Functional Approaches to Programming

~ Introduction ~

Didier Verna

didier@didierverna.net

didierverna.net @didierverna didier.verna in/didierverna

Outline

Build 2024-11-27 10:46:05+01:00

Plan

Build 2024-11-27 10:46:05+01:00

1930: Lambda-Calculus (Alonzo Church)

- Formal system for calculus *(top-down)*
- Rules
  - Variable: x
  - Abstraction: $(\lambda x.M)$
  - Application: (MN)
- Operators
  - α-conversion: $(\lambda x.M[x]) \rightarrow (\lambda y.M[y])$
  - β-reduction: $((\lambda x.M)E) \rightarrow (M[x := E])$
- Proof of Turing-completeness [Turing, 1937]
- Remark: Combinatory Logic (1920/1930)
  - Proof of theoretical equivalence [Curry, 1958]

Source: Princeton

1958: Lisp "LISt Processing" (John McCarthy)

- Origin [McCarthy, 1960]
  - "Advice Taker" Project
    - symbolic AI (expert system)
    - MIT AI Lab, IBM 700/7000
  - The AI language (a term coined by McCarthy)
    - 2^{nd} most ancient high level language
    - 1^{st} functional language
    - lambda-calculus + recursion
    - multi-paradigm & homoiconic
- Dialects chronology (simplified)
  - 1975: Scheme / Lisp Machines
  - 1980: Common Lisp (ANSI Standard 1994)
  - 1985: Emacs Lisp
  - 2005: Clojure
  - 2010: Racket

Source: Stanford

1958: Lisp "LISt Processing" (John McCarthy)

- Origin [McCarthy, 1960]
  - "Advice Taker" Project
    - symbolic AI (expert system)
    - MIT AI Lab, IBM 700/7000
  - The AI language (a term coined by McCarthy)
    - 2^{nd} most ancient high level language
    - 1^{st} functional language
    - lambda-calculus + recursion
    - multi-paradigm & homoiconic
- Dialects chronology (simplified)
  - 1975: Scheme / Lisp Machines
  - 1980: Common Lisp (ANSI Standard 1994)
  - 1985: Emacs Lisp
  - 2005: Clojure
  - 2010: Racket

Source: Wikimedia Commons

Other Languages

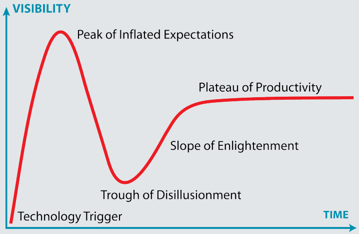
- Historical
  - IPL (1956) *Assembly level, very imperative*
  - APL (Iverson, 1960), FP (Backus, 1977) [Backus, 1978]
  - ML (Milner, 1973), Standard ML, Caml / OCaml (Leroy, 1996)
  - Miranda (Turner, 1985), Haskell (1990)
- More recent *Object / Functional mix*
  - Scala (Odersky, 2004)
  - F# (M$, 2005)
- In the mix
  - Python, Ruby, JavaScript, Julia, *etc.*
- Back to Functional
  - Lambda-expressions in C++11 / Java 8

 Build 2024-11-27 10:46:05+01:00

How Functional Fits in History

- Reminder: top-down approach
  - Frightening
    Literally, "extra-" ordinary
  - Difficult to implement
    Distant from the Von Neumann architecture
- AI Winter $\Longleftrightarrow$ FP Winter $\Longleftrightarrow$ Lisp Winter

The "Gartner Hype Cycle"



Source: Wikimedia Commons

The more top-down the approach, the more pronounced the hype cycle...

Plan

History

A Programming Paradigm

Characteristics of the Functional Approach

Summary

A Programming Paradigm

- **Reminders on the concept of paradigm**
  - General thought framework $\Longleftrightarrow$ Form of expression
  - Imperative
    - instruction (side effect)
    - execution
- **The functional paradigm**
  - Expression ($\neq$ Instruction)
  - Evaluation ($\neq$ Execution)

"What to do" rather than "How to do it"

From Imperative to Functional

"The sum of the squares of the integers between 1 and N"

C (imperative)

```
int ssq (int n)
{
  int i = 1, a = 0;
 loop:
  a += i*i; i += 1;
  if (i <= n+1)
    goto loop;

  return a;
}
```

C (recursive)

```
int ssq (int n)
{
  if (n == 1)
    return 1;
  else
    return n*n + ssq (n-1);
}
```

Lisp

```
(defun ssq (n)
  (if (= n 1)
      1
      (+ (* n n) (ssq (1- n)))))
```

Haskell

```
ssq :: Int -> Int
ssq 1 = 1
ssq n = n*n + ssq (n-1)
```

- Clarity
- Concision

Imperative Upside Down

"The square root of the sum of the squares of a and b"

C (imperative)

```
float hypo (float a, float b)
{
  float a2 = a*a;
  float b2 = b*b;
  float s = a2 + b2;

  return sqrt (s);
}
```

C (less imperative)

```
float hypo (float a, float b)
{
  return sqrt (a*a + b*b);
}
```

