Elixir's Set-Theoretic Type System

Robert Ellen

2024/11/12

Summary

Elixir

Elixir

The Erlang Ecosystem

What is the Erlang Ecosystem?

A group of languages, libraries, frameworks, and applications that are implemented on top of the Erlang virtual machine, the BEAM.

Languages include:

...plus dozens more

What is the Erlang Ecosystem?

Libraries and frameworks include:

OTP

What is the Erlang Ecosystem?

Built around a shared value in:

- massive concurrency
- fault-tolerance
- simplicity
- acknowledging the errors will occur so let's deal with them
  - "Let it crash"–have a plan to restart sub-systems when they crash

A brief history of Erlang

A brief history of Erlang

covered in my 2013 talk, but tonight...

A brief history of Erlang

A brief history of Erlang

- developed in the mid 1980s at Ericsson
- to run on next generation telephone switches
  - concurrent, fault-tolerant, distributed, soft real-time
  - strong, dynamic typing, impure, functional, simple
  - reports of 1200k LOC and “nine nines” of uptime on the AXD301 switch
  - reports of market penetration of > 50% in mobile telephony switches
- solved web-scale in the '80s

The BEAM and OTP

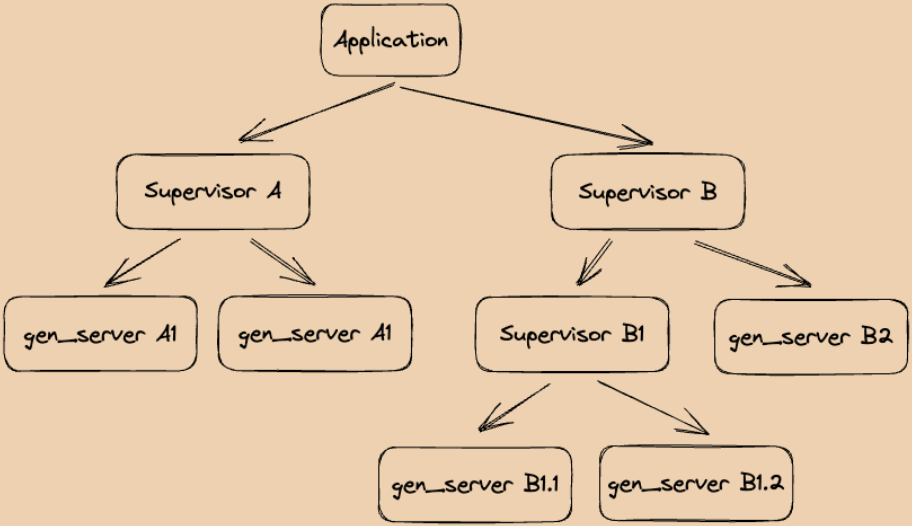
The BEAM

- Erlang's virtual machine / runtime system
- lightweight processes, SMP
- multi-generational, per-process garbage collector
- asynchronous, location-transparent, message-passing IPC
- hot-code-loading
- powerful REPL and introspection tools
- new JIT compiler (BEAM bytecode to machine code)

The OTP libraries

- Erlang's “standard library”
- dozens of modules providing various typical stdlib stuff
- core of which are for supervision trees
  - `gen_server` - actors / workers
  - supervisors - handling starting/stopping/restarting `gen_servers`

supervision trees



Elixir

There will be some comparing and contrasting of Gleam with Elixir...

Brief history of Elixir

- created by José Valim starting in 2012
- inspired by Erlang, Ruby, and, to a lesser extent, Lisp
- like Erlang, the language is quite stable

Features of Elixir

- Ruby-like syntax while retaining most of Erlang's semantics
  - ..., immutable data, HoF, side-effects anywhere, dynamically-typed, ...
- interoperate with Erlang
- hygienic macros
- highly ergonomic build tool: `mix`
- modern package manager: `hex`
- excellent unit test tool: exunit
- opinionated formatter
- drops strict SSA in favour of rebinding

Typing BEAM languages

Marlow & Wadler - 1997

“We can stop waiting for functional languages to be used in practice–that day is here!”

- threw away Hindley-Milner: $U = V$ – this would not work with existing Erlang codebases
- opted for strictly more general “semantic sub-typing” instead: $U \subseteq V$
- (Aiken & Wimmers - 1993)

eqWAlizer

Developed by Meta for WhatsApp

Static analysis tools

- rely on the Erlang Typespec notation - not checked by compiler
- dialyzer - success-typing - Linhahl & Sagonas, 2006
- gradualizer - gradual set-theoretic-inspired typing

Typed BEAM languages with alternate semantics

- Hamler - PureScript for the BEAM
- Caramel - ML for the BEAM
- Gleam - Rust/Ocaml/Elm inspired - HM type system
  - see my May 2024 talk
- ...

