Work supported by: supabase and fresha

A Type System for Elixir

Giuseppe Castagna Guillaume Duboc José Valim

Erlang 2023

Seattle - September the 4th

INSTITUT DE RECHERCHE EN INFORMATIQUE FONDAMENTALE

remote

Dashbit

Outline

Outline

Static Typing 101

A function that always errors

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

Static Typing 101

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, but 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, but 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, but 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, but 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 for 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 for 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 for 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
```

Singleton and Negation Types

Logical negation

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

Singleton and Negation Types

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

Singleton and Negation Types

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()          // i.e., a-lists, a-sets, a-ranges
```

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
```

Outline

Pattern Matching: Case expression

Using a case expression

```
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 a more precise type

```
$ (%{output: :ok, socket: socket()} -> {:accept,socket()}) and
  (%{output: :error, message: :timeout} -> :timeout) and
  (%{output: :error, message: {:delay,integer()}} -> {: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(r)
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 a more precise type

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

A Pinch of Formalization

α type variables, c constants, x variables

Base types	b	$::=$	$\texttt{int} \mid \texttt{atom} \mid \texttt{function} \mid \texttt{tuple}$
Types	t, s	$::=$	$b \mid c \mid \alpha \mid \overline{t} \to t \mid \{\overline{t}\} \mid t \vee t \mid t \wedge t \mid \neg t$
Expressions	e, f	$::=$	$c \mid x \mid \lambda(\overline{x}.\, e) \mid f(\overline{e}) \mid \{\overline{e}\} \mid \texttt{elem}(e, e)$
		$\mid$	$\texttt{let}\, x : t = e\, \texttt{in}\, e \mid \texttt{case}\, e\, \texttt{do}\, \overline{pg \to e}$
Patterns	p	$::=$	$x \mid c \mid \{\overline{p}\}$
Guards	g	$::=$	$g\, \texttt{and}\, g \mid g\, \texttt{or}\, g \mid \texttt{not}\, g \mid \texttt{is\_integer}(d)$
		$\mid$	$\texttt{is\_atom}(d) \mid \texttt{is\_tuple}(d) \mid \texttt{is\_function}(d, d)$
		$\mid$	$d == d \mid d\, !\!= d \mid d < d \mid d <= d$
Selectors	d	$::=$	$c \mid x \mid \texttt{elem}(d, d) \mid \texttt{tuple\_size}(d)$

Notation: $\overline{u} = u_1, \ldots, u_n$

A Pinch of Formalization

α type variables, c constants, x variables

Base types	b	$::=$	$\texttt{int} \mid \texttt{atom} \mid \texttt{function} \mid \texttt{tuple}$
Types	t, s	$::=$	$b \mid c \mid \alpha \mid \overline{t} \to t \mid \{\overline{t}\} \mid t \vee t \mid t \wedge t \mid \neg t$
Expressions	e, f	$::=$	$c \mid x \mid \lambda(\overline{x.\, e}) \mid f(\overline{e}) \mid \{\overline{e}\} \mid \texttt{elem}(e, e)$
		$\mid$	$\texttt{let}\, x : t = e\, \texttt{in}\, e \mid \texttt{case}\, e\, \texttt{do}\, \overline{pg \to e}$
Patterns	p	$::=$	$x \mid c \mid \{\overline{p}\}$
Guards	g	$::=$	$g\, \texttt{and}\, g \mid g\, \texttt{or}\, g \mid \texttt{not}\, g \mid \texttt{is\_integer}(d)$
