An Introduction to Functional Programming and Elixir

Howard Bussey 4/10/2025

About Me

- Software career started in 1969
- Some hardware design included
- Conferences, archival journal papers, etc
- Patents
- About two dozen programming languages
- Dyslexia tutor
    - Happy is the day your student doesn’t stumble over “semi-autobiographically” and can break it apart.
- I have no known financial connection to any Elixir business other than as a satisfied customer.
- I’m “at7heb” on GitHub; git@github.com:at7heb/Symposium-2025.git will be updated with these assets presently

About This Talk

- Functional programming
- A little bit of “magic”
    - At least that’s what I would have thought ten years ago
- Elixir characteristics
- Elixir advantages (and otherwise)
- When finished, I want you to know this about Elixir
    - Some of it’s advanced technology, when it might be useful, and when another choice might be better

Functional Programming Generalizations

Functional Programming Characteristics

From 50,000 Feet

- Pure functions
- Immutability
- First class functions
- Preferences
  - No side effects (ok for I/O, database, OS interface, …)
  - Recursion instead of loops
  - Compose simpler functions CR⁺C pattern — create, reduce⁺, convert

Pure Functions

- Always return same result given the same input
- Helps with testing
- Improved confidence in program correctness
- Helps with reasoning

(Except for functions that access databases, network resources, operating system resources, …)

Immutability

- The data cannot and does not change
  - One language enforced this at compile time
- Fewer side effects
- Greater code predictability

First Class Functions

- Functions can be bound to variables
- Functions can be used as function arguments
  - High order functions, like map(), reduce(), filter(), sort(), …
- Functions can be returned by a function
  - [Not recommended but] an Elixir function can also
    - Create the source code for a function
    - Compile the source code
    - Use or return the resulting function

Preferences

No side effects & use recursion

- No side effects

  —> testing is more straightforward

  —> code is easier to understand

  —> behavior is easier to predict

- Recursion

  —> immutability makes looping difficult

  —> Use tail recursion, optimized by compiler to prevent stack overflow

  ** Or avoid recursion by using each(), map() and reduce() functions

Preferences

Function Composition

- Compose functions - CR^+C (Create, $Reduce^+$, Convert)

  —> Use a chain of simpler functions for complex operation

  Like Unix pipe philosophy

—> ***Create*** functions makes a value of a desired type

—> ***Reducer*** functions copy and update a value, maintaining the type

—> ***Convert*** functions change the type

—> The type is usually a structure that holds necessary state

Functional Languages

- Clojure
- Elixir
- Erlang (which started Elixir & Gleam)
- F#
- Gleam
- Haskell
- Lisp

- Languages supporting functional style
  - JavaScript, Kotlin, Python, Rust, Scala, Swift, …

Maybe Some Magic?

Sufficiently advanced technology is indistinguishable from magic

—Arthur Clark

A bit of Elixir Magic

Numerical integration by process

- Calculate $\int_0^{\pi} \sin(\theta)\, d\theta$ — “the worst possible way”

- Create *n* processes, one for each slice of the numerical integration, the *k-th* calculating $\left(\frac{\pi}{n}\right) sin\left(\frac{k\pi}{n}\right)$

- Each process receives a message with k and n, calculates the area of the slice and sends it back

- Sum the areas of each slice and output the value

Elixir Solution

- The performance:
  - 1.8 or 2.7 µs per slice to
    - Do these steps *n* times: create Elixir process, send it the message, let it calculate the area, send back the result, and it exits (1.8 µs per slice)
    - Create *n* processes, then send them all messages, they all calculate the area, send back messages, and exit, and then the base process receives all the messages (2.7 µs per slice)
- I’m going to show you with the Elixir Livebook

- *[Switching back and forth with Livebook]*

- *[Switching back and forth with Livebook]*

- `Enum.each(range, function)`: apply *function* for each value in *range* & ignore value returned by *function*

- `spawn(module, function, list)`: start a process that runs *function* in *module*, passing argument(s) in *list*

- `send(pid, message)`: send *message* to process identified by *pid*

- `self()`: the *pid* of the running process

- `receive do pattern -> function end`: receive a message matching *pattern*, invoke *function*, and return the value

- `Enum.map(range, function)`: apply *function* for each value in *range* & return a list of returned values

- `Enum.sum(list)`: return the sum of the numbers in *list*

What you just saw

Elixir spun up a huge number of processes. Each process received a message with instructions, computed a result, and sent a message.

The orchestrating process received and processed all the messages.

Can your web server spin up a process to handle a request in two microseconds?

