?thesis
    by simp
qed


lemma ceillog2_rec:
  "ceillog2 n = (if n ≤ 1 then 0 else 1 + ceillog2 ((n + 1) div 2))"
proof (cases "n ≤ 1")
  case True
  thus ?thesis
    by (cases n) auto
next
  case False
```

```
  thus ?thesis
    by (cases "even n") (auto elim!: evenE oddE simp: ceillog2_rec_even
ceillog2_rec_odd)
qed

lemmas [code] = ceillog2_rec

end
```

1.1 Auxiliary material

```
theory Karatsuba_Sqrt_Library
imports
  Complex_Main
  "HOL-Library.Discrete"
  "HOL-Library.Log_Nat"
begin
```

1.2 Efficient simultaneous computation of *div* and *mod*

```
definition divmod_int :: "int ⇒ int ⇒ int × int" where
  "divmod_int a b = (a div b, a mod b)"

lemma divmod_int_code [code]:
  "divmod_int a b =
     (case divmod_integer (integer_of_int a) (integer_of_int b) of
       (q, r) ⇒ (int_of_integer q, int_of_integer r))"
  by (simp add: divmod_int_def divmod_integer_def)
```

1.2.1 Missing lemmas about *bitlen*

```
lemma drop_bit_eq_0_iff_nat:
  "drop_bit k (n :: nat) = 0 ⟷ bitlen n ≤ k"
  by (auto simp: drop_bit_eq_div div_eq_0_iff less_power_nat_iff_bitlen)

lemma drop_bit_eq_0_iff_int:
  assumes "n ≥ 0"
  shows   "drop_bit k (n :: int) = 0 ⟷ bitlen n ≤ k"
  by (metis assms drop_bit_eq_0_iff_nat drop_bit_nat_eq drop_bit_of_nat
nat_0_le nat_zero_as_int of_nat_0)

lemma drop_bit_bitlen_minus_1:
  assumes "n > 0"
  shows   "drop_bit (nat (bitlen n - 1)) n = 1"
proof -
  define s where "s = nat (bitlen n - 1)"
  have "bitlen n > 0"
    using assms by (simp add: bitlen_eq_zero_iff bitlen_nonneg order_less_le)
  have "drop_bit s n ≤ drop_bit s (mask (s+1))"
    unfolding drop_bit_eq_div mask_eq_exp_minus_1
```

```
    using ‹bitlen n > 0› bitlen_bounds[of n] assms
    by (intro zdiv_mono1)
       (auto simp: s_def nat_diff_distrib simp del: power_Suc)
  also have "drop_bit s (mask (s + 1) :: int) = 1"
    by (simp add: drop_bit_mask_eq)
  finally have "drop_bit s n ≤ 1" .
  moreover have "drop_bit s n ≠ 0"
    using assms ‹bitlen n > 0›
    by (subst drop_bit_eq_0_iff_int) (auto simp: s_def)
  moreover have "drop_bit s n ≥ 0"
    using assms by auto
  ultimately show "drop_bit s n = 1"
    by linarith
qed

lemma bitlen_pos: "n > 0 ⟹ bitlen n > 0"
  using bitlen_def bitlen_eq_zero_iff linorder_not_less by auto

lemma bit_bitlen_minus_1:
  assumes "n > 0"
  shows   "bit n (nat (bitlen n - 1))"
  using drop_bit_bitlen_minus_1[OF assms]
  by (simp add: bit_iff_and_drop_bit_eq_1)

lemma not_bit_ge_bitlen:
  assumes "int k ≥ bitlen n" "n ≥ 0"
  shows   "¬bit n k"
proof
  assume "bit n k"
  hence "odd (n div 2 ^ k)"
    by (auto simp: bit_iff_odd)
  hence "n ≥ 2 ^ k"
    using assms(2) linorder_not_le by fastforce
  hence "int k < bitlen n"
    by (metis bitlen_le_iff_power linorder_not_less nat_int)
  thus False
    using assms by auto
qed

lemma bitlen_eqI:
  assumes "bit n (nat k - 1)" "⋀i. int i ≥ k ⟹ ¬bit n i" "k > 0"
"n ≥ 0"
  shows   "bitlen n = k"
proof -
  from assms(1) have "n ≠ 0"
    by auto
  with ‹n ≥ 0› have "n > 0"
    by auto
  show ?thesis
```

```
  proof (cases "bitlen n" k rule: linorder_cases)
    assume "bitlen n > k"
    hence False
      using assms(2)[of "nat (bitlen n - 1)"] bit_bitlen_minus_1[of n]
‹n > 0›
      by (auto split: if_splits simp: bitlen_pos)
    thus ?thesis ..
  next
    assume "bitlen n < k"
    hence False
      using assms(1) ‹k > 0› not_bit_ge_bitlen[of n "nat k - 1"] ‹n >
0›
      by (auto simp: of_nat_diff)
    thus ?thesis ..
  qed auto
qed

lemma bitlen_drop_bit:
  assumes "n ≥ 0"
  shows   "bitlen (drop_bit k n) = max 0 (bitlen n - k)"
proof (cases "bitlen n > k")
  case False
  hence "drop_bit k n = 0"
    using assms by (subst drop_bit_eq_0_iff_int) auto
  thus ?thesis using False
    by simp
next
  case True
  hence "n ≠ 0"
    by auto
  with assms have "n > 0"
    by auto
  show ?thesis
  proof (rule bitlen_eqI)
    show "bit (drop_bit k n) (nat (max 0 (bitlen n - int k)) - 1)"
      using True bit_bitlen_minus_1[of n] ‹n > 0›
      by (auto simp: bit_drop_bit_eq nat_diff_distrib)
  next
    fix i :: nat
    assume "max 0 (bitlen n - int k) ≤ int i"
    hence "int (i + k) ≥ bitlen n"
      using True by simp
    thus "¬ bit (drop_bit k n) i"
      using ‹n > 0› by (auto simp: bit_drop_bit_eq not_bit_ge_bitlen)
  qed (use True ‹n > 0› in auto)
qed
```

1.2.2 Missing lemmas about *Discrete.sqrt*

