



INSTITUT DE RECHERCHE EN
INFORMATIQUE FONDAMENTALE



University of
BRISTOL

All Set: Bringing Static Types to Elixir

.....

Giuseppe CASTAGNA (CNRS, Université Paris Cité)

Guillaume DUBOC (University of Bristol)

*Elisir di sì perfetta,
di sì rara qualità,
ne sapessi la ricetta,
conoscessi chi ti fa!*

— G. Donizetti, *L'elisir d'amore*

Outline

1. Set-theoretic types and dynamic languages

- 1.1 Branching, overloading, and pattern matching
- 1.2 Semantic subtyping

2. Elixir basic typing

- 2.1 Unions, intersections, negations
- 2.2 Polymorphism

3. Guard analysis

- 3.1 Exhaustiveness and dead clauses
- 3.2 Narrowing and inference

4. Gradual typing

- 4.1 Strong function types
- 4.2 Propagation of `dynamic()`

5. Integration in Elixir

Why static typing for dynamic languages?

Static types for dynamic languages

- **Catch bugs early** and enable **tooling**: completion, navigation, safe refactoring
- **Checked documentation**: types state what functions expect and return; crucial when code is written by AI and read by programmers
- **Existing solutions fall short**: TypeScript, Flow, Hack are language-specific and unsound; rewriting in a typed language may take years for large codebases
- **Needed**: a type system that is **sound**, **gradual**, **automated**, and **open**

Why static typing for dynamic languages?

Static types for dynamic languages

- **Catch bugs early** and enable **tooling**: completion, navigation, safe refactoring
- **Checked documentation**: types state what functions expect and return; crucial when code is written by AI and read by programmers
- **Existing solutions fall short**: TypeScript, Flow, Hack are language-specific and unsound; rewriting in a typed language may take years for large codebases
- **Needed**: a type system that is **sound**, **gradual**, **automated**, and **open**

Regulatory compliance

EU and US regulations increasingly **require evidence of software assurance** (CRA, NIST, ...). A sound type system is a mechanically verifiable step in that direction, also for AI-generated code.

Why static typing for dynamic languages?

Static types for dynamic languages

- **Catch bugs early** and enable **tooling**: completion, navigation, safe refactoring
- **Checked documentation**: types state what functions expect and return; crucial when code is written by AI and read by programmers
- **Existing solutions fall short**: TypeScript, Flow, Hack are language-specific and unsound; rewriting in a typed language may take years for large codebases
- **Needed**: a type system that is **sound**, **gradual**, **automated**, and **open**

Regulatory compliance

EU and US regulations increasingly **require evidence of software assurance** (CRA, NIST, ...). A sound type system is a mechanically verifiable step in that direction, also for AI-generated code.

The challenge: dynamic code relies on idioms that classic type systems cannot describe

1. Branching and discriminated unions

```
type Status = "success" | "error";  
function report(s: Status): number | string {  
  if (s === "success") return 200;  
  return "failed";  
}
```

Typescript

The result is number or string depending on the branch

1. Branching and discriminated unions

```
type Status = "success" | "error";  
function report(s: Status): number | string {  
  if (s === "success") return 200;  
  return "failed";  
}
```

Typescript

The result is number or string depending on the branch

```
type Payment =  
  | { kind: "card";    cardNo: string }  
  | { kind: "paypal"; email: string };
```

Typescript

A discriminated union: the field kind tells which variant a payment is

1. Branching and discriminated unions

```
type Status = "success" | "error";  
function report(s: Status): number | string {  
  if (s === "success") return 200;  
  return "failed";  
}
```

Typescript

The result is number or string depending on the branch

```
type Payment =  
  | { kind: "card";    cardNo: string }  
  | { kind: "paypal"; email: string };
```

Typescript

A discriminated union: the field kind tells which variant a payment is

⇒ Union and singleton types are needed

"success", "error", "card": types with one value; Status, Payment: their unions

2. Overloading

Aka ad hoc polymorphism: functions that execute different code on arguments of different types

```
function double (x) {  
    return (typeof(x) === "number") ? 2*x : x.concat(x);  
}
```

JAVASCRIPT

If applied to a number returns a number, if applied to a string returns a string

2. Overloading

Aka ad hoc polymorphism: functions that execute different code on arguments of different types

```
function double (x) {  
  return (typeof(x) === "number") ? 2*x : x.concat(x);  
}
```

JAVASCRIPT

If applied to a number returns a number, if applied to a string returns a string

⇒ Intersection types are needed

`double : (number→number) ∧ (string→string)`

3. Precise typing of pattern matching

Accepted **type** of a pattern p : $\llbracket p \rrbracket = \{v \mid v \text{ matches } p\}$, e.g., $\llbracket (x, y) \rrbracket = \mathbb{1} \times \mathbb{1}$

Boolean type connectives are needed to type pattern matching:

$\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2 \text{ with } e : T$

3. Precise typing of pattern matching

Accepted **type** of a pattern p : $\llbracket p \rrbracket = \{v \mid v \text{ matches } p\}$, e.g., $\llbracket (x, y) \rrbracket = \mathbb{1} \times \mathbb{1}$

Boolean type connectives are needed to type pattern matching:

$\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2 \text{ with } e : T$

- To infer the type T_1 of e_1 we need $T \wedge \llbracket p_1 \rrbracket$;
- To infer the type T_2 of e_2 we need $(T \setminus \llbracket p_1 \rrbracket) \wedge \llbracket p_2 \rrbracket$;

3. Precise typing of pattern matching

Accepted **type** of a pattern p : $\llbracket p \rrbracket = \{v \mid v \text{ matches } p\}$, e.g., $\llbracket (x, y) \rrbracket = \mathbb{1} \times \mathbb{1}$

Boolean type connectives are needed to type pattern matching:

$\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2 \text{ with } e : T$

- To infer the type T_1 of e_1 we need $T \wedge \llbracket p_1 \rrbracket$;
- To infer the type T_2 of e_2 we need $(T \setminus \llbracket p_1 \rrbracket) \wedge \llbracket p_2 \rrbracket$;
- The type of the match expression is $T_1 \vee T_2$.
- Pattern matching is exhaustive if $T \leq \llbracket p_1 \rrbracket \vee \llbracket p_2 \rrbracket$.

