

What's new in Elixir

Giuseppe Castagna

CNRS · IRIF, Université Paris Cité

What's new in Elixir ... in theory

Giuseppe Castagna

CNRS · IRIF, Université Paris Cité

AS ANNOUNCED BY JOSÉ VALIM

Elixir 1.20's typechecker covers all of Elixir

```
def label(n) when is_integer(n), do: n <> " items"
```

✗ **warning: incompatible types in binary construction**
got type `integer()` but expected type `binary()`

AS ANNOUNCED BY JOSÉ VALIM

Elixir 1.20's typechecker covers all of Elixir ... does it?

```
def label(n) when is_integer(n), do: n <> " items"  
def notify(p) when is_pid(p), do: send(p, {:_error, :timeout})  
def call(m) when is_atom(m), do: m.run(:oops)
```



No warning

`pid()` and `module()` say nothing: any message can be sent, any function called.

PART 1

Typing processes

joint work with Luca Padovani and Daniele Varacca

```
def notify(p) when is_pid(p), do: send(p, {:error, :timeout})
```

`pid()` says nothing

Should this be well typed?

```
send(self(), true)  
receive do y -> IO.puts(y + y) end
```

Should this be well typed?

```
send(self(), true)
receive do y -> IO.puts(y + y) end
```

 **No**

The send puts a boolean in the mailbox; the receive uses it as an integer.

Should this be well typed?

```
send(self(), true)
receive do y -> IO.puts(y + y) end
```

 **No**

The send puts a boolean in the mailbox; the receive uses it as an integer.

Elixir 1.20 compiles it without a warning.

A guard, or a tag

```
send(self(), true)  
receive do y -> IO.puts(y + y) end
```

A guard, or a tag

```
send(self(), true)  
receive do y when is_integer(y) -> IO.puts(y + y) end
```



Yes, with a guard

The guard checks the value before it is used as an integer.

A guard, or a tag

```
send(self(), true)
receive do
  {:add, y} -> y + 1
end
```



Safe only under a contract

Every `:add` message must carry an integer as its second element.

Give processes a type

`pid()` Today: the type only says that the value is a process.

```
send(self(), true)
receive do
  {:add, y} -> y + 1
end
```

Give processes a type

● new syntax

`pid(message_type)` The type of a process, its **interface**: the **union** of the types of all the messages it can receive.

```
send(self(), true)
receive do
  {:add, y} -> y + 1
end
```

Give processes a type

● new syntax

pid(message_type) The type of a process, its **interface**: the **union** of the types of all the messages it can receive.

```
send(self(), true)
receive do
  {:add, y} -> y + 1
end
```

✓ `pid({:add, integer()} or boolean())`
accepted

Give processes a type

● new syntax

pid(message_type) The type of a process, its **interface**: the **union** of the types of all the messages it can receive.

```
send(self(), true)
receive do
  {:add, y} -> y + 1
end
```

✓ pid({:add, integer()} or boolean())

accepted

✗ pid({:add, integer()}))

sending true is rejected

Give processes a type

● new syntax

pid(message_type) The type of a process, its **interface**: the **union** of the types of all the messages it can receive.

```
send(self(), true)
receive do
  {:add, y} -> y + 1
end
```

✓ pid({:add, integer()} or boolean())

accepted

✗ pid({:add, integer()}))

sending true is rejected

✗ pid({:add, term()} or boolean())

y + 1 is rejected

An interface at spawn

```
send(self(), true)
receive do
  {:add, y} -> y + 1
end
```

An interface at spawn

● new syntax

```
mypid = self()
$interface { :add, integer() } or boolean()
spawn(fn ->
  send(self(), true)
  receive do
    { :add, y } -> send(mypid, y + 1)
  end
end)
```

An interface at spawn

● new syntax

```
mypid = self()
$interface {:add, integer()} or boolean()
spawn(fn ->
  send(self(), true)
  receive do
    {:add, y} -> send(mypid, y + 1)
  end
end)
```

i Two self()'s with different meanings ⇒ two types

An interface at spawn

● new syntax

```
mypid = self()
$interface { :add, integer() } or boolean()
spawn(fn ->
  send(self(), true)
  receive do
    { :add, y } -> send(mypid, y + 1)
  end
end)
```

i **Two self()'s with different meanings ⇒ two types**

Line 1: the **spawner** process `pid(integer())`

An interface at spawn

● new syntax

```
mypid = self()
$interface {:add, integer()} or boolean()
spawn(fn ->
  send(self(), true)
  receive do
    {:add, y} -> send(mypid, y + 1)
  end
end)
```

i Two self()'s with different meanings ⇒ two types

Line 1: the **spawner** process pid(integer())

Line 4: the **spawned** process pid({:add, integer()} or boolean())

A running example: one counter per client

● new syntax

RUNS IN THE DISPATCHER

```
$type worker =  
  :bump or  
  {:add, integer()} or  
  {:get_total, pid({:total, integer()})}  
  
$ (-> none())  
def dispatcher do  
  receive do  
    {:connect, client} ->  
      $interface worker  
      spawn(fn -> serve(client) end)  
      dispatcher()  
  end  
end
```

RUNS IN EACH WORKER

```
$ pid({:ready, pid(worker)}) -> none()  
defp serve(client) do  
  send(client, {:ready, self()})  
  send(self(), :bump) # counts the connection  
  loop(0)  
end  
  
$ integer() -> none()  
defp loop(n) do  
  receive do  
    :bump -> loop(n + 1)  
    {:add, k} -> loop(n + k)  
    {:get_total, from} -> send(from, {:total, n}); loop(n)  
  end  
end
```

