Functional Programming in Coq

Bobbie Soedirgo, Koo Zhengqun

August 20, 2020

Contents

1.1 Introduction

This assignment gets students familiar with defining fixpoints, specificially terminating fixpoints, where the parameter in the recursive call must be structurally smaller than the given parameter.

As Prof Danvy alluded to by mentioning the inductive hypothesis, the significance of this assignment is that it eases students into proving theorems via mathematical induction in Coq.

All fixpoint definitions are based on proof by mathematical induction. A proof sketch for addition is given in the same section where Prof Danvy mentioned the inductive hypothesis. Similarly, other definitions like multiplication, power, and fibonacci, are all defined based on similar proofs. However, all these proofs have been elided, because they are standard, and can be deduced from the inductive definitions.

1.2 Exercise 1

1.2.1 Addition

Code in this subsection was given by Prof Danvy, whereas comments are mine.

Tests

A suite of unit tests take in a candidate addition function, perform equality tests on naturals, and return true only if all tests return true.

```
Definition test_add (candidate: nat → nat → nat) : bool :=
  (candidate 0 0 =n= 0)
  &&
  (candidate 0 1 =n= 1)
  &&
  (candidate 1 0 =n= 1)
  &&
  (candidate 1 1 =n= 2)
  &&
  (candidate 1 2 =n= 3)
  &&
  (candidate 2 1 =n= 3)
  &&
  (candidate 2 2 =n= 4).
```

A test suite on a recursive function can never have 100% code coverage, because the recursive function has as parameter an inductive type, and this inductive type contains infinitely many elements (precisely all the naturals, but not the fixpoint ω).

This suite does not test over many addition parameters (up to 2), and also does not test all combinations of addition parameters.

However, this suite does test commutativity for some values of addition: 0+1=1=1+0, and 1+2=3=2+1.

Specification

Specification is already given by inductive proof at .

Implementation

The following is not tail-recursive, because there is a constructor over the recursive call.

```
Fixpoint add_v1 (i j : nat) : nat :=
  match i with
  | O ⇒ j
  | S i' ⇒ S (add_v1 i' j)
  end.
```

The following is tail-recursive.

```
Fixpoint add_v2 (i j : nat) : nat :=
  match i with
  | O ⇒ j
  | S i' ⇒ add_v2 i' (S j)
  end.
```

visit in the following is like *add_v1*, and *add_v3* passes both i j to *visit*.

A stark difference compared to *add_v1* is, *visit* recurses only on the parameter i instead of both i j in *add_v1*. This is because *visit* does not need to remember j across recursive calls: this is already handled since recursive calls to *visit* are in the scope of *add_v3*, which defines j.

```
Definition add_v3 (i j : nat) : nat :=
  let fix visit n :=
      match n with
      | O ⇒ j
      | S n' ⇒ S (visit n')
      end
  in visit i.
```

A similar thing can be done for *visit* in the following, where *visit* is like *add_v2*. Unlike *visit* in *add_v3*, here in *add_v4*, *visit* must have two parameters, because *add_v2* is tail-recursive, and accumulate the constructors S in one parameter of the recursive call (the other parameter is the principally decreasing argument).

```
Definition add_v4 (i j : nat) : nat :=
  let fix visit n m :=
      match n with
      | O ⇒ m
      | S n' ⇒ visit n' (S m)
      end
  in visit i j.
```

1.2.2 Multiplication

Tests

This suite of unit tests is defined on the same inputs as *test_add*.

```
Definition test_mul (candidate: nat → nat → nat) : bool :=
  (candidate 0 0 =n= 0)
  &&
  (candidate 0 1 =n= 0)
  &&
  (candidate 1 0 =n= 0)
```

&&
(*candidate* 1 1 =*n*= 1)
&&
(*candidate* 1 2 =*n*= 2)
&&
(*candidate* 2 1 =*n*= 2)
&&
(*candidate* 2 2 =*n*= 4).

A test suite on a recursive function can never have 100% code coverage, and this suite is no exception.

