


# Gradualizer
# The practical challenges of implementing a type checker

ElixirConf EU - Prague (09.04.2019)

# Who we are

appliscale

## Helping companies scale:

- BEAM applications
- DevOps: CICD pipelines, Cloud infrastructure
- Adtech consultancy
- Teams and way of working

# Disclaimer

- No type theory
- No history of failed attempts to type Erlang
- Not the author of Gradualizer
- No detailed user guide

# Agenda

- History, context and motivation
- How Gradualizer works
- Challenges
- Hope

# History, context and motivation

# Why type checking

- Static analysis - compile time
- Find more bugs early
- Goal: type check Elixir/Erlang
  not a new statically typed language for the BEAM
- BEAM already has Dialyzer (not a type checker)

# History of Dialyzer

- A DIscrepancy AnaLYZer for ERlang programs
- HiPE team @Uppsala
- Original paper 2004 (Tobias Lindahl/Kostis Sagonas)
- In OTP R10B ~2006
- Data flow analysis → Success typing
- BEAM byte code → Core Erlang

# History of Dialyzer cont.

- Initially no type specs
- EDoc: specs in comments
- EEP8: Types/specs added in OTP R12B (Dec 2007)
- Does not use but verifies specs (overspec/underspec)
- Typer: generate specs

# History of Gradualizer

- Gradual typing for Erlang
- Started by Josef Svenningsson late 2017
- Presentation: CodeBEAM STO 2018
- Still in alpha status

# What is gradual typing

- Combining static and dynamic typing
- Programmer in control of which regions of code are statically or dynamically typed
- Enables gradual evolution of code between the two typing disciplines
- TypeScript, Python, PHP, ...

# What is gradual typing

- Static types + “the dynamic type” (unknown aka **any**)
- Static subtype extended to compatible subtype
  - **any** is a subtype of every type
  - and every type is a subtype of **any**

# Compatible subtype

- Not transitive ⇒ harder constraint solving

  `atom() ~:: any() ~:: float()` ⇏ `atom() ~:: float()`
- Non-intuitive: subtype ⇏ more strict/precise

  `{:error, any()} ~:: {:error, atom()}`

# Gradualizer: Design decisions

- New type system is the existing type specs
- Without type specs no static typing
- Process communication is dynamically typed

# Notation changes

- in Elixir currently
  - `any()` : the top type, the set of all terms
  - `term()` : a shortcut (or alias) for `any()`
  - `'::'` : subtype relation
- for Gradualizer
  - `term()` : the top type, the set of all terms
  - `any()` : the dynamic type
  - `'::'` : compatible subtype relation

# How Gradualizer works

# Success typing vs static typing

- Success typing is optimistic
  Only reports error if none of the possibilities could succeed
- Static typing is pessimistic/strict
  All possibilities must succeed

# Success typing vs static typing example

```
@spec foo(float() | nil) :: any()
def foo(maybe_float), do: bar(maybe_float)

@spec bar(integer() | nil) :: any()
defp bar(maybe_int) do
    if maybe_int, do: Integer.to_string(maybe_int)
end
```

# Success typing vs static typing example

```
@spec foo(float() | nil) :: any()
def foo(maybe_float), do: bar(maybe_float)
```

Dialyzer ✔

# Success typing vs static typing example

```
@spec foo(float() | nil) :: any()
def foo(maybe_float), do: bar(maybe_float)
```

Gradualizer ✖

```
The variable maybe_float is expected to have type
integer() | nil but it has type float() | nil
```

# Dialyzer vs Gradualizer

- Dialyzer: the spec is wrong

```
"Invalid type specification for function Mod.fun/1.
The success typing is (_) -> actual_type()"
```

- Gradualizer: specs are taken seriously

```
"The expression E is expected to have type
return_type [but it has type actual_type]"
```

# Function type signatures

Type specs used for 2 things:

- Implementation:

  Checking the func clauses of the given func

- Usage:

  Checking calls to the given func

# Gradualizer is fast

- Checking a function:
  using only types and specs
- No wait for other funs ⇒ FAST
- (No type inference)
- Max 1 error per function

# Bidirectional type checking

Two modes of Gradualizer:

- Type  checking ↓
- Type propagation ↑

# Type checking mode ↓

- Expected return type of block is not `any()`
  (a type with at least some static parts)
- Propagate type info inward in nested expressions
  (downward in the abstract syntax tree)

# Type propagation mode ↑

- Expected return type of block is `any()`
  (the dynamic type)
- Propagate type info outward in nested expressions
  (upward in the abstract syntax tree)

# Type checking mode ↓ example

```
@spec ... :: {:error, {any(), integer()}}
do
  expr1
  expr2
  {:error, {:problem, M.f(a)}}
end
```

# Type propagation mode ↑ example

```
do
  expr1
  expr2
  {:error, {:problem, M.f(a)}}
end
:: {any(), {any(), mfa_type()}} OR any()
```

# User controls type propagation

- Types are only propagated from
  - function calls to spec'ed funs
  - expressions that contain at least one of the above
- If no type specs

  ⇒ All expressions have type `any()`

  ⇒ No type errors

# Challenges

# Existing rich type system

- Hard to understand semantics
  what is the expected meaning?
- Hard to implement that richness

# Existing rich type system

- basic types (`integer()`, `atom()`, `tuple()`, `list()`, ...)
- union
- literals aka singleton/unit types (`1`, `:foo`, `nil`, ...)
- refined types (infinite subset)
  (`pos_integer()`, `nonempty_maybe_improper_list()`, ...)