		$\mid$	$\texttt{is\_atom}(d) \mid \texttt{is\_tuple}(d) \mid \texttt{is\_function}(d, d)$
		$\mid$	$d == d \mid d\, !\!= d \mid d < d \mid d <= d$
Selectors	d	$::=$	$c \mid x \mid \texttt{elem}(d, d) \mid \texttt{tuple\_size}(d)$

Notation: $\overline{u} = u_1, \ldots, u_n$

A Pinch of Formalization

α type variables, c constants, x variables

Base types	b	$::=$	$\texttt{int} \mid \texttt{atom} \mid \texttt{function} \mid \texttt{tuple}$
Types	t, s	$::=$	$b \mid c \mid \alpha \mid \overline{t} \to t \mid \{\overline{t}\} \mid t \vee t \mid t \wedge t \mid \neg t$
Expressions	e, f	$::=$	$c \mid x \mid \lambda(\overline{x}.\, e) \mid f(\overline{e}) \mid \{\overline{e}\} \mid \texttt{elem}(e, e)$
		$\mid$	$\texttt{let}\, x : t = e\, \texttt{in}\, e \mid \texttt{case}\, e\, \texttt{do}\, \overline{pg \to e}$
Patterns	p	$::=$	$x \mid c \mid \{\overline{p}\}$
Guards	g	$::=$	$g\, \texttt{and}\, g \mid g\, \texttt{or}\, g \mid \texttt{not}\, g \mid \texttt{is\_integer}(d)$
		$\mid$	$\texttt{is\_atom}(d) \mid \texttt{is\_tuple}(d) \mid \texttt{is\_function}(d, d)$
		$\mid$	$d == d \mid d\, \texttt{!=}\, d \mid d < d \mid d <= d$
Selectors	d	$::=$	$c \mid x \mid \texttt{elem}(d, d) \mid \texttt{tuple\_size}(d)$

Notation: $\overline{u} = u_1, \ldots, u_n$

Types and subtyping

Types as sets of values

$$[\![\texttt{bool}]\!] = \{\texttt{true}, \texttt{false}\} \qquad [\![\texttt{int}]\!] = \{\texttt{0}, \texttt{1}, \texttt{-1}, \texttt{2}, \texttt{-2}, \ldots\}$$

$$[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!] \qquad [\![\neg t]\!] = \mathsf{Values} \setminus [\![t]\!]$$

Subtyping as set containment

$$t_1 \leq t_2 \quad \overset{\text{def}}{\iff} \quad [\![t_1]\!] \subseteq [\![t_2]\!]$$

Types for patterns and guards

We can associate to a pair pattern-guard p `when` g two types (i.e., two sets of values):

1. Surely accepted type: largest type that contains only values that match p `when` g (noted $\lfloor pg \rfloor$)
2. Possibly accepted type: smallest type that contains all values that match p `when` g (noted $\lceil pg \rceil$)

Types for patterns and guards

We can associate to a pair pattern-guard p `when` g two types (i.e., two sets of values):

1. Surely accepted type: largest type that contains only values that match p `when` g (noted $\wr pg \wr$)
2. Possibly accepted type: smallest type that contains all values that match p `when` g (noted $\lfloor pg \rfloor$)

Example

Let p `when` g be `x when (is_map(x) and map_size(x) == 2) or is_list(x)`

- $\lfloor pg \rfloor$ = `map() or list()` (it can accept only maps or lists)
- $\wr pg \wr$ = `list()` (it will surely accept all the lists)

Types for patterns and guards

We can associate to a pair pattern-guard p `when` g two types (i.e., two sets of values):

1. Surely accepted type: largest type that contains only values that match p `when` g (noted $\wr pg \wr$)
2. Possibly accepted type: smallest type that contains all values that match p `when` g (noted $\lfloor pg \rfloor$)

Example

Let p `when` g be `x when (is_map(x) and map_size(x) == 2) or is_list(x)`

- $\lfloor pg \rfloor$ = `map() or list()` (it can accept only maps or lists)
- $\wr pg \wr$ = `list()` (it will surely accept all the lists)

Types for patterns and guards

We can associate to a pair pattern-guard p `when` g two types (i.e., two sets of values):

1. Surely accepted type: largest type that contains only values that match p `when` g (noted $\wr pg \wr$)
