Bringing Types to Elixir

The design principles of the Elixir type system

Giuseppe Castagna **Guillaume Duboc** José Valim

guillaume.duboc@irif.fr

Why types?

Why types?

A dynamically typed function

```
def add(x) when is_integer(x), do: x + 1
```

$\Rightarrow$ Elixir manipulates types during execution.

Why types?

A dynamically typed function

```
def add(x) when is_integer(x), do: x + 1
```

⇒ Elixir manipulates types during execution.

Elixir Typespecs

```
defmodule StringHelpers do
  @type word() :: String.t()

  @spec long_word?(word()) :: boolean()
  def long_word?(word) when is_binary(word) do
    String.length(word) > 8
  end
end
```

⇒ Mostly for documentation, can be passed to external tools.

(Some) debugging for free

A function that always errors

```
def negate(x) when is_integer(x), do: not x
```

(Some) debugging for free

A function that always errors

```
def negate(x) when is_integer(x), do: not x
```

Type error message

```
Type error:
  |  def negate(x) when is_integer(x), do: not x
                                            ^^^^^

       the operator not expects boolean() as arguments,
       but the argument is integer()
```

Types as contracts between functions.

Types as contracts between functions.

Function A: Negate

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

Types as contracts between functions.

Function A: Negate

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

Types as contracts between functions.

Function A: Negate

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

Function B: Subtract

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

Types as contracts between functions.

Function A: Negate

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

Function B: Subtract

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

Question

- What if we modify the implementation of `negate`?

Union types: correct, not precise enough.

Adding a clause for booleans

```
def negate(x) when is_integer(x), do: -x
def negate(x) when is_boolean(x), do: not x
```

Union types: correct, not precise enough.

Adding a clause for booleans

```
$ (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
```

Union types: correct, not precise enough.

Adding a clause for booleans

```
$ (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
```

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

Union types: correct, not precise enough.

Adding a clause for booleans

```
$ (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
```

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

Type error in subtract

```
Type error:
  | 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()
```

Type intersections are used to represent functions.

A more precise annotation

```
def negate(x) when is_integer(x), do: -x
def negate(x) when is_boolean(x), do: not x
```

Type intersections are used to represent functions.

A more precise annotation

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

Type intersections are used to represent functions.

A more precise annotation

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

The function `subtract` type checks

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

Negation and singletons

Logical negation

```
def logical_neg(x) when x == false or x == nil, do: true
def logical_neg(x), do: false
```

Negation and singletons

Logical negation

```
$ (false or nil -> true) and
def logical_neg(x) when x == false or x == nil, do: true
def logical_neg(x), do: false
```

Types

- Singleton types are types containing exactly one value

Negation and singletons

Logical negation

```
$ (false or nil -> true) and (not (false or nil) -> false)
def logical_neg(x) when x == false or x == nil, do: true
def logical_neg(x), do: false
```

Types

- Singleton types are types containing exactly one value
- Negation types contain any value that is *not* in the negated type.

Polymorphism with Local Type Inference

Expressions and functions can have types containing type variables

```
def map([h | t], fun), do: [fun.(h) | map(t, fun)]
def map([], _fun), do: []
```

Polymorphism with Local Type Inference

Expressions and functions can have types containing type variables

```
$ ([a], (a -> b)) -> [b] when a: term(), b: term()
def map([h | t], fun), do: [fun.(h) | map(t, fun)]
def map([], _fun), do: []
```

Type Variables

- Quantified using a postfix `when` with upper bounds

Polymorphism with Local Type Inference

Expressions and functions can have types containing type variables

```
$ ([a], (a -> b)) -> [b] when a: term(), b: term()
def map([h | t], fun), do: [fun.(h) | map(t, fun)]
def map([], _fun), do: []
```

Type Variables

- Quantified using a postfix `when` with upper bounds

System deduces type variable instantiation

```
map([{0, true}], fn {x, y} when is_integer(x) -> x end)
# type [integer()]
```

Intersection and polymorphism

Logical OR

```
def logical_or(x, y) when x == false or x == nil, do: y
def logical_or(x, _), do: x
```

Intersection and polymorphism

Logical OR

```
$ (false or nil, a -> a) and

def logical_or(x, y) when x == false or x == nil, do: y
def logical_or(x, _), do: x
```

Intersection and polymorphism

Logical OR

```
$ (false or nil, a -> a) and
  (b, term() -> b) when a: term(), b: not(false or nil)
def logical_or(x, y) when x == false or x == nil, do: y
def logical_or(x, _), do: x
```

