Lectures Notes on Functional Programming

Software Verification

Nova School of Science and Technology
Mário Pereira `mjp.pereira@fct.unl.pt`

Version of September 8, 2025

Functional programming is a paradigm as old as the discipline of programming itself. LISP was the first non-trivial functional language, while today one can find many incarnations of the functional approach. One can cite OCaml, Haskell, Erlang, F# (only to name a few) as well-succeed and established industrial-scale functional languages. In the first part of our course we will be using the Coq proof assistant, an interactive verification framework that can be used to reason about pure functional programs. Coq is itself written in a functional language (*i.e.*, OCaml) and it features a programming language, Galllina, that allows one to write and *specify* functional code.

During this lecture we make our first contact with deductive program verification, through the lens of functional programming. Proving that a functional program is correct actually boils down to pure mathematical reasoning. This becomes crystal clear when one understands that functional programming is, indeed, about *functions* and treating functions as first-class objects. This connection between mathematics and computer programming is even more evident if one completely avoids *mutable state*. This might seem contra-intuitive at first sight (we have all learned about assignment, loops, arrays, etc., during our first computer programming course), however *pure* functional programming bears important advantages when it comes to program correctness, parallelism, and modularity. In the end, a functional program is just a set of mathematical functions, expecting arguments as input and producing some value as output. A particular execution is just functions composition. Showing that such functions obey some logical specification is just a matter of equational reasoning.

By the end of the lecture, I hope you are convinced and adopt the motto

Programmatic Thinking is Mathematical Thinking

The contents of this document is strongly inspired by notes from Michael Erdmann and Frank Pfenning, and Bernardo Toninho.

1 A Prime on Functional Programming

Before we even try do define what functional programming is, it might be interesting to take a step back and asks ourselves *what is programming?* This

may sound odd, since we have been programming for quite some years (myself, I am a programmer since the age of 15), and we are surely convicted that we can handle programming. But when one tries to *fundamentally* think about programming, what does come to the mind? I say, when thinking truly about programming we can even forget about computers! For me, programming is inherently the process of describing a *logical* and *systematic* approach to solve a particular problem or perform some operation. Following a French saying, by Xavier Leroy, that I really like:

> *Un peu de programmation éloigne de la logique mathématique ; beaucoup de programmation y ramène.*

(Homework: search for a good translation for this quote).

Programming is actually a communicative act. Good communication exists, so what is good programming? *Functional programming* is an improvement on our ability to communicate as programmers. Good programming should be descriptive. Good programming should be modular. Logically distinct parts should be separated, for separate maintenance and reuse. You should be able to think about a single area of a codebase without needing to concern yourself with unrelated logic. Good programming should be maintainable. Programs should be written with future maintainability in mind. This goes hand-in-hand with descriptivity and modularity, but code which is written in a declarative style, and has future expansion in mind, will be easiest to maintain.

I expect the reader to be familiar with some form of *imperative programming*, for instance, using the Java language. That is absolutely fine, Java is a great language that serves a very good purpose (here goes another quote: "*There are no good languages; only good programmers*"). However, let us take an analogy: suppose you are a master chef at a 5-star restaurant. An imperative program is like a fully crowded kitchen with no rules:

- Everyone uses the same ingredients and the same cookware.
- Each cook is an individual actor that can mess with the others, if care is not taken. The health of the kitchen depends on each individual chef.

On the other side, a *functional program* is like a kitchen where each cook has their own working space:

- Everyone has their own pots, pans, and ingredients, and they only share things when they finish producing their individual parts.
- Each cook only interacts when sharing finished results. This means it is impossible for one cook's actions to mess up another's cooking.

(Please, do not think I am saying imperative programming is wrong; I am only trying to make a point for the benefits of functional programming over other paradigms).

The attentive reader might have guessed that, in the above analogy, the mess in imperative programming is (potentially) caused by the use of *mutable state*. In stateful programs, we use commands like `x := 2` to change the world, to be one where `x + x` is `4`. This does not stop another part of the program from changing that later! In functional programs, on the other hand, we apply simplifying rules to expressions like `2 + 2`, to obtain the value of `4`. These

expressions are disjoint, in that evaluation of one expression is unrelated to the evaluation of another.