2. Possibly accepted type: smallest type that contains all values that match p `when` g (noted $\lfloor pg \rfloor$)

Example

Let p `when` g be `x when (is_map(x) and map_size(x) == 2) or is_list(x)`

- $\lfloor pg \rfloor$ = `map() or list()` (it can accept only maps or lists)
- $\wr pg \wr$ = `list()` (it will surely accept all the lists)

Properties

$- \vdash v : \wr pg \wr \quad \Rightarrow \quad v$ matches pg

$- \vdash v : \lfloor pg \rfloor \quad \Leftarrow \quad v$ matches pg

Types for patterns and guards

We can associate to a pair pattern-guard p `when` g two types (i.e., two sets of values):

1. Surely accepted type: largest type that contains only values that match p `when` g (noted $⌈pg⌋$)
2. Possibly accepted type: smallest type that contains all values that match p `when` g (noted $⌊pg⌉$)

Example

Let p `when` g be `x when (is_map(x) and map_size(x) == 2) or is_list(x)`

- $⌊pg⌉$ = `map() or list()` (it can accept only maps or lists)
- $⌈pg⌋$ = `list()` (it will surely accept all the lists)

Properties

$$\left.\begin{array}{lcll} \_ \vdash v : ⌈pg⌋ & \Rightarrow & v \text{ matches } pg & \text{under} \\ \_ \vdash v : ⌊pg⌉ & \Leftarrow & v \text{ matches } pg & \text{over} \end{array}\right\} \text{-approximation of the set } \{v \mid v \text{ matches } pg\}$$

Typing pattern matching

We use possible/surely accepted types to type case expressions:

Consider $\texttt{case}\ e\ \texttt{do}\ p_1 g_1 \rightarrow e_1\ ;\ ....; p_n g_n \rightarrow e_n$ with $e : t_0$

Typing pattern matching

We use possible/surely accepted types to type case expressions:

Consider $\texttt{case}\ e\ \texttt{do}\ p_1 g_1 \to e_1\ ;\ ....; p_n g_n \to e_n$ with $e : t_0$

the type t_i of all values that *may* be captured by the i-th branch is

$$t_i = (t_0 \setminus \bigvee_{j<i} \lceil p_j g_j \rfloor) \wedge \lfloor p_i g_i \rceil$$

Typing pattern matching

We use possible/surely accepted types to type case expressions:

Consider $\texttt{case}\ e\ \texttt{do}\ p_1 g_1 \rightarrow e_1\ ;\ ....; p_n g_n \rightarrow e_n$ with $e : t_0$

the type t_i of all values that *may* be captured by the i-th branch is

$$t_i = (t_0 \setminus \bigvee_{j<i} \lceil p_j g_j \rfloor) \wedge \lfloor p_i g_i \rceil$$

In words: the expression e_i will process values:

Typing pattern matching

We use possible/surely accepted types to type case expressions:

Consider $\texttt{case}\ e\ \texttt{do}\ p_1 g_1 \rightarrow e_1\ ;\ ....; p_n g_n \rightarrow e_n$ with $e : t_0$

the type t_i of all values that *may* be captured by the i-th branch is

$$t_i = (t_0 \setminus \bigvee_{j<i} \lceil p_j g_j \rfloor) \wedge \lfloor p_i g_i \rceil$$

In words: the expression e_i will process values:
- generated by e (i.e., in t_0)

Typing pattern matching

We use possible/surely accepted types to type case expressions:

Consider `case` e `do` $p_1 g_1 \to e_1$; ....; $p_n g_n \to e_n$ with $e : t_0$

the type t_i of all values that *may* be captured by the i-th branch is

$$t_i = (t_0 \setminus \bigvee_{j<i} \lceil p_j g_j \rfloor) \wedge \lfloor p_i g_i \rceil$$

In words: the expression e_i will process values:
- generated by e (i.e., in t_0)
- minus those *surely* captured by the previous branches (i.e., in $\bigvee_{j<i} \lceil p_j g_j \rfloor$)

Typing pattern matching

We use possible/surely accepted types to type case expressions:

