Programming Paradigms

(Lectures on High-performance Computing for Economists VII)

Jesús Fernández-Villaverde,\textsuperscript{1} Pablo Guerrón,\textsuperscript{2} and David Zarruk Valencia\textsuperscript{3}

September 14, 2020

\textsuperscript{1}University of Pennsylvania

\textsuperscript{2}Boston College

\textsuperscript{3}ITAM
Programming Approaches
A paradigm is the preferred approach to programming that a language supports.

Main paradigms in scientific computation (many others for other fields):

1. Imperative.
2. Structured.
3. Procedural.
4. Object-Oriented.
5. Functional.
• Multi-paradigm languages: C++, recent introduction of $\lambda$-calculus features.

• Different problems are better suited to different paradigms.

• You can always “speak” with an accent.

• Idiomatic programming.
Imperative, structured, and procedural
Imperative

• Oldest approach.

• Closest to the actual mechanical behavior of a computer ⇒ original imperative languages were abstractions of assembly language.

• A program is a list of instructions that change a memory state until desired end state is achieved.

• Useful for quite simple programs.

• Difficult to scale.

• Soon it led to spaghetti code.
• *Go To Statement Considered Harmful*, by Edsger Dijkstra in 1968.

• Structured program theorem (Böhm-Jacopini): sequencing, selection, and iteration are sufficient to express any computable function.

• Hence, structured: subroutines/functions, block structures, and loops, and tests.

• This is paradigm you are likely to be most familiar with.
- Evolution of structured programming.

- Divide the code in procedures: routines, subroutines, modules methods, or functions.

- Advantages:
  1. Division of work.
  2. Debugging and testing.
  4. Reusability.
OOP
Object-oriented programming I

- Predecessors in the late 1950s and 1960s in the LISP and Simula communities.

- 1970s: Smalltalk from the Xerox PARC.

- Large impact on software industry.

- Complemented with other tools such as design patterns or UML.

- Partial support in several languages: structures in C (and structs in older versions of Matlab).

- Slower adoption in scientific and HPC.

- But now even Fortran has OO support.
• Object: a composition of nouns (numbers, strings, or variables) and verbs (functions).

• Class: a definition of an object.

• Analogy with functional analysis in math.

• Object receive messages, processes data, and sends messages to other objects.
Object-orientated programming: basic properties

- Encapsulation.
- Inheritance.
- Polymorphism.
- Overloading.
- Abstraction penalty.
• Class *household*.

• We create the file *household.m*.

• We run *Example_Use_Class.m*.

• Public, private, and protected properties and methods.
Functional Programming
Functional programming

- Nearly as old as imperative programming.

- Created by John McCarthy with LISP (list processing) in the late 1950s.

- Many important innovations that have been deeply influential.

- Always admired in academia but with little practical use (except in Artificial Intelligence).
Theoretical foundation

- Inspired by Alonzo Church’s $\lambda$-calculus from the 1930s.

- Minimal construction of “abstractions” (functions) and substitutions (applications).

- Lambda Calculus is Turing Complete: we can write a solution to any problem that can be solved by a computer.

- John McCarthy is able to implement it in a practical way.

- Robin Milner creates ML in the early 1970’s.
Why functional programming?

- Recent revival of interest.

- Often functional programs are:
  1. Easier to read.
  2. Easier to debug and maintain.
  3. Easier to parallelize.

- Useful features:
  1. Hindley–Milner type system.
  2. Lazy evaluation.
  3. Closures.
Main idea

• All computations are implemented through functions: functions are first-class citizens.

• Main building blocks:
  1. Immutability: no variables gets changed (no side effects). In some sense, there are no variables.
  2. Recursions.
  3. Curried functions.
Interactions

How do we interact then?

1. Pure functional languages (like Haskell): only limited side changes allowed (for example, I/O) and tightly enforced to prevent leakage.

2. Impure functional languages (like OCaml or F#): side changes allowed at the discretion of the programmer.

Loops get substituted by recursion.

We can implement many insights from functional programming even in standard languages such as C++ or Matlab.
Main languages:

1. Mathematica.
2. Common Lisp/Scheme/Clojure.
3. Standard ML/Calm/OCalm/F#.
4. Haskell.
5. Erlang/Elixir.