As our baseline, during the first part of course we will be mainly interested on *pure functional programs*, *i.e.*, no mutable wickedness of any sort is allowed. On a more formal definition:

Definition 1.1 (Pure functional program)**.** A program is *functionally pure* if it does not have any observable side effects (*e.g.* reference assignment), and always returns the same outputs, given the same inputs.

The Coq proof assistant and the Gallina language will be our tools of choice. However, it is also safe to think in terms of OCaml programs, as Coq and OCaml share a common, pure, functional language subset. So, feel free to use the OCaml language as the vehicle to understand and explore functional programming.

A large amount of problems in computer science are of a pure nature. This means that they give the same outputs for the same inputs. An important motivation behind functional programming will end up being that we should prefer to solve pure problems with pure components. In other words, do not introduce state when it is not necessary.

2 Notation

For the sack of simplicity, we forget about the physical limits of a von Neumann machine, *e.g.*, memory or finite numbers representation. This means we can safely use a Gallina or OCaml object as its underlying mathematical representation. For instance, an OCaml integer value is actually treated as a mathematical integer, and not a 63-bit machine integer.

Henceforth, we will only care about *well-typed expressions* as these safely *evaluate* into a value, or in other words, *produce/compute* a final result. A *value* is a special kind of expression that evaluates to itself. A well-typed expression evaluates via zero or more reductions. Intuitively, a reduction is a simplification of an expression. For instance, the expression `2 * 3 * 7` evaluates to value `42`.

We write e for arbitrary expressions and v for values, and adopt the following notation:

$e \Downarrow v$	means expression e evaluates to value v
$e \Longrightarrow e'$	means expression e reduces to expression e' in a single step
$e \Longrightarrow^{k} e'$	means expression e reduces to expression e' in exactly k steps
$e \Longrightarrow^{*} e'$	means expression e reduces to expression e' in 0 or more steps

Our notion of *step* in the operational semantics is kept abstract and does not necessarily coincide with the actual operations performed by an implementation. Since our main concern is proving correctness rather than proving complexity bounds, the concrete number of steps is mostly irrelevant and we will frequently use the notation $e \Longrightarrow e'$ for reduction. Evaluation and reduction are related in the sense that if $e \Downarrow v$ then either e is equal to v already or $e \Longrightarrow e_1 \Longrightarrow \cdots \Longrightarrow v$ (and vice-versa). Values are special forms of expression in that they evaluate to themselves "in 0 steps". For any value v, there is no expression e such that $v \Longrightarrow e$. We assume that function application reduces in a single-step to its function body with the formal parameters substituted for the function arguments, as in math. We will assume that all primitive operations such as

addition, subtraction, conditional branching, etc., all reduce in a single-step when acting on values (*i.e.*, $3 + 2 \Longrightarrow 5$ and `if true then` e_1 `else` $e_2 \Longrightarrow e_1$).

3 Extensional Equivalence

Definition 3.1 (Extensional Equivalence)**.** In a pure functional language, we say that two expressions e and e' of the same (non-function) type are *equivalent*, written $e \equiv e'$, if one of the following occurs:

(a) the evaluation of e produces (*i.e.*, terminates with) the same value as the evaluation of e';

(b) the evaluation of e raises the same exception as does evaluation of e';

(c) the evaluation of e and of e' both loop forever.

In the first part of our course, we will focus only on *provably* terminating programs.

Note that the notion of extensional equivalence is potentially distinct from mathematical equality, written $e = e'$.

Example 3.1. The following two expressions are equivalent:

- `2 * 3 * 7`
- `42`

Example 3.2. Perhaps the "most famous" form of extensional equivalence all programmers know about:

- `if b then true else false`
- `b`,

where `b` is some arbitrary Boolean value.

4 Referential Transparency

Definition 4.1 (Referential Transparency)**.** In any pure functional programming language, an expression e can be replaced by another expression e' whenever $e \equiv e'$ holds, without affecting the final output of the program. This is called the *referential transparency* principle.

In particular, any function call can be replace by its body, with the proper formal arguments instantiated, or even by some optimization of the said function. This induces a powerful tool to reason about functional programs: this is exactly what one in mathematics calls the substitution of "equals for equals", a notion so familiar that one does it all the time without any concern. While this may sound obvious, this principle is extremely useful in practice, and it can lend support to program optimization or simplification steps that help develop better programs.

