Induction

15-150: Principles of Functional Programming – Lecture 02

Giselle Reis

What is functional programming?

Functional programming is a different programming paradigm. A programming paradigm is nothing more than a "way" of programming. Other programming paradigms are imperative (e.g. C, Python) and object oriented (e.g. Java, C++). As much as we like to use languages to illustrate each paradigm, this is a little bit misleading, since a paradigm is much more than a programming language, and many times one can use different paradigms in one language (e.g., Scala has both functional and object oriented features). Nevertheless, it is important to know what is the "flavor" of a language, since its design determines how well each construct works during execution time (e.g., python is not so efficient with recursions). Examples of functional programming languages include: SML, OCaml, Haskell, Lisp (used on emacs scripts), F#, Scala, among others.

So what are the features of this programming paradigm? As the name already gives away, it is all about functions. Now, we have heard about functions in at least two contexts: (1) programming functions, the small pieces of code that are used in many places and it is convenient to encapsulate them somewhere; (2) mathematical functions, a mapping from one set (domain) into another (co-domain). In functional programming, these two notions of functions are the same: **programming functions are mathematical functions (and vice versa)**.

To see how this is different from the way you have been programming so far, think of the differences between these two different functions. A mathematical function is a well-defined relation between two sets which, given an input, will return an output. It will *always* return an output and will *do nothing else*. In contrast, a programming function *may or may not* return a value, and it might do many other things in between calling and returning, such as printing on the screen, changing values of variables, throwing exceptions, etc (which are called *side-effects*). In functional programming, our functions will be as close as possible to a mathematical function.

Now, why would we want to do that? For starters, functions without side-effects are much easier to reason about: their return value does not depend on an internal state of the variables, for example. This will allow more straightforward proofs about your code, one that does not have to reason line-by-line, but about the concept. Additionally, having a programming language closer to mathematics will allow us to exploit the underlying mathematical structure of a problem to solve it.

Remark 1. *You might be wondering if everything that is done in a programming language can be done with mathematical functions. Well, if you are thinking only of "simple" functions (such as $f(x) = x^2$), then certainly not. Even going a bit further, to primitive recursive functions, will not get us there (they comprise the set of all computable functions that halt). But the set of* general *recursive functions (also called μ-recursive) is Turing complete. Another mathematical interpretation of computable functions is called λ-calculus. The equivalence (in terms of computational power) of these concepts is called the* Church-Turing thesis.

Induction

You might have heard about induction before as a proof method in mathematics. Induction is also the main method we will use in this course to prove properties about our code. To do this, one must *really* understand how induction works. Let's dissect it.

Inductive domains

The first thing to decide when developing a proof by induction is *what* we are inducting on. The only reason we can perform induction on a mathematical object or structure is because this structure is *inductively defined*. An inductive definition of a domain consists on some **basic building blocks** plus **simple construction rules** (optional).

Let's look at it with an example. Probably most of the induction proofs you have done so far induce on a natural number. Indeed, the set of natural numbers can be inductively defined[1]:

$$\begin{aligned} &0 \in \mathbb{N} && \longleftarrow \text{Basic building block} \\ &n+1 \in \mathbb{N} \text{ iff } n \in \mathbb{N} && \longleftarrow \text{Simple construction rule (i.e., successor)} \end{aligned}$$

Many other structures can be inductively defined. For example, arithmetic expressions (abbreviated here by exp):

$$\begin{aligned} &n \in \text{exp iff } n \in \mathbb{N} && \longleftarrow \text{Basic building block} \\ &n+m \in \text{exp iff } n, m \in \text{exp} && \longleftarrow \text{Simple} \\ &n-m \in \text{exp iff } n, m \in \text{exp} && \longleftarrow \text{construction} \\ &n*m \in \text{exp iff } n, m \in \text{exp} && \longleftarrow \text{rules} \\ &n/m \in \text{exp iff } n, m \in \text{exp} && \longleftarrow \text{(this one also!)} \end{aligned}$$

Inductive domains may have one or more building blocks, and zero or more construction rules. They may be finite (e.g. boolean), or infinite (e.g. natural numbers).