Protocols in Elixir

String.Chars protocol

```
defprotocol String.Chars do
  @doc """
  Converts a data type to a human-readable
  string representation.
  """
  def to_string(data)
end
```

Protocols in Elixir

String.Chars protocol

```
defprotocol String.Chars do
  @doc """
  Converts a data type to a human-readable
  string representation.
  """
  def to_string(data)
end
```

Union of Types Implementing a Protocol

- Denoted by `String.Chars.t()`
- Automatically filled in by the Elixir compiler
- Previously approximated by `term()`

Protocols in Elixir

String.Chars protocol

```
defprotocol String.Chars do
  @doc """
  Converts a data type to a human-readable
  string representation.
  """
  def to_string(data)
end
```

Union of Types Implementing a Protocol

- Denoted by `String.Chars.t()`
- Automatically filled in by the Elixir compiler
- Previously approximated by `term()`

```
String.Chars.t() = binary() or integer() or list() or ...
```

First-class support

Solved problems

- Parametrize protocols (5-year-old issue requesting it! #7541)

First-class support

Solved problems

- Parametrize protocols (5-year-old issue requesting it! #7541)

Current typespec for Enum.into

```
into(Enumerable.t(), Collectable.t()) :: Collectable.t()
```

First-class support

Solved problems

- Parametrize protocols (5-year-old issue requesting it! #7541)

Current typespec for Enum.into

```
into(Enumerable.t(), Collectable.t()) :: Collectable.t()
```

```
Enumerable.t(a) when a: term()
```

First-class support

Solved problems

- Parametrize protocols (5-year-old issue requesting it! #7541)

Current typespec for Enum.into

```
into(Enumerable.t(), Collectable.t()) :: Collectable.t()
```

```
Enumerable.t(a), Collectable.t(b) -> Collectable.t(a or b)
    when a: term(), b: term()
```

First-class support

Solved problems

- Parametrize protocols (5-year-old issue requesting it! #7541)
- Composition (via type intersections)

Current typespec for Enum.into

```
into(Enumerable.t(), Collectable.t()) :: Collectable.t()
```

```
Enumerable.t(a), Collectable.t(b) -> Collectable.t(a or b)
    when a: term(), b: term()
```

First-class support

```
$ type traversable(a) = Enumerable.t(a) and Collectable.t(a)
```

First-class support

```
$ type traversable(a) = Enumerable.t(a) and Collectable.t(a)
```

Echo

```
def echo(var) do
    Enum.into(var, var)
end
```

First-class support

```
$ type traversable(a) = Enumerable.t(a) and Collectable.t(a)
```

Echo

```
$ a -> a when a: traversable(string())
def echo(var) do
    Enum.into(var, var)
end
```

```
iex(1)> echo(IO.stream())
ah
ah
```

First-class support

```
$ type traversable(a) = Enumerable.t(a) and Collectable.t(a)
```

Echo

```
$ a -> a when a: traversable(string())
def echo_upcase(var) do
    Enum.into(var, var, &String.upcase/1)
end
```

First-class support

```
$ type traversable(a) = Enumerable.t(a) and Collectable.t(a)
```

Echo

```
$ a -> a when a: traversable(string())
def echo_upcase(var) do
    Enum.into(var, var, &String.upcase/1)
end
```

```
iex(1)> echo_upcase(IO.stream())
ah
AH
```

Pattern Matching: Case Statement

Using a case statement

```
def negate(x), do: (case x do
  x when is_integer(x) -> -x
  x when is_boolean(x) -> not x
end)
```

Using multiple function clauses

```
def negate(x) when is_integer(x), do: x
def negate(x) when is_boolean(x), do: x
```

Exhaustivity and redundancy in pattern matching

```
$ type result() =
    %{output: :ok, socket: socket()} or
    %{output: :error, message: :timeout or {:delay, integer()}}
```

Exhaustivity and redundancy in pattern matching

```
$ type result() =
    %{output: :ok, socket: socket()} or
    %{output: :error, message: :timeout or {:delay, integer()}}
```

Exhaustivity checking: handling all cases

```
$ result() -> string()
def handle(r) when r.output == :ok, do: "Msg received"
def handle(r) when r.message == :timeout, do: "Timeout"
```

Exhaustivity and redundancy in pattern matching

```
$ type result() =
    %{output: :ok, socket: socket()} or
    %{output: :error, message: :timeout or {:delay, integer()}}
```

Exhaustivity checking: handling all cases

```
$ result() -> string()
def handle(r) when r.output == :ok, do: "Msg received"
def handle(r) when r.message == :timeout, do: "Timeout"
#=> Type Warning: non-exhaustive pattern matching
```