This suite does not test over many addition parameters (up to 2), and also does not test all combinations of addition parameters.

However, this suite does test commutativity for some values of multiplication: 1*2=2=2*1.

Specification

The multiplication function multiplying i with j was defined using the distributivity law of addition and multiplication on naturals: $(k + k') * j = k * j + k' * j$ which says for an i of the form $k+k'$, where (i j k k' : *nat*), multiplication distributes over addition. In particular, when k is S O, the distributivity law states: $((SO) + k') * j = (SO) * j + k' * j$, and since S O is the identity of multiplication, we have $(SO) * j = j$, and so $((SO) + k') * j = j + k' * j$.

The base case is when i is 0, in which case $0j = 0$. Otherwise, $(1 + n)a = a + na$ where $1 + n = i$.

Implementation

```
Fixpoint mul_v1 (i j : nat) : nat :=
  match i with
  | O ⇒ O
  | S i' ⇒ add_v1 j (mul_v1 i' j)
  end.
```

1.2.3 Power

Tests

This suite of unit tests is defined on the same inputs as *test_add*.

```
Definition test_power (candidate: nat → nat → nat) : bool :=
  (candidate 0 0 =n= 1)
  &&
  (candidate 0 1 =n= 0)
  &&
  (candidate 1 0 =n= 1)
```

&&
(*candidate* 1 1 =*n*= 1)
&&
(*candidate* 1 2 =*n*= 1)
&&
(*candidate* 2 1 =*n*= 2)
&&
(*candidate* 2 2 =*n*= 4).

A test suite on a recursive function can never have 100% code coverage, and this suite is no exception.

The unit test 0^0 is mathematically undefined. But the definition of *power_v1* below, has as base case n of 1, because x^0 is 1 for $x \neq 0$.

power_v1 extends the return value 1 to x^0 for $x = 0$, by returning 1 on the structural case where n is O. Coq demands that *power_v1* must be a total function, in particular, the case where $x = 0$ and $n = 0$ must be defined, so we might as well define it by an extension of behavior of the power function.

So, to accomodate this extension of return values, the unit test 0^0 has an expected result of 1.

Specification

The power function taking x to the power n was defined using the distributivity law of multiplication and power on naturals: $x^{k+k'} = x^k * x^{k'}$ which says for an n of the form k+k', where (x n k k' : *nat*), power distributes over multiplication. In particular, when k is S O, the distributivity law states: $x^{(SO)+k'} = x^{SO} * x^{k'}$, and since S O is the identity of exponentiation, we have $x^{SO} = x$, and so $x^{(SO)+k'} = x * x^{k'}$.

The base case is when the power is 0, in which case the result is 1 (with a caveat on $x = 0$). Otherwise $x^{1+m} = x * x^m$ where $1 + m = n$.

Implementation

```
Fixpoint power_v1 (x n : nat) : nat :=
  match n with
  | O ⇒ 1
  | S n' ⇒ mul_v1 x (power_v1 x n')
  end.
```

We can also implement this with tail recursion using an accumulator, like so:

```
Definition power_v2 (x n : nat) : nat :=
  let fix f n acc :=
      match n with
      | O ⇒ acc
      | S n' ⇒ f n' (mul_v1 x acc)
```

```
    end
  in f n 1.
```

1.2.4 Factorial

Tests

This suite of unit tests is defined on the natural numbers from 0 to 6 as inputs.

```
Definition test_fac (candidate: nat → nat) : bool :=
  (candidate 0 =n= 1)
  &&
  (candidate 1 =n= 1)
  &&
  (candidate 2 =n= 2)
  &&
  (candidate 3 =n= 6)
  &&
  (candidate 4 =n= 24)
  &&
  (candidate 5 =n= 120)
  &&
  (candidate 6 =n= 720).
```

A test suite on a recursive function can never have 100% code coverage, and this suite is no exception.