# Existing rich type system cont.

- parameterized types
  (`list(integer()), %{required(atom()) => any()}`, ...)
- user-defined and remote types
  (aliases - allow recursive types)

# Spec's with type variables

- `@spec foo(x) :: x when x: list()`
  (extra info: interpretation depends on the tool)
  - just a type alias? (~ `foo(list()) :: list()`)
  - or bounded polymorphic type?
    ie if you pass an int-list it must return an int-list

# Spec’s with type variables

- Type vars occurring only once (OTP): treated as alias
- Unbounded polymorphism
  - OTP: `when x: term()`
  - Elixir: `when x: var`, `when x: any`

  redundant - dropped by Gradualizer

# Annotated types

```
@spec days_since_epoch(year :: integer(),
  month :: integer(), day :: integer()) :: integer()

@spec f(buffalo :: buffalo) … when buffalo: buffalo()

@spec f(annotation :: type_var) …
  when type_var: user_type()
```

# Overloaded spec's

- `@spec function(integer) :: atom`
  `@spec function(atom) :: integer`
- Type of captured function?
  `&function/1 ::`
  `( integer | atom -> integer | atom )`

# Overloaded spec's

- `@spec function(integer) :: atom`
  `@spec function(atom) :: integer`
- Type of captured function?
  `&function/1 :: intersection( (integer -> atom),`
  `(atom -> integer) )`

# Overloaded spec's

- ```
  @spec round(float) :: integer
  @spec round(value) :: value when value: integer
  ```
- no constraints (guards) for arrow types

  ```
  &Kernel.round/1 :: intersection((float->integer),
  (value -> value when value: integer) )
  ```
  ```

# Maps subtyping

- Key types can overlap (unlike overloaded specs)
- Not subtype: `map1 :: map2`

```
map1 :: %{optional(boolean) => integer}
map2 :: %{optional(true) => float,
          optional(atom) => integer}
```

# Variable bindings

- Bind types instead of values
- Variable scoping: implicit in Erlang AST
  (let expressions in Core Erlang)
- Details of semantics for variable binding propagation
  eg.: between tuple elements? try/else? guards? etc.

# Variable bindings example

- Erlang: `var1` bound outside case
- Elixir: not bound

```
case A.b() do
  1 -> var1 = 1
  2 -> var1 = 2
end
var1
```

# Pattern matching - type refinement

Clauses in orelse relation (order matters)

```
@spec ... :: String.t()
  case System.get_env(varname) do
    nil -> "banana";
    value -> value
  end
```

# Pattern matching - type refinement

- Subtract empty list from list ⇒ nonempty list

```
@spec f(list(integer()) :: integer()
def f([]), do: 0
def f(int_list), do: hd(int_list)
```

- Currently only singleton types can be subtracted (integers, atoms, empty list)

# Inline type annotation

- Annotate expressions
  (besides function signatures)

```
receive do
  {:ok, msg} ->
    int = msg :: integer()
```

# Inline type annotation

- Annotate expressions
  (besides function signatures)

```
receive do
  {:ok, msg} ->
    int = annotate_type(msg, integer())
```

# Inline type annotation

- “Staticalize” dynamic types (safe upcase)

```
int = annotate_type(msg, integer())
```

- Refine propagated type (semi safe downcast)

```
arity = assert_type(length(args), arity())
```

# Inline type annotation operators?

- “Staticalize” dynamic types (safe upcase)

```
int = msg :: integer()
```

- Refine propagated type (semi safe downcast)

```
arity = length(args) ?? arity()
```

# Summary

# Current status

- In alpha: no hex pkg, no version tag
- No crash on valid programs
- Lot of false alarms
- CLI, rebar3 plugin, mix task: Gradualixir

# Todo

- Constraint solving, Guards, Clause refinement, Completeness check
- Dialixir-style Elixir syntax in errors
- Protocols, Behaviours
- Inline type annotation (supported for Erlang)

# Place in the toolchain

- Side by side with Dialyzer
  - Won’t replace it ever
- Run before Dialyzer
  - Faster feedback
- Extra benefit of strictness:
  - Fix/improve types and spec’s in OTP

# Summary

- Might find type errors Dialyzer doesn’t
- Fast
- Still a long way to go

Please, try and report errors!

`https://github.com/josefs/Gradualizer`