```
lemma Discrete_sqrt_lessI:
  assumes "x < y ^ 2"
  shows   "Discrete.sqrt x < y"
  using assms Discrete.le_sqrt_iff linorder_not_less by blast

lemma Discrete_sqrt_conv_floor_sqrt:
  "Discrete.sqrt n = nat (floor (sqrt n))"
proof (rule Discrete.sqrt_unique)
  have "real (nat (floor (sqrt n)) ^ 2) = real_of_int ⌊sqrt (real n)⌋
^ 2"
    by simp
  also have "... ≤ sqrt (real n) ^ 2"
    by (intro power_mono) auto
  also have "... = real n"
    by simp
  finally show "nat (floor (sqrt n)) ^ 2 ≤ n"
    by linarith
next
  have "sqrt (real n) ^ 2 < (real_of_int ⌊sqrt (real n)⌋ + 1) ^ 2"
    by (rule power_strict_mono) auto
  hence "real n < (real_of_int ⌊sqrt (real n)⌋ + 1) ^ 2"
    by simp
  also have "... = real ((Suc (nat (floor (sqrt n)))) ^ 2)"
    by simp
  finally show "n < Suc (nat (floor (sqrt n))) ^ 2"
    by linarith
qed
```

1.3 Miscellaneous

```
lemma Let_cong:
  assumes "a = c" "⋀x. x = a ⟹ b x = d x"
  shows   "Let a b = Let c d"
  unfolding Let_def using assms by simp

lemma case_prod_cong:
  assumes "a = b" "⋀x y. a = (x, y) ⟹ f x y = g x y"
  shows   "(case a of (x, y) ⇒ f x y) = (case b of (x, y) ⇒ g x y)"
  using assms by (auto simp: case_prod_unfold)

end
theory Karatsuba_Sqrt
imports
  Complex_Main
  "HOL-Library.Discrete"
  "HOL-Library.Log_Nat"
  Ceil_Log2
  Karatsuba_Sqrt_Library
```

```
begin
```

1.4 Definition of an integer square root with remainder

```
definition sqrt_rem :: "nat ⇒ nat" where
  "sqrt_rem n = n - Discrete.sqrt n ^ 2"

lemma sqrt_rem_upper_bound: "sqrt_rem n ≤ 2 * Discrete.sqrt n"
proof -
  define s where "s = Discrete.sqrt n"
  have "n < (s + 1) ^ 2"
    unfolding s_def using Suc_sqrt_power2_gt[of n] by auto
  hence "n + 1 ≤ (s + 1) ^ 2"
    by linarith
  hence "n ≤ s ^ 2 + 2 * s"
    by (simp add: algebra_simps power2_eq_square)
  thus ?thesis
    unfolding s_def sqrt_rem_def by linarith
qed

lemma of_nat_sqrt_rem:
  "(of_nat (sqrt_rem n) :: 'a :: ring_1) = of_nat n - of_nat (Discrete.sqrt
n) ^ 2"
  by (simp add: sqrt_rem_def of_nat_diff)

definition sqrt_rem' where "sqrt_rem' n = (Discrete.sqrt n, sqrt_rem n)"

lemma Discrete_sqrt_code [code]: "Discrete.sqrt n = fst (sqrt_rem' n)"
  by (simp add: sqrt_rem'_def)

lemma sqrt_rem_code [code]: "sqrt_rem n = snd (sqrt_rem' n)"
  by (simp add: sqrt_rem'_def)
```

1.5 Heron's method

The method used here is a variant of Heron's method, which is itself essentially Newton's method specialised to square roots. This is already in the AFP under the name "Babylonian method". However, that entry derives a more general version for n-th roots and lacks some flexibility that is useful for us here, so we instead derive a simple version for the square root directly. We will use this version in the base case of the algorithm.

The starting value must be bigger than $\lfloor\sqrt{n}\rfloor$. We simply use $2^{\lceil\frac{1}{2}\log_2 n\rceil}$, which is easy to compute and fairly close to $\sqrt{n}$ already so that the Newton iterations converge very quickly.

```
context
  fixes n :: nat
begin
```

```
function sqrt_rem_aux :: "nat ⇒ nat × nat" where
  "sqrt_rem_aux x =
     (if x² ≤ n then (x, n - x²) else sqrt_rem_aux ((n div x + x) div
2))"
  by auto
termination proof (relation "Wellfounded.measure id")
  fix x assume x: "¬(x² ≤ n)"
  have "n div x * x ≤ n"
    by simp
  also from x have "n < x * x"
    by (simp add: power2_eq_square)
  finally have "n div x < x"
    using x by simp
  hence "(n div x + x) div 2 < x"
    by (subst div_less_iff_less_mult) auto
  thus "((n div x + x) div 2, x) ∈ measure id"
    by simp
qed auto

lemmas [simp del] = sqrt_rem_aux.simps

lemma sqrt_rem_aux_code [code]:
  "sqrt_rem_aux x = (
    let x2 = x*x; r = int n - int x2
    in if r ≥ 0 then (x, nat r) else sqrt_rem_aux (drop_bit 1 (n div
x + x)))"
  by (subst sqrt_rem_aux.simps)
     (auto simp: Let_def case_prod_unfold power2_eq_square nat_diff_distrib
drop_bit_eq_div
           simp flip: of_nat_mult)

lemma sqrt_rem_aux_decompose: "fst (sqrt_rem_aux x) ^ 2 + snd (sqrt_rem_aux
x) = n"
  by (induction x rule: sqrt_rem_aux.induct; subst (1 2) sqrt_rem_aux.simps)
auto

lemma sqrt_rem_aux_correct:
  assumes "x ≥ Discrete.sqrt n"
  shows   "fst (sqrt_rem_aux x) = Discrete.sqrt n"
  using assms
proof (induction x rule: sqrt_rem_aux.induct)
  case (1 x)
  show ?case
  proof (cases "x ^ 2 ≤ n")
    case True
    from True have "Discrete.sqrt n ≥ x"
      by (simp add: le_sqrtI)
    with "1.prems" show ?thesis using True
      by (subst sqrt_rem_aux.simps) auto
```