Elixir's Set-Theoretic Type System

Goal

- works on existing Elixir code-bases
- need range of techniques to give system a chance of being adopted

Features of the type system

- semantic sub-typing hence set-theoretic
- polymorphic with local type inference
  - type variables
  - requires some type annotations–but not everywhere

Semantic sub-typing

- establish sub-typing relationships between types based on the semantic meaning of values of the types
- semantic meaning derived from treating types as sets, values as set members
- set operations on types: union, intersection, and negation
- in comparison to Hindley-Milner, relax $U = V$ to $U \subseteq V$
- strictly more general - an extension to HM
- Frish et. al. referencing Aiken & Wimmers (also ref. by Marlow & Wadler)
- good idea because dynamically-typed languages variables can hold different types at run-time: hence union-ing

Polymorphic with local type inference

- type variables: `a`, `b` - no parentheses
- local type inference
  - functions must have type annotations
  - types are inferred for arguments and return types

Polymorphic with local type inference

```
$ (list(a), a → b) → list(b)
def map([], _), do: []
def map([x | xs], f), do:  [f.(x) | map(xs, f)]

x = map([1, 2, 3], &double/1)
# type system infers type of double and x
```

Guards and pattern-matching

- Elixir has rich run-time testing of types
- the type system can type captured variables and variables in guards

```
def elem_at([x | rest] = xs, pos) when is_integer(pos) do ...
```

Guards and pattern-matching

- “type narrowing” can check exhaustiveness of case expressions
- type system is conservative: case branches must handle `xs` being any map or list

```
def elem_at(xs, pos) when is_map(xs) or is_list(x) do
  case xs do
    %{} → # get for map
    [] → # get for list
    _ → # redundant
  end
end
```

Maps as “records” and “dictionaries”

- maps can represent records, dictionaries, and structs

```
ashley = %{name: "Ashley", age: 42}
# ashley :: %{:name => binary(), :age => integer()}

words = "The Elixir Type System ..."
word_count = wc(words) # :: %{optional(binary()) => integer()}
word_count["Elixir"] # 42

defstruct [:id , name: "", age: 0]
# %{
#   :__struct__ => :"User",
#   :id => term(),
#   :name => binary(),
#   :age => integer()
# }
```

Maps as “records” and “dictionaries”

- the type system treats maps as open or closed
  - open means there are potentially unknown keys
- strict or dynamic access changes type inference

```
user.first_name # user :: %{:first_name => term(), ...}

middle = person["middle_name"]
# person :: %{optional("middle_name") => term(), ...} => %{...}
# middle :: binary() or nil

ashley = %{name: "Ashley", age: 42}
# ashley :: %{:name => binary(), :age => integer()}
```

Maps as “records” and “dictionaries”

- sub-typing maps feels like structural sub-typing...

```
ashley = %{name: "Ashley", age: 42}
# ashley :: %{:name => binary(), :age => integer()}

ashley_at_school = %{name: "Ashley", age: 42, gpa: 6.75}
# ashley_at_school :: %{:name => binary(),
#                       :age => integer(),
#                       :gpa => float()}

def enroll(%{name: _, age: _} = person) do ...
```

- the type system innovates semantic sub-typing to handle maps
  - Castagna 2023

Gradual typing with `dynamic()`

Gradually introducing the system

- don't discount the chance of a deal-breaker in prod code taking them back to the drawing board

References

Aiken, A. and Wimmers, E. L. 93. Type inclusion constraints and type inference. In Proceed- ings of the Seventh ACM Conference on Functional Programming and Computer Architecture. Copenhagen, Denmark, 31–41

Giuseppe Castagna. Typing records, maps, and structs. Proc. ACM Program. Lang., 7(ICFP), September 2023. doi:10.1145/3607838.

Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. Journal of the ACM, 55(4):1–64, 2008. doi:10.1145/1391289.1391293

Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming, 2006. doi:10.1145/1140335.1140356.

References

Simon Marlow and Philip Wadler. A practical subtyping system for erlang. In Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming, ICFP ’97, page 136–149, New York, NY, USA, 1997. Association for Computing Machinery. doi:10.1145/258948.258962.

Thank you

I'll post slides soon.