Elixir's Set-Theoretic Type System

Robert Ellen

2024/11/12

Summary

- Q: ..., A: “A static type system!”
- rich history of static type systems for BEAM languages
- R&D project to build the definitive static type system for Elixir
- gradual semantic subtyping with innovations for Elixir's features
- gradual introduction into production Elixir compiler

Disclaimer

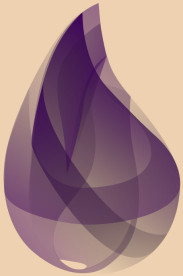
- I am not a student of programming language / type theory
- mistakes are my own
- check out the paper and references

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:



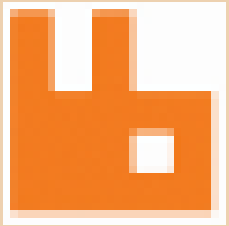
elixir

...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, dynamically typed, 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

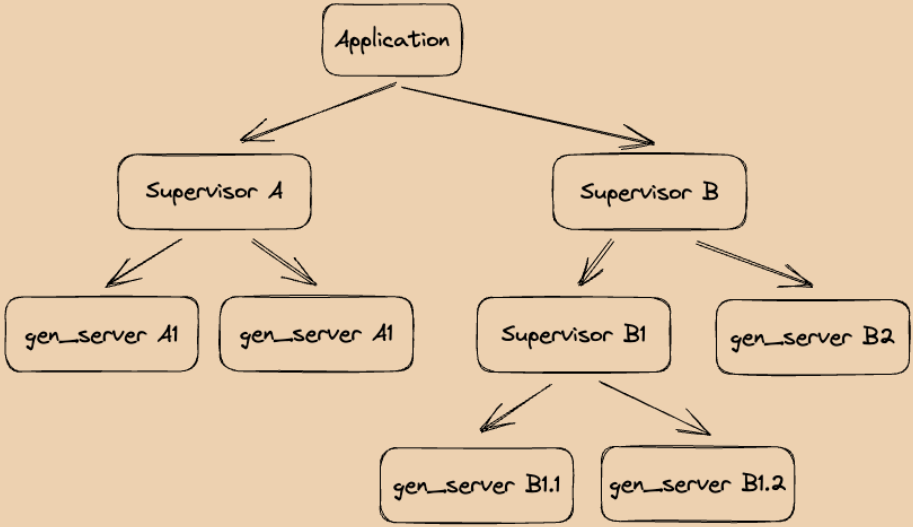
The BEAM

- Erlang's virtual machine / runtime system
- lightweight processes, SMP, low-latency IO, soft realtime, minimal locks
- 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, imminent release of v1.18
- has gone from web applications to embedded systems and numerical analysis and ML/AI

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
- ad-hoc polymorhism with protocols
- highly ergonomic build tool: `mix`
- modern package manager: `hex`
- excellent unit test tool: exunit
- opinionated formatter
- did I mention dynamically-typed?

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: $U \subseteq V$
- this approach developed by Aiken & Wimmers 1993

eqWAlizer

- Developed by Meta for WhatsApp
- set-theoretic gradual typing
- https://github.com/WhatsApp/eqwalizer

Static analysis tools

- rely on the Erlang Typespec notation - not checked by compiler
- dialyzer
  - success-typing based on whole-program analysis
  - will only fail if it can prove there's a problem (no false positives)
  - 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

Elixir's set-theoretic type system

A research and development project to gradually introduce a static type system to Elixir.

Giuseppe Castagna, Guillaume Duboc, and José Valim. The Design Principles of the Elixir Type System The Art, Science, and Engineering of Programming, 2024, Vol. 8, Issue 2, Article 4 https://doi.org/10.22152/programming-journal.org/2024/8/4

Requirements

- no modification to the syntax of Elixir expressions
- gradual typing
- extract maximum type information from patterns and guards
- mode to emit warnings when explicitly using gradual typing
- type annotations across language before advanced features
- initially - typing does not modify the compilation of Elixir code

Features of the type system

- semantic subtyping hence set-theoretic
- parametric polymorphism with local type inference
  - type variables
  - requires some type annotations–but not everywhere
- use patterns and guards
- typing maps in all use-cases
- gradual typing
- strong arrows
- typing ad-hoc polymorphism

Semantic subtyping

- establish subtyping 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

Semantic subtyping

- 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

Subtyping in type checking

- subtyping is a means to typechecking programs due to “subsumption”

$$\frac{e{:}T_1 \quad T_1 <: T_2}{e{:}T_2}$$

- given some expression e and types T_1 and T_2, if the type of e is T_1 and T_1 is a subtype of T_2 then e can be considered to also be of type T_2

Set-theoretic reasoning and type annotations

> the notation is based on Erlang/Elixir typespec

```
integer(), boolean(), binary(), float(), ... # indivisible types

atom(), :foo, # divisible types

list(element()) # element() is a concrete type

container(a) when a: term() # a is a type variable

%{required(:name) ⇒ binary(), ...}
# a map that must have at least a key :name of value binary()

%{required(:name) ⇒ binary(), required(:age) ⇒ integer()}
# a map that can have only :name and :age keys
```

Set-theoretic reasoning and type annotations

> special types

```
term() # top type

none() # bottom type

dynamic() # any type
```