3. Precise typing of pattern matching

Accepted **type** of a pattern p : $\llbracket p \rrbracket = \{v \mid v \text{ matches } p\}$, e.g., $\llbracket (x, y) \rrbracket = \mathbb{1} \times \mathbb{1}$

Boolean type connectives are needed to type pattern matching:

$\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2 \text{ with } e : T$

- To infer the type T_1 of e_1 we need $T \wedge \llbracket p_1 \rrbracket$;
- To infer the type T_2 of e_2 we need $(T \setminus \llbracket p_1 \rrbracket) \wedge \llbracket p_2 \rrbracket$;
- The type of the match expression is $T_1 \vee T_2$.
- Pattern matching is exhaustive if $T \leq \llbracket p_1 \rrbracket \vee \llbracket p_2 \rrbracket$.

Formally:

[Match]

$$\frac{\Gamma \vdash e : T \quad \Gamma, T \wedge \llbracket p_1 \rrbracket / p_1 \vdash e_1 : T_1 \quad \Gamma, T \setminus \llbracket p_1 \rrbracket / p_2 \vdash e_2 : T_2}{\Gamma \vdash \text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2 : T_1 \vee T_2} (T \leq \llbracket p_1 \rrbracket \vee \llbracket p_2 \rrbracket)$$

where T/p is the type for the capture variables in p when it is matched against values in T .

3. Precise typing of pattern matching

Accepted **type** of a pattern p : $\llbracket p \rrbracket = \{v \mid v \text{ matches } p\}$, e.g., $\llbracket (x, y) \rrbracket = \mathbb{1} \times \mathbb{1}$

Boolean type connectives are needed to type pattern matching:

$\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2 \text{ with } e : T$

- To infer the type T_1 of e_1 we need $T \wedge \llbracket p_1 \rrbracket$;
- To infer the type T_2 of e_2 we need $(T \setminus \llbracket p_1 \rrbracket) \wedge \llbracket p_2 \rrbracket$;
- The type of the match expression is $T_1 \vee T_2$.
- Pattern matching is exhaustive if $T \leq \llbracket p_1 \rrbracket \vee \llbracket p_2 \rrbracket$.

Formally:

$$\frac{\begin{array}{c} \text{[Match]} \\ \Gamma \vdash e : T \quad \Gamma, T \wedge \llbracket p_1 \rrbracket / p_1 \vdash e_1 : T_1 \quad \Gamma, T \setminus \llbracket p_1 \rrbracket / p_2 \vdash e_2 : T_2 \end{array}}{\Gamma \vdash \text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2 : T_1 \vee T_2} \quad (T \leq \llbracket p_1 \rrbracket \vee \llbracket p_2 \rrbracket)$$

where T/p is the type for the capture variables in p when it is matched against values in T .

3. Precise typing of pattern matching

Accepted **type** of a pattern p : $\llbracket p \rrbracket = \{v \mid v \text{ matches } p\}$, e.g., $\llbracket (x, y) \rrbracket = \mathbb{1} \times \mathbb{1}$

Boolean type connectives are needed to type pattern matching:

$\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2 \text{ with } e : T$

- To infer the type T_1 of e_1 we need $T \wedge \llbracket p_1 \rrbracket$;
- To infer the type T_2 of e_2 we need $(T \setminus \llbracket p_1 \rrbracket) \wedge \llbracket p_2 \rrbracket$;
- The type of the match expression is $T_1 \vee T_2$.
- Pattern matching is exhaustive if $T \leq \llbracket p_1 \rrbracket \vee \llbracket p_2 \rrbracket$.

⇒ Negation types are needed

To type e_2 we remove from T the values accepted by p_1 : $T \setminus \llbracket p_1 \rrbracket = T \wedge \neg \llbracket p_1 \rrbracket$

Semantic subtyping

Unions and intersections are common (TypeScript, Python, Scala); negations are rare.

Our approach: semantic subtyping, defined by what types mean.

Types: $T ::= \text{Bool} \mid \text{Int} \mid T \times T \mid T \rightarrow T \mid T \vee T \mid T \wedge T \mid \neg T \mid \perp \mid \top$ (possibly recursive)

Semantic subtyping

Unions and intersections are common (TypeScript, Python, Scala); negations are rare.

Our approach: semantic subtyping, defined by what types mean.

Types: $T ::= \text{Bool} \mid \text{Int} \mid T \times T \mid T \rightarrow T \mid T \vee T \mid T \wedge T \mid \neg T \mid \perp \mid \top$ (possibly recursive)

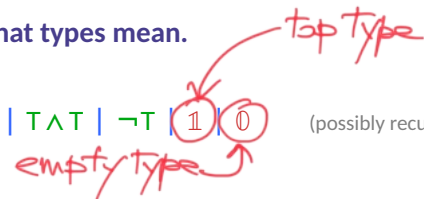
Set Theoretic Types 

Semantic subtyping

Unions and intersections are common (TypeScript, Python, Scala); negations are rare.