```
  next
    case False
    hence "x > 0"
      by (auto intro!: Nat.gr0I)
    have "0 < (x ^ 2 - n) ^ 2 / (4 * x ^ 2)"
      using ‹x > 0› False by (intro divide_pos_pos) auto
    also have "(x ^ 2 - n) ^ 2 / (4 * x ^ 2) = ((n / x + x) / 2) ^ 2 -
n"
      using ‹x > 0› False by (simp add: field_simps power2_eq_square
of_nat_diff)
    finally have "n < ((n / x + x) / 2) ^ 2"
      by simp
    hence "sqrt n ^ 2 < ((n / x + x) / 2) ^ 2"
      by simp
    hence "sqrt n < (n / x + x) / 2"
      by (rule power_less_imp_less_base) auto
    hence "nat (floor (sqrt n)) ≤ nat (floor ((n / x + x) / 2))"
      by linarith
    also have "nat (floor (sqrt n)) = Discrete.sqrt n"
      by (simp add: Discrete_sqrt_conv_floor_sqrt)
    also have "floor ((n / x + x) / 2) = (n div x + x) div 2"
      using floor_divide_real_eq_div[of 2 "n / x + x"] by (simp add: floor_divide_of_nat_eq
    finally have "Discrete.sqrt n ≤ (n div x + x) div 2"
      by simp
    from "1.IH"[OF False this] show ?thesis
      by (subst sqrt_rem_aux.simps) (use False in auto)
  qed
qed

lemma sqrt_rem_aux_correct':
  assumes "x ≥ Discrete.sqrt n"
  shows   "sqrt_rem_aux x = sqrt_rem' n"
  using sqrt_rem_aux_correct[OF assms] sqrt_rem_aux_decompose[of x]
  by (simp add: sqrt_rem'_def prod_eq_iff sqrt_rem_def)

definition sqrt_rem'_heron :: "nat × nat" where
  "sqrt_rem'_heron = sqrt_rem_aux (push_bit ((ceillog2 n + 1) div 2) 1)"

lemma sqrt_rem'_heron_correct:
  "sqrt_rem'_heron = sqrt_rem' n"
proof (cases "n = 0")
  case True
  show ?thesis unfolding sqrt_rem'_heron_def
    by (rule sqrt_rem_aux_correct') (auto simp: True)
next
  case False
  hence n: "n > 0"
    by auto
  show ?thesis unfolding sqrt_rem'_heron_def
```

```
  proof (rule sqrt_rem_aux_correct')
    have "real (Discrete.sqrt n) ≤ sqrt n"
      by (simp add: Discrete_sqrt_conv_floor_sqrt)
    also have "... = 2 powr log 2 (sqrt n)"
      using n by simp
    also have "log 2 (sqrt n) = log 2 n / 2"
      using n by (simp add: log_def ln_sqrt)
    also have "(2::real) powr ... ≤ 2 powr ((ceillog2 n + 1) div 2)"
    proof (intro powr_mono)
      have "log 2 (real n) ≤ real (ceillog2 n)"
        by (simp add: ceillog2_ge_log n)
      also have "... / 2 ≤ (ceillog2 n + 1) div 2"
        by linarith
      finally show "log 2 n / 2 ≤ (ceillog2 n + 1) div 2"
        by - simp_all
    qed auto
    also have "... = real (2 ^ ((ceillog2 n + 1) div 2))"
      by (subst powr_realpow) auto
    also have "2 ^ ((ceillog2 n + 1) div 2) = push_bit ((ceillog2 n +
1) div 2) 1"
      by (simp add: push_bit_eq_mult)
    finally show "Discrete.sqrt n ≤ push_bit ((ceillog2 n + 1) div 2)
1"
      by linarith
  qed
qed

end

lemmas [code] = sqrt_rem'_heron_correct [symmetric]
```

1.6 Main algorithm

1.6.1 Single step

```
definition splice_bit where
  "splice_bit i n x = take_bit n (drop_bit i x)"

lemma of_nat_splice_bit:
  "of_nat (splice_bit i n x) =
     splice_bit i n (of_nat x :: 'a :: linordered_euclidean_semiring_bit_operations)"
  by (simp add: splice_bit_def of_nat_take_bit of_nat_drop_bit)

definition karatsuba_sqrt_step where
  "karatsuba_sqrt_step a32 a1 a0 b =
     (let (s, r) = sqrt_rem' a32;
          (q, u) = ((r * b + a1) div (2 * s), (r * b + a1) mod (2 * s));
          s' = int (s * b + q);
          r' = int (u * b + a0) - int (q ^ 2)
      in  if r' ≥ 0 then (s', r') else (s' - 1, r' + 2 * s' - 1))"
```

```
definition karatsuba_sqrt_step' :: "nat ⇒ nat ⇒ int × int" where
  "karatsuba_sqrt_step' n k =
     (let (s, r) = map_prod int int (sqrt_rem' (drop_bit (2*k) n));
          (q, u) = divmod_int (push_bit k r + splice_bit k k n) (push_bit
1 s);
          s' = push_bit k s + q;
          r' = push_bit k u + take_bit k n - q ^ 2
      in  if r' ≥ 0 then (s', r') else (s' - 1, r' + push_bit 1 s' - 1))"
```

Note that unlike Zimmerman, we do not have any upper bound on a_3 since this bound turned out to be unnecessary for the correctness of the algorithm. As long as b^4 is not much smaller than n, there is no efficiency problem either, since the step will still strip away about half of the bits of n.

The advantage of this is that we do not have to do the "normalisation" done by Zimmerman to ensure that at least one of the two most significant bits of a_3 be set.

```
lemma karatsuba_sqrt_step_correct:
  fixes a32 a1 a0 :: nat
  assumes "a1 < b" "a0 < b" "4 * a32 ≥ b ^ 2" "even b"
  defines "n ≡ a32 * b ^ 2 + a1 * b + a0"
  shows   "karatsuba_sqrt_step a32 a1 a0 b =
             map_prod of_nat of_nat (sqrt_rem' n)"
proof -
  define s where "s = Discrete.sqrt a32"
  define r where "r = sqrt_rem a32"
  define q where "q = (r * b + a1) div (2 * s)"
  define u where "u = (r * b + a1) mod (2 * s)"
  define s' where "s' = int (s * b + q)"
  define r' where "r' = int (u * b + a0) - int (q ^ 2)"
  define s'' where "s'' = (if r' ≥ 0 then s' else s' - 1)"
  define r'' where "r'' = (if r' ≥ 0 then r' else r'  + 2 * s' - 1)"

  from assms have "b > 0"
    by auto
  have "s > 0"
    using assms by (auto simp: s_def intro!: Nat.gr0I)

  have "b ≤ 2 * s"
  proof -
    have "4 * (b div 2) ^ 2 = b ^ 2"
      using ‹even b› by (auto elim!: evenE simp: power2_eq_square)
    also have "... ≤ 4 * a32"
      by fact
    finally have "b div 2 ≤ s"
      unfolding s_def by (subst Discrete.le_sqrt_iff) auto
    thus "b ≤ 2 * s"
      using ‹even b› by (elim evenE) auto
```