Inductive function definition

Inductive functions can be defined over inductive domains. Like the domains, these functions will have base and inductive cases. They do not need to follow strictly the construction of the domain, although many times they will. This means that the base case will consider the basic building blocks and the inductive cases will take care of the construction rules. If the function does not take into account all construction rules, it will not be total.

A simple and well-known inductive function is factorial:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * n'! & \text{if } n = n' + 1 \end{cases}$$

Notice how its cases cover all the means of constructing natural numbers: there is a case for 0 and a case for numbers which are successors of something (i.e., $n = n' + 1$). Therefore, we can safely say that this function is total and its domain is $\mathbb{N}$.

Note how we only used $+1$, since this is the domain's construction rule, and avoided using subtraction.

We will use inductively defined functions mainly for two things:

1. **Computation:** the similarity of inductively defined and recursive functions is not accidental. Recursion is the programming mechanism used to implement inductive functions[2]. Since we are computing things, we will also learn how to analyze the complexity of our implementations.

2. **Proofs:** a big advantage of inductively defined functions is that we can straightforwardly prove properties about these functions. This means we will be able to straightforwardly prove properties about our code[3].

[1] iff ≡ if and only if

[2] Analogously, inductive domains are implemented via recursive datatypes. We will see this later.

[3] Proofs can be done on paper for the course. If you are curious on how to *program* such proofs, check out http://logic.qatar.cmu.edu/silkie.

Induction proofs

Since we are using inductively defined structures (domains and functions), nothing more natural than reasoning about them via proofs by induction. Let's prove something very simple about the factorial function above to understand exactly all the elements of an induction proof.

State the property that you want to prove.

Theorem 1. $\forall n \in \mathbb{N},\ n! \geq 1$.

Proof.

Describe the proof strategy used. Most of the times it will be some kind of induction.

We will prove this by structural induction on n.

Briefly explain the structure of the proof.

There are two cases: $n = 0$ and $n = n' + 1$.

Show the assumptions used on the base case.

Base case: $n = 0$

Re-state the property for the base case

To show: $0! \geq 1$

Develop the proof

Proof:

$$\begin{aligned} 0! &= 1 && \text{(by def)} \\ 0! &\geq 1 && \text{(by math)} \end{aligned}$$

Show the assumptions used on the inductive case.

Inductive case: $n = n' + 1$

Re-state the property for the inductive case.

To show: $(n' + 1)! \geq 1$

State the induction hypothesis.

IH: $n'! \geq 1$

Develop the proof

Proof:

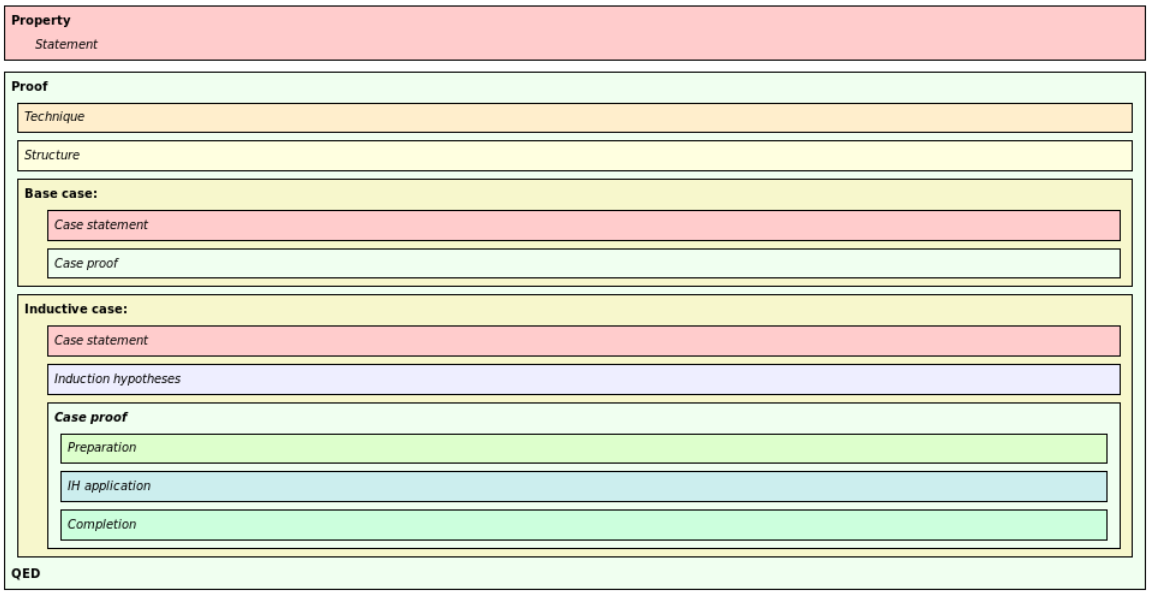
$$\begin{aligned} (n' + 1)! &= (n' + 1) * n'! && \text{(by def)} \\ (n' + 1)! &\geq (n' + 1) * 1 && \text{(by IH)} \\ (n' + 1)! &\geq 1 && \text{(by math)} \end{aligned}$$