Consider `case` e `do` $p_1 g_1 \to e_1$; ....; $p_n g_n \to e_n$ with $e : t_0$

the type t_i of all values that *may* be captured by the i-th branch is

$$t_i = (t_0 \setminus \bigvee_{j<i} \lceil p_j g_j \rfloor) \wedge \lfloor p_i g_i \rceil$$

In words: the expression e_i will process values:
- generated by e (i.e., in t_0)
- minus those *surely* captured by the previous branches (i.e., in $\bigvee_{j<i} \lceil p_j g_j \rfloor$)
- and that *may* match p_i `when` g_i (i.e., in $\lfloor p_i g_i \rceil$)

Typing pattern matching

We use possible/surely accepted types to type case expressions:

Consider `case` e `do` $p_1 g_1 \to e_1$; ....; $p_n g_n \to e_n$ with $e : t_0$

the type t_i of all values that *may* be captured by the i-th branch is

$$t_i = (t_0 \setminus \bigvee_{j<i} \lceil p_j g_j \rfloor) \wedge \lfloor p_i g_i \rceil$$

In words: the expression e_i will process values:
- generated by e (i.e., in t_0)
- minus those *surely* captured by the previous branches (i.e., in $\bigvee_{j<i} \lceil p_j g_j \rfloor$)
- and that *may* match p_i `when` g_i (i.e., in $\lfloor p_i g_i \rceil$)

$$\text{(case)}\ \frac{\Gamma \vdash e : t_0 \qquad (\forall i < n \quad \Gamma, (t_i/p_i) \vdash e_i : s_i)}{\Gamma \vdash \texttt{case}\ e\ \texttt{do}\ (p_i g_i \to e_i)_{i<n} : \bigvee_{\{i \mid t_i \not\simeq \mathbb{0}\}} s_i} \quad t_0 \leq \bigvee_{i \leq n} \lfloor p_i g_i \rceil$$

Typing pattern matching

We use possible/surely accepted types to type case expressions:

Consider `case` e `do` $p_1 g_1 \to e_1$; ....; $p_n g_n \to e_n$ with $e : t_0$

the type t_i of all values that *may* be captured by the i-th branch is

$$t_i = (t_0 \setminus \bigvee_{j<i} \lceil p_j g_j \rfloor) \wedge \lfloor p_i g_i \rceil$$

derive the types of the variables in p_i from t_i

In words: the expression e_i will process values:
- generated by e (i.e., in t_0)
- minus those surely captured by the previous branches (i.e., in $\bigvee_{j<i} \lceil p_j g_j \rfloor$)
- and that may match p_i `when` g_i (i.e., in $\lfloor p_i g_i \rceil$)

$$\text{(case)}\ \frac{\Gamma \vdash e : t_0 \qquad (\forall i < n \quad \Gamma, (t_i/p_i) \vdash e_i : s_i)}{\Gamma \vdash \texttt{case}\ e\ \texttt{do}\ (p_i g_i \to e_i)_{i<n} : \bigvee_{\{i \mid t_i \not\simeq \mathbb{0}\}} s_i} \qquad t_0 \leq \bigvee_{i \leq n} \lfloor p_i g_i \rceil$$

Typing pattern matching

We use possible/surely accepted types to type case expressions:

Consider `case` e `do` $p_1 g_1 \to e_1$; ....; $p_n g_n \to e_n$ with $e : t_0$

the type t_i of all values that *may* be captured by the i-th branch is

$$t_i = (t_0 \setminus \bigvee_{j<i} \lceil p_j g_j \rfloor) \wedge \lfloor p_i g_i \rceil$$

derive the types of the variables in p_i from t_i

In words: the expression e_i will process values:
- generated by e (i.e., in t_0)
- minus those surely captured by the previous branches (i.e., in $\bigvee_{j<i} \lceil p_j g_j \rfloor$)
- and that may match p_i `when` g_i (i.e., in $\lfloor p_i g_i \rceil$)

exaustivity condition

$$\text{(case)}\ \frac{\Gamma \vdash e : t_0 \qquad (\forall i < n \quad \Gamma, (t_i/p_i) \vdash e_i : s_i)}{\Gamma \vdash \texttt{case}\ e\ \texttt{do}\ (p_i g_i \to e_i)_{i<n} : \bigvee_{\{i \mid t_i \not\simeq \mathbb{0}\}} s_i} \qquad t_0 \leq \bigvee_{i \leq n} \lfloor p_i g_i \rceil$$

Typing pattern matching

We use possible/surely accepted types to type case expressions:

Consider `case` e `do` $p_1 g_1 \to e_1$; ....; $p_n g_n \to e_n$ with $e : t_0$