```
  qed

  have s'_r': "int n = s' ^ 2 + r'"
  proof -
    have *: "int a1 = int q * (2 * int s) + int u - int r * int b"
      using arg_cong[OF div_mod_decomp[of "r * b + a1" "2 * s"], of int,
folded q_def u_def]
      unfolding of_nat_add of_nat_mult by linarith
    have "int n = (int s ^ 2 + int r) * int b ^ 2 + int a1 * int b + int
a0"
      by (simp add: n_def s_def r_def of_nat_sqrt_rem algebra_simps power_numeral_reduce)
    also have "... = s' ^ 2 + r'"
      by (simp add: power2_eq_square algebra_simps * r'_def s'_def)
    finally show "int n = s' ^ 2 + r'" .
  qed
  hence s''_r'': "int n = s'' ^ 2 + r''"
    by (simp add: s''_def r''_def power2_eq_square algebra_simps)

  have "int n < (s' + 1) ^ 2"
  proof -
    define t where "t = Discrete.sqrt n - s * b"
    have "s ^ 2 * b ^ 2 ≤ a32 * b ^ 2"
      unfolding s_def by (intro mult_right_mono Discrete.sqrt_power2_le)
auto
    also have "... ≤ n"
      by (simp add: n_def)
    finally have "(s * b) ^ 2 ≤ n"
      by (simp add: power_mult_distrib)
    hence "Discrete.sqrt n ≥ s * b"
      by (simp add: le_sqrt_iff)
    hence sqrt_n_eq: "Discrete.sqrt n = s * b + t"
      unfolding t_def by simp

    have "int (2 * s * t * b) = 2 * int s * int b * int t"
      by simp
    also have "2 * int s * int b * int t ≤ 2 * int s * int b * int t
+ int t ^ 2"
      by simp
    also have "... = int ((s * b + t) ^ 2) - (int s * int b) ^ 2"
      unfolding of_nat_power of_nat_mult of_nat_add by algebra
    also have "s * b + t = Discrete.sqrt n"
      by (simp add: sqrt_n_eq)
    also have "Discrete.sqrt n ^ 2 ≤ n"
      by simp
    also have "n - (int s * int b) ^ 2 = int (a1 * b + a0) + (int a32
- int s ^ 2) * int b ^ 2"
      unfolding n_def of_nat_add of_nat_mult of_nat_power by algebra
    also have "int a32 - int s ^ 2 = int r"
      unfolding r_def by (simp add: of_nat_sqrt_rem s_def)
```

```
    also have "a0 < b"
      by fact
    also have "int (a1 * b + b) + int r * (int b)^2 = int ((a1 + 1 + r
* b) * b)"
      by (simp add: algebra_simps power2_eq_square)
    finally have "2 * s * t * b < (a1 + 1 + r * b) * b"
      unfolding of_nat_less_iff by - simp_all
    hence "2 * s * t < a1 + 1 + r * b"
      using ‹b > 0› mult_less_cancel2 by blast
    hence "2 * s * t ≤ r * b + a1"
      by linarith
    hence "t ≤ q"
      unfolding q_def using ‹s > 0›
      by (subst less_eq_div_iff_mult_less_eq) (auto simp: algebra_simps)
    with sqrt_n_eq have *: "Discrete.sqrt n ≤ s * b + q"
      by simp

    have "n < (Discrete.sqrt n + 1) ^ 2"
        using Suc_sqrt_power2_gt[of n] by simp
    also have "... ≤ (s * b + q + 1) ^ 2"
      by (intro power_mono add_mono *) auto
    finally have "int n < int ((s * b + q + 1) ^ 2)"
      by linarith
    thus "int n < (s' + 1) ^ 2"
      by (simp add: algebra_simps s'_def)
  qed

  have "q ≤ r"
  proof -
    have "q ≤ (r * b + a1) div b"
      unfolding q_def using ‹b ≤ 2 * s› ‹b > 0› by (intro div_le_mono2)
    also have "... = r"
      using ‹b > 0› assms by simp
    finally show "q ≤ r" .
  qed

  have "int (q ^ 2) < 2 * s'"
  proof (cases "q = 0")
    case False
    have "q ^ 2 ≤ 2 * s * b"
      unfolding power2_eq_square
    proof (intro mult_mono)
      show "q ≤ 2 * s"
        using ‹q ≤ r› sqrt_rem_upper_bound[of a32] unfolding r_def s_def
by linarith
    next
      show "q ≤ b"
      proof -
        have "r ≤ 2 * s"
```

```
          using ‹q ≤ r› unfolding r_def s_def using sqrt_rem_upper_bound[of
a32] by linarith
        hence "q ≤ (2 * s * b + a1) div (2 * s)"
          unfolding q_def by (intro div_le_mono add_mono mult_right_mono)
auto
        also have "... = b + a1 div (2 * s)"
          using assms ‹s > 0› by simp
        also have "a1 div (2 * s) = 0"
          using ‹b ≤ 2 * s› ‹a1 < b› by auto
        finally show "q ≤ b" by simp
      qed
    qed auto
    also have "2 * s * b < 2 * (s * b + q)"
      using ‹q ≠ 0› by (simp add: algebra_simps)
    also have "int ... = 2 * s'"
      by (simp add: s'_def)
    finally show ?thesis by - simp_all
  qed (use ‹s > 0› ‹b > 0› in ‹auto simp: s'_def›)

  have "r'' ≥ 0"
  proof (cases "r' ≥ 0")
    case False
    have "r' + 2 * s' > 0"
      unfolding r'_def using ‹int (q ^ 2) < 2 * s'› by linarith
    thus ?thesis
      unfolding r''_def by auto
  qed (auto simp: r''_def)

  have "s'' ≥ 0"
    using ‹0 ≤ r''› unfolding r''_def s''_def s'_def by auto

  have "s'' ^ 2 ≤ int n"
  proof -
    have "s'' ^ 2 = int n - r''"
      using s''_r'' by simp
    also have "... ≤ int n"
      using ‹r'' ≥ 0› by simp
    finally show "s'' ^ 2 ≤ n" .
  qed

  have "Discrete.sqrt n = nat s''"
  proof (rule Discrete.sqrt_unique)
    show "nat s'' ^ 2 ≤ n"
      using ‹s'' ^ 2 ≤ int n›
      by (metis nat_eq_iff2 of_nat_le_of_nat_power_cancel_iff zero_eq_power2
zero_le)
  next
    have "int n < (s'' + 1) ^ 2"
    proof (cases "r' ≥ 0")
```