Haskell

```
hypo :: Float -> Float -> Float
hypo a b = sqrt (a*a + b*b)
```

Lisp

```
(defun hypo (a b)
  (sqrt (+ (* a a) (* b b))))
```

But…

Haskell (100% prefix)

```
hypo :: Float -> Float -> Float
hypo a b = sqrt ((+) ((*) a a) ((*) b b))
```

Plan

History

A Programming Paradigm

Characteristics of the Functional Approach

Summary

Higher Order Functions

Aka 1st order, 1st class

- **Christopher Strachey** [Strachey, 72]
  - naming (variables)
  - aggregation (structures)
  - anonymous manipulation
  - function argument
  - function return value
  - dynamic construction
  - ...
- **Advantage:** increased expressivity (clarity, concision, *etc.*)
- Example: "mapping"

Lisp

```
(mapcar #'sqrt '(1 2 3 4 5))
```

Haskell

```
map sqrt [1..5]
```


Build 2024-11-27 10:46:05+01:00

Pure Functional Programming

The function, in the mathematical sense

$$ssq(x) = \begin{cases} 1 & \text{if } x = 1, \\ x^2 + ssq(x-1) & \text{otherwise.} \end{cases}$$

Haskell

```
ssq :: Int -> Int
ssq 1 = 1
ssq n = n*n + ssq (n-1)
```

- **Function**
  - Imperative: (procedure) sequence of computations with *side effects*, possibly with a return value
  - Purely functional: computation of a (return) value depending on input values (arguments)
- **Variable**
  - Imperative: storage of values which may *vary* over time (mutation)
  - Purely functional: unknown or arbitrary (but constant) value

Advantages of Purity

Purity $\Longrightarrow$ (more) safety

- **Parallelism**
  - Cf. Erlang
- **Function-local semantics**
  - Local tests / Local bugs
- **Program proofs**

Build 2024-11-27 10:46:05+01:00

Formal Proofs

Mathematical Induction

"Prove (please) that $\forall N, ssq(N) > 0$"

Purely functional

Haskell

```haskell
ssq :: Int -> Int
ssq 1 = 1
ssq n = n*n + ssq (n-1)
```

- It's true at rank 1
- Suppose it's true at rank $N - 1$...

Imperative

C

```c
int ssq (int n)
{
  int i = 1, a = 0;
 loop:
  a += i*i; i += 1;
  if (i <= n+1)
    goto loop;

  return a;
}
```

- Ugh...

Evaluation Principles

- **Question:** when to compute the value of an expression ?
- **Answers:**
  1. beforehand (Lisp, *strict evaluation*)
  2. on demand (Haskell, *lazy evaluation*)
- Example:

Lisp

```
(defun intlist (s)                    ;; KO
  (cons s (intlist (1+ s))))
```

Haskell

```
intlist :: Int -> [ Int ]             -- OK
intlist s = s : intlist (s + 1)
```

- **Advantage:** increased expressivity (clarity, concision, *etc.*)
- **Constraint:** purely functional only

Limitations of the Mathematical Formalism

How to express the concept of "square root"?

$$sqrt(x) = y| \begin{cases} y > 0 \\ y^2 = x \end{cases}$$

Lisp

```
(defun sqrt (x) ???)
```

Haskell

```
sqrt :: Float -> Float
sqrt x = ???
```

Lisp

```
(defun sqrtp (s x)
  (and (> s 0)
       (= (* s s) x)))
```

Haskell

```
sqrtp :: Float -> Float -> Bool
sqrtp s x = s > 0 && s*s == x
```

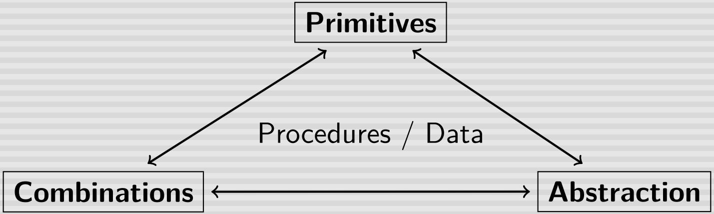
- By the end of the day, you still need to explain *how to do it*...
- But you just postpone the question: declarative or imperative ?

 Build 2024-11-27 10:46:05+01:00

Plan

Benefits of the Functional Approach

The 3 characteristics of (good) languages (*cf.* SICP [Abelson, 1996])



- Less distinction between procedures and data
- More power in combination
- More power in abstraction

Build 2024-11-27 10:46:05+01:00

Plan

Bibliography

Bibliography

Alan M. Turing.
Computability and λ-definability.
Journal of Symbolic Logic. Cambridge University Press. 2 (4): 153–163.

Haskell B. Curry, Robert Feys.
Combinatory Logic.
North-Holland Publishing Company.

John McCarthy.
Recursive Functions of Symbolic Expressions and their Computation by Machine, part I.
Communications of the ACM. 3: 184–195.

Bibliography

John W. Backus.
Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs.
Communications of the ACM. 2(8): 613-–641.

Joe E. Stoy and Christopher Strachey.
OS6 – An Experimental Operating System for a Small Computer.
The Computer Journal, 15(2): 117–124.

Harold Abelson and Gerald Jay Sussman.
Structure and Interpretation of Computer Programs (2nd ed.).
MIT Press.