Our approach: semantic subtyping, defined by what types mean.

Types: $T ::= \text{Bool} \mid \text{Int} \mid T \times T \mid T \rightarrow T \mid T \vee T \mid T \wedge T \mid \neg T \mid \boxed{1} \mid \boxed{0}$ (possibly recursive)



Semantic subtyping

Unions and intersections are common (TypeScript, Python, Scala); negations are rare.

Our approach: semantic subtyping, defined by what types mean.

Types: $T ::= \text{Bool} \mid \text{Int} \mid T \times T \mid T \rightarrow T \mid T \vee T \mid T \wedge T \mid \neg T \mid \perp \mid \emptyset$ (possibly recursive)

How does it work?

1. Types denote sets of values: $\llbracket t \rrbracket$
2. Connectives are set operations:
$$\llbracket t \vee s \rrbracket = \llbracket t \rrbracket \cup \llbracket s \rrbracket \quad \llbracket t \wedge s \rrbracket = \llbracket t \rrbracket \cap \llbracket s \rrbracket$$
$$\llbracket \neg t \rrbracket = \llbracket t \rrbracket^c$$
3. Subtyping is inclusion: $t \leq s \iff \llbracket t \rrbracket \subseteq \llbracket s \rrbracket$
4. Derive algorithms to decide $\leq$ and compute type operators

Semantic subtyping

Unions and intersections are common (TypeScript, Python, Scala); negations are rare.

Our approach: semantic subtyping, defined by what types mean.

Types: $T ::= \text{Bool} \mid \text{Int} \mid T \times T \mid T \rightarrow T \mid T \vee T \mid T \wedge T \mid \neg T \mid \perp \mid \emptyset$ (possibly recursive)

How does it work?

1. Types denote sets of values: $\llbracket t \rrbracket$
2. Connectives are set operations:
 $\llbracket t \vee s \rrbracket = \llbracket t \rrbracket \cup \llbracket s \rrbracket$ $\llbracket t \wedge s \rrbracket = \llbracket t \rrbracket \cap \llbracket s \rrbracket$
 $\llbracket \neg t \rrbracket = \llbracket t \rrbracket^c$
3. Subtyping is inclusion: $t \leq s \iff \llbracket t \rrbracket \subseteq \llbracket s \rrbracket$
4. Derive algorithms to decide $\leq$ and compute type operators

Why not subtyping rules?

Rule-based systems end up incomplete and fragile: e.g.,

$$(\text{Int} \vee \text{Bool}) \times \text{Int} \leq (\text{Int} \times \text{Int}) \vee (\text{Bool} \times \text{Int})$$

may not be derivable.

With semantic subtyping, all set-theoretic laws hold for **free**.

Outline

1. Set-theoretic types and dynamic languages

- 1.1 Branching, overloading, and pattern matching
- 1.2 Semantic subtyping

2. Elixir basic typing

- 2.1 Unions, intersections, negations
- 2.2 Polymorphism

3. Guard analysis

- 3.1 Exhaustiveness and dead clauses
- 3.2 Narrowing and inference

4. Gradual typing

- 4.1 Strong function types
- 4.2 Propagation of `dynamic()`

5. Integration in Elixir

Putting the theory to work: Elixir

The language

- Dynamic and functional, Ruby-like syntax, extensive metaprogramming
- Runs on the Erlang VM: distributed, fault-tolerant applications
- Nearly 100K developers; used by Adobe, Apple, BBC, Discord, PepsiCo, Spotify, ...

The type system

- Since 2021, in collaboration with José Valim (creator of Elixir) and the Dashbit team
- Based on our set-theoretic types: in the compiler since v1.17 (June 2024), covers the whole language since v1.20 (current version: 1.20.4, August 2026)

Types as contracts: union vs. intersection

Function A: negate

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

Function B: subtract

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

Types as contracts: union vs. intersection

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
```

\$ introduces a type signature. Elixir writes or, and, not, term(), none() for $\vee$, $\wedge$, $\neg$, $\mathbb{1}$, $\mathbb{0}$.

Types as contracts: union vs. intersection

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
```

\$ introduces a type signature. Elixir writes or, and, not, term(), none() for $\vee$, $\wedge$, $\neg$, $\mathbb{1}$, $\mathbb{0}$.

What if we modify the implementation of negate?

Types as contracts: union vs. intersection

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
```

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: union vs. intersection

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
```

Function B: subtract

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

Type warning in subtract (warnings never stop compilation)

```
warning: incompatible types given to Kernel.+/2: a + negate(b)  
given types: integer(), integer() or boolean()  
but expected one of: integer(), integer() ...
```

Types as contracts: union vs. intersection

Adding a clause for booleans

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

Function B: subtract

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

With the intersection type, subtract type checks

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
```

- false or nil: the values Elixir treats as false (falsy), a union of singleton types

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
```

- `false or nil`: the values Elixir treats as false (falsy), a union of singleton types
- `not (false or nil)`: all other values (truthy), a negation type

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
```

- `false or nil`: the values Elixir treats as false (falsy), a union of singleton types
- `not (false or nil)`: all other values (truthy), a negation type

Narrowing on the result of an application

```
def succ(x) when is_integer(x) or is_nil(x) do
  if logical_neg(x), do: 0,          # x : nil
    else: x + 1      # x : integer()
end
```

The intersection type of `logical_neg` narrows `x` in each branch

Polymorphism with 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 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 are quantified by a postfix **when**, with upper bounds (here the top type `term()`)

Polymorphism with 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 are quantified by a postfix **when**, with upper bounds (here the top type `term()`)

Instantiation of type variables is inferred (tallying)

