FPP 2023 Week 01 Exercises

Elaine Hou, Hana Miura, Rana Harris Farooq, Sean Lim, Zhu Wentao

August 24, 2023

Contents

1 Introduction

"The beginning is always today." - Mary Shelley

In the first week of the course Functional Programming and Proving (FPP), we recap some of the concepts learnt in Intro to CS and see how they set the foundation for proofs and formally reasoning about proofs. In a nutshell, "In Intro to CS, we test. In FPP, we prove."

Beyond that, we are also introduced to a pure and total functional language, Gallina, which is a programming language featured in the Coq Proof Assistant (tCPA). While there are some similarities between OCaml and Gallina, there are also a few key differences in syntax, program termination and side effects. Through this week's exercises, we will also get familiarised with these new features, which will be important for the rest of the course. Let us begin!

2 Mandatory Exercises

2.1 Exercise 00

(a) We have checked the page for updates and developments of the course.

(b) The chapter on *Introduction and motivation* helps us recap some of the concepts we learnt in Intro to CS and sets the foundation for the course on functional programming and proving.

In particular, the chapter first recaps the concept of programming languages and makes a distinction between the imperative and functional programming paradigms with respect to states. It then introduces the concept of Turing-completeness, which has the implication that we simulate one programming language with another using an interpreter and translate one programming language to another using a compiler. Furthermore, programming languages are prescribed by their grammar of which they must follow.

The chapter then recaps the concept of data and data types, which naturally leads to the introduction to statically typed programming languages, of which OCaml is one.

Just like how the structure of types are isomorphic to the structure of propositions, proofs are also logical constructs that can be expressed in a formally specified language. The chapter then introduces the Coq Proof Assistant (tCPA) and a pure and total functional language Gallina, both of which we will be using in this course. Following that, the chapter then marks out the main differences between OCaml and Gallina, which are things we will also be getting used to as the course progresses.

Overall, the chapter summarises the key ideas we have picked up in Intro to CS highlights how it lays the foundation for this course. It also highlights the overarching theme of the first lecture: "In Intro to CS, we test. In FPP, we prove."

2.2 Exercise 01

In this exercise, we are asked to implement some standard functions of the natural numbers in Gallina. The addition function has already been implemented for us in the accompanying file. We start with multiplication.

Multiplication Before we write any program, it is important to specify the unit tests. Here are the ones used to test the multiplication program.

```
Definition test_mul (candidate: nat -> nat -> nat) : bool :=
  (Nat.eqb (candidate 0 0) 0)
  &&
  (Nat.eqb (candidate 0 1) 0)
  &&
  (Nat.eqb (candidate 1 0) 0)
  &&
  (Nat.eqb (candidate 1 1) 1)
  &&
  (Nat.eqb (candidate 1 2) 2)
  &&
  (Nat.eqb (candidate 2 1) 2)
  &&
  (Nat.eqb (candidate 2 2) 4)
  &&
  (* commutativity: *)
  (Nat.eqb (candidate 2 10) (candidate 10 2))
  &&
  (* associativity: *)
  (Nat.eqb (candidate 2 (candidate 5 10))
           (candidate (candidate 2 5) 10))
  (* etc. *)
  .
```

We then implement multiplication in a recursive manner in Gallina.

```
Fixpoint mul_v1 (i j : nat) : nat :=
   match i with
    | O => O
```

```
    | S i' => j + (mul_v1 i' j)
  end.
```

In this recursive definition, we only have two cases because natural numbers only have two constructors o and S. The recursion here is based on the distributive laws of multiplication.

```
Compute (test_mul mul_v1).
```

This implementation passes the test.

We can also implement this in a tail-recursive manner using an accumulator, like this:

```
Fixpoint mul_v2_aux (i j a : nat) : nat :=
  match i with
  | O =>
    a
  | S i' =>
    mul_v2_aux i' j (j + a)
  end.

Definition mul_v2 (i j : nat) : nat :=
  mul_v2_aux i j 0.
```

In this implementation, the accumulator keeps track of the multiplicand throughout the process of multiplication and begins at o.

```
Compute (test_mul mul_v2).
```

This implementation passes the test.