It’s Not Magic, It’s Technology

Elixir Execution

The BEAM's Scheduler

- The BEAM, Bogdan's Erlang Abstract Machine, is the execution environment for Elixir
- Elixir processes — lightweight, "shared nothing"
  - Except processor resources and "atoms" (= literals)
  - Much lighter-weight than OS threads, e.g. 3 kB minimum size
- Preemptive context switching (default: one scheduler per processor core)
  - Based on priority and count of operations (a.k.a. BEAM reductions)
- Garbage collection is usually by process
  - Global garbage collection is rare (rarely needed)
- This is the advanced technology behind spinning up a process in a few microseconds

Learning Assessment 1

- You are an SME in your engineering domain.
- You need to write a new application, or rewrite an existing one
- Your bonus depends on how well you plan use of cheap, lightweight processes for the project
- Sketch or describe the plan
- What would be the costs?
- What would be the advantages?

Wait a minute

Assessment - My Answer

Simulating Asynchronous Transfer Mode Switch

- Using “Ring Reservation” at input ports to make output port reservations
- Two processes per input port a) cell queue process; b) ring reservation process
  - Cell queue process exchanges messages with associated ring reservation process
  - Ring reservation process exchanges messages with neighboring ring reservation processes
- Costs:
  - clock processes: switch fabric cell clock & ring reservation clock
  - Messages: port to ring to request output; ring to neighbor for reservation algorithm, ring to port to confirm output reservation
- Advantages
  - each port and each ring process have one function that handles a small number of messages
  - No shared data

Elixir History

- Elixir created by José Valim; first commit: January 2011; 1.0 Release September 2014
- Elixir uses the Beam, the abstract machine created for Erlang
  - Erlang systems are said to have nine nines of reliability; possibly for Ericsson telephone switches, which was the motivation for Erlang
  - Erlang development continues and benefits Elixir. E.g.just-in-time compilation
- Useful frameworks have been created for Elixir - web development, numerical processing, neural networks, IoT, multimedia, etc.

Elixir Characteristics

In addition to those from the 50,000' view

- Data Structures
  - Tuples, lists, maps, sets, structs
    - Structs are like maps with a named type and restricted keys
- Comprehensions
- Function chaining
- Lazy and "right now" execution
- Macros, a.k.a "Domain Specific Language"
  - Not like C's simple substitution macros, more like 1960s assembler macros

Some Elixir Data Structures

Tuples

- Tuples are ordered lists of values, which can be of any type
  - a = {1, 2, 3, “infinity”}
  - elem(a, 3) is “infinity”
- Tuple.insert_at(a, 3, :four) returns {1, 2, 3, :four, “infinity"}
  - :four is an atom, which is a literal value
- Access time is constant — elem(some_tuple, 0) takes same time as elem(some_tuple, 999)
- Often returned from function: {:ok, result} or {:error, “reason”}

Elixir Characteristic

Comprehension

- `for i <- 1..5, do: {i, 2*i}`
  is `[{1, 2}, {2, 4}, {3, 6}, {4, 8}, {5, 10}]`

- `for i <- 1..2, j <- [:a, :b, :c], do: {i, j}`
  is `[{1, :a}, {1, :b}, {1, :c}, {2, :a}, {2, :b}, {2, :c}]`

Learning Assessment 2

- Write pseudo code to reverse the elements in a tuple
  - {1, 2, :last} -> {:last, 2, 1}
  - Use these functions (indices are 0-based)
    - tuple_size({1, 2, :last} = 3
    - elem({1, 2, :last}, 2) = :last
  - Iterative: for v <- ***range***, do: ***expression*** returns list of all the values of the expression; List.to_tuple creates a tuple from the given list
- Recursive: put_elem(tuple, index, new_value)
  - put_elem({1, 2, :last}, 2, :really_last) = {1, 2, :really_last}

Wait 1/2 minute

Hints

- for index <- 0..tuple_size(t), do: expression
- Returns a list that is the expression evaluated for each index
- List.to_tuple(a_list) returns a tuple from the list

My solution - switch to Livebook

Assessment

My Solutions —

```
def reverse(t) when is_tuple(t) do
  t_length = tuple_size(t)
  reversed_list = for index <- 1..t_length//1, do:
                    elem(t, t_length - index)
  List.to_tuple(reversed_list)
end

defp reverse2_helper(t, index, size)
      when index < div(tuple_size(t), 2) do
  earlier = elem(t, index)
  later = elem(t, size - index - 1)
  temporary = put_elem(t, index, later)
  temporary2 = put_elem(temporary, size-index-1, earlier)
  reverse2_helper(temporary2, index + 1, size)
end
```