Set-theoretic reasoning and type annotations

```
# union
integer() or boolean()

# intersection
# say we have these typespec types
@type person :: %{:name ⇒ binary(), :age ⇒ integer()}
@type school_record :: %{:school ⇒ binary(), :gpa ⇒ float()}
# and we want a student() type
person() and school_record()

# negation
not integer()

# combinations - e.g. all atoms except true, false, nil
atom() and not(boolean() or nil)
```

Set-theoretic reasoning and type annotations

```
term() === not none()

none() === not term()

boolean() and integer() === none()
```

Set-theoretic reasoning and type annotations

- type annotation of functions with $ - only reserved symbol left unused!
- function arguments comma-separated
- function application →

```
$ (integer(), integer()) → integer()
def add(x,y), do: x + y
```

Set-theoretic reasoning and type annotations

> function types are set-theoretic too

```
$ integer() → integer()
def negate(x) when is_integer(x), do: -x

$ (integer(), integer()) → integer()
def subtract(a, b) when is_integer(a) and is_integer(b) do
  a + negate(b)
end
```

Set-theoretic reasoning and type annotations

> function types are set-theoretic too: let's expand negate to `boolean()`

```
$ (integer() or boolean()) → (integer() or boolean())
def negate(x) when is_integer(x), do: -x
def negate(x) when is_boolean(x), do: not x
```

Set-theoretic reasoning and type annotations

```
$ (integer() or boolean()) → (integer() or boolean())
def negate(x) when is_integer(x), do: -x
def negate(x) when is_boolean(x), do: not x

Type warning:
  | def subtract(a, b) when is_integer(a) and is_integer(b) do
  |   a + negate(b)
        ^ the operator + expects integer(), integer() as arguments
          but the second argument can be integer() or boolean()
```

Set-theoretic reasoning and type annotations

```
$ (integer() or boolean()) → (integer() or boolean())
def negate(x) when is_integer(x), do: -x
def negate(x) when is_boolean(x), do: not x

Type warning:
  | def subtract(a, b) when is_integer(a) and is_integer(b) do
  |   a + negate(b)
        ^ the operator + expects integer(), integer() as arguments
          but the second argument can be integer() or boolean()

# a more precise (and correct in regards to subtract) type
$ (integer() → integer()) and (boolean() → boolean())

# negate is in the set of functions integer() → integer()
# and the set of functions boolean() → boolean()
```

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}
# %{required(:name)⇒ binary(),
#   required(: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”

> subtyping maps feels like structural subtyping...

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

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

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

> but the type system innovates semantic subtyping to handle maps in all cases
>
> – Castagna 2023

Gradual typing with `dynamic()`

- as per requirements, avoid boiling the ocean in existing codebases
- gradual typing: see TypeScript, gradualizer
- a type that “materialises” into any other type
- a type that can be the subtype and supertype of any other type
  - `term()` can only be the later, so need a new type
- `dynamic()`

Gradual typing with `dynamic()`

- “sound gradual typing” - Siek & Taha 2006
- in the presence of dynamic typing, partial static typing still works
- a static type annotation/inference guarantees an expression either:
  - never returns
  - returns a value of the static type
  - emits a runtime exception
- necessitates the addition of runtime checks to the compiled program
- Elixir innovation: as per requirements, no change to the compiled program

Halting dynamic() propagation

> VM and programmer type checks halt the propagation of dynamic()
> functions with these checks are referred to as “strong arrows”

```
$ integer() → integer()
def id_strong(x) when is_integer(x), do: x

$ integer() → integer()
def id_weak(x), do: x

# due to "weak" vs "strong" arrows, the following
# is an acceptable type annotation for `ids(x)`
$ dynamic() → {dynamic(), integer()}
def ids(x), do: {id_weak(x), id_strong(x)}
```

“Solving” the expression problem with protocols

> “ad-hoc” polymorphism akin to typeclasses
> the type system will union all implementations of `String.Chars` to define a type `String.Chars.t()`

```
defmodule MyNewType do
  defstruct [:data]
end

defimpl String.Chars, for: MyNewType do
  def to_string(value) do ... end
end
```

“Solving” the expression problem with protocols

> combines with parametric polymorphism
> please say it is so!

```
# functory.ex
$ Functory.t(a), (a → b) → Functory.t(b)
    when a: term(),  b: term()
def map(xs, f)
...
# my_struct.ex
defimpl Functory, for: MyStruct do
  def map(xs, f) do ... end
end
```

Gradually introducing the system

- don't discount the chance of a deal-breaker in prod code taking them back to the drawing board
- phased approach to introducing the experimental system into production Elixir compiler
  - local inference of some Elixir types: v1.17
  - type all Elixir data types and add typing of (and use information inferred from) pattern matching and guards: v1.18
  - type annotations for functions: v1.?

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.

Giuseppe Castagna, Guillaume Duboc, and José Valim. The Design Principles of the Elixir Type System The Art, Science, and Engineering of Programming, 2024, Vol. 8, Issue 2, Article 4
https://doi.org/10.22152/programming-journal.org/2024/8/4

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

References

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.

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.

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In Scheme and Functional Programming Workshop, University of Chicago Technical Report TR-2006-06, pages 81–92, 2006.

Thank you

I'll post slides soon.