Module Server : a dispatcher spawns one counter per client;
each worker then serves its client directly

A running example: one counter per client

● new syntax

RUNS IN THE DISPATCHER

```
$type worker =  
  :bump or  
  {:add, integer()} or  
  {:get_total, pid({:total, integer()})}  
  
$ (-> none())  
def dispatcher do  
  receive do  
    {:connect, client} ->  
      $interface worker  
      spawn(fn -> serve(client) end)  
      dispatcher()  
  end  
end
```

RUNS IN EACH WORKER

```
$ pid({:ready, pid(worker)}) -> none()  
defp serve(client) do  
  send(client, {:ready, self()})  
  send(self(), :bump) # counts the connection  
  loop(0)  
end  
  
$ integer() -> none()  
defp loop(n) do  
  receive do  
    :bump -> loop(n + 1)  
    {:add, k} -> loop(n + k)  
    {:get_total, from} -> send(from, {:total, n}); loop(n)  
  end  
end
```

`worker` is an interface: the messages that a worker accepts

A running example: one counter per client

● new syntax

RUNS IN THE DISPATCHER

```
$type worker =  
  :bump or  
  {:add, integer()} or  
  {:get_total, pid({:total, integer()})}  
  
$ (-> none())  
def dispatcher do  
  receive do  
    {:connect, client} ->  
      $interface worker  
      spawn(fn -> serve(client) end)  
      dispatcher()  
  end  
end
```

RUNS IN EACH WORKER

```
$ pid({:ready, pid(worker)}) -> none()  
defp serve(client) do  
  send(client, {:ready, self()})  
  send(self(), :bump) # counts the connection  
  loop(0)  
end  
  
$ integer() -> none()  
defp loop(n) do  
  receive do  
    :bump -> loop(n + 1)  
    {:add, k} -> loop(n + k)  
    {:get_total, from} -> send(from, {:total, n}); loop(n)  
  end  
end
```

Each worker is spawned with the interface `worker`

A running example: one counter per client

● new syntax

RUNS IN THE DISPATCHER

```
$type worker =  
  :bump or  
  {:add, integer()} or  
  {:get_total, pid({:total, integer()})}  
  
$ (-> none())  
def dispatcher do  
  receive do  
    {:connect, client} ->  
      $interface worker  
      spawn(fn -> serve(client) end)  
      dispatcher()  
  end  
end
```

RUNS IN EACH WORKER

```
$ pid({:ready, pid(worker)}) -> none()  
defp serve(client) do  
  send(client, {:ready, self()})  
  send(self(), :bump) # counts the connection  
  loop(0)  
end  
  
$ integer() -> none()  
defp loop(n) do  
  receive do  
    :bump -> loop(n + 1)  
    {:add, k} -> loop(n + k)  
    {:get_total, from} -> send(from, {:total, n}); loop(n)  
  end  
end
```

A reply address: the request `{:get_total, from}` carries a pid that accepts the reply `{:total, integer()}`

A running example: one counter per client

● new syntax

RUNS IN THE DISPATCHER

```
$type worker =  
  :bump or  
  {:add, integer()} or  
  {:get_total, pid({:total, integer()})}  
  
$ (-> none())  
def dispatcher do  
  receive do  
    {:connect, client} ->  
      $interface worker  
      spawn(fn -> serve(client) end)  
      dispatcher()  
  end  
end
```

RUNS IN EACH WORKER

```
$ pid({:ready, pid(worker)}) -> none()  
defp serve(client) do  
  send(client, {:ready, self()})  
  send(self(), :bump) # counts the connection  
  loop(0)  
end  
  
$ integer() -> none()  
defp loop(n) do  
  receive do  
    :bump -> loop(n + 1)  
    {:add, k} -> loop(n + k)  
    {:get_total, from} -> send(from, {:total, n}); loop(n)  
  end  
end
```

A plain function type: `serve` needs a client that accepts `{:ready, pid(worker)}`

The interface is a contract

● new syntax

```
$type worker = :bump or { :add, integer() }  
               or { :get_total, pid({ :total, integer() }) }
```

The interface is a contract

● new syntax

```
$type worker = :bump or { :add, integer() }  
               or { :get_total, pid({ :total, integer() }) }
```

ON THE WAY IN

```
send(w, :bmup)
```



Rejected: wrong tag

`w` is a worker; `:bmup` is not in
worker.

The interface is a contract

● new syntax

```
$type worker = :bump or { :add, integer() }  
               or { :get_total, pid({ :total, integer() }) }
```

ON THE WAY IN

```
send(w, :bmup)  
send(w, { :get_total, 42 })
```



Rejected: wrong payload

Under `:get_total`, a worker expects a pid, not an integer.

The interface is a contract

● new syntax

```
$type worker = :bump or { :add, integer() }  
               or { :get_total, pid({ :total, integer() }) }
```

ON THE WAY IN

```
send(w, :bmup)  
send(w, { :get_total, 42 })
```



Rejected: wrong payload

Under `:get_total`, a worker expects a pid, not an integer.

ON THE WAY OUT

```
# in loop(n), n: integer()  
{ :add, k } -> loop(n + k)
```



Typed

The contract guarantees that `k` is an `integer()`, as `loop` expects.