Example 4.1. The following OCaml program

```
let rec fact x =                                                    OCaml
  if x <= 1 then 1 else x * fact (x-1)

let result = fact 5
```

is equivalent to this one:

```
let result = 120                                                    OCaml
```

For pure functional programs, such as those we will be reasoning about, because evaluation causes no side-effects, if we evaluate an expression twice, we obtain the same result. Moreover, the relative order in which one evaluates (non-overlapping) sub-expressions of a program makes no difference to the value of the program, so one may in principle use parallel evaluation strategies to speed up code while being sure that this does not affect the final value.

Example 4.2. In the following OCaml program

```
                                                                    OCaml
let rec fib x =
  if x = 1 then 1
  else if x = 2 then 1
  else fib (x-1) + fib (x-2)

let res1 = fib 7 + fib 7
let res2 = fib 7 + fib 7
```

both variables `res1` and `res2` hold the value `42`.

Example 4.3. In the following OCaml program

```
let counter =                                                       OCaml
  let r = ref 0 in
  fun () -> r := !r + 1; r

let res1 = counter ()
let res2 = counter ()
```

variables `res1` and `res2` hold distinct values by the end of execution. Can you tell why?

5 Proof by Simplification

Leveraging on the definitions and principles introduced in previous sections, we can now begin our journey on proving behavioral properties about functional programs. The first proof strategy we introduce is called *proof by simplification* and builds directly on the notion of referential transparency. Let us take a concrete example to begin with. Consider the following OCaml function:

```
let andb (b1: bool) (b2: bool) : bool =                    OCaml
  if b1 then b2
  else false
```

We can prove a simple property about it, in the form of a *Lemma*:

Lemma 5.1 (UNFOLD_andb)**.** $\forall$ b$_1$ b$_2$*: bool,*
andb b$_1$ b$_2$ = *if* b$_1$ *then* b$_2$ *else false.*

Before we even dig into some proof of the above statement, let us break our lemma into pieces. The $\forall$ b$_1$ b$_2$ part specifies that we are trying to prove some *universal* property about any two Boolean values b$_1$ and b$_2$. The $\forall$ signs reads "for all" and is normally referred to as the *universal quantifier* (as opposed to *existential quantifier* that we shall encounter later on the course). Next, the comma sign reads "then", which indicates we are now going to state some properties about the arguments of this lemma, *i.e.*, we are now in position to provide the *body* of the lemma. Finally, what the body says is a simple *unfold* property about the `andb` function. In other words, and thanks to referential transparency, the lemma says we can exchange the call `andb` b$_1$ b$_2$ for its body. This might seem obvious and not very interesting at first, but remember this is only possible because we are in the setting of functional programming.

Now, the proof of the above statement is extremely easy:

Proof.

$$\begin{array}{ll} & \texttt{andb b}_1\ \texttt{b}_2 = \texttt{if b}_1\ \texttt{then b}_2\ \texttt{else false} \\ \Longrightarrow & \texttt{if b}_1\ \texttt{then b}_2\ \texttt{else false} = \texttt{if b}_1\ \texttt{then b}_2\ \texttt{else false} \end{array}$$

□

We prove this statement directly, relying on function evaluation and the referential transparency principle. The proof ends because we have the same term on both sides of the equation. Generally, during the proof of a lemma, whenever we end up with an equation of the form $e = e$, for any arbitrary expression e, we say the proof is complete by *reflexivity*. **Note for the future**: although the above lemma might seem useless, it actually represents an important class of results about function calls, denoted as *unfold* principle. In practice, it allows one to take a *single* step in the evaluation of the function call. We will face many situations where Coq tries to do a little bit more computation than just unfolding the function call, which might lead our proof to a state that is "out of our control".

We can do a bit more with proofs by simplification than just unfolding function calls. Consider the following statement:

Lemma 5.2 (andb_true_b)**.** $\forall$ b : bool*, andb true* b = b*.*

The proof is, again, not very challenging:

Proof.

$$\begin{array}{lll} & \texttt{andb true b} = \texttt{b} & \\ \Longrightarrow & \texttt{if true then b else false} = \texttt{b} & \text{(by RT)} \\ \Longrightarrow & \texttt{b} = \texttt{b} & \text{(by simplification)} \end{array}$$

□

In this proof, we need to take an extra step and do a bit more than just evaluating the function call. In this case, after unfolding the definition of `andb`, we perform a *simplification* step: by the definition of `andb`, when the first argument of this function is equal to `true` it evaluates to its second argument. More generally, every term of the form `if true then` $\mathtt{b}_1$ `else` $\mathtt{b}_2$ reduces to $\mathtt{b}_1$.

