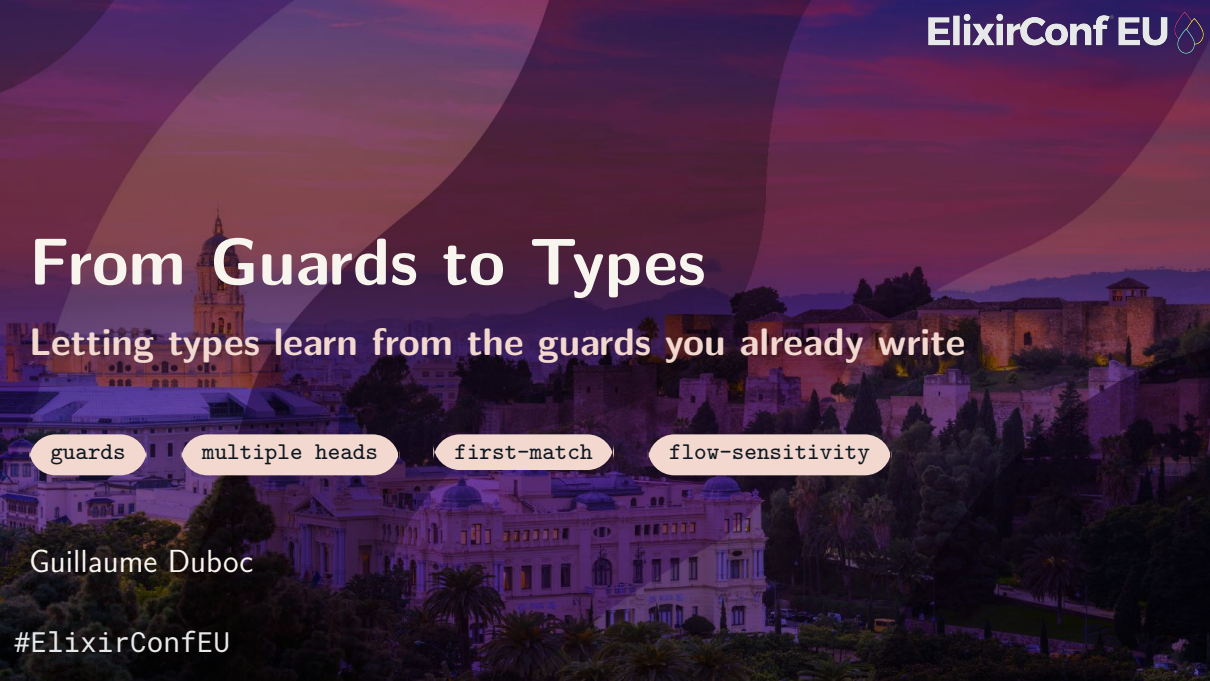

ElixirConf EU
From Guards to Types
Letting types learn from the guards you already write
guards
multiple heads
first-match
flow-sensitivity
Guillaume Duboc
#ElixirConfEU

Why Programmers Should Care

The release news is precise domain inference.

Patterns and guards will feed accepted input domains.

`inputs` `outputs` `warnings`

What improves

Precise domains

Accepted inputs can come from patterns and guards the code already has.

Better warnings

Residual clauses and dead-code warnings follow first-match semantics.

A Small Difference That Matters

```
id = fn x ->
  x
end
# id :: dynamic() -> dynamic()
```

- **No evidence gained**: The body sees the same uncertainty it started with.
- Operationally, nothing constrains `x`.

```
id_int = fn x when is_integer(x) ->
  x
end
# id_int :: dynamic() -> integer()
```

- **Guarded branch**: The body runs only after a successful proof.
- The VM checks the predicate; the type system can reclaim the result.

ElixirConf EU

Existing Semantics

Guards as technical evidence in ordinary Elixir.

Ordinary Elixir Already Uses This

Clause Selection

Multiple heads partition the input space into executable cases.

Residual Cases

Fallback clauses rely on the **negation** of earlier tests.

Operational Safety

Guards stop invalid operations before the branch logic begins.

The runtime is already carving the input space into meaningful cases.

ElixirConf EU

What Is Shipping

Elixir 1.20 is hard-working!

How Complex Guards Work Today

Head Domain

Patterns build the initial head domain.

Guard Refinement

Guards run left to right. Facts on `x` can refine `pair`, and selectors can feed facts back too.

Boolean Splitting

`or` splits branch-wise; `not` can recover structural complements too.

Precision Frontier

Precise clauses are stored and subtracted later; less precise guards can still sharpen the current clause.

1. Start from the domain that patterns already establish.

Head Domain Baseline

```elixir
def keep_ok({:ok, value} = reply) do
  reply
end
```

Already in 1.19

Recovered Facts

Facts

```
value : term()
reply : {:ok, term()}
```

Guard analysis starts from a pattern-shaped head domain, not from nothing.

How Complex Guards Work Today

Head Domain

Patterns build the initial head domain.

