Section 8 Solutions – Theorem Proving in Lean 4

Chun Ding • 1 Jun 2025

```
--ex 1
@[simp] def add : Nat → Nat → Nat
| 0, b => b
| a + 1, b => add a b + 1
@[simp] def mul : Nat → Nat → Nat
| 0, _ => 0
| a + 1, b => add (mul a b) b
@[simp] def exp : Nat → Nat → Nat
| _, 0 => 1
| a, b +1 => mul a (exp a b)
theorem add_zero : ∀ a, add a 0 = a
  | Nat.zero => rfl
  | Nat.succ a => congrArg Nat.succ (add_zero a)
theorem mul_1 : ∀ a, mul 1 a = a
  | Nat.zero => rfl
  | a + 1 => by simp[mul, add]
theorem exp_0 : ∀ a, exp 1 a = 1
  | Nat.zero => rfl
  | Nat.succ a => by simp[exp_0 a]

--ex 2

def reverse : List α → List α
| [] => []
| x :: xs => reverse xs ++ [x]
theorem reverse_reverse : ∀ xs : List α, reverse
        (reverse xs) = xs := by
  intro xs
  induction xs with
  | nil => rfl
  | cons x xs ih => simp [reverse,reverse_append,
        List.nil_append,List.singleton_append,ih]
where
  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 =>
      simp [List.cons_append, reverse, ih,
        List.append_assoc]

#print reverse

--ex 4
open  Nat
inductive Vector (α : Type u) : Nat → Type u
  | nil  : Vector α 0
  | cons : α → {n : Nat} → Vector α n → Vector α (n+1)

#eval  Vector.cons 5 Vector.nil

def append {α : Type u} {n m : Nat} : Vector α n →
        Vector α m → Vector α (n + m)
  | Vector.nil,        ys => by
    simp [Nat.zero_add]
    exact ys
  | Vector.cons x xs,  ys => by
    simp [Nat.succ_add]
    exact
    Vector.cons x (append xs ys)

--ex 5
inductive Expr where
  | const : Nat → Expr
  | var : Nat → Expr
  | plus : Expr → Expr → Expr
  | times : Expr → Expr → Expr
  deriving Repr

open Expr

def sampleExpr : Expr :=
  plus (times (var 0) (const 7)) (times (const 2) (var
        1))

def eval (v : Nat → Nat) : Expr → Nat
  | const n     => n
  | var n       => v n
```

```
  | plus e₁ e₂  => eval v e₁ + eval v e₂
  | times e₁ e₂ => eval v e₁ * eval v e₂

def sampleVal : Nat → Nat
  | 0 => 5
  | 1 => 6
  | _ => 0

#eval eval sampleVal sampleExpr --47

def simpConst : Expr → Expr
  | plus (const n₁) (const n₂)  => const (n₁ + n₂)
  | times (const n₁) (const n₂) => const (n₁ * n₂)
  | e                           => e

def fuse : Expr → Expr
  | plus e₁ e₂  => simpConst (plus (fuse e₁) (fuse
      e₂))
  | times e₁ e₂ => simpConst (times (fuse e₁) (fuse
      e₂))
  | e => e

theorem simpConst_eq (v : Nat → Nat) : ∀ (e : Expr),
      eval v (simpConst e) = eval v e
  | plus e1 e2 => by cases e1 <;> cases e2 <;> rfl
  | times e1 e2 => by cases e1 <;> cases e2 <;> rfl
  | var x => rfl
  | const n => rfl

theorem fuse_eq (v : Nat → Nat) : ∀ e : Expr, eval v
      (fuse e) = eval v e
  | const n     => rfl
  | var n       => rfl
  | plus e₁ e₂  => by rw
      [fuse,simpConst_eq,eval,eval,fuse_eq,fuse_eq]
  | times e₁ e₂ => by rw
      [fuse,simpConst_eq,eval,eval,fuse_eq,fuse_eq]
```