Some Elixir Data Structures

Lists

- Elixir lists are linked lists, each element comprising a value at the head and a list as the tail
  - The last element's tail is an empty list: []
- Access complexity is O(<*list length*>)
- `a = [1 | [2 | [3]]]` is printed as [1, 2, 3]
  - `a = [1, 2, 3]` is the same — shorthand
- `hd(a)` is 1, `tl(a)` is [2, 3], `hd([42])` is 42, `tl([42])` is []
- `a ++ a` is [1, 2, 3, 1, 2, 3]
- Often used to hold things to process

Some Elixir Data Structures

Maps

- Elixir maps are sets of key/value pairs. Keys are unique within a map.
- `%{a_key => a_value, another_key => some_value}`
- Access time depends on the distribution of the keys.
- Often used to hold composite state.
  - Simplified compiler state:
    - `%{:source => ["main(argc, **argv) {}"],`
      `:symbols => %{},`
      `:a_out => [ ... ]}`

Elixir Characteristic

Function Chaining

- Create, Reduce+, Convert is awkward as
  - `convert(reducer3(reducer2(reducer1(create(p1, p2)), aux_param)), :to_string)`
- Convenience:

```
create(p1, p2) |> reducer1() |> reducer2(aux_param) |>
reducer3() |> convert(:to_string)
```

Elixir Characteristic

Lazy and “Right Now!” Execution

- “Right Now!” execution completes each step
- “Lazy” execution creates a stream; each element of a stream is processed when available
- Use streams to reduce memory requirements
    - For instance, to extract metadata from an EXIF file, use streams to avoid holding a 400 megapixel image file in memory

Elixir Characteristic

Macros

- Elixir macros allow processing of the abstract syntax tree generated from the source code
- `if rem(a, 2) == 0, do: :even, else: :odd` becomes a case statement
- Macros are used extensively to create domain specific languages for frameworks
- E.g. this fragment for HTML processing

```
scope "/", OglWeb do
  pipe_through :browser
  live "/", HomeLive
  live "/plans", PlansLive
  live "/tutors", TutorsLive
  live "/students/:tutor_id", StudentsForTutorLive
  live "/student-page", StudentPageLive
end
```

Elixir Characteristics

In addition to those from the 50,000’ view

- Ubiquitous pattern matching
  - To identify which function to execute
  - To destructure a list, map, or tuple
  - To destructure a byte stream
  - Examples

```
[only | []] = [1] (successful match, only is 1)

[only | []] = [1, 2] (unsuccessful match)

{:ok, root} = square_root(10) matches, root is 3.162…

def reverse([a | [] ]), do: a

case square_root(x) do
    {:ok, root} -> root
    {:error, “negative argument”} -> raise “square root negative argument: #{x}”
    {:error, unknown} -> raise “unknown square root error: #{unknown}”
end
```

Elixir Characteristics

Statements

Statement	purpose	Example
defmodule	Namespace functions	defmodule QuickSort do ...
def & def	Define functions	def next(j) do: j + 1 defp helper(work_list) do ... ...
if, cond, case	Conditional value	its_odd = (if 1 == rem(number,2), do: true, else: false)
send and receive	Messages	area = receive do {:result, value} -> value end)
raise	Raise exception	raise “square root of negative number”
throw	Like raise for any value	throw 0
try … catch try … rescue	Handle raise/throw	root = try do sqrt(n) rescue “square root of negative number” -> 0 end

Learning Assessment 3

What's missing?

- At least two statement types are common in languages like C and Java but aren't listed. What are some?

Wait

Assessment 3

Solution

- Elixir doesn’t have looping statements (no for, no while…)
    - Recursion is often used instead. For instance, handle the first element of a list and use recursion to handle the rest of the list
    - Enum.each(range, function), Enum.map(range, function), and Enum.reduce(range, accumulator, function) can replace recursion
- Elixir doesn’t have type declarations (yet).
    - Elixir is dynamically typed
    - Pattern matching can provide some type safety. “Guards” can explicitly check types — “is_integer(n)”
- Elixir has (but I have omitted) use, import, and alias statements for incorporating code that isn’t in the current module

Learning Assessment 4

- Pseudo-code to reverse a list using recursion

Wait

Hints

- The reverse of a list with one element is the list; this terminates the recursion
- Split a list with more than one element into the head and the tail
- The reverse is the head appended to (the end of) the reverse of the tail

My solution - switch to Livebook

Assessment 4

Solution

```
def reverse(l) when is_list(l) and length(l) <= 1, do: l

def reverse([head | tail] = l) when is_list(l) do
  reverse(tail) ++ [head]
end
```

Modules and Frameworks

“You don’t have to write it all yourself”

- Enum & Stream — filter, map, reduce, sort (Enum, not Stream)
- String, List, Tuple, Map, Mapset, Struct — creators, accessors, mutators, converters
- Web pages & SPAs — Phoenix & LiveView
- IoT — Nerves (Custom Linux kernel, process 1 is the BEAM instead of init. Handles over the air updates)
- GraphQL — Absinthe
- Numerical Processing — NX (leverage Google’s XLA work)
- Machine Learning — Axon (e.g. run pre-trained models from https://huggingface.co)

(More) Frameworks

Don’t write it yourself if you don’t have to!