Guard Refinement

Guards run left to right. Facts on `value` can refine `pair`, and selectors can feed facts back too.

Boolean Splitting

`or` splits branch-wise; `not` can recover structural complements too.

Precision Frontier

Precise clauses are stored and subtracted later; less precise guards can still sharpen the current clause.

2. Then guards refine that domain and push facts through the structure.

Dependency Propagation

Elixir 1.19.5

Same clause. Older compiler.

```
case pair do
  {x, y} when is_integer(x) ->
    elem(pair, 0) + 1
end
```

What The Checker Sees

Facts

```
pair : dynamic()
elem(pair, 0) : dynamic()
elem(pair, 1) : dynamic()
```

The fact about `x` never comes back to `pair`.

Dependency Propagation

Current

Same clause. Current compiler.

```
case pair do
  {x, y} when is_integer(x) ->
    elem(pair, 0) + 1
end
```

What The Checker Sees

Facts

```
pair : dynamic({integer(), term()})
elem(pair, 0) : integer()
elem(pair, 1) : dynamic()
```

The fact about `x` is pushed back onto `pair`.

Selector Misuse

Elixir 1.19.5

The dependency never comes back.

```
case pair do
  {x, y} when is_integer(x) ->
    String.length(elem(pair, 0))
end
```

Result

Accepted

```
dynamic()
```

The misuse slips through because slot 0 stays unknown.

Selector Misuse

Current

Same code. Current compiler.

```
case pair do
  {x, y} when is_integer(x) ->
    String.length(elem(pair, 0))
end
```

Result

Rejected

```
String.length/1
receives integer()
```

The propagated fact is strong enough to catch the bug.

Non-Precise Guards

Elixir 1.19.5

The checker does not even recover the list shape yet.

```
case list do
  value when hd(value) == :ok ->
    value
end
```

Result Type

Type

```
dynamic()
```

Even the fact that the list is non-empty is left on the table.

Non-Precise Guards

Current

Same case. Current compiler.

```
case list do
  value when hd(value) == :ok ->
    value
end
```

Result Type

Type

```
dynamic(non_empty_list())
```

Even an inexact guard can still sharpen the shape.

Impossible Guards

Elixir 1.19.5

Contradictory facts still produce no diagnostic.

```
case x do
  value when is_pid(value)
             and is_atom(value) ->
        value
end
```

Result

Accepted

```
no diagnostic
```

Two incompatible facts can still pass by silently.

Impossible Guards

Current

Same case. Current compiler.

```
case x do
  value when is_pid(value)
             and is_atom(value) ->
        value
end
```

Result

Rejected

```
guard will
never succeed
```

Mutually exclusive facts are now recognized as impossible.

How Complex Guards Work Today

Head Domain

Patterns build the initial head domain.

Guard Refinement

Guards run left to right. Facts on `x` can refine `pair`, and selectors can feed facts back too.

Boolean Splitting

`or` splits branch-wise; `not` can recover structural complements too.

Precision Frontier

Precise clauses are stored and subtracted later; less precise guards can still sharpen the current clause.

3. `or` splits branch-wise, and `not` can recover exact complements.

Negation Can Be Exact Too

`not` is not just loss of information. Many complements are structural and exact.

Guards

```
not is_map_key(x, :foo)
not (tuple_size(x) != 2)
not (map_size(x) == 0)
not (length(x) != 0)
not (length(x) <= 0)
not (tuple_size(x) <= 2)
```

Recovered Types

```
%{..., foo: not_set()}
{term(), term()}
map() and not empty_map()
empty_list()
non_empty_list(term())
{term(), term(), term(), ...}
```

Exact Accepted Domains

Elixir 1.19.5

Complex guards still collapse.

```
defguard is_stream_chunk(x) when
  tuple_size(x) == 2 and
    (elem(x, 0) == :data and is_binary(elem(x, 1)) or
     elem(x, 0) == :halt and elem(x, 1) == :normal)
```

Accepted Type

Type

```
dynamic()
```

The guard runs, but its accepted domain is still opaque.

Exact Accepted Domains

Current

Same guard. Current compiler.

```
defguard is_stream_chunk(x) when
  tuple_size(x) == 2 and
    (elem(x, 0) == :data and is_binary(elem(x, 1)) or
     elem(x, 0) == :halt and elem(x, 1) == :normal)
```

Accepted Type

Type

```
dynamic(
  {:data, binary()} or
  {:halt, :normal})
```

The guard itself now describes an exact tagged union.

Map-Shaped Domains

Elixir 1.19.5

Field guards still collapse.

```
defguard is_resource(x) when
  (x.kind == :file and is_binary(x.path)) or
  (x.kind == :fd and is_integer(x.fd))
```

Accepted Type

Type

```
dynamic()
```

The field checks run, but the map shape is still lost.