Exponentiation Similarly, before implementing the function for exponentiation, we first write the unit tests.

```
Definition test_power (candidate: nat -> nat -> nat) : bool :=
  (Nat.eqb (candidate 0 0) 1)
  &&
  (Nat.eqb (candidate 0 1) 0)
  &&
  (Nat.eqb (candidate 1 0) 1)
  &&
  (Nat.eqb (candidate 1 1) 1)
  &&
  (Nat.eqb (candidate 1 2) 1)
  &&
  (Nat.eqb (candidate 2 1) 2)
  &&
  (Nat.eqb (candidate 2 2) 4)
  &&
  (Nat.eqb (candidate 3 2) 9)
  &&
  (Nat.eqb (candidate 3 (S 1)) (3 * candidate 3 1))
  (* etc. *)
  .
```

We then implement exponentiation in a recursive manner in Gallina.

```
Fixpoint power_v1 (x n : nat) : nat :=
  match n with
  | O => 1
  | S n' => x * power_v1 x n'
  end.
```

In this recursive definition, we only have two cases because natural numbers only have two constructors o and S. The recursion here is based on the exponent rules.

```
Compute (test_power power_v1).
```

This implementation passes the test.

We can also implement this in a tail-recursive manner using an accumulator, like this:

```
Fixpoint power_v2_aux (x n a : nat) : nat :=
  match n with
  | 0 => a
  | S n' => power_v2_aux x n' (x * a)
  end.

Definition power_v2 (x n : nat) : nat :=
  power_v2_aux x n 1.
```

In this implementation, the accumulator keeps track of the exponent throughout the process of exponentiation and begins at 1.

```
Compute (test_power power_v2).
```

This implementation passes the test.

Factorial Function Similarly, before implementing the function for the factorial function, we first write the unit tests.

```
Definition test_fac (candidate: nat -> nat) : bool :=
  (Nat.eqb (candidate 0) 1)
  &&
  (Nat.eqb (candidate 1) 1)
  &&
  (Nat.eqb (candidate 2) 2)
  &&
  (Nat.eqb (candidate 3) 6)
  &&
  (Nat.eqb (candidate 4) 24)
  &&
  (Nat.eqb (candidate 5) 120)
  &&
  (Nat.eqb (candidate (S 4)) (S 4 * candidate 4))
  (* etc. *)
  .
```

We then implement the factorial function in a recursive manner in Gallina:

```
Fixpoint fac_v1 (n : nat) : nat :=
  match n with
  | O => 1
  | S n' => n * fac_v1 n'
  end.
```

In this recursive definition, we only have two cases because natural numbers only have two constructors o and S. The recursion here is based on the definition of the factorial function.

```
Compute (test_fac fac_v1).
```

This implementation passes the test.

We can also implement this in a tail-recursive manner using an accumulator, like this:

```
Fixpoint fac_v2_aux (n a : nat) :nat :=
  match n with
  | 0 => a
  | S n' => fac_v2_aux n' (n * a)
  end.

Definition fac_v2 (n : nat) : nat :=
  fac_v2_aux n 1.

Compute (test_fac fac_v2).
```

This implementation passes the test.

Fibonacci Function Similarly, before implementing the function for the Fibonacci function, we first write the unit tests.

```
Definition test_fib (candidate: nat -> nat) : bool :=
  (Nat.eqb (candidate 2) 1)
  &&
  (Nat.eqb (candidate 3) 2)
  &&
  (Nat.eqb (candidate 4) 3)
  &&
  (Nat.eqb (candidate 5) 5)
  &&
  (Nat.eqb (candidate 6) 8)
  &&
  (Nat.eqb (candidate 7) 13)
  &&
  (Nat.eqb (candidate 8) 21)
  &&
  (Nat.eqb (candidate 12) 144)
  &&
  (Nat.eqb (candidate (S (S 11))) (candidate 11 + candidate (S 11)))
  (* etc. *)
  .
```

We then implement the Fibonacci function in a recursive manner in Gallina. To only use the two constructors o and S, we can do like the interludes suggest:

```
Fixpoint fib_v3 (n : nat) : nat :=
  match n with
  | 0 => 0
  | S n' => match n' with
            | 0 => 1
            | S n'' => fib_v3 n' + fib_v3 n''
            end
  end.

Compute (test_fib fib_v3).
```

This implementation passes the test.