- Database interface — Ecto
- Domain Driven Design — Ash
- Data processing pipeline — Broadway
- Robust execution of background jobs — Oban
- Unit testing and Test Driven Development — ExUnit and DocTest
- JSON — jason
- Certificates — certify
- Event handling — Telemetry

Success Stories

- Three YC alumni use Elixir (at least) (according to Grok)
  - Discord
  - Brex
  - Gigster
- From the web (https://www.freshcodeit.com/blog/leading-companies-using-elixir-7-use-cases) November 2023
  - Discord, Spotify, Pepsi, Toyota Connected, Pinterest, Square Enix, Sketch

Learning Assessment 5

Pick an application in your domain

- List three Modules or Frameworks that are immediately useful
- List three Module or Framework capabilities for your application's future needs
- Homework: Search for those capabilities that are needed; modules or frameworks may exist already
    - Use search prompts like "Elixir framework for trisecting the angle with a compass and straightedge"; Replace "framework" with "module" or "package"

Thinking Time

Assessment 5 Example Solution

My domain is web app for tutoring dyslexic students online

- Immediately useful modules or frameworks:
  - Struct to represent tutoring information; Phoenix + Liveview for web page; Ecto and Ash for domain data; ExUnit for testing (and test-driven development?)
- Future needs:
  - Oban to send periodic status & progress emails; Telemetry to create tutor requests and student response events
    - “Handler” functions would catch the events and add them to the lesson record — helps when tutor plans the next lesson

My Opinion

- Elixir is great
  - Functions only handle what is passed in, so I can easily understand what data the function is working with
  - Functions tend to be small, so I can easily understand each
  - Functions tend to use pattern matching in the definition, so it is obvious what code handles each case
  - Functions compose with CR+C pattern, so I can generate new code or understand old code more easily
  - (Not discussed) The developer experience is great - package management, project control, testing, etc.
  - (Not discussed) Ericsson's "Open Telecom Package" — OTP has great support for writing highly available and reliable applications
  - Scripting
- Elixir isn't great
  - Operating Systems code, interface to C/C++ libraries, applications with random access to large arrays, string processing where Snobol4 is great, numerical analysis codes (e.g. the analytic geometry for and APT system) where Fortran or Python are great

Summary

- Elixir is a good fit for many applications; it has the usual capabilities (except pointers to data)
- Elixir has kept up with the times — Neural networks, ML models, numerical processing, TDD, DDD, etc.
- Elixir code can be written and understood with focus on just a few things
  - It doesn’t overwhelm my working memory
- Many solutions for DevOps

Resources

Getting Started

Abbreviated lists; use your favorite search engine

- Websites (searched “elixir getting started”)
  - https://hexdocs.pm/elixir/introduction.html
  - https://elixir-lang.org/
  - https://elixirschool.com/en/lessons/basics/basics
- Training — search with your favorite search engine for “elixir training”
- Conferences (searched “elixir conferences”
  - ElixirConf: https://elixirconf.com/ , CodeBeam America: https://codebeamamerica.com/ , Gig City Elixir: https://www.gigcityelixir.com/
- LiveBook: https://livebook.dev/

Elixir Books from pragprog.com

Testing Elixir

Learn Functional Programming with Elixir

Metaprogramming Elixir

Machine Learning in Elixir

Adopting Elixir

Exploring Graphs with Elixir

From Ruby to Elixir

Engineering Elixir Applications

Genetic Algorithms in Elixir

Programmer Passport: Elixir

Network Programming in Elixir and Erlang

Build a Binary Clock with Elixir and Nerves

Build a Weather Station with Elixir and Nerves

Concurrent Data Processing in Elixir

Property-Based Testing with PropEr, Erlang, and Elixir

Craft GraphQL APIs in Elixir with Absinthe

Designing Elixir Systems with OTP

Functional Web Development with Elixir, OTP, and Phoenix

Functional Programming: A PragPub Anthology

Ash Framework

Programming Phoenix LiveView

Programming Ecto

Programming Elixir 1.6

Real-Time Phoenix

Seven Web Frameworks in Seven Weeks

Real-World Event Sourcing

Programming Phoenix 1.4

Building Table Views with Phoenix LiveView

Seven More Languages in Seven Weeks

Programmer Passport: OTP

Elixir Books from manning.com

Phoenix in Action

Elixir in Action (3rd edition)

The Little Elixir & OTP Guidebook