Map-Shaped Domains

Current

Same guard. Current compiler.

```
defguard is_resource(x) when
  (x.kind == :file and is_binary(x.path)) or
  (x.kind == :fd and is_integer(x.fd))
```

Accepted Type

Type

```
dynamic(%{..., kind: :fd, fd: integer()}
    or %{..., kind: :file, path: binary()})
```

The same guard now recovers an exact map-shaped union.

Contradicted Uses

Elixir 1.19.5

The refined space is still too weak to expose the bug.

```
def add_to_chunk(x)
  when tuple_size(x) == 2 and
    ((elem(x, 0) == :data and is_binary(elem(x, 1))) or
     (elem(x, 0) == :halt and elem(x, 1) == :normal)) do
    elem(x, 1) + 1
end
```

Result

Accepted

```
dynamic(number())
```

The contradiction stays hidden, so the arithmetic still typechecks.

Contradicted Uses

Current

Same case. Current compiler.

```
def add_to_chunk(x)
  when tuple_size(x) == 2 and
    ((elem(x, 0) == :data and is_binary(elem(x, 1))) or
     (elem(x, 0) == :halt and elem(x, 1) == :normal)) do
    elem(x, 1) + 1
end
```

Result

Rejected

```
elem(x, 1) is
binary() or :normal
```

Once the guard is understood, the contradiction becomes a real error.

How Complex Guards Work Today

Head Domain

Patterns build the initial head domain.

Guard Refinement

Guards run left to right. Facts on `x` can refine `pair`, and selectors can feed facts back too.

Boolean Splitting

`or` splits branch-wise; `not` can recover structural complements too.

Precision Frontier

Precise clauses are stored and subtracted later; less precise guards can still sharpen the current clause.

4. Finally, precise clauses can be stored and subtracted from later ones.

Residual Clauses

Elixir 1.19.5

The first guard is not subtracted precisely.

```
case reply do
  {:ok, value} when is_binary(value) ->
    {:text, value}

  {:ok, value} ->
    {:other, value}
end
```

Result Type

Type

```
dynamic({:other or :text, term()})
```

Clause 2 forgets that clause 1 already captured binaries.

Residual Clauses

Current

Same code. Current compiler.

```
case reply do
  {:ok, value} when is_binary(value) ->
    {:text, value}

  {:ok, value} ->
    {:other, value}
end
```

Result Type

Type

```
dynamic({:other, not binary()} or
        {:text, binary()})
```

Clause 2 is typed on the residual left by clause 1.

Repeated Guarded Clauses

Elixir 1.19.5

No redundancy warning yet.

```
case pair do
  {x, y} when is_integer(x) and is_integer(y) ->
    {:ints, x, y}

  {x, y} when is_integer(x) and is_integer(y) ->
    {:again, x, y}
end
```

Result

Accepted

```
dynamic(
  {:again or :ints, term(),
    term()}
)
```

The overlap is seen, but not subtracted from later clauses.

Repeated Guarded Clauses

Current

Same case. Current compiler.

```
case pair do
  {x, y} when is_integer(x) and is_integer(y) ->
    {:ints, x, y}

  {x, y} when is_integer(x) and is_integer(y) ->
    {:again, x, y}
end
```

Result

Warning

```
redundant clause on
{integer(), integer()}
```

First-match semantics now reaches the warning too.

What This Buys Programmers

Accepted Inputs

Complex guards can contribute real accepted domains instead of disappearing into `dynamic()`.

Better Warnings

Residual clauses and dead-code warnings line up with actual clause order.

Less Duplication

Tagged unions and case splits do not need to be restated from scratch in a separate contract.

Reverse Arrows

After The Call

Facts

```
def id(x), do: x
def id_int(x) when is_integer(x), do: x
id(x)      # x : dynamic()
id_int(x)  # x : integer()
```

Key Idea

Successful calls can refine inputs backwards, like guards do inside a clause.

```
def render_tagged({:int, value}),
  do: Integer.to_string(value)

def render_tagged({:not_int, value}),
  do: {:not_int, value}
```

Recovered Domain

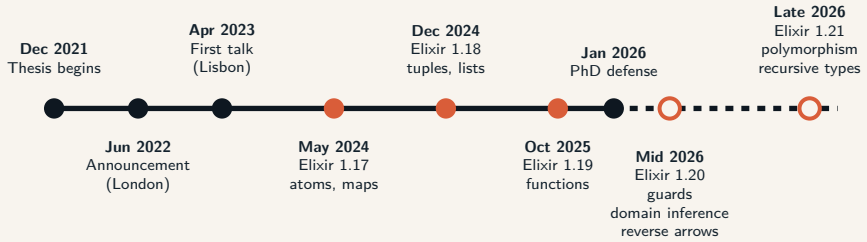
Type

```
{:int, integer()} or {:not_int, term()}
```

Error caught:

```
render_tagged({:int, "not an integer"})
```

The Journey



3 years from thesis start to compiler releases.

ElixirConf EU

Guards as evidence.

Types that follow the code Elixir programmers already write.

Feedback

Slides and code examples

Website

gldubc.github.io/
#elixirconf

#ElixirConfEU