The interface is a contract

● new syntax

```
$type worker = :bump or { :add, integer() }  
               or { :get_total, pid({ :total, integer() }) }
```

ON THE WAY IN

```
send(w, :bmup)  
send(w, { :get_total, 42 })
```



Rejected: wrong payload

Under `:get_total`, a worker expects a pid, not an integer.

ON THE WAY OUT

```
# in loop(n), n: integer()  
{ :add, k } -> loop(n + k)
```



Typed

The contract guarantees that `k` is an `integer()`, as `loop` expects.

Messages checked on the way in, payloads typed on the way out.

Self-sends

● new syntax

```
# in the dispatcher
$interface worker
spawn(fn -> serve(client) end)

# in the worker
defp serve(client) do
  send(client, {:ready, self()})
  send(self(), :bump)
  loop(0)
end
```

Self-sends

● new syntax

```
# in the dispatcher
$interface worker
spawn(fn -> serve(client) end)

# in the worker
defp serve(client) do
  send(client, {:ready, self()})
  send(self(), :bump)
  loop(0)
end
```



A self-send

It lands in the sender's own mailbox, so it is checked against the sender's own interface: `:bump` is in `worker`.

Self-sends

● new syntax

```
# in the dispatcher
$interface worker
spawn(fn -> serve(client) end)

# in the worker
defp serve(client) do
  send(client, {:ready, self()})
  send(self(), :tick)
  loop(0)
end
```



A stray :tick

The type inferred for `serve` records that it sends `:tick` to the process that runs it; `:tick` is not in `worker`, so the `spawn` is rejected. No other process to blame.

Functions travel in messages

● new syntax

```
$type job      = (-> term())
$type worker = :bump or { :add, integer() }
              or { :get_total, pid({ :total, integer() }) }
              or { :run, job }

# a new branch in loop/1
{ :run, job } -> job.(); loop(n)
```

Functions travel in messages

● new syntax

```
$type job      = (-> term())
$type worker = :bump or {:add, integer()}
              or {:get_total, pid({:total, integer()})}
              or {:run, job}

# a new branch in loop/1
{:run, job} -> job.(); loop(n)
```



A new capability

Workers now accept jobs: functions sent in a message, and run by the worker.

Functions travel in messages

● new syntax

```
$type job      = (-> term())
$type worker = :bump or {:add, integer()}
              or {:get_total, pid({:total, integer()})}
              or {:run, job}

# a new branch in loop/1
{:run, job} -> job.(); loop(n)
```

i

A plain arrow

`(-> term())` is any function with no argument. **Nothing is written about mailboxes.**

Functions travel in messages

● new syntax

```
$type job      = (-> term())
$type worker = :bump or { :add, integer() }
              or { :get_total, pid({ :total, integer() }) }
              or { :run, job }

# a new branch in loop/1
{ :run, job } -> job.(); loop(n)
```

i

Inferred

From `job.()`, jobs run in the worker: their `self()` and `receive` are the worker's. What they read and write there is (invisibly) recorded in their type.

Jobs that fit, and jobs that overstep

● new syntax

In a client, `w` has type `pid(Server.worker)`: the type `worker` exported by module `Server`.

```
send(w, { :run, fn -> send(self(), :bump) end })
```



Accepted

The job runs in the worker, so its `self()` is `w`: `:bump` is in the worker's interface.

Jobs that fit, and jobs that overstep

● new syntax

In a client, `w` has type `pid(Server.worker)`: the type `worker` exported by module `Server`.

```
send(w, {:run, fn -> send(self(), :tick) end})
```



Rejected: it writes

It would put `:tick` in the worker's mailbox, and the worker does not handle `:tick`.

Jobs that fit, and jobs that overstep

● new syntax

In a client, `w` has type `pid(Server.worker)`: the type `worker` exported by module `Server`.

```
send(w, {:run, fn -> receive do x -> x + 1 end end})
```

✗ Rejected: it reads

Its `receive` scans the worker's mailbox. The first message may be `:bump`, and `:bump + 1` does not typecheck.

Jobs that fit, and jobs that overstep

● new syntax

In a client, `w` has type `pid(Server.worker)`: the type `worker` exported by module `Server`.

```
send(w, {:run, fn ->
  receive do
    {:add, 0} -> send(self(), :bump)
    {:add, n} -> send(self(), {:add, n + 1})
  end
end})
```



Accepted

It reads `:add` with an integer, and writes `:bump` and `{:add, integer()}: the worker's interface covers both.`

Jobs that fit, and jobs that overstep

● new syntax

In a client, `w` has type `pid(Server.worker)`: the type `worker` exported by module `Server`.

```
send(w, { :run, fn ->
  receive do
    { :add, 0 } -> send(self(), :bump)
    { :add, n } -> send(self(), { :add, n + 1 })
  end
end })
```

Errors are reported **where the closure is written**, in the client,
not **where it runs**, in the worker.

What an interface cannot say

```
receive do y -> IO.puts(y + y) end  
send(self(), true)  
receive do b -> IO.puts(not b) end
```

What an interface cannot say

```
receive do y -> IO.puts(y + y) end  
send(self(), true)  
receive do b -> IO.puts(not b) end
```



It sends itself true

So its interface must accept booleans.

What an interface cannot say

```
receive do y -> IO.puts(y + y) end  
send(self(), true)  
receive do b -> IO.puts(not b) end
```



Rejected, whatever the interface

`y` may be a boolean, and `y + y` does not typecheck.