```
      case True
      show ?thesis
        using True ‹int n < (s' + 1) ^ 2› by (simp add: s''_def)
    next
      case False
      have "int n < s' ^ 2"
        using False s'_r' by auto
      thus ?thesis using False by (simp add: s''_def)
    qed
    also have "(s'' + 1) ^ 2 = int (Suc (nat s'') ^ 2)"
      using ‹s'' ≥ 0› by simp
    finally show "n < Suc (nat s'') ^ 2"
      by linarith
  qed
  moreover from this have "int (sqrt_rem n) = r''"
    using s''_r'' ‹s'' ≥ 0› unfolding of_nat_sqrt_rem by auto
  hence "sqrt_rem n = nat r''"
    by linarith
  moreover have "karatsuba_sqrt_step a32 a1 a0 b = (s'', r'')"
    unfolding karatsuba_sqrt_step_def sqrt_rem'_def n_def s''_def r''_def
r'_def s'_def
              r_def s_def u_def q_def Let_def case_prod_unfold
    by (simp add: divmod_def)
  ultimately show ?thesis using ‹r'' ≥ 0› ‹s'' ≥ 0›
    by (simp add: n_def sqrt_rem'_def)
qed

lemma karatsuba_sqrt_step'_correct:
  fixes k n :: nat
  assumes k: "k > 0" and bitlen: "int k ≤ (bitlen n + 1) div 4"
  defines "a32 ≡ drop_bit (2*k) n"
  defines "a1 ≡ splice_bit k k n"
  defines "a0 ≡ take_bit k n"
  shows   "karatsuba_sqrt_step' n k = map_prod int int (sqrt_rem' n)"
proof -
  define n' where "n' = drop_bit (2*k) n"
  have less: "a0 < 2 ^ k" "a1 < 2 ^ k"
    by (auto simp: a0_def a1_def splice_bit_def)
  have mod_less: "x mod y < 2 ^ k" if "y ≤ 2 ^ k" "y > 0" for x y :: int
  proof -
    have "x mod y < y"
      using that by (intro pos_mod_bound) auto
    also have "... ≤ 2 ^ k"
      using that by simp
    finally show ?thesis .
  qed

  have n_eq: "n = a32 * 2 ^ (2 * k) + a1 * 2 ^ k + a0"
  proof -
```

```
    have "n = push_bit (2*k) (drop_bit (2*k) n) + take_bit (2*k) n"
      by (simp add: bits_ident)
    also have "take_bit (2*k) n = take_bit (2*k) (push_bit k (drop_bit
k n) + take_bit k n)"
      by (simp add: bits_ident)
    also have "... = push_bit k (splice_bit k k n) + take_bit k n"
      by (subst bit_eq_iff)
         (auto simp: bit_take_bit_iff bit_push_bit_iff bit_disjunctive_add_iff
splice_bit_def)
    also have "push_bit (2 * k) (drop_bit (2 * k) n) + (push_bit k (splice_bit
k k n) + take_bit k n) =
                drop_bit (2 * k) n * 2 ^ (2 * k) + splice_bit k k n *
2 ^ k + take_bit k n"
      by (simp add: push_bit_eq_mult)
    finally show ?thesis by (simp add: a32_def a1_def a0_def)
  qed

  have "a32 > 0"
  proof (rule Nat.gr0I)
    assume "a32 = 0"
    hence "bitlen (int n) ≤ 2 * int k"
      by (simp add: a32_def drop_bit_eq_0_iff_nat)
    with bitlen and ‹k > 0› show False
      by linarith
  qed

  have *: "(2 ^ k) ^ 2 ≤ 4 * a32"
  proof -
    have "int ((2 ^ k) ^ 2) = (2 ^ (2 * k) :: int)"
      by (simp add: power_mult add: mult_ac)
    also have "... ≤ int (4 * a32) ⟷ bitlen (int a32 * 2 ^ 2) ≥ 2 *
k + 1"
      by (subst bitlen_ge_iff_power) (auto simp: nat_add_distrib nat_mult_distrib)
    also have "bitlen (int a32 * 2 ^ 2) = bitlen a32 + 2"
      using ‹a32 > 0› by (subst bitlen_pow2) auto
    also have "bitlen (int n) ≥ 2 * int k"
      using assms(1,2) by linarith
    hence "bitlen (int a32) = bitlen (int n) - 2 * int k"
      by (simp add: a32_def of_nat_drop_bit bitlen_drop_bit)
    also have "(int (2 * k + 1) ≤ bitlen (int n) - 2 * int k + 2) ⟷
True"
      using assms(2) by simp
    finally show ?thesis
      unfolding of_nat_le_iff by simp
  qed

  have "n = a32 * 2 ^ (2 * k) + a1 * 2 ^ k + a0"
    by (simp add: n_eq)
  also have "map_prod int int (sqrt_rem' ...) = karatsuba_sqrt_step a32
```