Evenness/Oddness Similarly, before implementing the function for testing evenness, we first write the unit tests.

```
Definition test_evenp (candidate: nat -> bool) : bool :=
  (bool_eqb (candidate 2) true)
```

```
  &&
  (bool_eqb (candidate 3) false)
  &&
  (bool_eqb (candidate 4) true)
  &&
  (bool_eqb (candidate 5) false)
  &&
  (bool_eqb (candidate 24) true)
  &&
  (bool_eqb (candidate 75) false)
  &&
  (bool_eqb (candidate 301) false)
  &&
  (bool_eqb (candidate 430) true)
  &&
  (bool_eqb (candidate (S 2)) (negb (candidate 2)))
  (* etc. *)
  .
```

We then implement testing the evenness function in a recursive manner in Gallina:

```
Fixpoint even_v1 (n : nat) : bool :=
  match n with
  | O => true
  | S O => false
  | S (S n') => even_v1 n'
  end.
```

In this recursive definition, we only have cases for o and S because natural numbers only have two constructors o and S. The recursion here is based on the fact that even and odd numbers appear in an alternating order.

```
Compute (test_evenp even_v1).
```

This implementation passes the test.

We can also implement this in a tail-recursive manner using an accumulator, like this:

```
Fixpoint even_v2_aux (n : nat) (a : bool) : bool :=
  match n with
  | O => a
  | S p => even_v2_aux p (negb a)
  end.

Definition even_v2 (n : nat) : bool :=
  even_v2_aux n true.

Compute (test_evenp even_v2).
```

This implementation passes the test.

The implementation for testing oddness is similar, except that we should initialise the accumulator with *false* instead.

Overall, we have gotten more familiar with the syntax of Gallina through this exercise. It also helps us refresh our memory of structural recursion and accumulators, both of which are concepts in the course Intro to CS.

2.3 Exercise 02

In this exercise, we aim to implement some standard functions of the binary trees in Gallina. As start with the number of leaves by including a function using an accumulator.

Number of Leaves In addition to the provided function, we can create a recursive function using an accumulator in Gallina that checks the number of leaves as the following.

```
Fixpoint number_of_leaves_acc (t : binary_tree_nat) (a : nat) : nat :=
  match t with
    Leaf_nat n => 1 + a
  | Node_nat t1 t2 => number_of_leaves_acc t1 (number_of_leaves_acc t2 a)
  end.
```

Number of Nodes Before writing any program, it is crucial to specify the unit tests. The following is the unit test used to test the number of nodes.

```
Definition test_number_of_nodes (candidate: binary_tree_nat -> nat) : bool :=
  (Nat.eqb (candidate (Leaf_nat 1))
        1)
  &&
  (Nat.eqb (candidate (Node_nat (Leaf_nat 1)
                                (Leaf_nat 2)))
          3)
           (* etc. *)
  .
```

We then implement this function in a recursive manner in Gallina as the following:

```
Fixpoint number_of_nodes_v1 (t: binary_tree_nat) : nat :=
 match t with
   Leaf_nat n => 0
   | Node_nat t1 t2 => S((number_of_nodes_v1 t1) + (number_of_nodes_v1 t2))
    end.
```

This functions ensures to add additional node to account for the leaf; thus, passes the unit test.

```
Compute (test_number_of_nodes number_of_nodes_v1).
```

Smallest Leaf Here, we aim to find the smallest leaf in a given binary tree.

The unit test is as follows.

```
Definition test_smallest_leaf (candidate: binary_tree_nat -> nat) : bool :=
  (Nat.eqb (candidate (Node_nat (Leaf_nat 1)
                                (Leaf_nat 2)))
     1)
  &&
    (Nat.eqb (candidate (Node_nat (Leaf_nat 3)(Node_nat (Leaf_nat 4) (Leaf_nat 5)))) 3)

  (* etc. *)
  .
```

We then implement the function in a recursive manner in Gallina as the following.

```
Fixpoint smallest_leaf_v1 (t: binary_tree_nat) : nat :=
match t with
  Leaf_nat n => n
  | Node_nat t1 t2 => min (smallest_leaf_v1 t1) (smallest_leaf_v1 t2)
  end.
```

In this function, we recursively compare the payload of the node by taking the minimum per each node.