What an interface cannot say

```
receive do y -> IO.puts(y + y) end  
send(self(), true)  
receive do b -> IO.puts(not b) end
```

? Yet it may be safe

If a **single** integer arrives: the first receive takes it, the second takes `true`.

What an interface cannot say

```
receive do y -> IO.puts(y + y) end  
send(self(), true)  
receive do b -> IO.puts(not b) end
```

? Yet it may be safe

If a **single** integer arrives: the first receive takes it, the second takes `true`.

An interface is a set: it says **which** messages may arrive,
not how many, nor in which combinations.

BACK TO THE RUNNING EXAMPLE

Mailbox types

At capacity, the dispatcher answers `:busy` . The mailbox type of a client:

●● new syntax

```
:busy + ({:ready, pid(worker)} . {:total, integer()}))
```

`_ . _` = together `_+_` = either / or

Mailbox types

At capacity, the dispatcher answers `:busy` . The mailbox type of a client:

● new syntax

```
:busy + ({:ready, pid(worker)} . {:total, integer()}))
```

`_ . _` = together `_+_` = either / or

i Either turned away, or served

A single `:busy` , or exactly one `:ready` and one `:total` : no more, no less.

Mailbox types

At capacity, the dispatcher answers `:busy`. The mailbox type of a client:

●● new syntax

```
:busy + ({:ready, pid(worker)} . {:total, integer()}))
```

`_ . _` = together `_+_` = either / or

✗ Rejected

A client that still waits for a total after `:busy`: past the `:busy`, nothing is left to read.

Mailbox types

At capacity, the dispatcher answers `:busy` . The mailbox type of a client:

●● new syntax

```
:busy + ({:ready, pid(worker)} . {:total, integer()}))
```

`_ . _` = together `_+_` = either / or



What is checked: balance

Every send meets a receive clause that handles it.

Every receive clause meets a matching send.

No process ends with unread messages; a lock process cannot be released twice.

PART 2

Typing modules

joint work with Aghilas Boussaa, Antonin Bretagne, and José Valim

```
def call(m) when is_atom(m), do: m.run(:oops)
```

`module()` says nothing either

GenServer today: three problems ... among many

```
defmodule Counter do
  @behaviour GenServer
  @type state() :: %{optional(atom()) => integer()}
  @spec init(:ok) :: {:ok, state()}
  def init(:ok), do: {:ok, %{}}
  @spec handle_call({:bump, atom()}, pid(), state()) :: {:reply, integer(), state()}
  def handle_call({:bump, k}, _from, s),
    do: {:reply, Map.get(s, k, 0) + 1, Map.update(s, k, 1, &(&1 + 1))}
end

GenServer.start_link(Counter, :not_ok)
# generic code, for any module implementing GenServer
def step(module, msg, %{ } = st), do: module.handle_call(msg, self(), st)
```

A GenServer today: Counter counts calls per key, in a map.
Elixir 1.20 compiles all of this without a warning

GenServer today: three problems ... among many

```
defmodule Counter do
  @behaviour GenServer
  @type state() :: %{optional(atom()) => integer()}
  @spec init(:ok) :: {:ok, state()}
  def init(:ok), do: {:ok, %{}}
  @spec handle_call({:bump, atom()}, pid(), state()) :: {:reply, integer(), state()}
  def handle_call({:bump, k}, _from, s),
    do: {:reply, Map.get(s, k, 0) + 1, Map.update(s, k, 1, &(&1 + 1))}
end

GenServer.start_link(Counter, :not_ok)

# generic code, for any module implementing GenServer
def step(module, msg, %{ } = st), do: module.handle_call(msg, self(), st)
```

?

1. start_link does not know init

`Counter.init/1` accepts only `:ok`: the server crashes at run time, and the caller with it.

GenServer today: three problems ... among many

```
defmodule Counter do
  @behaviour GenServer
  @type state() :: %{optional(atom()) => integer()}
  @spec init(:ok) :: {:ok, state()}
  def init(:ok), do: {:ok, %{}}
  @spec handle_call({:bump, atom()}, pid(), state()) :: {:reply, integer(), state()}
  def handle_call({:bump, k}, _from, s),
    do: {:reply, Map.get(s, k, 0) + 1, Map.update(s, k, 1, &(&1 + 1))}
end

GenServer.start_link(Counter, :not_ok)
# generic code, for any module implementing GenServer
def step(module, msg, %{ } = st), do: module.handle_call(msg, self(), st)
```

? 2. Generic code may assume a shape

Nothing says that the state of `module` is a map: `step` works with `Counter`, until `Counter` changes its representation.

GenServer today: three problems ... among many

```
defmodule Counter do
  @behaviour GenServer
  @type state() :: %{optional(atom()) => integer()}
  @spec init(:ok) :: {:ok, state()}
  def init(:ok), do: {:ok, %{}}
  @spec handle_call({:bump, atom()}, pid(), state()) :: {:reply, integer(), state()}
  def handle_call({:bump, k}, _from, s),
    do: {:reply, Map.get(s, k, 0) + 1, Map.update(s, k, 1, &(&1 + 1))}
end

GenServer.start_link(Counter, :not_ok)
# generic code, for any module implementing GenServer
def step(module, msg, %{ } = st), do: module.handle_call(msg, self(), st)
```

? 3. Every callback spec is restated

`GenServer` types its callbacks with `term()` : to give `Counter` its own state, each of their specs must be written again.