```
a1 a0 (2^k)"
    by (subst karatsuba_sqrt_step_correct)
       (use * less ‹k > 0› in ‹auto simp: mult_ac simp flip: power_mult›)
  also have "karatsuba_sqrt_step a32 a1 a0 (2^k) =
           (let (s, r) = map_prod int int (sqrt_rem' a32);
                (q, u) = ((r * 2^k + a1) div (2 * s), (r * 2^k + a1) mod
(2 * s));
                s' = s * 2^k + q;
                r' = u * 2^k + a0 - q ^ 2
            in  if r' ≥ 0 then (s', r') else (s' - 1, r' + 2 * s' - 1))"
    unfolding karatsuba_sqrt_step_def
    by (simp add: case_prod_unfold Let_def divmod_def zdiv_int zmod_int)
  also have "... = karatsuba_sqrt_step' n k"
    unfolding karatsuba_sqrt_step'_def karatsuba_sqrt_step_def
    by (intro Let_cong case_prod_cong arg_cong2[of _ _ _ _ "divmod"]
              arg_cong[of _ _ "map_prod int int"]
              arg_cong[of _ _ sqrt_rem'] arg_cong[of _ _ int]
              arg_cong2[of _ _ _ _ "(-) :: int ⇒ _"] refl if_cong
              arg_cong2[of _ _ _ _ Pair] arg_cong2[of _ _ _ _ "(+)"])
       (auto simp: map_prod_def sqrt_rem'_def divmod_def a32_def a1_def
a0_def of_nat_splice_bit
                  of_nat_drop_bit of_nat_take_bit divmod_int_def mult_ac
push_bit_eq_mult)
  finally show ?thesis ..
qed
```

1.6.2 Full algorithm

Our algorithm is parameterised with a "limb size" and a cutoff. The cutoff value describes the threshold for the base case, i.e. the size of inputs (in bits) for which we fall back to Heron's method.

The algorithm splits the input number into four parts in such a way that the bit size of the lower three parts is a multiple of 2^l (where l is the limb size). This may be useful to avoid unnecessary bit shifting, since one one always splits the input number exactly at limb boundaries. However, whether this actually helps depends on how bit shifting of arbitrary-precision integers is actually implemented in the runtime.

There is only one rather weak condition on the limb size and cutoff. Which values work best must be determined experimentally.

```
locale karatsuba_sqrt =
  fixes cutoff limb_size :: nat
  assumes cutoff: "2 ^ (2 + limb_size) ≤ cutoff + 2"
begin

function karatsuba_sqrt_aux :: "nat ⇒ int × int" where
  "karatsuba_sqrt_aux n = (
    let sz = bitlen n
```

```
    in  if sz ≤ int cutoff then
          case sqrt_rem'_heron n of (s, r) ⇒ (int s, int r)
        else let
          k = push_bit limb_size (drop_bit (2 + limb_size) (nat (bitlen
n + 1)));
          (s, r) = karatsuba_sqrt_aux (drop_bit (2*k) n);
          (q, u) = divmod_int (push_bit k r + splice_bit k k n) (push_bit
1 s);
          s' = push_bit k s + q;
          r' = push_bit k u + take_bit k n - q ^ 2
      in  if r' ≥ 0 then (s', r') else (s' - 1, r' + push_bit 1 s' - 1))"

  by auto
termination proof (relation "measure id", goal_cases)
  case (2 n x k)
  have "2 ^ (2 + limb_size) ≤ cutoff + 2"
    using cutoff by simp
  also have "cutoff + 2 < nat (bitlen (int n) + 2)"
    using 2 by simp
  finally have "2 ^ (2 + limb_size) ≤ nat (bitlen (int n) + 1)"
    by linarith
  hence "k > 0"
    by (auto simp: push_bit_eq_mult drop_bit_eq_div 2(3) nat_add_distrib
div_greater_zero_iff)
  hence "2 ^ 0 < (2 ^ (2 * k) :: nat)"
    using 2 by (intro power_strict_increasing Nat.gr0I)
               (auto simp: div_eq_0_iff nat_add_distrib not_le)
  moreover have "n > 0"
    using 2 by (auto intro!: Nat.gr0I)
  ultimately have "drop_bit (2 * k) n < n"
    by (auto simp: drop_bit_eq_div intro!: div_less_dividend)
  thus ?case
    by simp
qed auto

lemmas [simp del] = karatsuba_sqrt_aux.simps

lemma karatsuba_sqrt_aux_correct: "karatsuba_sqrt_aux n = map_prod int
int (sqrt_rem' n)"
proof (induction n rule: karatsuba_sqrt_aux.induct)
  case (1 n)
  define sz where "sz = bitlen n"
  show ?case
  proof (cases "sz ≤ cutoff")
    case True
    thus ?thesis
       by (subst karatsuba_sqrt_aux.simps)
          (auto simp: sqrt_rem'_heron_correct sqrt_rem'_def sz_def)
  next
```

```
    case False
    define k where "k = push_bit limb_size (drop_bit (2 + limb_size)
(nat (bitlen n + 1)))"
    have n_eq: "n = drop_bit (2 * k) n * (2 ^ k)² + splice_bit k k n *
2 ^ k + take_bit k n"
    proof -
      have "n = push_bit (2*k) (drop_bit (2*k) n) + take_bit (2*k) n"
        by (simp add: bits_ident)
      also have "take_bit (2*k) n = take_bit (2*k) (push_bit k (drop_bit
k n) + take_bit k n)"
        by (simp add: bits_ident)
      also have "... = push_bit k (splice_bit k k n) + take_bit k n"
        by (subst bit_eq_iff)
           (auto simp: bit_take_bit_iff bit_push_bit_iff bit_disjunctive_add_iff
splice_bit_def)
      also have "push_bit (2 * k) (drop_bit (2 * k) n) + (push_bit k (splice_bit
k k n) + take_bit k n) =
                   drop_bit (2 * k) n * (2 ^ k)² + splice_bit k k n *
2 ^ k + take_bit k n"
        by (simp add: push_bit_eq_mult flip: power_mult)
      finally show ?thesis .
    qed

    have "karatsuba_sqrt_aux n = karatsuba_sqrt_step' n k"
      using False "1.IH"[of sz k]
      by (subst karatsuba_sqrt_aux.simps)
         (simp_all add: karatsuba_sqrt_step'_def of_nat_splice_bit
                        of_nat_take_bit of_nat_drop_bit sz_def k_def Let_def)
    also have "... = map_prod int int (sqrt_rem' n)"
    proof (subst karatsuba_sqrt_step'_correct)
      have "k ≤ nat (bitlen (int n) + 1) div 4"
        by (simp add: k_def nat_add_distrib div_mult2_eq push_bit_eq_mult
drop_bit_eq_div)
      moreover have "bitlen (int n) + 1 ≥ 0"
        by (auto simp: bitlen_def)
      ultimately show "int k ≤ (bitlen (int n) + 1) div 4"
        by linarith
    next
      show "k > 0"
      proof (rule Nat.gr0I)
        assume "k = 0"
        hence "nat sz + 1 < 2 ^ nat (int limb_size + 2)"
          by (auto simp: k_def div_eq_0_iff sz_def drop_bit_eq_div nat_add_distrib
bitlen_def)
        hence "sz + 1 < int (2 ^ nat (int limb_size + 2))"
          by linarith
        also have "... = int (2 ^ (2 + limb_size))"
          by (simp add: nat_add_distrib)
        also have "... ≤ int (cutoff + 2)"
```

