Isabelle/HOL Exercises
Projects

BIGNAT - Specification and Verification

Representation

```
type_synonym
  bigNat = "nat list"

primrec val :: "nat ⇒ bigNat ⇒ nat" where
  "val d [] = 0"
| "val d (n#ns) = n + d*(val d ns)"

primrec valid :: "nat ⇒ bigNat ⇒ bool" where
  "valid d [] = (0<d)"
| "valid d (n#ns) = ((n<d) ∧ (valid d ns))"
```

Auxiliary lemmas

```
lemma aux: "m < d * d ⟹ m div d < (d::nat)"
proof -
  assume m: "m < d * d"
  show ?thesis
  proof (rule classical)
    presume "d ≤ m div d"
    then have "d * d ≤ d * (m div d)" by simp
    also have "d * (m div d) ≤ m" by (simp add: mult_div_cancel)
    finally show ?thesis using m by arith
  qed auto
qed

lemma auxa:"a < d ⟹ b < d ⟹ (a + b) div d < (d::nat)"
proof -
  assume a: "a < d" "b < d"
  { assume "d = 0" with a have ?thesis by simp
  } moreover
  { assume "d = 1" with a have ?thesis by simp
  } moreover
  { from a have "a + b < 2 * d" by simp
    also assume "2 <= d" then have "2 * d <= d * d" by simp
```

```
    finally have "a + b < d * d" .
    then have "(a + b) div d  < d" by (rule aux)
  }
  ultimately show ?thesis by arith
qed

lemma auxb:"a < d ⟹ b < d ⟹ c < d ⟹ (a + b + c) div d < (d::nat)"
proof -
  assume a: "a < d" "b < d" "c < d"
  { assume "d = 0" with a have ?thesis by simp
  } moreover
  { assume "d = 1" with a have ?thesis by simp
  } moreover
  { assume "d = 2" with a have ?thesis by  (cases a, auto)
  } moreover
  { from a have "a + b + c < 3 * d" by simp
    also assume "3 <= d" then have "3 * d <= d * d" by simp
    finally have "a + b + c < d * d" .
    then have "(a + b + c) div d  < d" by (rule aux)
  }
  ultimately show ?thesis by arith
qed

lemma le_iff_lSuc:"(a ≤ b) = (a < Suc b)"
  by arith

lemma auxc:" ⟦ a ≤ d; b ≤ d; c ≤ d⟧ ⟹ (a * b + c) div (Suc d) ≤ d"
proof -
  assume a:"a ≤ d" and b:"b ≤ d" and c:"c ≤ d"
  then have d:"a * b + c <= d * d +d"
    by (auto intro: add_le_mono mult_le_mono)
  then have e:"d * d + d = d * (Suc d)" by clarsimp
  from d have f:"(a * b + c) div (Suc d) <= (d * Suc d) div (Suc d)"
    by (auto simp:e intro:div_le_mono)
  have "(d * Suc d) div (Suc d) = d" by (simp only:div_mult_self_is_m)
  with f show ?thesis by simp
qed

lemma auxd:" ⟦ a < d; b < d; c < d⟧ ⟹ (a * b + c) div d < (d::nat)"
proof (cases d)
  assume "a < d" "d = 0" then show ?thesis by simp
next
  fix n thm le_iff_lSuc[THEN iffD1]
```

```
  assume d:"d = Suc n" and a:"a < d" "b < d" "c < d"
  then show "(a * b + c) div d < d"
    by (auto dest:le_iff_lSuc[THEN iffD2]
            intro:le_iff_lSuc[THEN iffD1] auxc)
qed
```

Addition

```
primrec carry :: "nat ⇒ nat ⇒ bigNat ⇒ bigNat" where
  "carry d c [] = [c]"
| "carry d c (m#ms) = ((m+c) mod d) # carry d ((m+c) div d) ms"

fun add :: "nat ⇒ nat ⇒ bigNat ⇒ bigNat ⇒ bigNat" where
  "add d c []     ns     = carry d c ns"
| "add d c ms     []     = carry d c ms"
| "add d c (m#ms) (n#ns) = ((m+n+c) mod d) # (add d ((m+n+c) div d) ms ns)"

lemma add_empty[simp]: "add d c ms [] = carry d c ms"
  apply (case_tac ms)
    apply simp_all
done

lemma val_carry[simp]: "⋀c. val d (carry d c ms) = val d ms + c"
proof (induct ms)
  case Nil show ?case by simp
next
  case (Cons m ms c) thus ?case by (simp add: add_mult_distrib2)
qed

lemma val_add:"val d (add d c ms ns) =
  val d ms + val d ns + c"
proof (induct d c ms ns rule:add.induct)
  case 1 show ?case by simp
next
  case (2 d c m ms)
  show ?case by (simp add:add_mult_distrib2)
next
  case (3 d c m ms n ns)
  thus ?case by (simp add:add_mult_distrib2)
qed

lemma carry_valid:"⋀c. ⟦ valid d ms; c < d ⟧ ⟹
  valid d (carry d c ms)"
```

```
apply (induct ms)
 apply (auto simp:auxa)
done

lemma add_valid:"⟦ valid d ms; valid d ns; c < d⟧ ⟹
  valid d (add d c ns ms)"
apply (induct d c ms ns rule:add.induct)
   apply (auto intro:carry_valid simp: auxa auxb)
apply (simp only:add_ac)
done
```

Multiplication

```
primrec mult1 :: "nat ⇒ nat ⇒ nat ⇒ bigNat ⇒ bigNat" where
  "mult1 d c b [] = [c]"
| "mult1 d c b (a#as) = ((a*b+c) mod d) #
                        (mult1 d ((a*b+c) div d) b as)"

primrec mult :: "nat ⇒ bigNat ⇒ bigNat ⇒ bigNat" where
  "mult d as [] = []"
| "mult d as (b#bs) = add d 0 (mult1 d 0 b as) (0#mult d as bs)"

lemma val_mult1[simp]:"⋀c. val d (mult1 d c b as) =
                       (val d as *b + c)"
proof (induct as)
  case Nil show ?case by simp
next
  case (Cons a as c) thus ?case
    by (simp add:add_mult_distrib add_mult_distrib2)
qed

lemma val_mult:"val d (mult d as bs) = val d as * val d bs"
apply (induct bs)
 apply (auto simp add:add_mult_distrib2 val_add)
done

lemma mult1_valid:"⋀c. ⟦ valid d ms; n < d; c < d⟧ ⟹
  valid d (mult1 d c n ms)"
apply (induct ms)
 apply (auto intro:auxd)
done

lemma  mult_valid:"⟦ valid d ms; valid d ns⟧ ⟹
```

```
  valid d (mult d ns ms)"
apply (induct ms)
 apply (auto)
apply (rule add_valid)
  apply auto
apply (rule mult1_valid)
  apply auto
done

end
```