This passes the unit test.

```
Compute (test_smallest_leaf smallest_leaf_v1).
```

Weight Here, we aim to find the sum of the payloads in the leaves of a given binary tree. The given function is as follows.

```
Definition specification_of_weight (weight : binary_tree_nat -> nat) : Prop :=
  (forall n : nat,
      weight (Leaf_nat n) = n)
  /\
  (forall t1 t2 : binary_tree_nat,
      weight (Node_nat t1 t2) = weight t1 + weight t2).
```

The unit test is as follows.

```
Definition test_weight (candidate: binary_tree_nat -> nat) : bool :=
  (Nat.eqb (candidate (Node_nat (Leaf_nat 1) (Leaf_nat 2))) 3)
  &&
    (Nat.eqb (candidate (Node_nat (Leaf_nat 5) (Leaf_nat 6))) 11)
  &&
    (Nat.eqb (candidate (Node_nat (Leaf_nat 10) (Leaf_nat 12))) 22).
```

We then implement the weight function in a recursive manner in Gallina as the following.

```
Fixpoint weight_v1 (t : binary_tree_nat) : nat :=
  match t with
    Leaf_nat n =>
    n
  | Node_nat t1 t2 =>
    weight_v1 t1 + weight_v1 t2
  end.
```

This passes the unit test.

```
Compute (test_weight weight_v1).
```

Longest path Here, we aim to find out the length of the longest path from the root of a given binary tree to its leaves, which is also referred to as the height of the tree.

The unit test is as follows.

```
Definition test_longest_path (candidate: binary_tree_nat -> nat) : bool :=
  (Nat.eqb (candidate (Node_nat (Leaf_nat 1) (Leaf_nat 2))) 1)
  &&
  (Nat.eqb (candidate (Node_nat (Node_nat (Leaf_nat 1)(Leaf_nat 2))
  (Node_nat (Leaf_nat 3) (Leaf_nat 4)))) 2)
  &&
  (Nat.eqb (candidate (Node_nat (Node_nat (Leaf_nat 3) (Leaf_nat 5))
  (Node_nat (Leaf_nat 6) (Node_nat (Leaf_nat 4)(Leaf_nat 1))))) 3)

  (* etc. *)
  .
```

We then implement this in a recursive manner in Gallina as the following.

```
 Fixpoint length_of_longest_path_v1 (t : binary_tree_nat) : nat :=
   match t with
     Leaf_nat n => 0
   | Node_nat t1 t2 => S(max(length_of_longest_path_v1 t1)(length_of_longest_path_v1 t2))
   end.
```

We take the height 0 for leaves as there is no path and we take the maximum of the length of the previous node and add an additional path for the full height of the tree.

This passes the unit test.

```
Compute (test_length_of_longest_path length_of_longest_path_v1).
```

Shortest path Like the previous paragraph on the longest path, we now aim to find the length of the shortest path from the root of a given binary tree to its leaves.

Likewise, the unit test is as follows.

```
Definition test_shortest_path (candidate: binary_tree_nat -> nat) : bool :=
  (Nat.eqb (candidate (Node_nat (Node_nat (Leaf_nat 1)(Leaf_nat 2))
  (Node_nat (Leaf_nat 3) (Leaf_nat 4)))) 2)
  && (Nat.eqb (candidate (Node_nat (Leaf_nat 3)(Leaf_nat 4)))1)
       && (Nat.eqb (candidate (Node_nat (Node_nat
       (Node_nat (Leaf_nat 1)(Leaf_nat 2)) (Leaf_nat 3))(Leaf_nat 4)))1)
  (* etc. *)
  .
```

We then implement this function in a recursive manner in Gallina.

```
Fixpoint length_of_shortest_path_v1 (t : binary_tree_nat) : nat :=
  match t with
    Leaf_nat n => 0
  | Node_nat t1 t2 => S(min(length_of_shortest_path_v1 t1)(length_of_shortest_path_v1 t2))
  end.
```

In the function above, instead of taking the maximum, we now take the minimum recursively to find the length of the shortest path.