the type t_i of all values that *may* be captured by the i-th branch is

$$t_i = (t_0 \setminus \bigvee_{j<i} \lceil p_j g_j \rfloor) \wedge \lceil p_i g_i \rceil$$

use only the branches that may be selected (redundancy)

derive the types of the variables in p_i from t_i

In words: the expression e_i will process values:

- generated by e (i.e., in t_0)
- minus those surely captured by the previous branches (i.e., in $\bigvee_{j<i} \lceil p_j g_j \rfloor$)
- and that may match p_i `when` g_i (i.e., in $\lceil p_i g_i \rceil$)

exaustivity condition

$$\text{(case)}\ \frac{\Gamma \vdash e : t_0 \qquad (\forall i < n \quad \Gamma, (t_i/p_i) \vdash e_i : s_i)}{\Gamma \vdash \texttt{case}\ e\ \texttt{do}\ (p_i g_i \to e_i)_{i<n} : \bigvee_{\{i \mid t_i \not\simeq \mathbb{0}\}} s_i} \qquad t_0 \leq \bigvee_{i \leq n} \lceil p_i g_i \rceil$$

The actual typing rule

$$\text{(case)}\ \frac{\Gamma \vdash e : t \qquad (\forall i{\leq}n)\,(\forall j{\leq}m_i)\ \ (t_{ij} \not\leq \mathbb{O}\ \Rightarrow\ \Gamma, \left(t_{ij}/_{p_i}\right) \vdash e_i : s)}{\Gamma \vdash \mathtt{case}\ e\ \mathtt{do}\ \left(p_i g_i \rightarrow e_i\right)_{i<n} : s}\ t \leq \bigvee_{i\leq n} \lfloor p_i g_i \rceil$$

where $\Gamma\,; t \vdash [(p_i g_i)]_{i\leq n} \rightsquigarrow [(s_{ij}, \mathfrak{b}_{ij})]_{i\leq n, j\leq m_i}$ *and* $t_{ij} = (t \wedge s_{ij}) \setminus \bigvee_{\{(h,k)\,|\,(h,k) \overset{L}{<} (i,j) \text{ and } \mathfrak{b}_{hk}\}} s_{hk}$

The actual typing rule

$$\text{(case)}\ \frac{\Gamma \vdash e : t \qquad (\forall i{\leq}n)\,(\forall j{\leq}m_i)\ \ (t_{ij} \not\leq \mathbb{O} \ \Rightarrow\ \Gamma, \left(t_{ij}/_{p_i}\right) \vdash e_i : s)}{\Gamma \vdash \mathtt{case}\ e\ \mathtt{do}\ \left(p_i g_i \rightarrow e_i\right)_{i<n} : s}\ t \leq \bigvee_{i\leq n} \lfloor p_i g_i \rceil$$

where $\underbrace{\Gamma\,;\, t \vdash [(p_i g_i)]_{i\leq n} \rightsquigarrow [(s_{ij}, \mathfrak{b}_{ij})]_{i\leq n, j\leq m_i}}_{\text{guard analysis}}$ *and* $t_{ij} = (t \wedge s_{ij}) \setminus \bigvee_{\{(h,k)\,|\,(h,k) \overset{L}{<} (i,j) \text{ and } \mathfrak{b}_{hk}\}} s_{hk}$

The actual typing rule

$$\text{(case)}\ \frac{\Gamma \vdash e : t \qquad (\forall i{\leq}n)\,(\forall j{\leq}m_i)\ \ (t_{ij} \not\leq \mathbb{O} \ \Rightarrow\ \Gamma, \left(t_{ij}/_{p_i}\right) \vdash e_i : s)}{\Gamma \vdash \texttt{case}\ e\ \texttt{do}\ \left(p_i g_i \rightarrow e_i\right)_{i<n} : s}\ t \leq \bigvee_{i\leq n} \lfloor p_i g_i \rceil$$

where $\Gamma\,;\, t \vdash [(p_i g_i)]_{i\leq n} \rightsquigarrow [(s_{ij}, \mathfrak{b}_{ij})]_{i\leq n, j\leq m_i}$ *and* $t_{ij} = (t \wedge s_{ij}) \setminus \bigvee_{\{(h,k)\,|\,(h,k)\overset{L}{<}(i,j) \text{ and } \mathfrak{b}_{hk}\}} s_{hk}$

lists

The actual typing rule

$$\text{(case)}\ \frac{\Gamma \vdash e : t \qquad (\forall i{\leq}n)\,(\forall j{\leq}m_i)\ \ (t_{ij} \not\leq \mathbb{O} \ \Rightarrow\ \Gamma, \left(t_{ij}/_{p_i}\right) \vdash e_i : s)}{\Gamma \vdash \texttt{case}\ e\ \texttt{do}\ \left(p_i g_i \rightarrow e_i\right)_{i<n} : s}\ t \leq \bigvee_{i\leq n} \lfloor p_i g_i \rceil$$

where $\Gamma\,;\, t \vdash [(p_i g_i)]_{i\leq n} \rightsquigarrow [(s_{ij}, \mathfrak{b}_{ij})]_{i\leq n, j\leq m_i}$ *and* $t_{ij} = (t \wedge s_{ij}) \setminus \bigvee_{\{(h,k)\,|\,(h,k) \overset{L}{<} (i,j) \text{ and } \mathfrak{b}_{hk}\}} s_{hk}$

lists

a Type

a Boolean

The actual typing rule