```
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
```

Intersection and polymorphism

$\forall b \leq \text{True thy.}$

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
```

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
```

Bounded polymorphism (t, u types; for b in `logical_or`, u is `not(false or nil)`)

$$\forall(\alpha \leq u).t \stackrel{\text{def}}{=} \forall\alpha. t[\alpha := \alpha \wedge u]$$

Outline

1. Set-theoretic types and dynamic languages

- 1.1 Branching, overloading, and pattern matching
- 1.2 Semantic subtyping

2. Elixir basic typing

- 2.1 Unions, intersections, negations
- 2.2 Polymorphism

3. Guard analysis

- 3.1 Exhaustiveness and dead clauses
- 3.2 Narrowing and inference

4. Gradual typing

- 4.1 Strong function types
- 4.2 Propagation of `dynamic()`

5. Integration in Elixir

Guards are more than type tests

A multi-clause function is a **case** on its arguments: each clause has a pattern and a guard.

Guards combine type tests, comparisons, field access, and, or, not

```
def f(x) when (is_map(x) and map_size(x) == 2) or is_list(x), do: ...
```

Guards are more than type tests

A multi-clause function is a **case** on its arguments: each clause has a pattern and a guard.

Guards combine type tests, comparisons, field access, and, or, not

```
def f(x) when (is_map(x) and map_size(x) == 2) or is_list(x), do: ...
```

The values that pass this guard do not form a type: no type describes “maps of size 2”.

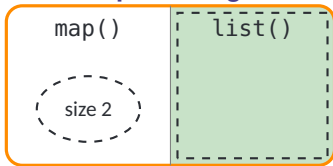
Guards are more than type tests

A multi-clause function is a **case** on its arguments: each clause has a pattern and a guard.

Guards combine type tests, comparisons, field access, and, or, not

```
def f(x) when (is_map(x) and map_size(x) == 2) or is_list(x), do: ...
```

The values that pass this guard do not form a type: no type describes “maps of size 2”.



dashed: the values that pass the guard

Two types approximate them:

- **surely accepted:** `list()`
every list passes
- **possibly accepted:** `map() or list()`
only maps and lists can pass

Dead clauses and exhaustiveness

Same recipe as for patterns (slide 5): clause i receives the input values **minus** those **surely accepted** by the previous clauses, **and possibly accepted** by its own guard.

- To infer the type T_1 of e_1 we need $T \wedge \{p_1\}$;
- To infer the type T_2 of e_2 we need $(T \setminus \{p_1\}) \wedge \{p_2\}$;
- The type of the match expression is $T_1 \vee T_2$.
- Pattern matching is exhaustive if $T \leq \{p_1\} \vee \{p_2\}$.

Formally:

$$\begin{array}{c}
 \text{[Match]} \\
 \frac{\Gamma \vdash e : T \quad \Gamma, T \wedge \{p_1\} / p_1 \vdash e_1 : T_1 \quad \Gamma, T \setminus \{p_1\} / p_2 \vdash e_2 : T_2}{\Gamma \vdash \text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2 : T_1 \vee T_2} \quad (T \leq \{p_1\} \vee \{p_2\})
 \end{array}$$

Handwritten annotations: Red arrows point from the list items to the corresponding parts of the formal rule. Green arrows point from the list items to the guard condition $(T \leq \{p_1\} \vee \{p_2\})$.

Dead clauses and exhaustiveness

Same recipe as for patterns (slide 5): clause i receives the input values **minus** those **surely accepted** by the previous clauses, **and possibly accepted** by its own guard.

- To infer the type T_1 of e_1 we need $T \wedge [p_1]$;
- To infer the type T_2 of e_2 we need $(T \setminus [p_1]) \wedge [p_2]$;
- The type of the match expression is $T_1 \vee T_2$.
- Pattern matching is exhaustive if $T \leq [p_1] \vee [p_2]$.

Formally:

$[p]$: surely accepted, $[p]$: possibly accepted by p and its guard

$$\frac{\begin{array}{c} \text{[Match]} \\ \Gamma \vdash e : T \quad \Gamma, T \wedge [p_1] / p_1 \vdash e_1 : T_1 \quad \Gamma, T \setminus [p_1] / p_2 \vdash e_2 : T_2 \end{array}}{\Gamma \vdash \text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2 : T_1 \vee T_2} (T \leq [p_1] \vee [p_2])$$

Dead clauses and exhaustiveness

Same recipe as for patterns (slide 5): clause i receives the input values **minus** those **surely accepted** by the previous clauses, **and possibly accepted** by its own guard.

- To infer the type T_1 of e_1 we need $T \wedge [p_1]$;
 - To infer the type T_2 of e_2 we need $(T \setminus [p_1]) \wedge [p_2]$;
 - The type of the match expression is $T_1 \vee T_2$.
 - Pattern matching is exhaustive if $T \leq [p_1] \vee [p_2]$.
- Formally:** $[p]$: surely accepted, $[p]$: possibly accepted by p and its guard

$$\begin{array}{c}
 \text{[Match]} \\
 \hline
 \frac{\Gamma \vdash e : T \quad \Gamma, T \wedge [p_1] / p_1 \vdash e_1 : T_1 \quad \Gamma, T \setminus [p_1] / p_2 \vdash e_2 : T_2}{\Gamma \vdash \text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2 : T_1 \vee T_2} (T \leq [p_1] \vee [p_2])
 \end{array}$$

Dead clauses and exhaustiveness

Same recipe as for patterns (slide 5): clause *i* receives the input values **minus** those **surely accepted** by the previous clauses, **and possibly accepted** by its own guard.

Nothing left: a dead clause (Elixir 1.20)