This passes the test.

```
Compute (test_length_of_shortest_path length_of_shortest_path_v1).
```

Mirrors The given mirror function is the following.

```
Definition specification_of_the_mirror_function (mirror : binary_tree_nat -> binary_tree_nat)
: Prop :=
  (forall n : nat,
      mirror (Leaf_nat n) = Leaf_nat n)
  /\
  (forall t1 t2 : binary_tree_nat,
      mirror (Node_nat t1 t2) = Node_nat (mirror t1) (mirror t2)).
```

The unit test of the mirror function is as follows.

```
Definition test_mirror (candidate: binary_tree_nat -> binary_tree_nat) : bool :=
  (beq_binary_tree_nat (candidate (Leaf_nat 1))
                       (Leaf_nat 1))
  &&
  (beq_binary_tree_nat (candidate (Node_nat (Leaf_nat 1)
                                            (Leaf_nat 2)))
                       (Node_nat (Leaf_nat 2)
                          (Leaf_nat 1)))
  &&
    (beq_binary_tree_nat (candidate (Node_nat (Leaf_nat 3) (Leaf_nat 4)))
                         (Node_nat (Leaf_nat 4) (Leaf_nat 3)))
  (* etc. *)
  .
```

We then implement the mirror function in a recursive manner in Gallina:

```
Fixpoint mirror_v1 (t : binary_tree_nat) : binary_tree_nat :=
  match t with
    Leaf_nat n => Leaf_nat n
  | Node_nat t1 t2 =>
      Node_nat (mirror_v1 t2) (mirror_v1 t1)
  end.
```

In this recursive definition, leaf stays the same while we swap the order for nodes in order to reflect the mirror function.

```
Compute (test_mirror mirror_v1).
```

This passes the test.

Mobile The unit test is as follows.

```
Definition test_balancedp (candidate: binary_tree_nat -> nat) : bool :=
  (Nat.eqb (candidate (Node_nat (Leaf_nat 1) (Leaf_nat2))
  (Node_nat (Leaf_nat 2) (Leaf_nat 1))) true)
  &&
  (Nat.eqb (candidate (Node_nat (Node_nat (Node_nat (Leaf_nat 3) (Leaf_nat 4)))
  (Node_nat (Node_nat (Leaf_nat 5)(Leaf_nat 2)))) true)
  &&
  (Nat.eqb (candidate (Node_nat (Leaf_nat 10) Node_nat (Leaf_nat 2) (Leaf_nat 3)) false)
  &&
  (Nat.eqb (candidate (Node_nat (Node_nat (Leaf_nat 1) (Leaf_nat 3))
  (Node_nat (Leaf_nat 5) (Leaf_nat 7))) false)
  (* etc. *)
  .
```

We then implement the function in a recursive manner in Gallina as the following.

```
Fixpoint balancedp_v1 (t : binary_tree_nat) : bool :=
  match t with
    Leaf_nat n => true
  | Node_nat t1 t2 =>
      let difference := Nat.abs(weight_v1 t1 - weight_v1 t2)
      in balancedp_v1 t1 && balancedp_v2 t2 && difference = 0
  end.
```

In this function, we recursively check the difference in the weight between each side of the binary tree. When both sides are balanced, the difference should equal to 0.

2.4 Exercise 05

In this exercise, we are asked to write a Gallina expression of type polymorphic_binary_tree (nat * bool) and polymorphic_binary_tree (polymorphic_binary_tree nat).

2.4.1 part a

```
Compute(PLeaf (nat*bool) (2,true)).
(*
 = PLeaf (nat * bool) (2, true)
 : polymorphic_binary_tree (nat * bool)
*)
```

2.4.2 part b

```
Compute(PLeaf(polymorphic_binary_tree nat) (PLeaf nat 2)).
(* = PLeaf (polymorphic_binary_tree nat) (PLeaf nat 2)
   : polymorphic_binary_tree (polymorphic_binary_tree nat)
*)
```

We had to first look at the function type and then reverse engineer the above type examples using PLeaf (V → polymorphic_binary_tree V). For part b, it was tricky to give an instance (PLeaf nat 2); therefore, we first found out a similar a Gallina expression of type nat.

```
Compute(PLeaf nat 2).
(*  = PLeaf nat 2
     : polymorphic_binary_tree nat
*)
```

This allowed us to then build the expression for part b as shown above.

