Revisiting “Why FP Matters” by John Hughes (1990)

March 2018
V0.1

Greg Baran
gbkoala@gmail.com

Welcome to FP Chicago

Today, we'll offer some reasons that curious folk might want to look closer at functional programming

We revisit a paper written by John Hughes back in 1990 called "Why FP Matters" (now with Chalmers in Sweden)

It's his second most-read paper - he invented Quickcheck and other useful things, such as Haskell (with many colleagues of course - first most-read)

Hughes Sources

The original paper is from 1990:

https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf

A keynote presentation at Lambda Days 2017 by Hughes and his wife, Mary Sheeran (other versions from YOW, Erlang Days, etc.):

https://www.youtube.com/watch?v=1qBHf8DrWR8

An interview on the subject with Hughes by Infoq:

https://www.infoq.com/interviews/john-hughes-fp

Roots

Where FP comes from

Some Early FP Landmarks

Church’s lambda calculus 1930s-40s

McCarthy LISP 1960

Landin 1965: “Next 700 Programming Languages”: laws should be satisfied - map reverse L == reverse map L

Backus 1978 Turing award paper: “Functional Style and Algebra of Programs” - how to meet Landin’s requirement

Promo

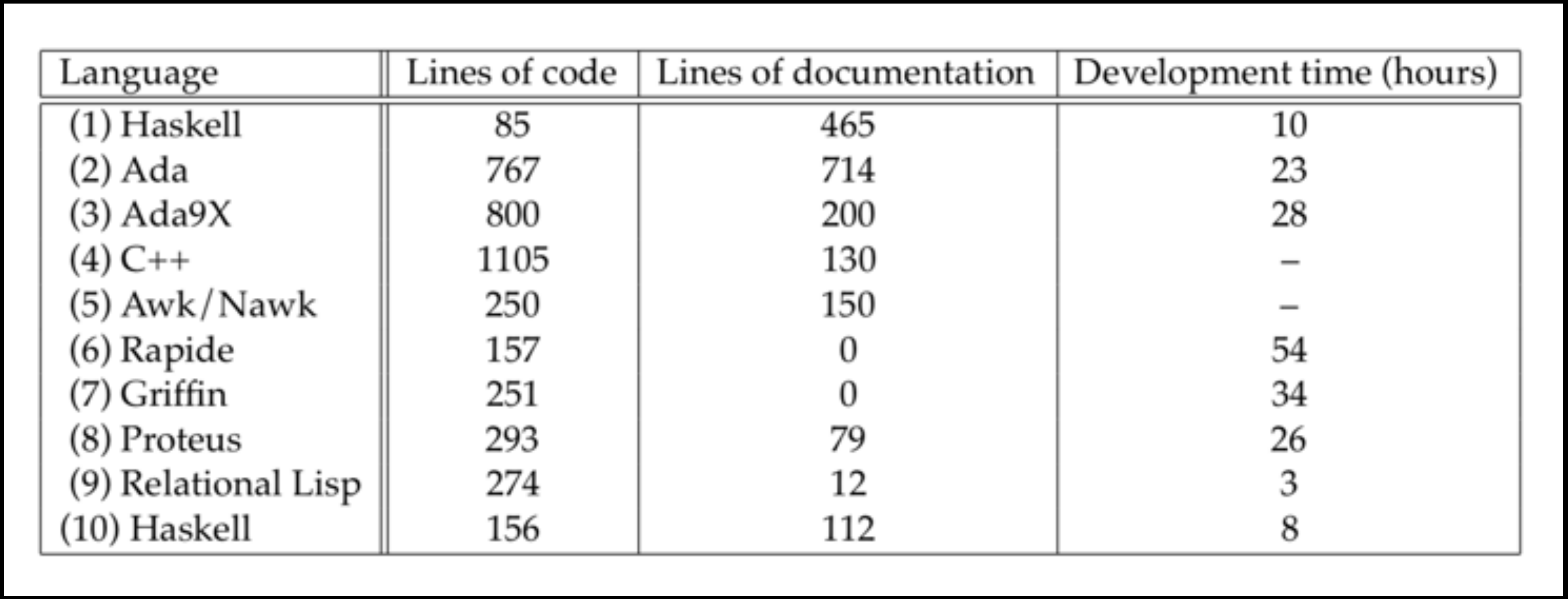
Comparison with other languages

The Navy's Programming Slam

In 1993, the US Navy hosted a competition to identify the programming language that would be best suited for a target-tracking application, which was basically a geometry problem

Ten entries were made, including Haskell

"Haskell vs. Ada vs. C++ vs. Awk vs. … An Experiment in Software Prototyping Productivity", Paul Hudak, Mark P. Jones (1994)



Language	Lines of code	Lines of documentation	Development time (hours)
(1) Haskell	85	465	10
(2) Ada	767	714	23
(3) Ada9X	800	200	28
(4) C++	1105	130	–
(5) Awk/Nawk	250	150	–
(6) Rapide	157	0	54
(7) Griffin	251	0	34
(8) Proteus	293	79	26
(9) Relational Lisp	274	12	3
(10) Haskell	156	112	8

Haskell (1) programmed by Jones, a postdoc at Yale at the time — it includes 29 lines of inferable typing info in the 85 lines

To check how easy Haskell was to learn, Hudak asked a recent college grad to learn Haskell in 8 days, and give it a go
That's Haskell (10)

Disbelief!