A behaviour, **today** and **tomorrow**

```
defmodule GenServer do
  @type init_arg() :: term()
  @type call_msg() :: term()
  @type reply() :: term()
  @type state() :: term()
  @callback init(init_arg()) :: {:ok, state()}
  @callback handle_call(call_msg(), pid(), state()) ::
    {:reply, reply(), state()}

end
```

Today the compiler only checks that the callbacks exist; their types are left to Dialyzer

A behaviour, **today** and **tomorrow**

● new syntax

```
defmodule GenServer do
  $param init_arg
  $param call_msg
  $param reply
  $opaque state
  $callback init(init_arg) = {:ok, state}
  $callback handle_call(call_msg, pid(), state) = {:reply, reply, state}
end
```

Tomorrow the compiler checks the types: parameters, opaque state, callbacks

A behaviour, **today** and **tomorrow**

● new syntax

```
defmodule GenServer do
  $param init_arg
  $param call_msg
  $param reply
  $opaque state
  $callback init(init_arg) = {:ok, state}
  $callback handle_call(call_msg, pid(), state) = {:reply, reply, state}
end
```

`$param` : type parameters, fixed by each implementation and visible to callers: `start_link` can check its argument against `init_arg` **(problem 1)**

A behaviour, **today** and **tomorrow**

● new syntax

```
defmodule GenServer do
  $param init_arg
  $param call_msg
  $param reply
  $opaque state
  $callback init(init_arg) = {:ok, state}
  $callback handle_call(call_msg, pid(), state) = {:reply, reply, state}
end
```

\$opaque : each implementation chooses its `state` , and nobody else can look inside, not even `GenServer` **(problem 2)**

A behaviour, **today** and **tomorrow**

● new syntax

```
defmodule GenServer do
  $param init_arg
  $param call_msg
  $param reply
  $opaque state
  $callback init(init_arg) = {:ok, state}
  $callback handle_call(call_msg, pid(), state) = {:reply, reply, state}
end
```

`$callback` : typed contracts, **statically** checked *on every implementation (problem 3)* and at every call

What a module declares

● new syntax

```
defmodule Counter do
  $param key
  $behaviour GenServer[init_arg: :ok, reply: integer(),      # fixes GenServer's param's
                        call_msg: {:bump, key} or :reset]
  $opaque state = %{key => integer()}                        # defines the module state
  def init(:ok), do: {:ok, %{}}
  def handle_call({:bump, k}, _from, s),
    do: {:reply, count(s, k) + 1, Map.update(s, k, 1, &(&1 + 1))}
  def handle_call(:reset, _from, _s), do: {:reply, 0, %{}}
  $ (state, key) -> integer()
  def count(s, k), do: Map.get(s, k, 0)
end
```



An implementation of GenServer

A counter per key: `{:bump, k}` replies with the new count.

What a module declares

● new syntax

```
defmodule Counter do
  $param key
  $behaviour GenServer[init_arg: :ok, reply: integer(),      # fixes GenServer's param's
                        call_msg: {:bump, key} or :reset]
  $opaque state = %{key => integer()}                        # defines the module state
  def init(:ok), do: {:ok, %{}}
  def handle_call({:bump, k}, _from, s),
    do: {:reply, count(s, k) + 1, Map.update(s, k, 1, &(&1 + 1))}
  def handle_call(:reset, _from, _s), do: {:reply, 0, %{}}
  $ (state, key) -> integer()
  def count(s, k), do: Map.get(s, k, 0)
end
```

Type parameters

`Counter` is a **parametric module**: it is not stuck to `atom()` keys.

What a module declares

● new syntax

```
defmodule Counter do
  $param key
  $behaviour GenServer[init_arg: :ok, reply: integer(),      # fixes GenServer's param's
                        call_msg: {:bump, key} or :reset]
  $opaque state = %{key => integer()}                        # defines the module state
  def init(:ok), do: {:ok, %{}}
  def handle_call({:bump, k}, _from, s),
    do: {:reply, count(s, k) + 1, Map.update(s, k, 1, &(&1 + 1))}
  def handle_call(:reset, _from, _s), do: {:reply, 0, %{}}
  $ (state, key) -> integer()
  def count(s, k), do: Map.get(s, k, 0)
end
```



Type parameters

THE ONLY NEW SYNTAX OUTSIDE TYPE ANNOTATIONS

The previous counter module for atom keys is now `Counter[key: atom()]`

What a module declares

● new syntax

```
defmodule Counter do
  $param key
  $behaviour GenServer[init_arg: :ok, reply: integer(),      # fixes GenServer's param's
                        call_msg: {:bump, key} or :reset]
  $opaque state = %{key => integer()}                        # defines the module state
  def init(:ok), do: {:ok, %{}}
  def handle_call({:bump, k}, _from, s),
    do: {:reply, count(s, k) + 1, Map.update(s, k, 1, &(&1 + 1))}
  def handle_call(:reset, _from, _s), do: {:reply, 0, %{}}
  $ (state, key) -> integer()
  def count(s, k), do: Map.get(s, k, 0)
end
```



The behaviour it implements

With its arguments: replies are integers, messages use its own parameter `key`.