The converse lemma is also easily provable:

Lemma 5.3 (andb_false_b). $\forall\, \mathtt{b} : \mathtt{bool}$, *andb false* $\mathtt{b} =$ *false*.

Exercise 1. Prove Lemma 5.3. □

As a final set of examples, consider the following definition for natural numbers, using Peano arithmetic:

```
type nat = O | S nat
```
OCaml

On top of such definition, we can provide a handful of functions to manipulate natural numbers. For instance:

```
let rec plus (n: nat) (m: nat) : nat =
  match n with
  | O -> m
  | S n' -> S (plus n' m)

let rec mult (n: nat) (m: nat) : nat =
  match n with
  | O -> O
  | S n' -> plus m (mult n' m)
```
OCaml

We can prove the following results:

Lemma 5.4 (plus_O_n). $\forall\, \mathtt{n} : \mathtt{nat}$, *plus O* $\mathtt{n} = \mathtt{n}$.

and also

Lemma 5.5 (mult_O_l). $\forall\, \mathtt{n}$, *mult O* $\mathtt{n} =$ *O*.

The proof of Lemma 5.4 is as follows:

Proof.

$$\begin{array}{rll} & \mathtt{plus\ O\ n} = \mathtt{n} & \\ \Longrightarrow^* & \mathtt{n} = \mathtt{n} & \text{(by simplification)} \end{array}$$

□

During this proof, we perform two steps of simplification simultaneously: we exchange function call for its body and simplify *pattern-matching* into the first branch. This is exactly the behavior one can expect when calling the `simpl` tactic in Coq.

Exercise 2. Prove Lemma 5.5. □

6 Proof by Rewriting

Simplification is, indeed, a useful mechanism. It allows one to prove statements using nothing but pure (symbolic) execution of functional expressions. However, simplification is pointless when it comes to prove properties about arbitrary values. Take, for instance, the following statement:

Lemma 6.1 (plus_id). $\forall n\ m : \mathtt{nat}$,
$n = m \rightarrow$
$\mathit{plus}\ n\ n = \mathit{plus}\ m\ m$.

Before we even begin proving this Lemma, let us break it into pieces:

- first, the already familiar universal quantification of variables;
- then, on the left hand-side of symbol $\rightarrow$, the *premise* that n is actually the same value as m;
- finally, the *conclusion* or *goal* of the Lemma that states plus n n = plus m m.

The body of the above statement is of the form $P \rightarrow Q$, for some propositions P and Q. This formula reads "P implies Q". Or, from a more practical point of view, "by assuming P, one must be able to prove Q". When writing down our proofs, this corresponds to moving hypothesis P into the *proof context* and from there on we are able to use that fact to prove the goal. From a pure logical approach, this is exactly the rule for *implication introduction* from Natural Deduction, normally expressed as follows:

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \rightarrow Q}$$

Now, back to business. Since both n and m are arbitrary numbers, simplification cannot help us out here. What one must do is to explore the equality $n = m$ in order to replace n by m in the goal and finish the proof. This proceeds as follows:

Proof. - From hypothesis:

$$n = m \tag{H}$$

- By rewriting hypothesis H in the goal we get

$$m + m = m + m$$

- The proof is done by reflexivity.

□

Other than simply using hypotheses from the context, rewriting finds its main strength when it comes to use other lemmas to finish one's proof. Let us assume the following two results, which we shall prove later:

Lemma 6.2 (mult_n_0). $\forall n : \mathtt{nat}$, $0 = \mathit{mult}\ n\ 0$

Lemma 6.3 (mult_n_Sm). $\forall n\ m : \mathtt{nat}$, $\mathit{plus}\ (\mathit{mult}\ n\ m)\ n = \mathtt{mult}\ n\ (S\ m)$.

Now, we can use this as an *auxiliary* lemma to prove the following:

Lemma 6.4 (mult_n_1)**.** $\forall n : \mathtt{nat}$, *mult* n (S O) $= n$.