$$\text{(case)}\ \frac{\Gamma \vdash e : t \qquad (\forall i{\leq}n)\,(\forall j{\leq}m_i)\ \ (t_{ij} \not\leq \mathbb{O} \ \Rightarrow\ \Gamma, \left(t_{ij}/_{p_i}\right) \vdash e_i : s)}{\Gamma \vdash \texttt{case}\ e\ \texttt{do}\ \left(p_i g_i \rightarrow e_i\right)_{i<n} : s}\ t \leq \bigvee_{i\leq n} \lbag p_i g_i \rbag$$

where $\Gamma\,; t \vdash [(p_i g_i)]_{i\leq n} \rightsquigarrow [(s_{ij}, \mathfrak{b}_{ij})]_{i\leq n, j\leq m_i}$ *and* $t_{ij} = (t \wedge s_{ij}) \setminus \bigvee_{\{(h,k)\,|\,(h,k) \overset{L}{<} (i,j) \text{ and } \mathfrak{b}_{hk}\}} s_{hk}$

Guard Analysis

Consider again `pg = x when (is_map(x) and map_size(x) == 2) or is_list(x)`

The actual typing rule

$$\text{(case)}\ \frac{\Gamma \vdash e : t \qquad (\forall i{\leq}n)\ (\forall j{\leq}m_i)\ \ (t_{ij} \not\leq \mathbb{O} \ \Rightarrow\ \Gamma, \left(t_{ij}/_{p_i}\right) \vdash e_i : s)}{\Gamma \vdash \texttt{case}\ e\ \texttt{do}\ \left(p_i g_i \rightarrow e_i\right)_{i<n} : s}\ t \leq \bigvee_{i \leq n} \lbag p_i g_i \rbag$$

where $\Gamma\,;t \vdash [(p_i g_i)]_{i\leq n} \rightsquigarrow [(s_{ij}, \mathfrak{b}_{ij})]_{i\leq n, j\leq m_i}$ *and* $t_{ij} = (t \wedge s_{ij}) \setminus \bigvee_{\{(h,k)\,|\,(h,k) \overset{L}{<} (i,j) \text{ and } \mathfrak{b}_{hk}\}} s_{hk}$

Guard Analysis

Consider again `pg = x when (is_map(x) and map_size(x) == 2) or is_list(x)`

$\Gamma, \texttt{term()} \vdash [pg] \rightsquigarrow$ [(`map()`,`false`), (`list()`,`true`)]

The actual typing rule

$$\text{(case)}\ \frac{\Gamma \vdash e : t \qquad (\forall i{\leq}n)\ (\forall j{\leq}m_i)\ \ (t_{ij} \not\leq \mathbb{O} \ \Rightarrow\ \Gamma, \left(t_{ij}/_{p_i}\right) \vdash e_i : s)}{\Gamma \vdash \texttt{case}\ e\ \texttt{do}\ \left(p_i g_i \rightarrow e_i\right)_{i<n} : s}\ t \leq \bigvee_{i \leq n} \lceil p_i g_i \rceil$$

where $\Gamma\,;\, t \vdash [(p_i g_i)]_{i \leq n} \rightsquigarrow [(s_{ij}, \mathfrak{b}_{ij})]_{i \leq n, j \leq m_i}$ *and* $t_{ij} = (t \wedge s_{ij}) \setminus \bigvee_{\{(h,k)\,|\,(h,k) \overset{L}{<} (i,j) \text{ and } \mathfrak{b}_{hk}\}} s_{hk}$

Guard Analysis

Consider again `pg = x when (is_map(x) and map_size(x) == 2) or is_list(x)`

$\Gamma, \texttt{term()} \vdash [pg] \rightsquigarrow$ [(`map()`,`false`), (`list()`,`true`)]

$\lceil pg \rfloor$ = `list()` (all types in the list with `true` flag)

$\lfloor pg \rceil$ = `map() or list()` (all types in the list with any flag)

The actual typing rule

$$\text{(case)}\ \frac{\Gamma \vdash e : t \qquad (\forall i{\leq}n)\ (\forall j{\leq}m_i)\ \ (t_{ij} \not\leq \mathbb{O} \ \Rightarrow\ \Gamma, \left(t_{ij}/_{p_i}\right) \vdash e_i : s)}{\Gamma \vdash \texttt{case}\ e\ \texttt{do}\ \left(p_i g_i \rightarrow e_i\right)_{i<n} : s}\ t \leq \bigvee_{i \leq n} \lceil p_i g_i \rceil$$

where $\Gamma\,;\, t \vdash [(p_i g_i)]_{i\leq n} \rightsquigarrow [(s_{ij}, \mathfrak{b}_{ij})]_{i\leq n, j\leq m_i}$ *and* $t_{ij} = (t \wedge s_{ij}) \setminus \bigvee_{\{(h,k)\,|\,(h,k) \overset{L}{<} (i,j) \text{ and } \mathfrak{b}_{hk}\}} s_{hk}$

Guard Analysis

Consider again `pg = x when (is_map(x) and map_size(x) == 2) or is_list(x)`

$\Gamma, \texttt{term}() \vdash [pg] \rightsquigarrow$ [(`map()`,`false`), (`list()`,`true`)]

$\lceil pg \rceil$ = `list()` (all types in the list with `true` flag)

$\lfloor pg \rfloor$ = `map() or list()` (all types in the list with any flag)

Fine-grained analysis needed to infer the type of