```
def red(x) when is_integer(x) or is_boolean(x), do: 1
def red(x) when is_integer(x), do: 2
```

warning: the following clause is redundant:

```
def red(x) when is_integer(x)
```

it has type: integer()
previous clauses have already matched on the following types:
boolean() or integer()

Dead clauses and exhaustiveness

Same recipe as for patterns (slide 5): clause *i* receives the input values **minus** those **surely accepted** by the previous clauses, **and possibly accepted** by its own guard.

Nothing left: a dead clause (Elixir 1.20)

```
def red(x) when is_integer(x) or is_boolean(x), do: 1
def red(x) when is_integer(x), do: 2
```

warning: the following clause is redundant:

```
def red(x) when is_integer(x)
```

it has type: integer()
previous clauses have already matched on the following types:
boolean() or integer()

Exhaustiveness depends on the declared input type

```
$ list() -> _ # _ : result type left to inference
def f(x) when (is_map(x) and map_size(x) == 2) or is_list(x), do: ...
```

Covered by the **surely accepted** type list(): exhaustive

Dead clauses and exhaustiveness

Same recipe as for patterns (slide 5): clause *i* receives the input values **minus** those **surely accepted** by the previous clauses, **and possibly accepted** by its own guard.

Nothing left: a dead clause (Elixir 1.20)

```
def red(x) when is_integer(x) or is_boolean(x), do: 1
def red(x) when is_integer(x), do: 2
```

warning: the following clause is redundant:

```
def red(x) when is_integer(x)
```

it has type: integer()
previous clauses have already matched on the following types:
boolean() or integer()

Exhaustiveness depends on the declared input type

```
$ map() or list() -> _ # _ : result type left to inference
def f(x) when (is_map(x) and map_size(x) == 2) or is_list(x), do: ...
```

Covered only by the **possibly accepted** type: may not be exhaustive

Dead clauses and exhaustiveness

Same recipe as for patterns (slide 5): clause i receives the input values **minus** those **surely accepted** by the previous clauses, **and possibly accepted** by its own guard.

Nothing left: a dead clause (Elixir 1.20)

```
def red(x) when is_integer(x) or is_boolean(x), do: 1
def red(x) when is_integer(x), do: 2
```

warning: the following clause is redundant:

```
def red(x) when is_integer(x)
```

it has type: integer()
previous clauses have already matched on the following types:
boolean() or integer()

Exhaustiveness depends on the declared input type

```
$ term() -> _ # _ : result type left to inference
def f(x) when (is_map(x) and map_size(x) == 2) or is_list(x), do: ...
```

Not covered: not exhaustive

Narrowing: each clause sees a more precise type

A response type

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

Handling a response

```
$ response() -> _  
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)}
```

clause type of r

result

Narrowing: each clause sees a more precise type

A response type

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

Handling a response

```
$ response() -> _  
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)}
```

clause	type of r	result
1	%{output: :ok, socket: socket()}	{:accepted, socket()}

Narrowing: each clause sees a more precise type

A response type

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

Handling a response

```
$ response() -> _  
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)}
```

clause	type of r	result
1	%{output: :ok, socket: socket()}	{:accepted, socket()}
2	%{output: :error, message: :timeout}	:timeout

Narrowing: each clause sees a more precise type

A response type

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

Handling a response

```
$ response() -> _  
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)}
```

clause	type of r	result
1	%{output: :ok, socket: socket()}	{:accepted, socket()}
2	%{output: :error, message: :timeout}	:timeout
3	%{output: :error, message: {:delay, integer()}}	{:retry, integer()}

Narrowing: each clause sees a more precise type

A response type

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

Handling a response

```
$ response() -> _  
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)}
```

Each row is an arrow: the inferred type of handle (only the input type was declared)

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

Narrowing: each clause sees a more precise type

A response type

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

A bug caught by narrowing (Elixir 1.20)

```
case fetch() do      # fetch() returns a response()  
  r when r.output == :ok -> r.socket  
  r -> r.socket  
end
```

```
warning: unknown key .socket in expression: r.socket  
the given type does not have the given key:  
  %{message: :timeout or {:delay, integer()}, output: :error}
```

Splitting guards: several arrows for one clause

One clause, a disjunctive guard

```
def baz(x, y) when is_boolean(x) or is_integer(y), do: {y, x}
```

Splitting guards: several arrows for one clause

One clause, a disjunctive guard

```
def baz(x, y) when is_boolean(x) or is_integer(y), do: {y, x}
```

The guard is split into its or branches; the body is typed once per branch:

```
is_boolean(x) (boolean(), term() -> {term(), boolean()})
```

Splitting guards: several arrows for one clause

One clause, a disjunctive guard

```
def baz(x, y) when is_boolean(x) or is_integer(y), do: {y, x}
```

The guard is split into its or branches; the body is typed once per branch:

```
is_boolean(x) (boolean(), term() -> {term(), boolean()})
```

```
is_integer(y) (term(), integer() -> {integer(), term()})
```

Splitting guards: several arrows for one clause

One clause, a disjunctive guard

```
def baz(x, y) when is_boolean(x) or is_integer(y), do: {y, x}
```

The guard is split into its or branches; the body is typed once per branch:

```
is_boolean(x) (boolean(), term() -> {term(), boolean()})
```

```
is_integer(y) (term(), integer() -> {integer(), term()})
```

Inferred type: the intersection of the two arrows

```
$ (boolean(), term() -> {term(), boolean()}) and  
  (term(), integer() -> {integer(), term()})
```

Branches are analyzed in evaluation order: swapping the operands of or can change the arrows.

Splitting guards: several arrows for one clause

One clause, a disjunctive guard