Specification

A factorial function is defined on n with two base cases: O with return value 1, and $S\ O$ with return value 1. For the recursive case, we assume the inductive hypothesis for the recursive call to the factorial function with parameter n' where $n = S\ n$', and with return value $ih = n'!$. Mathematically, $ih = (n-1)!$. Then, we can define $n!$ by multiplying ih with n.

The base case is when $n = 0$, in which case the result is 1. The inductive case is $(1+n)! = (1+n) * n!$.

Implementation

```
Fixpoint fac_v1 (n : nat) : nat :=
  match n with
  | O ⇒ 1
  | S n' ⇒ mul_v1 n (fac_v1 n')
  end.
```

We can also implement this with tail recursion using an accumulator, like so:

```
Definition fac_v2 (n : nat) : nat :=
  let fix f n acc :=
      match n with
      | O ⇒ acc
      | S n' ⇒ f n' (mul_v1 n acc)
      end
  in f n 1.
```

1.2.5 Fibonacci

Tests

This suite of unit tests is defined on the natural numbers from 0 to 6 as inputs.

```
Definition test_fib (candidate: nat → nat) : bool :=
  (candidate 0 =n= 0)
  &&
  (candidate 1 =n= 1)
  &&
  (candidate 2 =n= 1)
  &&
  (candidate 3 =n= 2)
  &&
  (candidate 4 =n= 3)
  &&
  (candidate 5 =n= 5)
  &&
  (candidate 6 =n= 8).
```

A test suite on a recursive function can never have 100% code coverage, and this suite is no exception.

Specification

A fibonacci function on n has base cases 0 with return value 0, and 1 with return value 1, and the recursive case is on $n \geq 2$ which assumes two inductive hypotheses via two recursive calls to the fibonacci function with parameters n - 1 and n - 2, with values *ih_1* and *ih_2* respectively. Mathematically, denoting the fibonacci function by fib, $ih_1 = fib(n-1)$ and $ih_2 = fib(n-2)$. Then, $fib(n)$ can be defined as $fib(n-1) + fib(n-2)$.

Implementation

When attempting to define fibonacci for the first time, I actually did not read part of the lecture notes, and I made the exact same mistake as Brynja from the conversation. Brynja

did:

```
Fixpoint fib_v2 (n : nat) : nat :=
match n with
| 0 ⇒ 0
| 1 ⇒ 1
| S (S n'') ⇒ fib_v2 ??? + fib_v2 n''
end.
```

While I did:

```
Fixpoint fib_v1 (n : nat) : nat :=
match n with
| O ⇒ 0
| S O ⇒ 1
| S (S n') ⇒ add_v1 (fib_v1 n') (fib_v1 (S n'))
end.
```

which is the same modulo commutativity of addition, and assuming *add_v1* is correct.

As alluded to by Prof Danvy, the fibonacci fixpoint cannot be defined by recursing on *S n'*.

This is in spite that it is intuitively obvious that *fib_v1* does decrease on its principal argument *n*: *fib_v1 (S (S n'))* recurses into *fib_v1 n'* and *fib_v1 (S n')*, where *n'* and *S n'* are structurally smaller than *S (S n')*.

The problem is: Coq thinks *fib_v1* does not decrease on the principal argument *n*, and Coq thinks the fixpoint does not terminate.

The corrected *fib_v1* below was actually given by Thomas Tan, but almost exactly matches the solution given by Prof Danvy.

```
Fixpoint fib_v1 (n : nat) : nat :=
  match n with
  | O ⇒ 0
  | S O ⇒ 1
  | S n' ⇒ match n' with
            | O ⇒ 1
            | S n'' ⇒ add_v1 (fib_v1 n') (fib_v1 n'')
            end
  end.
```

1.2.6 Evenness

Tests

This suite of unit tests is defined on the natural numbers from 0 to 6 as inputs.

```
Definition test_even (candidate: nat → bool) : bool :=
  (candidate 0)
```

&&
(*negb* (*candidate* 1))
&&
(*candidate* 2)
&&
(*negb* (*candidate* 3))
&&
(*candidate* 4)
&&
(*negb* (*candidate* 5))
&&
(*candidate* 6).

