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

2 Functional Programming

2.1 Correctness

(see 1.2.1)

2.1.1 Obtaining Mathematical Knowledge

1. Conjecture
   The product of all prime numbers between and including 2 and p, increased by 1, is again a prime number.

2. Examples
   For $p = 2, 3, 5, 7, 11, 379$ the conjecture is confirmed.

3. Counterexample
   For $p = 17$ the conjecture is refuted.

1. Theorem
   $(a + b)^2 = a^2 + 2ab + b^2$

2. Proof
   $(a + b)^2 = (a + b)(a + b) = a(a + b) + b(a + b) = aa+ab+ba+bb = aa+ab+ab+bb = aa+2ab+bb = a^2 + 2ab + b^2$

   with a **finite** number of steps we have thus shown that something holds for an **infinite** number of values

2.1.2 Consequence

- programming languages should simplify proofs (and therefore also program development itself)
- and thus may enhance program reliability

2.2 Referential Transparency

2.2.1 A More Formal Proof

$$
\begin{aligned}
&(a+b)^2 \\
=&\{\text{def square}\} \\
&(a+b)\cdot(a+b) \\
=&\{\text{distri}\} \\
&a\cdot(a+b)+b\cdot(a+b) \\
=&\{\text{distri twice}\} \\
&a\cdot a+a\cdot b+b\cdot a+b\cdot b \\
=&\{\text{commu multi}\} \\
&a\cdot a+a\cdot b+a\cdot b+b\cdot b \\
=&\{\text{neutral multi twice}\} \\
&a\cdot a+1\cdot(a\cdot b)+1\cdot(a\cdot b)+b\cdot b \\
=&\{\text{distri}\} \\
&a\cdot a+(1+1)\cdot(a\cdot b)+b\cdot b \\
=&\{\text{def 2}\} \\
&a\cdot a+2\cdot(a\cdot b)+b\cdot b \\
=&\{\text{def square twice}\} \\
&a^2+2ab+b^2
\end{aligned}
$$

- this proof sill handles associativity implicitly
- this format for *calculational proofs* is due to FEIJEN and DIJKSTRA
- a corresponding `calc` statement is available in Dafny

2.2.2 Equality

A fundamental mathematical concept

- four inference rules of a logic
- Reflexivity: $\frac{}{X=X}$
- Symmetry: $\frac{X=Y}{Y=X}$
- Transitivity: $\frac{X=Y, Y=Z}{X=Z}$
- LEIBNIZ: $\frac{X=Y}{E[v\leftarrow X]=E[v\leftarrow Y]}$
- X, Y, Z, E: expressions, v: variable, $E[v \leftarrow X]$: textual substitution of all (free) occurrences of v by (X) in E

2.2.3 Example LEIBNIZ

- from numbers: $x \cdot (y + z) = x \cdot y + x \cdot z$
- therefore, by LEIBNIZ (and Substitution):

$$\underbrace{(a \cdot (a + b))}+b \cdot (a + b) \tag{1}$$