```
def baz(x, y) when is_boolean(x) or is_integer(y), do: {y, x}
```

The guard is split into its or branches; the body is typed once per branch:

```
is_boolean(x) (boolean(), term() -> {term(), boolean()})
```

```
is_integer(y) (term(), integer() -> {integer(), term()})
```

What it allows to type

```
{n, b} = baz(true, 2)  # both arrows apply: {integer(), boolean()}  
{n + 1, not b}         # accepted (with one arrow: n, b : term(), rejected)
```

Outline

1. Set-theoretic types and dynamic languages

- 1.1 Branching, overloading, and pattern matching
- 1.2 Semantic subtyping

2. Elixir basic typing

- 2.1 Unions, intersections, negations
- 2.2 Polymorphism

3. Guard analysis

- 3.1 Exhaustiveness and dead clauses
- 3.2 Narrowing and inference

4. Gradual typing

- 4.1 Strong function types
- 4.2 Propagation of `dynamic()`

5. Integration in Elixir

Gradual typing: no big-bang migration

Two non-negotiable requirements for real-world adoption by existing languages:

1. You do not have to type-annotate your entire codebase at once.
2. Types do not modify the run-time semantics of existing programs.

The `dynamic()` type

- May turn into any other type at runtime: typed and untyped code coexist seamlessly
- 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()`.

The gradual typing conundrum

Typing every expression by `dynamic()` is trivially sound ...but hardly useful.

The gradual typing conundrum

Typing every expression by `dynamic()` is trivially sound ...but hardly useful.

The system must fulfill two opposite requirements

1. it must use `dynamic()` as little as possible so as to be useful, and
2. it must use `dynamic()` enough so as not to hinder the versatility of gradual typing

The gradual typing conundrum

Typing every expression by `dynamic()` is trivially sound ... but hardly useful.

The system must fulfill two opposite requirements

1. it must use `dynamic()` as little as possible so as to be useful, and
2. it must use `dynamic()` enough so as not to hinder the versatility of gradual typing

The first requirement is fulfilled by the typing of **strong functions**, which leverage the run-time checks written by the programmer, or performed by the VM

The second requirement is fulfilled by the **propagation of `dynamic()`**.

Strong function types

(Weak) function type

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

Strong function types

(Weak) function type

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

Let `x_dyn` be of type `dynamic()`

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

`x_dyn` may be a string, and
`id_weak` returns it unchanged

Strong function types

(Weak) function type

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

Let `x_dyn` be of type `dynamic()`

```
id_weak(x_dyn)           x_dyn may be a string, and  
# type dynamic()        id_weak returns it unchanged
```

Strong function type

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

Strong function types

(Weak) function type

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

Let `x_dyn` be of type `dynamic()`

<code>id_weak(x_dyn)</code>	<code>id_strong(x_dyn)</code>
<code># type dynamic()</code>	<code># type integer()</code>

Strong function type

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

Strong function types

(Weak) function type

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

Let `x_dyn` be of type `dynamic()`

<code>id_weak(x_dyn)</code>	<code>id_strong(x_dyn)</code>
<code># type dynamic()</code>	<code># type integer()</code>

Strong function type

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

Strong function: on any argument, even outside its domain, it returns a result in its codomain or fails

Strong function types

(Weak) function type

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

Let `x_dyn` be of type `dynamic()`

<code>id_weak(x_dyn)</code>	<code>id_strong(x_dyn)</code>
<code># type dynamic()</code>	<code># type integer()</code>

Strong function type (by the programmer)

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

Strong function: on any argument, even outside its domain, it returns a result in its codomain or fails

Strong function types

(Weak) function type

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

Let `x_dyn` be of type `dynamic()`

<code>id_weak(x_dyn)</code>	<code>id_strong(x_dyn)</code>
<code># type dynamic()</code>	<code># type integer()</code>

Strong function type (by the programmer)

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

Strong function type (by the VM)

```
$ number() -> number()  
def double(x), do: x + x
```

Strong function types

(Weak) function type

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

Let `x_dyn` be of type `dynamic()`

<code>id_weak(x_dyn)</code>	<code>id_strong(x_dyn)</code>
<code># type dynamic()</code>	<code># type integer() and dynamic()</code>

Strong function type (by the programmer)

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

Strong function type (by the VM)

```
$ number() -> number()  
def double(x), do: x + x
```

Strong function types

(Weak) function type

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

PROPAGATION OF *dynamic()* !

Let *x_dyn* be of type *dynamic()*

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

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

Strong function type (by the programmer)

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

Strong function type (by the VM)

```
$ number() -> number()  
def double(x), do: x + x
```

Propagation of `dynamic()`

Apply the (strong function) `negate()` function to a `dynamic()` argument: `x_dyn`

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

Propagation of `dynamic()`

Apply the (strong function) `negate()` function to a `dynamic()` argument: `x_dyn`

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

Propagation of dynamic()

Apply the (strong function) negate() function to a dynamic() argument: x_dyn

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

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

Propagation of `dynamic()`

Apply the (strong function) `negate()` function to a `dynamic()` argument: `x_dyn`

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

```
def subtract(a, b), do: a + negate(b)    # a, b not annotated, hence dynamic()
```

Propagation of dynamic()

Apply the (strong function) `negate()` function to a `dynamic()` argument: `x_dyn`

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

```
def subtract(a, b), do: a + negate(b)    # a, b not annotated, hence dynamic()  
#=> Without propagation: type error, + undefined for booleans
```

Propagation of `dynamic()`

Apply the (strong function) `negate()` function to a `dynamic()` argument: `x_dyn`