Proof. - We use lemma 6.3 to rewrite the goal as follows:

$$\mathtt{plus}\ (\mathtt{mult}\ n\ \mathtt{O})\ n = n$$

- Now, using lemma 6.2, we write the sub-expression that is inside the parentheses:

$$\mathtt{plus}\ \mathtt{O}\ n = n$$

- Finally, this is exactly the statement of lemma 5.4. Either we use it to rewrite our current goal or proceed by simplification.

□

7 Proof by Case Analysis

Consider the following definition for deciding equality on natural numbers:

```
let rec eqb (n: nat) (m: nat) : bool =                    OCaml
  match n with
  | O -> begin match m with
         | O -> true
         | S _ -> false
         end
  | S n' -> begin match m with
            | O -> false
            | S m' -> eqb n' m'
            end
```

Let us try to prove an apparently obvious result about this function:

Lemma 7.1 (plus_1_neq_0)**.** $\forall n : \mathtt{nat}$, *eqb (plus* n (S O)) O $=$ false.

Obviously, the successor of any natural number is always different from zero. But, we just cannot expect to prove the above statement by simplification: the definition of `eqb` pattern-matches on its first argument, which we now nothing about in the case of this lemma.

It is often the case, when proving functional programs correct, that one needs to proceed by *case analysis*. We have already encountered two forms of conditional branching in functional programming: on one hand, classical `if-then-else` expressions, which allows one to check against a Boolean value; on the other hand, pattern-matching, which allows one to check against the *structure* of a value. Pattern-matching has been a distinctive feature of functional languages for many years. It is now being slowly adopted by other mainstream languages from the imperative realm.

In formal proofs, case analysis boils down to considering all the possible forms (aka, *shape* or *structure*) a value can take. In the case of a natural number n , either $n = \mathtt{O}$, or $n = \mathtt{S}\ n'$, for some n'. With such an information at hand, simplification is now possible. The proof of lemma 7.1 proceeds as follows:

Proof. By case analysis on the structure of n.

- Sub-case [n = O]:
  - We have:

$$\begin{array}{rll} & \texttt{eqb (plus O (S O)) O} = \texttt{false} & \\ \Longrightarrow & \texttt{eqb (S O) O} = \texttt{false} & \text{(by simplification)} \\ \Longrightarrow^* & \texttt{false} = \texttt{false} & \text{(by simplification)} \end{array}$$

- Sub-case [n = S n′]:
  - We have:

$$\begin{array}{rll} & \texttt{eqb (plus (S n') (S O)) O} = \texttt{false} & \\ \Longrightarrow & \texttt{eqb (S (plus (n' (S O)))) O} = \texttt{false} & \text{(by simplification)} \\ \Longrightarrow^* & \texttt{false} = \texttt{false} & \text{(by simplification)} \end{array}$$

□

For the second sub-case, the first simplification simply accounts for the definition of function `plus`. Then, the extra steps of computation explore the fact that, when the first argument of `eqb` is of the form `S n′`, we get into the second branch of the pattern-matching. On that branch, if the second argument of `eqb` is `O`, then this function immediately returns `false`.

The case analysis approach can be used with any *inductively*-defined datatype. Although we have, so far, skipped it and used it as builtin values, the common definition for Boolean values is as follows:

```
type bool = True | False                                        OCaml
```

The literals `true` and `false` are just syntactic-sugar for the corresponding *type constructor*. So, we can indeed perform case analysis on Boolean values. Consider the following definition:

```
let negb (b: bool) : bool =                                     OCaml
  if b then false else true
```

The following lemma is valid:

Lemma 7.2 (negb_Involutive)**.** ∀ b : bool, *negb (negb* b*)* = b.

Proof. By case analysis on b.

- Sub-case [b = true]:

$$\begin{array}{rll} & \texttt{negb (negb true)} = \texttt{true} & \\ \Longrightarrow^* & \texttt{true} = \texttt{true} & \text{(by simplification)} \end{array}$$

- Sub-case [b = false]:

$$\begin{array}{rll} & \texttt{negb (negb false)} = \texttt{false} & \\ \Longrightarrow^* & \texttt{false} = \texttt{false} & \text{(by simplification)} \end{array}$$

□

Exercise 3. Prove the following statement:
∀ b c : bool, `andb b c` = `andb c b`. □

Exercise 4. Prove the following statement:
∀ b : bool, `if b then true else false`. □