Exhaustivity and redundancy in pattern matching

```
$ type result() =
    %{output: :ok, socket: socket()} or
    %{output: :error, message: :timeout or {:delay, integer()}}
```

Exhaustivity checking: handling all cases

```
$ result() -> string()
def handle(r) when r.output == :ok, do: "Msg received"
def handle(r) when r.message == :timeout, do: "Timeout"
#=> Type Warning: non-exhaustive pattern matching
```

Redundancy checking: detecting unused clauses

```
$ result() -> string()
def handle(r) when r.output == :ok, do: "Msg received"
def handle(r) when r.output == :error, do: "Error raised"
def handle(%{socket: _}), do: "Socket found"
```

Exhaustivity and redundancy in pattern matching

```
$ type result() =
    %{output: :ok, socket: socket()} or
    %{output: :error, message: :timeout or {:delay, integer()}}
```

Exhaustivity checking: handling all cases

```
$ result() -> string()
def handle(r) when r.output == :ok, do: "Msg received"
def handle(r) when r.message == :timeout, do: "Timeout"
#=> Type Warning: non-exhaustive pattern matching
```

Redundancy checking: detecting unused clauses

```
$ result() -> string()
def handle(r) when r.output == :ok, do: "Msg received"
def handle(r) when r.output == :error, do: "Error raised"
def handle(%{socket: _}), do: "Socket found"
#=> Type Warning: unused branch
```

Narrowing: inferring more precise types

```
$ type result() =
    %{output: :ok, socket: socket()} or
    %{output: :error, message: :timeout or {:delay, integer()}}
```

Automatically infer a return type

```
$ result() -> _
def handle(r) when r.output == :ok, do: {:accepted, r.socket}
def handle(r) when is_atom(r.message), do: r.message
def handle(r), do: {:retry, elem(r.message, 1)}
```

Narrowing: inferring more precise types

```
$ type result() =
    %{output: :ok, socket: socket()} or
    %{output: :error, message: :timeout or {:delay, integer()}}
```

Automatically infer a return type

```
$ result() -> _
def handle(r) when r.output == :ok, do: {:accepted, r.socket}
def handle(r) when is_atom(r.message), do: r.message
def handle(r), do: {:retry, elem(r.message, 1)}
#=> Return type: {:accept, socket()} or :timeout or {:retry, integer()}
```

Narrowing: inferring more precise types

```
$ type result() =
    %{output: :ok, socket: socket()} or
    %{output: :error, message: :timeout or {:delay, integer()}}
```

Automatically infer a return type

```
$ result() -> _
def handle(r) when r.output == :ok, do: {:accepted, r.socket}
def handle(r) when is_atom(r.message), do: r.message
def handle(r), do: {:retry, elem(r.message, 1)}
#=> Return type: {:accept, socket()} or :timeout or {:retry, integer()}
```

Inferring the function's more precise type

```
$ (%{output: :ok, socket: socket()} -> {:accept,socket()}) and
  (%{output: :error, message: :timeout} -> :timeout) and
  (%{output: :error, message: {:delay,integer()}} -> {:retry,integer()})
```

Map types

```
$ type t() = %{foo: atom(),
               optional(:bar) => atom(),
               optional(atom()) => integer()}
```

Map types

```
$ type t() = %{foo: atom(),
               optional(:bar) => atom(),
               optional(atom()) => integer()}
```

With `m` of type `t()`

```
m.foo
# type atom()
```

Map types

```
$ type t() = %{foo: atom(),
               optional(:bar) => atom(),
               optional(atom()) => integer()}
```

With `m` of type `t()`

```
m.foo
# type atom()
m.bar
#=> Type error: key :bar may be undefined
```

Map types

```
$ type t() = %{foo: atom(),
               optional(:bar) => atom(),
               optional(atom()) => integer()}
```

With `m` of type `t()`

```
m.foo
# type atom()
m.bar
#=> Type error: key :bar may be undefined
m[:bar]
# type atom() or nil
```

Map types

```
$ type t() = %{foo: atom(),
               optional(:bar) => atom(),
               optional(atom()) => integer()}
```

With `m` of type `t()`

```
m.foo
# type atom()
m.bar
#=> Type error: key :bar may be undefined
m[:bar]
# type atom() or nil
m[:other]
# type integer() or nil
```

Map types

```
$ type t() = %{foo: atom(),
               optional(:bar) => atom(),
               optional(atom()) => integer()}
```

With `m` of type `t()`

```
m.foo
# type atom()
m.bar
#=> Type error: key :bar may be undefined
m[:bar]
# type atom() or nil
m[:other]
# type integer() or nil
m[42]
# type nil   (...and a warning)
```

Gradual typing

Principles

- Blend statically typed and dynamically typed code
- Gradual migration instead of converting entire codebase at once
- Introduce `dynamic()` type for type-checking in dynamic typing mode

The `dynamic()` Type

- May turn into any other type at runtime
- Offers a more relaxed type safety guarantee
- If some program `e` has type `integer()` and evaluates to a value, then this value is of type `integer()`.