```
$ (integer() -> integer()) and (boolean() -> boolean())  
def negate(x) when is_integer(x), do: -x  
def negate(x) when is_boolean(x), do: not x  
#=> negate(x_dyn): (integer() or boolean()) and dynamic()  
  
def subtract(a, b), do: a + negate(b)    # a, b not annotated, hence dynamic()
```

Propagation of `dynamic()`

Apply the (strong function) `negate()` function to a `dynamic()` argument: `x_dyn`

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

```
def subtract(a, b), do: a + negate(b)    # a, b not annotated, hence dynamic()  
#=> subtract: (dynamic(), dynamic() -> number())
```

Propagation of `dynamic()`

Apply the (strong function) `negate()` function to a `dynamic()` argument: `x_dyn`

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

```
def subtract(a, b), do: a + negate(b)    # a, b not annotated, hence dynamic()  
#=> subtract: (dynamic(), dynamic() -> number()) and dynamic()
```

Propagation of `dynamic()`

Apply the (strong function) `negate()` function to a `dynamic()` argument: `x_dyn`

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

```
def subtract(a, b), do: a + negate(b)    # a, b not annotated, hence dynamic()  
#=> subtract: (dynamic(), dynamic() -> number()) and dynamic()
```

```
def concat(a, b), do: a <> negate(b)  
warning: incompatible types in binary construction: <<..., negate(b)::binary>>  
got type: dynamic(boolean() or integer()), expected type: binary()  
(dynamic(t) is how Elixir prints t and dynamic())
```

Propagation of `dynamic()`

Apply the (strong function) `negate()` function to a `dynamic()` argument: `x_dyn`

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

```
def subtract(a, b), do: a + negate(b)    # a, b not annotated, hence dynamic()  
#=> subtract: (dynamic(), dynamic() -> number()) and dynamic()
```

```
def concat(a, b), do: a <> negate(b)  
warning: incompatible types in binary construction: <<..., negate(b)::binary>>  
got type: dynamic(boolean() or integer()), expected type: binary()  
(dynamic(t) is how Elixir prints t and dynamic())
```

`t` and `dynamic()` is accepted where `s` is expected iff `t` and `s` is not `none()`: `dynamic()` is forced to materialize into `s`, and is propagated to the result.

Outline

1. Set-theoretic types and dynamic languages

- 1.1 Branching, overloading, and pattern matching
- 1.2 Semantic subtyping

2. Elixir basic typing

- 2.1 Unions, intersections, negations
- 2.2 Polymorphism

3. Guard analysis

- 3.1 Exhaustiveness and dead clauses
- 3.2 Narrowing and inference

4. Gradual typing

- 4.1 Strong function types
- 4.2 Propagation of `dynamic()`

5. Integration in Elixir

Elixir 1.20: typing without annotations

- No annotation, no syntax change, no type variables: parameters are `dynamic()`
- Types are refined by patterns and guards (guard analysis, Section 3)
- Inferred types are checked by gradual typing (Section 4)
- Multi-clause functions get intersection types from their guards (Section 3), with `dynamic()` propagated to the results (Section 4); for `negate`:

```
(integer() -> integer() and dynamic()) and  
(boolean() -> boolean() and dynamic())
```

The error is still caught

```
def bad(b) when is_boolean(b), do: 1 + negate(b)  
warning: incompatible types given to Kernel.+/2: 1 + negate(b)  
given types: integer(), dynamic(boolean()) (i.e., boolean() and dynamic())
```

Elixir 1.20: typing without annotations

- No annotation, no syntax change, no type variables: parameters are `dynamic()`
- Types are refined by patterns and guards (guard analysis, Section 3)
- Inferred types are checked by gradual typing (Section 4)
- Multi-clause functions get intersection types from their guards (Section 3), with `dynamic()` propagated to the results (Section 4); for `negate`:

```
(integer() -> integer() and dynamic()) and  
(boolean() -> boolean() and dynamic())
```

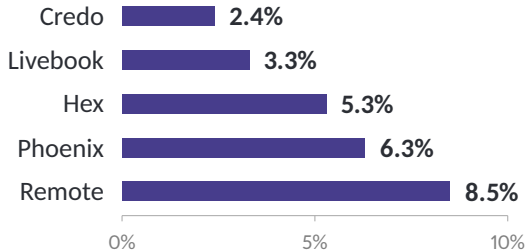
The error is still caught

```
def bad(b) when is_boolean(b), do: 1 + negate(b)  
warning: incompatible types given to Kernel.+/: 1 + negate(b)  
given types: integer(), dynamic(boolean()) (i.e., boolean() and dynamic())
```

Type signatures (the \$ of this talk), and with them type variables, are planned after efficient recursive and parametric types, map traversal, and typed structs are implemented.

Compilation cost on production code

Checking as a share of total compilation time, Elixir v1.19



Remote

19.5 s

to check **24,292 modules**

Over 1 million lines of code
228.8 s total compilation

Precision on real Elixir code

17 open-source projects, without user type annotations

86.1%

Exact pattern/guard pairs

141,755 of 164,701 analyzed pairs

The analysis identifies exactly which inputs match.

61.7%

Informative inferred returns

43,158 of 69,976 inferred arrows

The return type differs from `dynamic()`. For example, `dynamic(integer())`.

Source: Castagna & Duboc, Core Elixir paper. Empirical evaluation and compilation measurements.

Narrowing through a nested condition

Elixir adaptation of *If-T*: 12/13 core items pass

Research compiler

