Advanced Programming Paradigms

N. Kälin

September 19, 2020

Contents

1 Introduction

1.1 Programming Paradigms

Paradigm: (definitions from Merriam-Webster)

- a theory or group of ideas about how something should be done, made, or thought about
- example, pattern; especially: an outstandingly clear or typical example or archetype (a perfect example of something)

Programmin Paradigm: fundamental style of programming

- In which notions do we think about a program?
- Which aspects can be **explicitly** described, which cannot?
  - state
  - concurrency and parallelism
  - nondeterminism

Software quality: (according to Bertrand Meyer)

- reliability
  - **correctness**
  - robustness
- modularity
  - extendibility
  - reusability
- compatibility, efficiency, portability, ease of use, timeliness

1.1.1 Imperative Programming

- based on explicitly reading and updating **state**
- immediate abstraction of von Neumann computer
- theoretical base: *Turing* machine
- concepts:
  - data structures: variables, records, arrays, pointers
  - computation:
    - expressions:
      literal, identifier, operation, function call
    - commands (instructions, "statements"):
      assignment, composition, goto, conditional, loop, procedure call
  - abstraction: functions, procedures
- examples: Ada, Algol, C, Cobol, Fortran, Moudla, Pascal

1.1.2 Object-Oriented Programming

- strongly based on imperative paradigm
- further support for abstraction and modularization
  - Abstract Data Types (ADTs)
  - information hiding
- concepts:
  - objects as instances of classes: data + procedures put together
  - encapsulation (private, protected, public)
  - inheritance for modularity and for variant records
  - subtyping, polymorphism and dynamic binding
  - genericity (from some imperative and most functional languages)
- examples: C++, C#, Eiffer, Java, Objective-C, Simula, Smalltalk

1.1.3 Functional Programming

- based on λ-calculus and reduction
- subexpressions are replaced by simpler, but equivalent subexpressions until no longer possible
- concepts:
  - no state, no commands; just expression
  - identifiers denote values, not variables (storage cells)
  - no commands implies no loops; just recursion
  - functions: recursive, anonymous, curried, higher-order (DSLs)
  - recursive algebraic data types and pattern matching
  - polymorphic and overloaded types
  - type inference
  - eager or lazy evaluation
  - simple equational reasoning about programs
- examples: F#, Haskell, Lisp, ML, OCaml

1.1.4 Logic Programming

- based on first-order logic (predicate logic)
- logical formulas express relations declaratively
- machine solves formulas through resolution
- works for specialized formulas like *HORN* clauses
- efficient only if programmer guides the solution process
- example: Prolog

1.1.5 Further Programming Paradigms

- constraint programming
- concurrent programming
- parallel programming

1.1.6 Multiparadigm Programming

- several paradigms can be combined into a single language
- each paradigm has its realm; today's large applications embrace many such realms; a single language simplifies interoperability
- examples:
  - functional with imperative features: ML
  - object-oriented with functional features: C#
  - functional with object-oriented features: F#, OCaml
  - functional + object-oriented: Scala
  - functional + logic: Curry (based on Haskell)

1.2 Correctness and Verification

1.2.1 Correctness

- prime quality, *conditio sine qua non*
- relative notion: program should be correct with respect to its **specification**
  - example: program that computes the sine perfectly well but should compute the root is clearly not correct
- but how can one know whether a program is correct or not?
  - by *testing*, one can find faults (bugs)
  - by *proving*, one can show the absence of faults

1.2.2 Testing versus Proving

better: Tests **and** Proofs

- testing
  - choose particular input
  - determine correct result for that input using test oracle
  - run program under test on the chosen input
  - compare obtained and correct result
    - if different: fault found
    - if equal: no relevant information obtained
- proving
  - do **not** choose a particular input
  - do **not** execute the program
  - instead apply mathematical rules to program and specification

1.2.3 Verification As a Matter Of Course (VAMOC)

(according to Bertrand Meyer)

- software controls more and more of our daily lives
- software becomes more and more complicated
- testing does not suffice; verification is needed in addition
- verification tools become more and more powerful
- examples: Spec# and Dafny for specification and verification of object-oriented programs

1.2.4 Types

- 'good' expressions can be typed at compile time
- ill-typed expressions will not compile
- thus corresponding run-time errors cannot occur
- type checking and inference is mostly fully automatic
- light-weight formal method
- first step towards program verification