



INSTITUT DE RECHERCHE EN
INFORMATIQUE FONDAMENTALE

Typing Dynamic Languages with Set-Theoretic Types: The Case of Elixir

Giuseppe CASTAGNA

(joint work with Guillaume Duboc and José Valim)

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

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 Simple function types
- 2.2 Unions, intersections, negations
- 2.3 Polymorphism

3. Guard Analysis

- 3.1 Case expressions: redundancy, exhaustiveness, and narrowing
- 3.2 A Pinch of Formalization

4. Gradual Typing

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

5. Integration in Elixir

Set-theoretic types

We consider the following possibly recursive types:

$$T ::= \text{Bool} \mid \text{Int} \mid T \times T \mid T \rightarrow T \mid T \vee T \mid T \wedge T \mid \neg T \mid \underline{1} \mid 0$$

Set-theoretic types

We consider the following possibly recursive types:

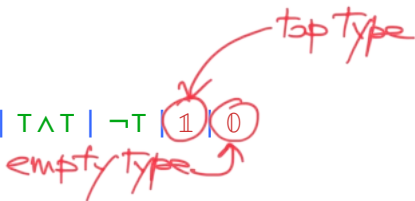
$T ::= \text{Bool} \mid \text{Int} \mid T \times T \mid T \rightarrow T \mid T \vee T \mid T \wedge T \mid \neg T \mid \underline{1} \mid 0$

Set-theoretic Types 

Set-theoretic types

We consider the following possibly recursive types:

$T ::= \text{Bool} \mid \text{Int} \mid T \times T \mid T \rightarrow T \mid T \vee T \mid T \wedge T \mid \neg T \mid \boxed{1} \mid \boxed{0}$



Set-theoretic types

We consider the following possibly recursive types:

$T ::= \text{Bool} \mid \text{Int} \mid T \times T \mid T \rightarrow T \mid T \vee T \mid T \wedge T \mid \neg T \mid 1 \mid 0 \mid \alpha$

type variables $\rightarrow$

Set-theoretic types

We consider the following possibly recursive types:

$T ::= \text{Bool} \mid \text{Int} \mid T \times T \mid T \rightarrow T \mid T \vee T \mid T \wedge T \mid \neg T \mid \underline{1} \mid 0 \mid \alpha \mid c$

singleton types 

Set-theoretic types

We consider the following possibly recursive types:

$$T ::= \text{Bool} \mid \text{Int} \mid T \times T \mid T \rightarrow T \mid T \vee T \mid T \wedge T \mid \neg T \mid \underline{1} \mid \underline{0} \mid \alpha \mid c$$

Necessary to precisely capture the programming style used with dynamic languages:

1. non-uniform structures/branching (union types, singletons)
2. overloading (intersection types)
3. pattern matching (negation types, ...)

Note: henceforward I will sometimes use $T_1 | T_2$ to denote $T_1 \vee T_2$

Set-theoretic types

We consider the following possibly recursive types:

$$T ::= \text{Bool} \mid \text{Int} \mid T \times T \mid T \rightarrow T \mid T \vee T \mid T \wedge T \mid \neg T \mid \underline{1} \mid \underline{0} \mid \alpha \mid c$$

Necessary to precisely capture the programming style used with dynamic languages:

1. non-uniform structures/branching (union types, singletons)
2. overloading (intersection types)
3. pattern matching (negation types, ...)

Let us see each point more in detail

Note: henceforward I will sometimes use $T_1 | T_2$ to denote $T_1 \vee T_2$

1. Branching and discriminated unions

```
type Status = "pending" | "success" | "error";
type Result = string | number | boolean;
function processStatus(input: Status): Result {
  switch (input) {
    case "pending":
      return "Processing...";           // returns string
    case "success":
      return 200;                       // returns number
    case "error":
      return false;                     // returns boolean
    default:
      return "Unknown status";          // returns string
  }
}
```

1. Branching and discriminated unions

```
type Status = "pending" | "success" | "error";
type Result = string | number | boolean;
function processStatus(input: Status): Result {
  switch (input) {
    case "pending":
      return "Processing...";           // returns string
    case "success":
      return 200;                       // returns number
    case "error":
      return false;                     // returns boolean
    default:
      return "Unknown status";          // returns string
  }
}

type PaymentMethod =
  | { type: "credit_card"; cardNumber: string; cvv: string }
  | { type: "paypal"; email: string }
  | { type: "bank_transfer"; accountNumber: string; routingNumber: string };
```

2. Overloading: basic use

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);  
}
```

If applied to an integer returns an integer, if applied to a string returns a string

2. Overloading: basic use

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);  
}
```

If applied to an integer returns an integer, if applied to a string returns a string

Use set-theoretic types

2. Overloading: basic use

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);  
}
```

If applied to an integer returns an integer, if applied to a string returns a string

Use set-theoretic types

- Naive solution: union types

$$(\text{number} \mid \text{string}) \rightarrow (\text{number} \mid \text{string})$$

2. Overloading: basic use

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);  
}
```

If applied to an integer returns an integer, if applied to a string returns a string

Use set-theoretic types

- Naive solution: union types

$(\text{number} \mid \text{string}) \rightarrow (\text{number} \mid \text{string})$