A test suite on a recursive function can never have 100% code coverage, and this suite is no exception.

Specification

The evenness function takes in parameter n, and has base cases O and $S\ O$ with return values *true* and *false* respectively. The recursive case on $n \geq 2$ assumes the inductive hypothesis via a recursive call on the mathematical value of $n-2$, which returns value *ih*. Split by cases on *ih*. If $n-2$ is even, then n is even. If $n-2$ is odd, then n is odd. Hence, the recursive case of the evenness function is *ih*.

The $S\ O$ case can be merged into the $S\ (S\ n')$ case by defining $S\ n'$ as *negb* (*even_v1 n'*).

Implementation

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

1.2.7 Oddness

Tests

This suite of unit tests is defined on the natural numbers from 0 to 6 as inputs.

```
Definition test_odd (candidate: nat → bool) : bool :=
  (negb (candidate 0))
  &&
  (candidate 1)
  &&
```

(*negb* (*candidate* 2))
&&
(*candidate* 3)
&&
(*negb* (*candidate* 4))
&&
(*candidate* 5)
&&
(*negb* (*candidate* 6)).

A test suite on a recursive function can never have 100% code coverage, and this suite is no exception.

Specification

The oddness function takes in parameter n, and has base cases O and $S\ O$ with return values *false* and *true* respectively. The recursive case on $n \geq 2$ assumes the inductive hypothesis via a recursive call on the mathematical value of $n-2$, which returns value *ih*. Split by cases on *ih*. If $n-2$ is even, then n is even. If $n-2$ is odd, then n is odd. Hence, the recursive case of the oddness function is *ih*.

The $S\ O$ case can be merged into the $S\ (S\ n')$ case by defining $S\ n'$ as *negb* (*odd_v1 n'*).

Implementation

```
Fixpoint odd_v1 (n : nat) : bool :=
  match n with
  | O ⇒ false
  | S n' ⇒ negb (odd_v1 n')
  end.
```

While it's necessary to have 2 base cases for each of *even_v1* and *odd_v1*, we can reduce this to one base case each if we implement them together with mutual recursion:

```
Fixpoint even_v2 (n : nat) : bool :=
  match n with
  | O ⇒ true
  | S n' ⇒ odd_v2 n'
  end with
odd_v2 n :=
  match n with
  | O ⇒ false
  | S n' ⇒ even_v2 n'
  end.
```

1.3 Exercise 2

1.3.1 Number of Nodes

Tests

This suite of unit tests is defined on all trees of height 0 to height 2, modulo the content in the leaves.

```
Definition test_number_of_nodes (candidate: binary_tree → nat) : bool :=
  (candidate (Leaf 1) =n= 0)
  &&
  (candidate (Node (Leaf 1) (Leaf 2)) =n= 1)
  &&
  (candidate (Node (Leaf 0) (Node (Leaf 0) (Leaf 0))) =n= 2)
  &&
  (candidate (Node (Node (Leaf 0) (Leaf 0)) (Leaf 0)) =n= 2)
  &&
  (candidate (Node (Node (Leaf 0) (Leaf 0)) (Node (Leaf 0) (Leaf 0))) =n= 3).
```

A test suite on a recursive function can never have 100% code coverage, and this suite is no exception.

Specification

A tree which is a leaf has 0 nodes. A tree which is a node has two subtrees. By inductive hypotheses, the first subtree has *ih_1* number of nodes, and the second subtree has *ih_2* number of nodes. Then, the total number of nodes in the tree which is a node is $1+ih_1+ih_2$.