```
# x, y : term()
if if(is_integer(x),
      do: is_binary(y),
      else: false) do
  x + byte_size(y)
else
  0
end
```

Inside the outer branch

x : integer()

y : binary()

The outer condition can only be true when both tests succeed.

Elixir accepts the program.

Paper snapshot: 7 of 11 other implementations miss this item.

TypeScript, Flow, mypy, Pyright, Luau, ty, Pyrefly.

Status of the integration

- Very positive feedback from developers on versions 1.17 to 1.20
- No syntactic changes and no annotations required: particularly appreciated
- Errors found in tested and reviewed code, in production for months or years

Status of the integration

- Very positive feedback from developers on versions 1.17 to 1.20
- No syntactic changes and no annotations required: particularly appreciated
- Errors found in tested and reviewed code, in production for months or years

Real-world Bug Discoveries

- **Phoenix** (deployed on 12,000+ websites): an exception handler reads the message field, which not every exception defines: the error branch can itself crash
- **Remote** (24,000 modules): `String.Chars` called on a tuple in a rarely executed error branch, undetected by tests
- **Dead code**: 14 fixes, at least 179 lines removed in 9 projects (Postgrex, Flame, LiveView, Ecto, ExDoc, Livebook, Ash, Phoenix, Spitfire)
- **v1.20**: porting one company application to 1.20 revealed about 500 problems missed by 1.19, mostly dead code but also real bugs

Future work: typing modules

Today the type `module()` says nothing about what a module implements

```
defmodule Square do
  @behaviour Shape
  def area(side), do: side * side
end
```

today (Elixir 1.20)

```
@spec compute(module(), term()) :: term()
def compute(m, x), do: m.area(x)
```

```
compute(Square, "big")
```

X crashes at run time

Shape is a *behaviour* (an interface for modules) requiring `@callback area(term()) :: term()`

Future work: typing modules

Today the type `module()` says nothing about what a module implements

```
defmodule Square do
  $behaviour Shape[unit: integer()]
  def area(side), do: side * side
end
```

tomorrow

```
$ (Shape[unit: a], a) -> a when a: term()
def compute(m, x), do: m.area(x)
```

```
compute(Square, "big")
```

✓ error: the unit of Square is integer()

Shape is a *behaviour* declaring a type parameter `unit` and **\$callback** `area(unit) = unit`

Future work: typing processes

Today the type `pid()` says nothing about the messages a process accepts

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

✗ crashes at run time

today (Elixir 1.20)

Future work: typing processes

Today the type `pid()` says nothing about the messages a process accepts

```
$interface integer()  
spawn(fn ->  
  send(self(), true)  
  receive do y -> IO.puts(y + y) end  
end)
```

tomorrow

✓ error: true is not an integer()

Future work: typing processes

Today the type `pid()` says nothing about the messages a process accepts

```
$interface boolean()  
spawn(fn ->  
  send(self(), true)  
  receive do y -> IO.puts(y + y) end  
end)
```

✓ error: y is not an integer()

tomorrow

Future work: record type comprehensions

Today `Map.intersect/2` and its kin return `dynamic()`: the keys are lost

```
m = Map.intersect(%{a: 1, b: "x"}, %{b: :ok, c: 2})  
m.a  
m.b + 1
```

today (Elixir 1.20)

✗ crashes at run time

✗ crashes at run time

The same comprehensions type `Map.take`, `drop`, `merge`, `filter`, ...

Modules: with A. Boussaa, A. Bretagne, J. Valim. Processes: with L. Padovani, D. Varacca. Syntax not final.

Future work: record type comprehensions

Today `Map.intersect/2` and its kin return `dynamic()`: the keys are lost

```
$ (a, b) -> %{k in keyof(a) and keyof(b) | b[k]}
```

tomorrow

```
m = Map.intersect(%{a: 1, b: "x"}, %{b: :ok, c: 2})
```

```
m.a
```

✓ error: no key :a

```
m.b + 1
```

✓ error: m.b is :ok, not a number

The same comprehensions type `Map.take`, `drop`, `merge`, `filter`, ...

Modules: with A. Boussaa, A. Bretagne, J. Valim. Processes: with L. Padovani, D. Varacca. Syntax not final.

Conclusion

- **Types are sets of values.** Unions, intersections and negations are set operations, subtyping is inclusion: they describe tagged unions, overloading and pattern matching, the everyday idioms of dynamic functional code.
- **Inferred from patterns and guards.** With no annotation, multi-clause functions get intersection types, each clause sees a narrowed type, and the compiler reports non-exhaustive matches and dead clauses.
- **Gradual, without casts.** `dynamic()` plus strong functions: run-time semantics unchanged, existing code compiles as before.
- **In production.** Elixir type-checks the whole language since v1.20 and finds real bugs; the same theory is used by Python's `ty`, Erlang and R tools, Luau, Ballerina.
- **Next.** Type signatures, then modules, processes, record type comprehensions.

Conclusion

- **Types are sets of values.** Unions, intersections and negations are set operations, subtyping is inclusion: they describe tagged unions, overloading and pattern matching, the everyday idioms of dynamic functional code.
- **Inferred from patterns and guards.** With no annotation, multi-clause functions get intersection types, each clause sees a narrowed type, and the compiler reports non-exhaustive matches and dead clauses.
- **Gradual, without casts.** `dynamic()` plus strong functions: run-time semantics unchanged, existing code compiles as before.
- **In production.** Elixir type-checks the whole language since v1.20 and finds real bugs; the same theory is used by Python's `ty`, Erlang and R tools, Luau, Ballerina.
- **Next.** Type signatures, then modules, processes, record type comprehensions.

**Set-theoretic types fit dynamic languages built on pattern matching and guards:
the theory is ready, and it runs in production.**