2.5 Exercise 06

Using the comparison function for polymorphic binary trees, we are able to implement functions that test the structural equality of binary trees of pairs of natural numbers and booleans and those of binary trees of natural numbers.

2.5.1 Part a

```
Definition eqb_nat := Nat.eqb.
Definition eqb_bool := Bool.eqb.

Definition eqb_pair_nat_bool (p1 p2 : (nat * bool)) : bool :=
    match p1, p2 with
    | (n1, b1), (n2, b2) => eqb_nat n1 n2 && eqb_bool b1 b2
    end.

Definition eqb_binary_tree_of_pair_nat_bool (t1 t2 : polymorphic_binary_tree (nat * bool))
: bool :=
eqb_polymorphic_binary_tree (nat * bool) eqb_pair_nat_bool t1 t2.
```

The function testing the structural equality of binary trees of pairs of natural numbers and booleans is obtained by instantiating the comparison function with the type of pairs of nats and bools and the equality predicate for pairs of nats and bools that we constructed.

2.5.2 Part b

```
  Definition eqb_binary_tree_of_nats (t1 t2 : polymorphic_binary_tree nat) : bool :=
eqb_polymorphic_binary_tree nat eqb_nat t1 t2.

Definition eqb_binary_tree_of_binary_tree_of_nats
(t1 t2 : polymorphic_binary_tree (polymorphic_binary_tree nat)) : bool :=
eqb_polymorphic_binary_tree (polymorphic_binary_tree nat) eqb_binary_tree_of_nats t1 t2.
```

The function testing the structural equality of binary trees of binary trees of natural numbers is obtained by instantiating the comparison function with the type of binary trees of nats and the equality predicate for binary trees of nats which we are given.

3 Optional Exercises

3.1 Exercise 03: Functions for Binary Trees with Node Payloads

In this exercise we are provided with a new binary tree type that has payloads only on the nodes. We write the node and leaf counting functions based on this new type definition

3.1.1 Exercise 03: Definition

The provided type.

```
Inductive binary_tree_nat' : Type :=
| Leaf_nat' : binary_tree_nat'
| Node_nat': binary_tree_nat' -> nat -> binary_tree_nat' -> binary_tree_nat'.
```

3.1.2 Exercise 03: Counting Leaves

We adapt the unit test provided in the nodes but swap the payloads onto the nodes and off the leaves.

Unit Test

```
Definition test_number_of_leaves_ex3 (candidate: binary_tree_nat' -> nat) : bool :=
  (Nat.eqb (candidate (Leaf_nat'))
           1)
  &&
  (Nat.eqb (candidate (Node_nat' (Leaf_nat') 1
                                 (Leaf_nat')))
           2)
  .
```

Function We use the leaf counting function provided in the notes but make it compatible with the new type definition.

```
Fixpoint number_of_leaves_ex3_v1 (t : binary_tree_nat') : nat :=
  match t with
    Leaf_nat' =>
    1
  | Node_nat' t1 n t2 =>
    (number_of_leaves_ex3_v1 t1) + (number_of_leaves_ex3_v1 t2)
  end.
```

Testing

```
Compute (test_number_of_leaves_ex3 number_of_leaves_ex3_v1).
```

3.1.3 Exercise 03: Counting Nodes

We constructed a unit test for the nodes. Since this was not provided, we adapted the test for the leaves. We added a new test case to cover situations where a node is succeeded by two nodes. This could identify issues with the node to node traversal logic.

