Functional programming vs. declarative programming

Sergei Winitzki

Academy by the Bay 2024

2024-04-14

Outline

Functional programming is interesting, but in a weird way

- FP is (almost) engineering
- Other programming paradigms are (almost) artisanship

What is declarative programming:

- A language for problem requirements, understandable to people
- The same language is mechanically translated into code

Declarative programming is domain-specific (DSL)

- Different domains have evolved different languages

Implementing DSLs is a "killer app" for functional programming

- FP is declarative for recursive manipulation of labeled trees

Is the Functional Programming community weird?

The FP community is unlike other programmers' communities

- Others are focused on a chosen programming language (Java, Python, JavaScript, etc.), and on designing and using libraries and frameworks
  - *"setup this YAML config, override this method, use this annotation"*
- People in the FP community talk in a very different way
  - *"referential transparency, algebraic data types, monoid laws, parametric polymorphism, free applicative functors, monad transformers, Yoneda lemma, Curry-Howard isomorphism, profunctor lenses, catamorphisms"*
    - A glossary of FP terminology (more than 100 terms)
  - From SBTB 2018: *The Functor, Applicative, Monad talk*
    - By 2018, everyone *expects* to hear these concepts mentioned

As a former theoretical physicist, I recognize that sort of jargon

- The FP jargon is used similarly to an engineer's jargon
- It is based on math but heavily adapted to the engineering domain

Programming as engineering vs. artisanship

- FP is similar to engineering in how it uses math-based tools
  - Mechanical, electrical, chemical engineering use calculus, complex variables, classical and quantum mechanics, electrodynamics, thermodynamics, physical chemistry
  - FP uses category theory, type theory, logic proof theory, λ-calculus
- Engineers use *a lot* of special terminology
  - Examples from mechanical, electrical, chemical engineering: rank-4 tensors, Lagrangians with non-holonomic constraints, Fourier transform of the delta function, inverse Z-transform, Gibbs free energy 1, 2
  - Examples from FP: rank-*N* types, continuation-passing transformation, polymorphic lambda functions, free monads, hylomorphisms
- As in engineering, the special terminology in FP is *not* self-explanatory
  - "Gibbs free energy" is not energy that J. W. Gibbs provides for free

All that stuff is mathematics-based knowledge that needs learning

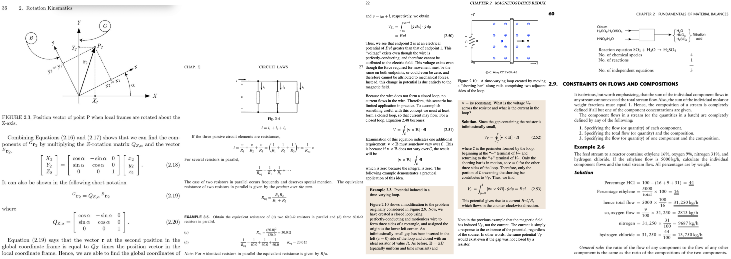
- Programmers today neither study as engineers, nor work as engineers

Books on engineering vs. artisanship

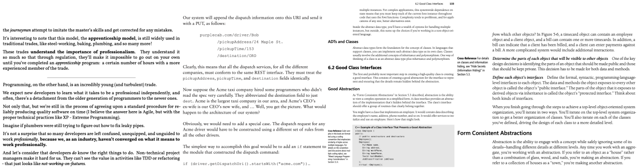
Functional programming looks like engineering

Today's "software engineering" resembles artisanship

Books on mechanical, electrical, chemical engineering design:



Books on software design and architecture:



Questions that have rigorous answers

In engineering, certain questions about device design have rigorous answers
In FP, certain questions about code design have rigorous answers

- The answers are *not* a matter of opinion or experience
- The answers are found via mathematical derivations and reasoning
- The answers guide the design

Examples of reasoning tasks:

- Can we implement these APIs?

```
f :: (r -> Either z a) -> Either z (r -> a) -- Haskell
g :: Either z (r -> a) -> r -> Either z a
def f[Z, R, A](r: R => Either[Z, A]): Either[Z, R => A] // Scala
def g[Z, R, A](e: Either[Z, R => A]): R => Either[Z, A]
```

- Can use the data structure `F` as a monad in our code?

```
type F a = Maybe (a, a, a) -- Haskell
bind :: F a -> (a -> F b) -> F b
type F[A] = Option[(A, A, A)] // Scala
def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
```

A definition of "declarative"

Programming is "declarative" when *"specifications are programs"*

- "Being declarative" is not a property of a programming language alone

A language L is **declarative for an application domain** D if:

- The domain D has a good specification formalism F
  - "good" = visual, pragmatically convenient, complete for the domain D
- There is a syntactic translation from F to L
- The resulting program correctly implements the specification

Less formally:

- A declarative language is a "readable DSL" for the given domain

Example: declarative FORTRAN 77

- Application domain: numerical mathematical expressions
- Specification: a mathematical formula involving *numbers* and *functions*
- Example specification:

$$f(x,p,q) = \frac{\sin px}{x^2} - \frac{(\sin qx)^2}{x^3}$$

  - Implementation: `F(X,P,Q) = SIN(P*X)/X**2-(SIN(Q*X))**2/X**3`

- For more complicated tasks, FORTRAN is not declarative

$$\tilde{X}_k = Y_k - \sum_{j=k+1}^{n} A_{kj} X_j, \quad \forall k \in [1..n]$$

```
      X(N)=Y(N)/A(N,N)
      DO 10 K=N-1,1,-1
         S=0.
         DO 20 J=K+1,N
20          S=S+A(K,J)*X(J)
10       X(K)=(Y(K)-S)/A(K,K)
```

(example code, 1987)

- Application domain: recursively defined, algebraic data structures
- Specifications: inductive definitions of functions on ADTs
- Example (from R. Sedgewick, *Algorithms in C*, 1998)

Definition 5.1 *A* **binary tree** *is either an external node or an internal node connected to a pair of binary trees, which are called the left subtree and the right subtree of that node.*

This definition makes it plain that the binary tree itself is an ab-

```
data BTree α = BTNode α | BTVertex α (BTree α) (BTree α)
```

```
enum BTree[+A]:     //Scala
  case BTNode[A](a:  A)
  case BTVertex[A](a:  A, left:  BTree[A], right:  BTree[A])
```

Example: declarative Haskell 98, continued

Definition 5.6 *The* **level** *of a node in a tree is one higher than the level of its parent (with the root at level 0). The* **height** *of a tree is the maximum of the levels of the tree's nodes. The* **path length** *of a tree is the sum of the levels of all the tree's nodes. The* **internal path**

```
height ::  BTree α → Int     -- Haskell
height (BTNode _) = 0
height (BTVertex _ t1 t2) = 1 + max (height t1) (height t2)

def height[A]: BTree[A] => Int = {     //Scala
    case BTNode(_) => 0
    case BTVertex(_, t1, t2) =>
      1 + math.max(height(t1), height(t2))
}
```

Example: non-declarative Haskell

For a different application domain, Haskell is *not* declarative!

- Downloading data from server (from “*Real World Haskell*”, 2008)

```
download dbh logf =
    do pclist <- getPodcasts dbh
       mapM_ procPodcast pclist
       logf "Download complete."
    where procPodcast pc =
              do logf $ "Considering " ++ (castURL pc)
                 episodelist <- getPodcastEpisodes dbh pc
                 let dleps = filter (\ep -> epDone ep == False)
                             episodelist
                 mapM_ procEpisode dleps
          procEpisode ep =
              do logf $ "Downloading " ++ (epURL ep)
                 getEpisode dbh ep
```

Declarative programming: Stories of success and failure

Success stories (in the sense of achieving declarative programming)

- Infix arithmetic: numerical math
- SQL: relational queries
- Autolayout: GUI layout on iOS and MacOS
- Haskell, Scala: Parsing, type-checking, evaluation of DSLs
- Prolog: logic puzzles (next slide)
  - for comparison, see: Matrix multplication with Prolog
- Chemical Abstract Machine: dining philosophers problem

Failure stories (in the sense of turning out non-declarative)

- Programming languages resembling English (COBOL, AppleScript)
- "Mark-up" languages (XML, HTML)
- Languages where everything is a built-in feature (PL/I, ABAP)

Prolog as a DSL for logic puzzles

All jumping creatures are green. All small jumping creatures are martians. All green martians are intelligent. Ngtrks is small and green. Pgvdrk is a jumping martian. Who is intelligent? (inpired by *Invasion from Aldebaran*)

```
$ cat > martians.pl
green(X) :- jumping(X).
martian(X) :- small(X), jumping(X).
intelligent(X) :- green(X), martian(X).
small(ngtrks). green(ngtrks).
jumping(pgvdrk). martian(pgvdrk).
question :-
  intelligent(X), format('~w is intelligent.~n', X), halt.
^D
$ brew install swi-prolog
$ swipl -o martians -q -t question -c martians.pl
$ ./martians
pgvdrk is intelligent.
```

The joy of implementing DSLs in FP languages

Two choices for implementing a DSL:

- Embedded DSL
- External DSL

In both cases, FP works great

- For embedded DSLs:
  - Free monads, GADTs, non-leaky abstractions, strict typing
- For external DSLs:
  - Parser combinators, recursion schemes, GADTs, HOAS / PHOAS

Anecdotal evidence: one-person languages with compilers in Haskell

- Agda (Ulf Norell, 2007)
- Idris (Edwin Brady, 2007)
- Elm (Evan Czaplicki, 2012)
- PureScript (Phil Freeman, 2013)
- Dhall (Gabriella Gonzalez, 2016)
  - Time to re-implement Dhall in Scala: 2 months, 4K LOC

Conclusions

Functional programming has a steep learning curve

- Using FP techniques makes programmers' work closer to *engineering*
- Most artisans don't want to become engineers

Declarative programming means a symbiosis between human tradition and formal mathematics

- Best implemented by math-based programming paradigms
  - FP and logic programming

Implementing DSLs is one of FP's "killer apps"

- Even a small DSL is a productivity boost when it is declarative for the chosen domain