What a module declares

● new syntax

```
defmodule Counter do
  $param key
  $behaviour GenServer[init_arg: :ok, reply: integer(), # fixes GenServer's param's
                        call_msg: {:bump, key} or :reset]
  $opaque state = %{key => integer()} # defines the module state
  def init(:ok), do: {:ok, %{}}
  def handle_call({:bump, k}, _from, s),
    do: {:reply, count(s, k) + 1, Map.update(s, k, 1, &(&1 + 1))}
  def handle_call(:reset, _from, _s), do: {:reply, 0, %{}}
  $ (state, key) -> integer()
  def count(s, k), do: Map.get(s, k, 0)
end
```



Its opaque type

A map, but only inside `Counter`.

What a module declares

● new syntax

```
defmodule Counter do
  $param key
  $behaviour GenServer[init_arg: :ok, reply: integer(), # fixes GenServer's param's
                        call_msg: {:bump, key} or :reset]
  $opaque state = %{key => integer()} # defines the module state
  def init(:ok), do: {:ok, %{}}
  def handle_call({:bump, k}, _from, s),
    do: {:reply, count(s, k) + 1, Map.update(s, k, 1, &(&1 + 1))}
  def handle_call(:reset, _from, _s), do: {:reply, 0, %{}}
  $ (state, key) -> integer()
  def count(s, k), do: Map.get(s, k, 0)
end
```



Callbacks

They take their types from `GenServer`: no spec to restate. **Problem 3 solved.**

What a module declares

● new syntax

```
defmodule Counter do
  $param key
  $behaviour GenServer[init_arg: :ok, reply: integer(),      # fixes GenServer's param's
                        call_msg: {:bump, key} or :reset]
  $opaque state = %{key => integer()}                        # defines the module state
  def init(:ok), do: {:ok, %{}}
  def handle_call({:bump, k}, _from, s),
    do: {:reply, count(s, k) + 1, Map.update(s, k, 1, &(&1 + 1))}
  def handle_call(:reset, _from, _s), do: {:reply, 0, %{}}
  $ (state, key) -> integer()
  def count(s, k), do: Map.get(s, k, 0)
end
```



Other exported functions

They need a `$` signature, as in Part 1. Here `state` is a map, so `Map.get` is fine.

Modules as typed values

```
@spec replay(module(), term(), [term()]) :: {[term()], term()}
def replay(server, st, msgs) do
  Enum.map_reduce(msgs, st, fn msg, acc ->
    {:_reply, n, acc} = server.handle_call(msg, self(), acc)
    {n, acc}
  end)
end
```

A unit-testing helper: replays calls on a state, with no process, for any GenServer

Modules as typed values

```
@spec replay(module(), term(), [term()]) :: {[term()], term()}
def replay(server, st, msgs) do
  Enum.map_reduce(msgs, st, fn msg, acc ->
    {reply, n, acc} = server.handle_call(msg, self(), acc)
    {n, acc}
  end)
end
```

A unit-testing helper: replays calls on a state, with no process, for any GenServer
Today: module() and term() , that is, any module, any state, any reply

Modules as typed values

● new syntax

```
$type call = { :bump, atom() } or :reset
$type bumper = GenServer[init_arg: :ok, call_msg: call, reply: integer()]
$ (server :: bumper, server.state, [call]) -> {[integer()], server.state}
def replay(server, st, msgs) do
  Enum.map_reduce(msgs, st, fn msg, acc ->
    { :reply, n, acc } = server.handle_call(msg, self(), acc)
    { n, acc }
  end)
end
```

A behaviour is a type: `bumper` is any module implementing `GenServer` with these arguments, such as `Counter[key: atom()]`

Modules as typed values

● new syntax

```
$type call = { :bump, atom() } or :reset
$type bumper = GenServer[init_arg: :ok, call_msg: call, reply: integer()]
$ (server :: bumper, server.state, [call]) -> {[integer()], server.state}
def replay(server, st, msgs) do
  Enum.map_reduce(msgs, st, fn msg, acc ->
    { :reply, n, acc } = server.handle_call(msg, self(), acc)
    { n, acc }
  end)
end
```

`server.state` : the state of the module passed as `server`, and no other; the replies are integers, as `bumper` says

Modules as typed values

● new syntax

```
$type call = { :bump, atom() } or :reset
$type bumper = GenServer[init_arg: :ok, call_msg: call, reply: integer()]
$ (server :: bumper, server.state, [call]) -> {[integer()], server.state}
def replay(server, st, msgs) do
  Enum.map_reduce(msgs, st, fn msg, acc ->
    { :reply, n, acc } = server.handle_call(msg, self(), acc)
    { n, acc }
  end)
end
```

The body does not change: only the signature is new

Errors caught

● new syntax

```
alias Visits = Counter[key: atom()]      # state: a map
alias EtsVisits = EtsCounter[key: atom()] # state: an ETS table
{:ok, st} = Visits.init(:ok)
```

CODE

ELIXIR 1.20 TODAY

WITH MODULE TYPES

Errors caught

● new syntax

```
alias Visits = Counter[key: atom()]      # state: a map
alias EtsVisits = EtsCounter[key: atom()] # state: an ETS table
{:ok, st} = Visits.init(:ok)
```

CODE	ELIXIR 1.20 TODAY	WITH MODULE TYPES
<code>in replay : server.hanle_call(...)</code>	crashes at run time	✗ <code>GenServer</code> has no callback <code>hanle_call</code>
<code>replay(String, st, [:reset])</code>	crashes at run time	✗ <code>String</code> does not implement <code>GenServer</code>

```
$type bumper = GenServer[init_arg: :ok, call_msg: call, reply: integer()]
$ (server :: bumper, server.state, [call]) -> {[integer()], server.state}
def replay(server, st, msgs)          # call = {:bump, atom()} or :reset
```