Unit Test

```
Definition test_number_of_nodes_ex3 (candidate: binary_tree_nat' -> nat) : bool :=
  (Nat.eqb (candidate (Leaf_nat'))
           0)
  &&
  (Nat.eqb (candidate (Node_nat' (Leaf_nat') 1
                                 (Leaf_nat')))
           1)
  &&
  (Nat.eqb (candidate (Node_nat' (Node_nat' (Leaf_nat') 3
                                            (Leaf_nat')) 5
                                 (Node_nat' (Leaf_nat') 10
                                            (Leaf_nat'))))
           3)
  .
```

Function We adapt the leaf counting function in the notes but update it to match this new type definition and make the increment occur on the node pattern instead of the leaf. After feedback, we realised that, thinking structurally, we should be using a value construction (i.e. S) instead of a computation (i.e. +1)

```
Fixpoint number_of_nodes_ex3_v1 (t : binary_tree_nat') : nat :=
  match t with
    Leaf_nat' =>
    0
  | Node_nat' t1 n t2 =>
    S ((number_of_nodes_ex3_v1 t1) + (number_of_nodes_ex3_v1 t2))
  end.
```

Testing

```
Compute (test_number_of_nodes_ex3 number_of_nodes_ex3_v1).
```

3.2 Exercise 04: Binary Trees with Payloads in Leaves and Nodes

In this exercise we create a definition for a binary tree that has natural numbers payloads both in the nodes and leaves. We then write node and leaf counting functions that work with this new definition.

3.2.1 Exercise 04: Definition

Referring to the lecture notes, to create a binary tree with payloads both in nodes and leaves we add the nat type to both the Leaf and Node constructors.

```
Inductive binary_tree_ex4 : Type :=
| Leaf : nat -> binary_tree_ex4
| Node : binary_tree_ex4 -> nat -> binary_tree_ex4 -> binary_tree_ex4.
```

3.2.2 Exercise 04: Counting Leaves

Unit Test As usual we prepare our test before writing our function. We modify the test given in the nodes to match the constructors of our new definition.

```
Definition test_number_of_leaves_ex4 (candidate: binary_tree_ex4 -> nat) : bool :=
  (Nat.eqb (candidate (Leaf 1))
           1)
  &&
  (Nat.eqb (candidate (Node (Leaf 1) 1
                            (Leaf 2)))
           2)
  .
```

Function We adapt the leaf counting function in the notes to align with our new binary tree type and its constructors. This means changing the type signature to reflect the name of our new type and changing the pattern match on the Node to include our payload, n.

```
Fixpoint number_of_leaves_ex4_v1 (t : binary_tree_ex4) : nat :=
  match t with
    Leaf n =>
    1
  | Node t1 n t2 =>
    (number_of_leaves_ex4_v1 t1) + (number_of_leaves_ex4_v1 t2)
  end.
```

Testing We pass our function to our unit test, and it works.

```
Compute (test_number_of_leaves_ex4 number_of_leaves_ex4_v1).
```

3.2.3 Exercise 04: Counting Nodes

Unit Test We adapt the unit test we write for Exercise 03, updating it to align with the new type definition in this exercise.

```
Definition test_number_of_nodes_ex4 (candidate: binary_tree_ex4 -> nat) : bool :=
  (Nat.eqb (candidate (Leaf 1))
           0)
  &&
  (Nat.eqb (candidate (Node (Leaf 1) 1
                            (Leaf 2)))
           1)
  &&
  (Nat.eqb (candidate (Node (Node (Leaf 8) 3 (Leaf 10)) 5
                            (Node (Leaf 8) 10 (Leaf 12))))
           3)
  .
```

Function We take our solution for Exercise 03 and update it to fit the new type definition.

```
Fixpoint number_of_nodes_ex4_v1 (t : binary_tree_ex4) : nat :=
  match t with
    Leaf n =>
    0
  | Node t1 n t2 =>
    S ((number_of_nodes_ex4_v1 t1) + (number_of_nodes_ex4_v1 t2))
  end.
```

Testing We test likewise.

```
Compute (test_number_of_nodes_ex4 number_of_nodes_ex4_v1).
```

4 Conclusion

To conclude, we begin to see how concepts such as structural recursion and programming using an accumulator are highly relevant as we progress through our journey in computer science. These links remind us to regularly revise and refresh our memory of previous concepts introduced.

Moreover, we have become more comfortable with programming in Gallina and the relevant notations through these exercises. Beyond that, we are now more aware of how these common functions are actually defined and the way computation takes place. All these will be of great help not only in writing proofs, but also reasoning about them in general.