```
def baz(x, y) when is_boolean(x) or is_integer(y), do: {y,x}
#=> ((term(),integer()) -> {integer(),term()}) and
#   ((boolean(),term()) -> {term(),boolean()})
```

Outline

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[ m.foo ]
# type atom() or 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[ m.foo ]
# type atom() or integer() or nil
m[41+1]
# type nil   (...and a warning)
```

Map types

See ICFP 23 talk on Wed 6th, at 15:30

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[ m.foo ]
# type atom() or integer() or nil
m[41+1]
# type nil   (...and a warning)
```

Outline

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()`.

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()`.

Same guarantees as *sound gradual typing* but without inserting casts at compilation

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 stronger 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() and dynamic()
```

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() and dynamic()
```

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

Outline

Current Progress

Project Milestones

1. Formalize a (rather) complete type system
2. Define a syntax for types
3. Implement a prototype
4. Present our design choices
5. Publish our approach
6. Integrate the system with the Elixir compiler

Current Progress

Project Milestones

1. Formalize a (rather) complete type system
2. Define a syntax for types
3. Implement a prototype
4. Present our design choices
5. Publish our approach
6. Integrate the system with the Elixir compiler

Feedback welcome!

4 5 $\Rightarrow$ A paper: *The Design Principles of the Elixir Type System* (Castagna, Duboc, Valim). *The Art, Science, and Engineering of Programming,* 8(2), 2024, to appear.
Preprint at `https://arxiv.org/abs/2306.06391`

Current Progress

Project Milestones

1. Formalize a (rather) complete type system
2. Define a syntax for types
3. Implement a prototype
4. Present our design choices
5. Publish our approach
6. Integrate the system with the Elixir compiler

Feedback welcome!

4 5 ⇒ A paper: *The Design Principles of the Elixir Type System* (Castagna, Duboc, Valim). *The Art, Science, and Engineering of Programming,* 8(2), 2024, to appear. Preprint at `https://arxiv.org/abs/2306.06391`

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

Current Progress

Project Milestones

1. Formalize a (rather) complete type system
2. Define a syntax for types
3. Implement a prototype
4. Present our design choices
5. Publish our approach
6. Integrate the system with the Elixir compiler

Feedback welcome!

4 5 ⇒ A paper: *The Design Principles of the Elixir Type System* (Castagna, Duboc, Valim). *The Art, Science, and Engineering of Programming,* 8(2), 2024, to appear. Preprint at `https://arxiv.org/abs/2306.06391`

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

... did I say **highly** experimental?

Future Work

- Behaviors (first-class modules)
- Maps: row polymorphism
- Type reconstruction and occurrence typing
- Typing processes and communications
- Behavioral Types (e.g., mailbox types, session types, ...)

Future Work

- Behaviors (first-class modules)
- Maps: row polymorphism
- Type reconstruction and occurrence typing
- Typing processes and communications
- Behavioral Types (e.g., mailbox types, session types, ...)

Behaviors

The visible parts of an Elixir module are:

```
defmodule GenServer do
 @type option() :: {:debug, debug()} | {:name, name()} | ...      //transparent
 @type result() :: {:reply, reply(), state()} | ...               //transparent
 @type state() :: term()                                          //opaque
 @type request() :: term()                                        //parameter
 @type reply() :: reply()                                         //parameter
    ⋮
 @callback init(init_arg :: term()) :: {:ok, state()} | :error    //mandatory
 @callback handle_call(request(),pid(),state()) :: result()       //optional
 @callback handle_cast(request(),state()) :: result()             //optional
    ⋮
 @spec start(module(), any(), options()) :: on_start()            //higher-order
    ⋮
end
```

Behaviors

The visible parts of an Elixir module are:

```
defmodule GenServer do
 @type option() :: {:debug, debug()} | {:name, name()} | ...      //transparent
 @type result() :: {:reply, reply(), state()} | ...               //transparent
 @type state() :: term()                                          //opaque
 @type request() :: term()                                        //parameter
 @type reply() :: reply()                                         //parameter
    ⋮
 @callback init(init_arg :: term()) :: {:ok, state()} | :error    //mandatory
 @callback handle_call(request(),pid(),state()) :: result()       //optional
 @callback handle_cast(request(),state()) :: result()             //optional
    ⋮
 @spec start(module(), any(), options()) :: on_start()            //higher-order
    ⋮
end
```

Behaviors

The visible parts of an Elixir module are:

```
defmodule GenServer do
 @type option() :: {:debug, debug()} | {:name, name()} | ...      //transparent
 @type result() :: {:reply, reply(), state()} | ...               //transparent
 @type state() :: term()                                          //opaque
 @type request() :: term()                                        //parameter
 @type reply() :: reply()                                         //parameter
    ⋮
 @callback init(init_arg :: term()) :: {:ok, state()} | :error    //mandatory
 @callback handle_call(request(),pid(),state()) :: result()       //optional
 @callback handle_cast(request(),state()) :: result()             //optional
    ⋮
 @spec start(module(), any(), options()) :: on_start()            //higher-order
    ⋮
end
```

Replace `module()` by a type describing the visible parts of the expected module

Behaviors

The visible parts of an Elixir module are:

```
defmodule GenServer do
 @type option() :: {:debug, debug()} | {:name, name()} | ...      //transparent
 @type result() :: {:reply, reply(), state()} | ...               //transparent
 @type state() :: term()                                          //opaque
 @type request() :: term()                                        //parameter
 @type reply() :: reply()                                         //parameter
    ⋮
 @callback init(init_arg :: term()) :: {:ok, state()} | :error    //mandatory
 @callback handle_call(request(),pid(),state()) :: result()       //optional
 @callback handle_cast(request(),state()) :: result()             //optional
    ⋮
 @spec start(module(), any(), options()) :: on_start()            //higher-order
    ⋮
end
```

Replace `module()` by a type describing the visible parts of the expected module

Behaviors

The visible parts of an Elixir module are:

```
defmodule GenServer(request, reply) do
 type option() = {:debug, debug()} | {:name, name()} | ...        //transparent
 type result() = {:reply, reply, state()} | ...                   //transparent
 type state()                                                     //opaque

    ⋮

 callback init :: term() -> {:ok, state()} | :error               //mandatory
 callback? handle_call :: request, pid(), state() -> result()     //optional
 callback? handle_cast :: request, state() -> result()            //optional
    ⋮
 function start :: GenServer(request, reply), any(), options() -> on_start()
    ⋮
end
```

Row variables

```
def delete_foo(map), do: Map.delete(map, :foo)
```

Row variables

```
$ map() -> %{optional(:foo) => none(), ...}
def delete_foo(map), do: Map.delete(map, :foo)
```

Row variables

```
$ map() -> %{optional(:foo) => none(), ...}
def delete_foo(map), do: Map.delete(map, :foo)
(delete_foo(%{foo: 12, bar: true})).bar
# => Type error
```

Row variables

```
$ %{ a } -> %{optional(:foo) => none(), a} when a : fields()
def delete_foo(map), do: Map.delete(map, :foo)
(delete_foo(%{foo: 12, bar: true})).bar
# => type boolean()
```

Row variables

```
$ %{ a } -> %{optional(:foo) => none(), a} when a : fields()
def delete_foo(map), do: Map.delete(map, :foo)
(delete_foo(%{foo: 12, bar: true})).bar
# => type boolean()
```

Type reconstruction

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

Row variables

```
$ %{ a } -> %{optional(:foo) => none(), a} when a : fields()
def delete_foo(map), do: Map.delete(map, :foo)
(delete_foo(%{foo: 12, bar: true})).bar
# => type boolean()
```

Type reconstruction

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

Occurrence typing

```
$ (a -> boolean(), [a]) -> [a]
    when a: term(), b: term()
def filter(fun, []), do: []
def filter(fun, [h | t]) do
  if fun.(h), do: [h | filter(fun, t)], else: filter(fun, t)
end
```

Row variables

```
$ %{ a } -> %{optional(:foo) => none(), a} when a : fields()
def delete_foo(map), do: Map.delete(map, :foo)
(delete_foo(%{foo: 12, bar: true})).bar
# => type boolean()
```

Type reconstruction

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

Occurrence typing

```
$ ((a -> true) and (b -> false), [a]) -> [a and not b]
    when a: term(), b: term()
def filter(fun, []), do: []
def filter(fun, [h | t]) do
  if fun.(h), do: [h | filter(fun, t)], else: filter(fun, t)
end
```

Recap: Current Progress

Project Milestones

1. Formalize a (rather) complete type system
2. Define a syntax for types
3. Implement a prototype
4. Present our design choices
5. Publish our approach
6. Integrate the system with the Elixir compiler

Feedback welcome!

- A paper: *The Design Principles of the Elixir Type System* (Castagna, Duboc, Valim).
  Preprint at `https://arxiv.org/abs/2306.06391`
- A prototype: `https://typex.fly.dev/` (highly experimental!)

Advertising

1. We are looking for students/postdocs to develop the future work
2. We are looking for sponsors for this work