```
      using cutoff by linarith
    finally show False
      using False by simp
  qed
 qed (use n_eq in auto)
 finally show ?thesis .
qed
qed
```

```
definition karatsuba_sqrt where
  "karatsuba_sqrt n = (case karatsuba_sqrt_aux n of (s, r) ⇒ (nat s,
nat r))"

theorem karatsuba_sqrt_correct: "karatsuba_sqrt n = sqrt_rem' n"
  by (simp add: karatsuba_sqrt_def karatsuba_sqrt_aux_correct case_prod_unfold)

end
```

1.6.3 Concrete instantiation

We pick a cutoff of 1024 and a limb size of 64 as reasonable default values.

```
definition karatsuba_sqrt_default where
  "karatsuba_sqrt_default = karatsuba_sqrt.karatsuba_sqrt 1024 6"

definition karatsuba_sqrt_default_aux where
  "karatsuba_sqrt_default_aux = karatsuba_sqrt.karatsuba_sqrt_aux 1024
6"

interpretation karatsuba_sqrt_default:
  karatsuba_sqrt 1024 6
  rewrites "karatsuba_sqrt.karatsuba_sqrt 1024 6 ≡ karatsuba_sqrt_default"
       and "karatsuba_sqrt.karatsuba_sqrt_aux 1024 6 ≡ karatsuba_sqrt_default_aux"
  by unfold_locales (auto simp: nat_add_distrib karatsuba_sqrt_default_aux_def
karatsuba_sqrt_default_def)

lemmas [code] =
  karatsuba_sqrt_default.karatsuba_sqrt_aux.simps[unfolded power2_eq_square]
  karatsuba_sqrt_default.karatsuba_sqrt_def
  karatsuba_sqrt_default.karatsuba_sqrt_correct [symmetric]
```

1.7 Using sqrt_rem to compute floors and ceilings of sqrt

```
definition sqrt_nat_ceiling :: "nat ⇒ nat" where
  "sqrt_nat_ceiling n = nat (ceiling (sqrt (real n)))"

definition sqrt_int_floor :: "int ⇒ int" where
  "sqrt_int_floor n = floor (sqrt (real_of_int n))"

definition sqrt_int_ceiling :: "int ⇒ int" where
```

```
  "sqrt_int_ceiling n = ceiling (sqrt (real_of_int n))"

lemma sqrt_nat_ceiling_code [code]:
  "sqrt_nat_ceiling n = (case sqrt_rem' n of (s, r) ⇒ if r = 0 then s
else s + 1)"
proof -
  have n: "(Discrete.sqrt n)² + sqrt_rem n = n"
    by (auto simp: sqrt_rem_def)
  have "sqrt n = sqrt (Discrete.sqrt n ^ 2 + sqrt_rem n)"
    by (simp add: sqrt_rem_def)
  also have "ceiling ... = Discrete.sqrt n + (if sqrt_rem n = 0 then 0
else 1)"
  proof (cases "sqrt_rem n = 0")
    case False
    have "n < (Discrete.sqrt n + 1)²"
      using Suc_sqrt_power2_gt le_eq_less_or_eq by auto
    hence "real n < real ((Discrete.sqrt n + 1)²)"
      by linarith
    hence "sqrt (Discrete.sqrt n ^ 2 + sqrt_rem n) ≤ Discrete.sqrt n
+ 1"
      by (subst n) (auto intro!: real_le_lsqrt simp flip: of_nat_add)
    moreover have "Discrete.sqrt n < sqrt (Discrete.sqrt n ^ 2 + sqrt_rem
n)"
      by (rule real_less_rsqrt) (use False in auto)
    ultimately have "ceiling (sqrt (Discrete.sqrt n ^ 2 + sqrt_rem n))
= Discrete.sqrt n + 1"
      by linarith
    thus ?thesis
      using False by simp
  qed auto
  finally show ?thesis
    by (simp add: sqrt_nat_ceiling_def sqrt_rem'_def nat_add_distrib)
qed

lemma sqrt_int_floor_code [code]:
  "sqrt_int_floor n =
     (if n ≥ 0 then int (Discrete.sqrt (nat n)) else -int (sqrt_nat_ceiling
(nat (-n))))"
  by (auto simp: sqrt_int_floor_def sqrt_nat_ceiling_def Discrete_sqrt_conv_floor_sqrt
                 real_sqrt_minus ceiling_minus)

lemma sqrt_int_ceiling_code [code]:
  "sqrt_int_ceiling n =
     (if n ≥ 0 then int (sqrt_nat_ceiling (nat n)) else -int (Discrete.sqrt
(nat (-n))))"
  using sqrt_int_floor_code[of "-n"]
  by (cases n "0 :: int" rule: linorder_cases)
     (auto simp: sqrt_int_ceiling_def sqrt_int_floor_def sqrt_nat_ceiling_def[of
0]
```

```
                 real_sqrt_minus floor_minus)

end
theory Karatsuba_Sqrt_Float
imports
  Karatsuba_Sqrt
  "HOL-Library.Interval_Float"
begin
```

1.8 Floating-point approximation of *sqrt*

