Section 7 Solutions – Theorem Proving in Lean 4

Chun Ding • 30 May 2025

```
namespace Hidden

--ex1
open Nat
def predecessor : Nat → Nat
  | Nat.zero => Nat.zero
  | Nat.succ n => n

def subtract (m n : Nat) : Nat :=
  match m, n with
  | Nat.zero, _ => Nat.zero
  | _, Nat.zero => m
  | Nat.succ m', Nat.succ n' => subtract m' n'

#eval subtract 5 3 -- 2
#eval subtract 3 5 -- 0

--ex2
def length (as: List α) : Nat :=
  match as with
  | [] => 0
  | _ :: as' => 1 + length as'

#eval length [1, 2, 3] -- 3

def reverse (as: List α) : List α :=
  match as with
  | [] => []
  | a :: as' => reverse as' ++ [a]

#eval reverse [1, 2, 3] -- [3, 2, 1]

theorem lengthIsAGroupHomomorphism : ∀ (as bs : List
        α), length (as ++ bs) = length as + length
        bs := by
```

```
  intro as bs
  induction as with
  | nil => rw [List.nil_append, length, Nat.zero_add]
  | cons a as ih =>
    rw [List.cons_append, length, length, ih,
        Nat.add_assoc]

theorem length_reverse (as: List α) : length (reverse
        as) = length as := by
  induction as with
  | nil => rfl
  | cons a as' ih =>
    rw [reverse, lengthIsAGroupHomomorphism, length,
        length, length, Nat.add_zero,Nat.add_comm,ih]

theorem reverse_append (as bs: List α) : reverse (as +
        + bs) = reverse bs ++ reverse as := by
  induction as with
  | nil => rw [reverse, List.nil_append,
        List.append_nil]
  | cons a as' ih =>
    rw [reverse, List.cons_append, reverse, ih,
        List.append_assoc]

theorem reverse_reverse (as: List α) : reverse
        (reverse as) = as := by
  induction as with
  | nil => rfl
  | cons a as' ih =>
    rw [reverse, reverse_append, ih, reverse, reverse,
        List.nil_append, List.singleton_append]

--ex3
inductive Term where
  | const : Nat → Term
  | var   : Nat → Term
  | plus  : Term → Term → Term
  | times : Term → Term → Term
deriving Repr
def eval : Term → (Nat → Nat) → Nat
  | Term.const n => fun _ => n
  | Term.var n => fun f => f n
  | Term.plus t1 t2 => fun f => eval t1 f + eval t2 f
  | Term.times t1 t2 => fun f => eval t1 f * eval t2 f
```

```
#eval eval (Term.plus (Term.const 1) (Term.var 0)) (fu
        n n => n + 2) -- 3

end Hidden
```