The Navy reviewers rejected the Yale solution because it did not have the usual data-structure definition boiler plate and related noise.

They thought it was a specification, not an implementation.

Nowadays, one could write a similarly succinct solution using Scala, Ocaml, Rust, Elixir/Erlang, etc. (Interesting to try)

Program glue for better modularity

Higher-order functions

Lazy evaluation

Modularity

Hughes's main premise:

Modularity helps quality

- divide the problem into subproblems
- solve the subproblems
- combine the solutions

How you can divide depends directly on the ways in which solutions can be glued together, so improving modularity means making *new kinds of glue*

Want smaller and simpler and more general modules, glued together with the new glues

(Complicated scope rules and provision for separate compilation help only with clerical details)

Higher-order functions

Demonstrate:

Functional languages allow functions that are indivisible in conventional programming languages to be expressed as a combinations of parts — a general higher-order function and some particular specializing functions

Such higher-order functions allow many operations to be programmed very easily

Whenever a new datatype is defined, higher-order functions should be written for processing it

(Sec. 3, p. 5 to 8)

(Code: *glue1.hs, glue2.hs, glue3.hs, glue4.hs*)

Lazy evaluation

What is laziness? Produce data and evaluate arguments only when required

UNIX shell pipelines are lazy:

```
cat addr.csv | grep “Bob Dobbs” | sed 1q > bob.txt
```

Same thing in FP:

consume until good enough (generate better solutions)

UNIX pipe(7) man page

"If a process attempts to read from an empty pipe, then read(2) will block until data is available. If a process attempts to write to a full pipe, then write(2) blocks until sufficient data has been read from the pipe to allow the write to complete."

This is lazy, implemented by a flow-controlled FIFO

Laziness Examples

Separation of generator of approximate square roots and the decision that the result is good enough:

The generator is infinite, but the consumer stops the process when satisfied (Sec. 4.1)

A second example shows numerical differentiation using the same methods (Sec. 4.2)

(Code: *progglue1.hs, progglue2.hs*)

Numerical Integration

Another example is covered in the paper (Sec. 4.3) on integration, analogous to numerical differentiation:

within eps (integrate f a b)

relative eps (integrate f a b)

integrate generates better and better solutions, *within* or *relative* consumes these until the result is good enough, then stops

Alpha-Beta Algorithm

There’s a final section (5) that covers a basic AI technique for efficiently finding the best alternative, originally for games

Hughes uses TicTacToe to illustrate how to enumerate alternatives

Data61 FP Course (e.g)

The Data61 Functional Programming on-line course (*https://github.com/data61/fp-course*) lets you reconstruct all of the main parts of the Haskell prelude as an exercise - learn by doing

It includes an FP exercise based on TicTacToe which is an interesting challenge (under *projects*) - see next slide

Thanks to Mark Hibberd and Tony Morris for the course. It also has good community support.

TicTacToe Challenge

The goal is to write an API for the tic-tac-toe game. An API user, should be able to play a game of tic-tac-toe using this API, but importantly, it should be impossible for the API user to break the rules of the game. Specifically, if an attempt is made to break a rule, the API should reject the program. This is often done by way of a **compile-time type error**.

It is strongly advised that functional programming techniques are used to achieve the goal. This is because ensuring that the API adheres to the rules of tic-tac-toe, while rejecting an incorrect program, is difficult using a non-FP language. No specific programming language is prescribed.

Conclusion

- modularity is the key to successful programming
- mechanisms for separate compilation are not enough
- decomposing a problem into parts depends directly on our ability to glue solutions together
- FP languages provide two new kinds of glue, higher-order functions and lazy evaluation
- smaller and more general modules can be reused more widely, easing subsequent programming
- thus FP solutions are so much smaller and easier to write than conventional ones

Questions

References

J. Backus, Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. 1977 Turning Award paper, CACM, Vol. 21, Num. 8, pp. 613-641 (August 1978)

W.E. Carlson, P. Hudak, and M.P. Jones. An experiment using Haskell to prototype “geometric region servers” for navy command and control. Research Report 1031, Yale University (November 1993)

J. Hughes, Why Functional Programming Matters. From “Research Topics in Functional Programming” ed. D. Turner, Addison-Wesley, pp 17–42 (1990)

P. J. Landin, The Next 700 Programming Languages. CACM, Vol. 9, Num. 3, pp. 157-166 (March 1966)