Taming `dynamic()` with Elixir type tests

(Weak) function type

```
$ integer() -> integer()
def id_weak(x), do: x
```

Taming dynamic() with Elixir type tests

(Weak) function type

```
$ integer() -> integer()
def id_weak(x), do: x
```

x_dyn of type dynamic()

```
id_weak(x_dyn)
# type dynamic()
```

Taming dynamic() with Elixir type tests

(Weak) function type

```
$ integer() -> integer()
def id_weak(x), do: x
```

x_dyn of type dynamic()

```
id_weak(x_dyn)
# type dynamic()
```

Strong function type

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

Taming dynamic() with Elixir type tests

(Weak) function type

```
$ integer() -> integer()
def id_weak(x), do: x
```

x_dyn of type dynamic()

```
id_weak(x_dyn)
# type dynamic()
```

```
id_strong(x_dyn)
# type integer()
```

Strong function type

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

Taming dynamic() with Elixir type tests

(Weak) function type

```
$ integer() -> integer()
def id_weak(x), do: x
```

x_dyn of type dynamic()

```
id_weak(x_dyn)
# type dynamic()
```

```
id_strong(x_dyn)
# type integer()
```

Strong function type

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

- Functions with strong types guarantee correct output or runtime type-check failure
- Ensures type safety guarantee without modifying Elixir compilation

Passing dynamic() around

```
def negate(x) when is_integer(x), do: -x
def negate(x) when is_boolean(x), do: not x
```

Passing dynamic() around

```
def negate(x) when is_integer(x), do: -x
def negate(x) when is_boolean(x), do: not x
#=> Possible type: integer() or boolean()
```

```
def subtract(a, b) do
    a + negate(b)
end
```

Passing dynamic() around

```
def negate(x) when is_integer(x), do: -x
def negate(x) when is_boolean(x), do: not x
#=> Possible type: integer() or boolean()
```

```
def subtract(a, b) do
    a + negate(b)
end
#=> Type Error: + undefined for booleans
```

Passing dynamic() around

```
def negate(x) when is_integer(x), do: -x
def negate(x) when is_boolean(x), do: not x
#=> Result type: (integer() or boolean()) and dynamic()
```

```
def subtract(a, b) do
    a + negate(b)
end
```

Passing dynamic() around

```
def negate(x) when is_integer(x), do: -x
def negate(x) when is_boolean(x), do: not x
#=> Result type: (integer() or boolean()) and dynamic()
```

```
def subtract(a, b) do
    a + negate(b)
end
#=> Result type: integer()
```

Passing dynamic() around

```
def negate(x) when is_integer(x), do: -x
def negate(x) when is_boolean(x), do: not x
#=> Result type: (integer() or boolean()) and dynamic()
```

```
def subtract(a, b) do
    a + negate(b)
end
#=> Result type: integer()
```

```
def concat(a, b) do
    a ++ negate(b)
end
#=> Type Error: incompatible types.
```

Current Progress

Project Milestones

- Formalized a complete type system
- Defined a syntax for types
- Implemented a prototype
- Present our design choices
- Publishing our approach
- Integrating the system with the Elixir compiler

Current Progress

Project Milestones

- Formalized a complete type system
- Defined a syntax for types
- Implemented a prototype
- Present our design choices
- Publishing our approach
- Integrating the system with the Elixir compiler

Feedback welcome!

⇒ `https://www.irif.fr/~gduboc`

Current Progress

Project Milestones

- Formalized a complete type system
- Defined a syntax for types
- Implemented a prototype
- Present our design choices
- Publishing our approach
- Integrating the system with the Elixir compiler

Feedback welcome!

⇒ `https://www.irif.fr/~gduboc`

⇒ `https://typex.fly.dev/` (highly experimental!)

Current Progress

Project Milestones

- Formalized a complete type system
- Defined a syntax for types
- Implemented a prototype
- Present our design choices
- Publishing our approach
- Integrating the system with the Elixir compiler

Feedback welcome!

⇒ `https://www.irif.fr/~gduboc`

⇒ `https://typex.fly.dev/` (highly experimental!) ... did I say **highly** experimental?