Isabelle/HOL Exercises
Projects

The Euclidean Algorithm – Inductively

Rules without base case

Show that the following

```
inductive_set evenempty :: "nat set" where
Add2Ie: "n ∈ evenempty ⟹ Suc(Suc n) ∈ evenempty"
```

defines the empty set:

```
lemma evenempty_empty: "evenempty = {}"
by (auto elim: evenempty.induct)
```

The Euclidean algorithm

Define inductively the set *gcd*: *(a,b,g) ∈ gcd* means that *g* is the greatest common divisor of *a* und *b*. The definition should closely follow the Euclidean algorithm.

Reminder: The Euclidean algorithm repeatedly subtracts the smaller from the larger number, until one of the numbers is 0. Then, the other number is the gcd.

```
inductive_set gcd :: "(nat × nat × nat) set" where
gcdZero: "(u, 0, u) ∈ gcd" |
gcdStep: "⟦ (u - v, v, g) ∈ gcd; 0 < v; v ≤ u ⟧ ⟹ (u, v, g) ∈ gcd" |
gcdSwap: "⟦ (v, u, g) ∈ gcd; u < v ⟧ ⟹ (u, v, g) ∈ gcd"
```

Now, compute the gcd of 15 and 10:

```
lemma "(15, 10, ?g) ∈ gcd"
  apply (rule gcdStep) apply simp
  apply (rule gcdSwap)
  apply (rule gcdStep) apply simp
  apply (rule gcdStep) apply simp
  apply (rule gcdSwap)
  apply (rule gcdZero)
  apply simp+
done
```

How does your algorithm behave on special cases as the following?

```
lemma "(0, 0, ?g)  ∈ gcd"
by (rule gcdZero)
```

Show that the gcd is really a divisor (for the proof, you need an appropriate lemma):

```
lemma gcd_divides: "(a,b,g) ∈ gcd ⟹ g dvd a ∧ g dvd b"
lemma dvd_minus: "⟦ v ≤ u;  (g::nat) dvd u - v; g dvd v⟧ ⟹ g dvd u"
  apply (clarsimp simp add: dvd_def)
  apply (rule_tac x="k + ka" in exI)
  apply (simp add: add_mult_distrib2)
done

lemma gcd_divides: "(a,b,g) ∈ gcd ⟹ g dvd a ∧ g dvd b"
  apply (induct rule: gcd.induct)
  apply simp
  apply (simp add: dvd_minus)
  apply simp
done
```

Show that the gcd is the greatest common divisor:

```
lemma gcd_greatest [rule_format]: "(a,b,g) ∈ gcd ⟹
  0 < a ∨ 0 < b ⟶ (∀ d. d dvd a ⟶ d dvd b ⟶ d ≤ g)"
lemma dvd_leq: "⟦ 0 < v; (d::nat) dvd v ⟧ ⟹ d ≤ v"
  by (clarsimp simp add: dvd_def)

lemma dvd_minus2: "⟦ (d::nat) dvd u; d dvd v ⟧ ⟹ d dvd u - v"
  apply (clarsimp simp add: dvd_def)
  apply (rule_tac x="k-ka" in exI)
  apply (simp add: diff_mult_distrib2)
done

lemma gcd_greatest [rule_format]: "(a,b,g) ∈ gcd ⟹
  0 < a ∨ 0 < b ⟶ (∀ d. d dvd a ⟶ d dvd b ⟶ d ≤ g)"
  apply (induct rule: gcd.induct)
  apply (clarsimp simp add: dvd_leq)

  apply clarsimp
  apply (case_tac "v = u")
  apply simp
  apply (blast dest: dvd_minus2)+
done
```

Here as well, you will have to prove a suitable lemma. What is the precondition `0 < a` $\vee$ `0 < b` good for?

So far, we have only shown that *gcd* is correct, but your algorithm might not compute a result for all values `a,b`. Thus, show completeness of the algorithm:

```
lemma gcd_defined: "∀ a b. ∃ g. (a, b, g) ∈ gcd"
```

The following lemma, proved by course-of-value recursion over *n*, may be useful. Why does standard induction over natural numbers not work here?

```
lemma gcd_defined_aux [rule_format]:
  "∀ a b. (a + b) ≤ n ⟶ (∃ g. (a, b, g) ∈ gcd)"
  apply (induct rule: nat_less_induct)
  apply clarify
```

The idea is to show that *gcd* yields a result for all `a, b` whenever it is known that *gcd* yields a result for all `a', b'` whose sum is smaller than `a + b`.

In order to prove this lemma, make case distinctions corresponding to the different clauses of the algorithm, and show how to reduce computation of *gcd* for `a, b` to computation of *gcd* for suitable smaller `a', b'`.

```
lemma gcd_defined_aux [rule_format]:
  "∀ a b. (a + b) ≤ n ⟶ (∃ g. (a, b, g) ∈ gcd)"
apply (induct rule: nat_less_induct)
apply clarify

apply (case_tac b)

— Application of gcdZero
apply simp
apply (rule exI)
apply (rule gcdZero)

apply (rename_tac n a b b')
apply simp

apply (case_tac "b ≤ a")

— Application of gcdStep
apply simp
apply (drule_tac x=a in spec, drule mp)
apply arith
apply (elim allE impE)
prefer 2
apply (elim exE)
apply (rule exI)
apply (rule gcdStep, assumption)
```

```
apply simp+

apply (case_tac a)
apply simp

— Application of gcdSwap, followed by gcdZero
apply (drule_tac x=0 in spec, drule mp) apply arith
apply (drule_tac x=0 in spec, drule_tac x=0 in spec, drule mp)
apply simp
apply (elim exE)
apply (rule exI)
apply (rule gcdSwap) apply (rule gcdZero)
apply simp

— Application of gcdSwap, followed by gcdStep
apply (drule_tac x=b in spec, drule mp) apply arith
apply (elim allE impE)
prefer 2
apply (elim exE)
apply (rule exI)
apply (rule gcdSwap)
apply (rule gcdStep) apply assumption
apply arith+
done

lemma gcd_defined: "∀ a b. ∃ g. (a, b, g) ∈ gcd"
  apply clarify
  apply (rule_tac n="a + b" in gcd_defined_aux)
  apply simp
done
```