□

In general, a proof will have as many cases as the inductive function definition.

Other types of induction exist. For example strong/complete induction states that a property holds for every smaller element, not only the immediate predecessor. Structural induction relies on the inductive hypothesis on structurally smaller objects (e.g. the arithmetic expression $6 * 7$ is structurally smaller than $(2 * 20) + 2$). This kind of induction is used a lot in computer science, since most of the time we are dealing with data-*structures*.

For the course of this semester we will develop lots of proofs by induction, therefore, the following template might be helpful (credits to Prof. Iliano Cervesato).



Exercise: Use the template to prove $\forall n \geq 4,\ n! > 2^n$.

The meta-theory of induction (extra)

During the course we use induction as a tool to prove properties about programs. Let's take a look at the internal workings of this tool and try to understand the reasoning behind a proof by induction.

The simplest type of induction is on natural numbers. Suppose we are trying to prove some property P. The first thing we do is to prove it for zero, i.e., $P(0)$. Then we assume that it holds *for some generic* k and show it is also the case for $k+1$, i.e., $\forall k.P(k) \to P(k+1)$. We conclude thus that P holds for every number. Logically this is expressed by the following formula:

$$P(0) \land \big(\forall k.(P(k) \to P(k+1))\big) \to \forall x.P(x)$$

Take a moment to read this formula again and convince yourself that it represents the induction principle for natural numbers.

Now, we have seen that natural numbers are an inductive domain which can be described by a base case z (for zero) and the inductive case s (for the successor function). We can use this to increase the level of abstraction of the induction principle:

$$P(z) \land \big(\forall k.(P(k) \to P(s(k)))\big) \to \forall x.P(x)$$

Now suppose we change out inductive domain to lists. How does the induction principle look like? Well, lists are described by a base case [] and an inductive case ::, therefore:

$$P([]) \land \big(\forall k.(P(k) \to \forall h.P(h :: k))\big) \to \forall x.P(x)$$

This is almost the same, except that we have to generalize h, the head of the list, by saying that $P(h :: k)$ holds for *any* h.

Indeed, a proof by (structural) induction on lists involves the proof of the base case $P([])$, and the inductive case $\forall h.P(h :: k)$ from the induction hypothesis $P(k)$.

Can we do this for any inductive domain? Of course!

Trying to be more general, let's imagine an inductive domain with n base cases: $b_1, ..., b_n$ and m type constructors $c_1, ..., c_m$ (taking any number of arguments). The induction principle will be roughly:

$P(b_1) \land ... \land P(b_n)$	base cases
$\big(\forall y_1, ..., y_k.P(y_1) \land ... \land P(y_k)$	IHs (as many as needed for the constructors)
$\to P(c_1(...)) \land P(c_m(...))\big)$	inductive cases (possibly using the y_is in the arguments)
$\to \forall x.P(x)$	conclusion: P holds for every element in the domain

We are skimming over some details, but that is the general idea. When doing proofs by induction, try to make your inductive domain fit this schema and the base cases, IHs and inductive cases will pop-out.