Implementation

```
Fixpoint number_of_nodes_v0 (t : binary_tree) : nat :=
  match t with
  | Leaf _ ⇒ O
  | Node t1 t2 ⇒ S (add_v1 (number_of_nodes_v0 t1) (number_of_nodes_v0 t2))
  end.
```

1.3.2 Smallest Leaf

Tests

This suite of unit tests is defined on all trees of height 0 to height 2, modulo the content in the leaves.

```
Definition test_smallest_leaf (candidate: binary_tree → nat) : bool :=
  (candidate (Leaf 1) =n= 1)
```

&&
(*candidate* (*Node* (*Leaf* 1) (*Leaf* 2)) =*n*= 1)
&&
(*candidate* (*Node* (*Leaf* 0) (*Node* (*Leaf* 1) (*Leaf* 1))) =*n*= 0)
&&
(*candidate* (*Node* (*Node* (*Leaf* 1) (*Leaf* 1)) (*Leaf* 0)) =*n*= 0)
&&
(*candidate* (*Node* (*Node* (*Leaf* 1) (*Leaf* 1)) (*Node* (*Leaf* 1) (*Leaf* 0))) =*n*= 0).

A test suite on a recursive function can never have 100% code coverage, and this suite is no exception.

Each unit test has a *Leaf* which contains a *nat* which is the smallest of all *Leaf*s in the tree. Each unit test then asserts that the candidate function acting on the tree returns this *nat*.

Specification

A tree which is a leaf containing *nat* of value n has the smallest leaf of n. A tree which is a node has two subtrees. By inductive hypotheses, the first subtree has the smallest leaf of *ih_1*, and the second subtree has the smallest leaf of *ih_2*. Then, the smallest leaf in the tree is a node is $min(ih_1, ih_2)$.

Implementation

We first define the comparison function $\leq$ on $m\ n : nat$. As base cases, if m is O, then $m \leq n$ is *true*, and if n is O, then $m \leq n$ is *false*. As inductive case, we assume the inductive hypothesis that *ih* is the value of $m' \leq n'$, where $Sm' = m$ and $Sn' = n$. Then, $m' \leq n'$ if and only if $m \leq n$, so the return value of $\leq$ is *ih*.

```
Fixpoint leq_nat (m n : nat) : bool :=
  match m with
  | O ⇒ true
  | S m' ⇒ match n with
            | O ⇒ false
            | S n' ⇒ leq_nat m' n'
            end
  end.
```

We also define convenience functions, each with an `if-else` statement. An `if-else` statement can be thought of as syntactic sugar for a `match`-statement, so these convenience functions can be thought of as structurally defined.

```
Definition min_nat (m n : nat) : nat := if leq_nat m n then m else n.
```

```
Definition max_nat (m n : nat) : nat := if leq_nat m n then n else m.
```

Implementation

The implementation follows the specification to the tee. The implementation uses *min_nat*.

```
Fixpoint smallest_leaf_v0 (t : binary_tree) : nat :=
  match t with
  | Leaf n ⇒ n
  | Node t1 t2 ⇒ min_nat (smallest_leaf_v0 t1) (smallest_leaf_v0 t2)
  end.
```

1.3.3 Weight

Tests

This suite of unit tests is defined on all trees of height 0 to height 2, modulo the content in the leaves.

```
Definition test_weight (candidate: binary_tree → nat) : bool :=
  (candidate (Leaf 1) =n= 1)
  &&
  (candidate (Node (Leaf 1) (Leaf 2)) =n= 3)
  &&
  (candidate (Node (Leaf 0) (Node (Leaf 1) (Leaf 1))) =n= 2)
  &&
  (candidate (Node (Node (Leaf 1) (Leaf 1)) (Leaf 0)) =n= 2)
  &&
  (candidate (Node (Node (Leaf 1) (Leaf 1)) (Node (Leaf 1) (Leaf 1))) =n= 4).
```

A test suite on a recursive function can never have 100% code coverage, and this suite is no exception.

Specification

A tree which is a leaf containing *nat* of value n has a weight of n. A tree which is a node has two subtrees. By inductive hypotheses, the first subtree has a weight of *ih_1*, and the second subtree a weight of *ih_2*. Then, the weight of the tree which is a node is $ih_1 + ih_2$.