Errors caught

● new syntax

```
alias Visits = Counter[key: atom()]      # state: a map
alias EtsVisits = EtsCounter[key: atom()] # state: an ETS table
{:ok, st} = Visits.init(:ok)
```

CODE	ELIXIR 1.20 TODAY	WITH MODULE TYPES
in replay : server.hanle_call(...)	crashes at run time	✗ GenServer has no callback hanle_call
replay(String, st, [:reset])	crashes at run time	✗ String does not implement GenServer
map_size(st)	accepted	✗ st is opaque outside Counter
replay(EtsVisits, st, [:reset])	crashes at run time	✗ st is not an EtsVisits.state

```
$type bumper = GenServer[init_arg: :ok, call_msg: call, reply: integer()]
$ (server :: bumper, server.state, [call]) -> {[integer()], server.state}
def replay(server, st, msgs)           # call = {:bump, atom()} or :reset
```

Errors caught

● new syntax

```
alias Visits = Counter[key: atom()]      # state: a map
alias EtsVisits = EtsCounter[key: atom()] # state: an ETS table
{:ok, st} = Visits.init(:ok)
```

CODE	ELIXIR 1.20 TODAY	WITH MODULE TYPES
in <code>replay : server.handle_call(...)</code>	crashes at run time	✗ <code>GenServer</code> has no callback <code>handle_call</code>
<code>replay(String, st, [:reset])</code>	crashes at run time	✗ <code>String</code> does not implement <code>GenServer</code>
<code>map_size(st)</code>	accepted	✗ <code>st</code> is opaque outside <code>Counter</code>
<code>replay(EtsVisits, st, [:reset])</code>	crashes at run time	✗ <code>st</code> is not an <code>EtsVisits.state</code>
<code>replay(Visits, st, [{:bump, "a"}])</code>	accepted	✗ <code>"a"</code> is not an <code>atom()</code>
in <code>Counter : :reset</code> replies <code>:ok</code>	accepted	✗ the reply must be an <code>integer()</code>

```
$type bumper = GenServer[init_arg: :ok, call_msg: call, reply: integer()]
$ (server :: bumper, server.state, [call]) -> {[integer()], server.state}
def replay(server, st, msgs)           # call = {:bump, atom()} or :reset
```

Errors caught

● new syntax

```
alias Visits = Counter[key: atom()]      # state: a map
alias EtsVisits = EtsCounter[key: atom()] # state: an ETS table
{:ok, st} = Visits.init(:ok)
```

CODE	ELIXIR 1.20 TODAY	WITH MODULE TYPES
in replay : server.hanle_call(...)	crashes at run time	✗ GenServer has no callback hanle_call
replay(String, st, [:reset])	crashes at run time	✗ String does not implement GenServer
map_size(st)	accepted	✗ st is opaque outside Counter
replay(EtsVisits, st, [:reset])	crashes at run time	✗ st is not an EtsVisits.state
replay(Visits, st, [{:bump, "a"}])	accepted	✗ "a" is not an atom()
in Counter : :reset replies :ok	accepted	✗ the reply must be an integer()
1 GenServer.start_link(Visits, :not_ok)	crashes at run time	✗ the init_arg of Visits is :ok
2 in replay : fn msg, %{ } = acc -> ...	accepted	✗ the state of server is opaque

Errors caught

● new syntax

New: what goes through module variables, abstract types, and contracts.

CODE	ELIXIR 1.20 TODAY	WITH MODULE TYPES
in <code>replay : server.hanle_call(...)</code>	crashes at run time	✗ <code>GenServer</code> has no callback <code>hanle_call</code>
<code>replay(String, st, [:reset])</code>	crashes at run time	✗ <code>String</code> does not implement <code>GenServer</code>
<code>map_size(st)</code>	accepted	✗ <code>st</code> is opaque outside <code>Counter</code>
<code>replay(EtsVisits, st, [:reset])</code>	crashes at run time	✗ <code>st</code> is not an <code>EtsVisits.state</code>
<code>replay(Visits, st, [{:bump, "a"}])</code>	accepted	✗ <code>"a"</code> is not an <code>atom()</code>
in <code>Counter : :reset</code> replies <code>:ok</code>	accepted	✗ the reply must be an <code>integer()</code>
1 <code>GenServer.start_link(Visits, :not_ok)</code>	crashes at run time	✗ the <code>init_arg</code> of <code>Visits</code> is <code>:ok</code>
2 in <code>replay : fn msg, %{} = acc -> ...</code>	accepted	✗ the state of <code>server</code> is opaque

What you can now specify

● new syntax

Exported types transparent or opaque (Part 1)

```
pid({:ready, pid(Server.worker)})
```

Parametric modules

```
Counter[key: atom()]
```

Parametric behaviours

```
$type bumper = GenServer[init_arg: :ok, call_msg: {:bump, atom()} or :reset, reply: integer()]
```

Behaviours as types instead of module()

```
$ bumper -> bumper
```

Dependent signatures

```
$ (s :: bumper, s.state) -> s.state
```

Modules in data, and as results

```
$type config = %{:server: bumper}  
$callback default_server() = bumper
```

Elixir as it is

● new syntax

```
defmodule Counter do
  $param key
  $behaviour GenServer[init_arg: :ok, call_msg: {:bump, key} or :reset, reply: integer()]
  $opaque state = %{key => integer()}
  def init(:ok), do: {:ok, %{}}
  $ (-> integer())
  def check() do
    {:ok, a} = init(:ok)           # local call
    {:ok, b} = Counter[key: key].init(:ok) # remote call
    map_size(a) + map_size(b)      # warning on b
  end
end
```