```
definition shift_int :: "int ⇒ int ⇒ int"
  where "shift_int k n = (if k ≥ 0 then n * 2 ^ nat k else n div 2 ^
(nat (-k)))"

lemma shift_int_code [code]:
  "shift_int k n = (if k ≥ 0 then push_bit (nat k) n else drop_bit (nat
(-k)) n)"
  by (simp add: shift_int_def push_bit_eq_mult drop_bit_eq_div)

definition lb_sqrt :: "nat ⇒ float ⇒ float" where
  "lb_sqrt prec x = (let n = mantissa x; e = exponent x; k = nat (2 *
int prec - bitlen n);
                       k' = (if even k = even e then k else k + 1) in
     normfloat (Float (sqrt_int_floor (shift_int k' n)) (shift_int (-1)
(e - k'))))"

definition ub_sqrt :: "nat ⇒ float ⇒ float" where
  "ub_sqrt prec x = (let n = mantissa x; e = exponent x; k = nat (2 *
prec - bitlen n);
                       k' = (if even k = even e then k else k + 1) in
     normfloat (Float (sqrt_int_ceiling (shift_int k' n)) (shift_int (-1)
(e - k'))))"

lemma lb_sqrt: "lb_sqrt prec x ≤ sqrt x"
proof -
  define n where "n = mantissa x"
  define e where "e = exponent x"
  define k where "k = nat (2 * int  prec - bitlen n)"
  define k' where "k' = (if even k = even e then k else k + 1)"
  have "even (e - k')"
    by (auto simp: k'_def)
  define e'' where "e'' = (e - k') div 2"
  have e'': "k' = e - 2 * e''"
    using ‹even (e - k')› by (auto simp: e''_def)
  have "real_of_float (lb_sqrt prec x) = of_int ⌊sqrt (n * 2 powi int
k')⌋ * 2 powi ((e - k') div 2)"
    by (simp add: lb_sqrt_def n_def e_def k_def k'_def
                 Let_def powr_real_of_int' shift_int_def add_ac nat_add_distrib
```

```
                 sqrt_int_floor_def sqrt_int_ceiling_def)
  also have "... ≤ sqrt (n * 2 powi int k') * 2 powi ((e - k') div 2)"
    by (intro mult_right_mono) auto
  also have "... = sqrt (of_int n * 2 powi e) * (2 powi e'' / sqrt (2
powi (2 * e'')))"
    unfolding e'' by (simp add: power_int_diff real_sqrt_divide)
  also have "2 powi (2 * e'') = (2 powi e'' :: real) ^ 2"
    by (simp add: mult.commute power_int_mult)
  also have "sqrt ... = 2 powi e''"
    by simp
  also have "real_of_int n * 2 powi e = real_of_float (Float n e)"
    by (simp add: powr_real_of_int')
  also have "Float n e = x"
    by (simp add: n_def e_def Float_mantissa_exponent)
  finally show ?thesis
    by simp
qed

lemma ub_sqrt: "ub_sqrt prec x ≥ sqrt x"
proof -
  define n where "n = mantissa x"
  define e where "e = exponent x"
  define k where "k = nat (2 * int  prec - bitlen n)"
  define k' where "k' = (if even k = even e then k else k + 1)"
  have "even (e - k')"
    by (auto simp: k'_def)
  define e'' where "e'' = (e - k') div 2"
  have e'': "k' = e - 2 * e''"
    using ‹even (e - k')› by (auto simp: e''_def)
  have "sqrt x = sqrt (Float n e)"
    by (simp add: n_def e_def Float_mantissa_exponent)
  also have "... = sqrt (of_int n * 2 powi e) * (2 powi e'' / sqrt (2
powi (2 * e'')))"
    by (simp add: mult.commute power_int_mult powr_real_of_int')
  also have "... = sqrt (of_int n * 2 powi (e - 2 * e'')) * 2 powi e''"
    by (simp add: real_sqrt_divide power_int_diff)
  also have "... = sqrt (of_int n * 2 powi int k') * 2 powi ((e - k')
div 2)"
    unfolding e'' by simp
  also have "... ≤ ⌈sqrt (of_int n * 2 powi int k')⌉ * 2 powi ((e - k')
div 2)"
    by (intro mult_right_mono) auto
  also have "... = real_of_float (ub_sqrt prec x)"
    by (simp add: ub_sqrt_def n_def e_def k_def k'_def
                 Let_def powr_real_of_int' shift_int_def add_ac nat_add_distrib
                 sqrt_int_floor_def sqrt_int_ceiling_def)
  finally show ?thesis .
qed
```

```
context
  includes interval.lifting
begin

lift_definition sqrt_float_interval :: "nat ⇒ float interval ⇒ float
interval" is
  "λprec (l, u). (lb_sqrt prec l, ub_sqrt prec u)"
proof goal_cases
  case (1 prec lu)
  obtain l u where [simp]: "lu = (l, u)"
    by (cases lu)
  have "real_of_float (lb_sqrt prec l) ≤ sqrt l"
    by (rule lb_sqrt)
  also have "... ≤ sqrt u"
    using 1 by auto
  also have "... ≤ real_of_float (ub_sqrt prec u)"
    by (rule ub_sqrt)
  finally show ?case
    by simp
qed

lemma sqrt_float_intervalI:
  fixes x :: real and X :: "float interval"
  assumes "x ∈ set_of (real_interval X)"
  shows   "sqrt x ∈ set_of (real_interval (sqrt_float_interval prec X))"
  using assms
proof (transfer, goal_cases)
  case (1 x lu prec)
  obtain l u where [simp]: "lu = (l, u)"
    by (cases lu)
  from 1 have x: "real_of_float l ≤ x" "x ≤ real_of_float u"
    by simp_all
  have "real_of_float (lb_sqrt prec l) ≤ sqrt x"
    using lb_sqrt[of prec l] x(1) by (meson dual_order.trans real_sqrt_le_iff)
  moreover have "real_of_float (ub_sqrt prec u) ≥ sqrt x"
    using ub_sqrt[of u prec] x(2) by (meson dual_order.trans real_sqrt_le_iff)
  ultimately show ?case
    by simp
qed

lemma sqrt_float_interval:
  "sqrt ` set_of (real_interval X) ⊆ set_of (real_interval (sqrt_float_interval
prec X))"
  using sqrt_float_intervalI[of _ X] by blast

end

end
```

1.9 Tests

```
theory Karatsuba_Sqrt_Test
imports
  Karatsuba_Sqrt_Float
  "HOL-Library.Code_Target_Numeral"
begin

value "sqrt_rem' 123456"
value "sqrt_rem 123456"
value "Discrete.sqrt 123456"
value "sqrt_int_floor 123456"
value "sqrt_nat_ceiling 123456"
value "sqrt_int_ceiling 123456"
value "sqrt_float_interval 64 (Ivl 123456 123456)"

end
```

References

[1] Y. Bertot, N. Magaud, and P. Zimmermann. A proof of GMP square root. *Journal of Automated Reasoning*, 29(3):225–252, 2002.

[2] P. Zimmermann. Karatsuba Square Root. Research Report RR-3805, INRIA, 1999.