Implementation

```
Fixpoint weight_v0 (t : binary_tree) : nat :=
  match t with
  | Leaf n ⇒ n
  | Node t1 t2 ⇒ weight_v0 t1 + weight_v0 t2
  end.
```

1.3.4 Height

Tests

This suite of unit tests is defined on all trees of height 0 to height 2, modulo the content in the leaves.

```
Definition test_height (candidate: binary_tree → nat) : bool :=
  (candidate (Leaf 1) =n= 0)
  &&
  (candidate (Node (Leaf 1) (Leaf 2)) =n= 1)
  &&
  (candidate (Node (Leaf 0) (Node (Leaf 1) (Leaf 1))) =n= 2)
  &&
  (candidate (Node (Node (Leaf 1) (Leaf 1)) (Leaf 0)) =n= 2)
  &&
  (candidate (Node (Node (Leaf 1) (Leaf 1)) (Node (Leaf 1) (Leaf 1))) =n= 2).
```

A test suite on a recursive function can never have 100% code coverage, and this suite is no exception.

Specification

A tree which is a leaf containing *nat* of value n has a height of 0. A tree which is a node has two subtrees. By inductive hypotheses, the first subtree has a height of *ih_1*, and the second subtree a height of *ih_2*. Then, the height of the tree which is a node is $1 + max(ih_1 + ih_2)$.

Implementation

```
Fixpoint height_v0 (t : binary_tree) : nat :=
  match t with
  | Leaf _ ⇒ 0
  | Node t1 t2 ⇒ S (max_nat (height_v0 t1) (height_v0 t2))
  end.
```

1.3.5 Mirror

Tests

This suite of unit tests is defined on all trees of height 0 to height 2, modulo the content in the leaves.

```
Definition test_mirror (candidate: binary_tree → binary_tree) : bool :=
  (candidate (Leaf 1) =bt= (Leaf 1))
  &&
  (candidate (Node (Leaf 1) (Leaf 2)) =bt= (Node (Leaf 2) (Leaf 1)))
```

```
  &&
  (let t := (Node (Leaf 1) (Leaf 2))
   in candidate (candidate t) =bt= t)
  &&
  (candidate (Node (Leaf 0) (Node (Leaf 1) (Leaf 1))) =bt= (Node (Node (Leaf 1) (Leaf
1)) (Leaf 0)))
  &&
  (let t := (Node (Node (Leaf 2) (Leaf 1)) (Leaf 0))
   in candidate (candidate t) =bt= t).
```

A test suite on a recursive function can never have 100% code coverage, and this suite is no exception.

Specification

The mirror function acting on a tree which is a leaf containing *nat* of value n is the identity function. A tree which is a node has two subtrees. By inductive hypotheses, the first subtree has a mirror of *ih_1*, and the second subtree a mirror of *ih_2*. Then, the mirror of the tree which is a node is a node which has *ih_1* as the first subtree, and *ih_2* as the second subtree.

Implementation

```
Fixpoint mirror_v0 (t : binary_tree) : binary_tree :=
  match t with
  | Leaf n ⇒ Leaf n
  | Node t1 t2 ⇒ Node (mirror_v0 t2) (mirror_v0 t1)
  end.
```

1.4 Conclusion

Implementing fixpoints is straightforward: just follow the proof by structural induction.

However, we must take care to make sure that the recursive call only has parameters that are structurally smaller than the given parameters, see *fib_v1*. In the context of Coq, we must make sure Coq knows this. Performing structural induction through `match`-statements convinces Coq that there is a principally decreasing argument, and thus that the fixpoint is terminating.

The *S O* case has been removed not only from the *even_v1* and *odd_v1*, but also removed from *mul_v1*, *power_v1*, *fac_v1* and *fib_v1*. The reasoning is as follows:

- Bobbie: So I want to ask this before I continue: is there anything wrong with omitting the *S O* case for mul, power, fac, and others? I haven't read until the end so I might be missing a punchline

- Zhengqun: Good observation, I did not consider that! I've tried removing $S\ O$ from mul to fib, and all the unit tests pass. So I think $S\ O$ is being captured by the inductive case $S\ n'$, where n' is O, and this is well-defined because we already defined the function on O.