Elixir as it is

● new syntax

```
defmodule Counter do
  $param key
  $behaviour GenServer[init_arg: :ok, call_msg: {:bump, key} or :reset, reply: integer()]
  $opaque state = %{key => integer()}
  def init(:ok), do: {:ok, %{}}
  $ (-> integer())
  def check() do
    {:ok, a} = init(:ok)           # local call
    {:ok, b} = Counter[key: key].init(:ok) # remote call
    map_size(a) + map_size(b)      # warning on b
  end
end
```

`init(:ok)` is a local call: **this version** of `Counter`, where `state` is a map

Elixir as it is

● new syntax

```
defmodule Counter do
  $param key
  $behaviour GenServer[init_arg: :ok, call_msg: {:bump, key} or :reset, reply: integer()]
  $opaque state = %{key => integer()}
  def init(:ok), do: {:ok, %{}}
  $ (-> integer())
  def check() do
    {:ok, a} = init(:ok)           # local call
    {:ok, b} = Counter[key: key].init(:ok) # remote call
    map_size(a) + map_size(b)      # warning on b
  end
end
```



warning: incompatible types given to map_size/1

got type `Counter[key: key].state` but expected type `map()`
After a hot code upgrade, a remote call may return a different `state`.

Elixir as it is

● new syntax

```
defmodule Counter do
  $param key
  $behaviour GenServer[init_arg: :ok, call_msg: {:bump, key} or :reset, reply: integer()]
  $opaque state = %{key => integer()}
  def init(:ok), do: {:ok, %{}}
  $ (-> integer())
  def check() do
    {:ok, a} = init(:ok)           # local call
    {:ok, b} = Counter[key: key].init(:ok) # remote call
    map_size(a) + map_size(b)      # warning on b
  end
end
```

Also as in Elixir:

- a nested module is separate: it does not see its parent's parameters, types and private functions;
- a module implements a behaviour only if it declares it (`$behaviour`, as `@behaviour` today).

What's new in Elixir ... in theory

Processes

Messages checked on the way in, payloads typed on the way out, even for code that travels.

```
$interface worker  
spawn(fn -> serve(client) end)
```

Mailbox types

How many messages, in which combinations; no message left unread.

```
:busy + ({:ready, pid(worker)} . {:total, integer()})
```

Modules

Behaviours as types, opaque and parametric modules, dependent signatures.

```
$type bumper = GenServer[init_arg: :ok,  
  call_msg: {:bump, atom()} or :reset, reply: integer()]  
$ (s :: bumper, s.state) -> {integer(), s.state}
```

Thank you! I am ready to answer your questions

What's new in Elixir ... in theory

Processes

theory done, ready for implementation

Messages checked on the way in, payloads typed on the way out, even for code that travels.

```
$interface worker  
spawn(fn -> serve(client) end)
```

Mailbox types

experimental, on paper only

How many messages, in which combinations; no message left unread.

```
:busy + ({:ready, pid(worker)} . {:total, integer()}))
```

Modules

some work left before implementation

Behaviours as types, opaque and parametric modules, dependent signatures.

```
$type bumper = GenServer[init_arg: :ok,  
  call_msg: {:bump, atom()} or :reset, reply: integer()]  
$ (s :: bumper, s.state) -> {integer(), s.state}
```

Thank you!

I am ready to answer your questions

What's new in Elixir ... in theory

Processes

theory done, ready for implementation

Messages checked on the way in, payloads typed on the way out, even for code that travels.

```
$interface worker  
spawn(fn -> serve(client) end)
```

Mailbox types

experimental, on paper only

How many messages, in which combinations; no message left unread.

```
:busy + ({:ready, pid(worker)} . {:total, integer()}))
```

Modules

some work left before implementation

Behaviours as types, opaque and parametric modules, dependent signatures.

```
$type bumper = GenServer[init_arg: :ok,  
  call_msg: {:bump, atom()} or :reset, reply: integer()]  
$ (s :: bumper, s.state) -> {integer(), s.state}
```

Thank you!

I am ready to answer your questions ... in theory

A LONG TIME FROM NOW, IN A GALAXY FAR, FAR AWAY....

● new syntax

Putting it all together

```
# in GenServer: the pid of a server that accepts calls c and replies r
$type server(c, r) = pid({:"$gen_call", {pid({reference(), r}), reference()}}, c)})
$ (GenServer[init_arg: a, call_msg: c, reply: r], a) -> {:ok, server(c, r)}
    when a: term(), c: term(), r: term()
def start_link(module, init_arg)
$ (server(c, r), c) -> r when c: term(), r: term()
def call(server, msg)
# a client of Counter, which implements
#   GenServer[init_arg: :ok, call_msg: {:bump, key} or :reset, reply: integer()]
alias Visits = Counter[key: atom()]
{:ok, pid} = GenServer.start_link(Visits, :ok)
GenServer.call(pid, {:bump, :home}) + 1      # an integer
GenServer.call(pid, {:bump, "a"})           # rejected: "a" is not an atom()
GenServer.start_link(Visits, :not_ok)       # rejected: the init_arg is :ok
```

A sketch, in neither theory yet: module types give the pid its process type (Part 1)
Parametric types and type variables keep the code generic