2. Overloading: basic use

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);  
}
```

If applied to an integer returns an integer, if applied to a string returns a string

Use set-theoretic types

- Naive solution: union types

$(\text{number} \mid \text{string}) \rightarrow (\text{number} \mid \text{string})$

- Better solution: intersection types

$(\text{number} \rightarrow \text{number}) \ \& \ (\text{string} \rightarrow \text{string})$

2. Overloading: basic use

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);  
}
```

If applied to an integer returns an integer, if applied to a string returns a string

Use set-theoretic types

- Naive solution: union types

$(\text{number} \mid \text{string}) \rightarrow (\text{number} \mid \text{string})$

- Better solution: intersection types

$(\text{number} \rightarrow \text{number}) \ \& \ (\text{string} \rightarrow \text{string})$

2. Overloading: advanced use

```
function toBoolean(x) {  
  if (x === 0 || x === "" || x == null || Number.isNaN(x)) return false;  
  return true;  
}
```

```
function logicalOr(x, y) {  
  if (toBoolean(x)) return x; return y;  
}
```

2. Overloading: advanced use

```
function toBoolean(x) {  
  if (x === 0 || x === "" || x == null || Number.isNaN(x)) return false;  
  return true;  
}  
  
function logicalOr(x, y) {  
  if (toBoolean(x)) return x; return y;  
}
```

Fine grained typing to capture the code semantics

```
type Falsy = 0 | "" | null | undefined | NaN;  
type Truthy =  $\neg$ Falsy;  
toBoolean : (Truthy  $\rightarrow$  true)  $\wedge$  (Falsy  $\rightarrow$  false)  
logicalOR : (( $\alpha$   $\wedge$  Truthy  $\times \mathbb{1}$ )  $\rightarrow \alpha$ )  $\wedge$  ((Falsy  $\times \beta$ )  $\rightarrow \beta$ )
```

3. Precise typing of pattern matching (I)

Consider the following pattern matching expression

`match e with  $p_1 \rightarrow e_1$  |  $p_2 \rightarrow e_2$`

where patterns are defined as follows:

$p ::= x \mid (p, p) \mid p \mid p \mid p \wedge p$

3. Precise typing of pattern matching (I)

Consider the following pattern matching expression

`match e with  $p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2$`

where patterns are defined as follows:

$p ::= x \mid (p, p) \mid p \mid p \mid p \wedge p$

If we interpret types as set of values

$t = \{v \mid v \text{ is a value of type } t\}$

3. Precise typing of pattern matching (I)

Consider the following pattern matching expression

`match e with  $p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2$`

where patterns are defined as follows:

$p ::= x \mid (p, p) \mid p \mid p \mid p \wedge p$

If we interpret types as set of values

$t = \{v \mid v \text{ is a value of type } t\}$

then **the set of all values that match a pattern is a type**

$\llbracket p \rrbracket = \{v \mid v \text{ is a value that matches } p\}$

$$\begin{aligned}\llbracket x \rrbracket &= \mathbb{1} \\ \llbracket (p_1, p_2) \rrbracket &= \llbracket p_1 \rrbracket \times \llbracket p_2 \rrbracket \\ \llbracket p_1 \mid p_2 \rrbracket &= \llbracket p_1 \rrbracket \vee \llbracket p_2 \rrbracket \\ \llbracket p_1 \wedge p_2 \rrbracket &= \llbracket p_1 \rrbracket \wedge \llbracket p_2 \rrbracket\end{aligned}$$

2. Precise typing of pattern matching (I)

Consider the following pattern matching expression

`match e with  $p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2$`

where patterns are defined as follows:

$p ::= x \mid (p, p) \mid p \mid p \mid p \wedge p$

If we interpret types as set of values

$t = \{v \mid v \text{ is a value of type } t\}$

then **the set of all values that match a pattern is a type**

$\{p\} = \{v \mid v \text{ is a value that matches } p\}$

accepted
Type of p



$$\begin{aligned}\{x\} &= \mathbb{1} \\ \{(p_1, p_2)\} &= \{p_1\} \times \{p_2\} \\ \{p_1 \mid p_2\} &= \{p_1\} \vee \{p_2\} \\ \{p_1 \wedge p_2\} &= \{p_1\} \wedge \{p_2\}\end{aligned}$$

3. Precise typing of pattern matching (I)

Consider the following pattern matching expression

`match e with  $p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2$`

where patterns are defined as follows:

$p ::= x \mid (p, p) \mid p \mid p \mid p \wedge p$

If we interpret types as set of values

$t = \{v \mid v \text{ is a value of type } t\}$

then **the set of all values that match a pattern is a type**

$\{p\} = \{v \mid v \text{ is a value that matches } p\}$

$$\begin{aligned}\{x\} &= \mathbb{1} \\ \{(p_1, p_2)\} &= \{p_1\} \times \{p_2\} \\ \{p_1 \mid p_2\} &= \{p_1\} \vee \{p_2\} \\ \{p_1 \wedge p_2\} &= \{p_1\} \wedge \{p_2\}\end{aligned}$$

Let us see how to type pattern matching.

3. Precise typing of pattern matching (II)

Boolean type connectives are needed to type pattern matching:

3. Precise typing of pattern matching (II)

Boolean type connectives are needed to type pattern matching:

```
match e with  $p_1 \rightarrow e_1$  |  $p_2 \rightarrow e_2$ 
```

3. Precise typing of pattern matching (II)

Boolean type connectives are needed to type pattern matching:

`match e with p1 -> e1 | p2 -> e2`

Suppose that $e : T$ and let us write $T_1 \searrow T_2$ for $T_1 \wedge \neg T_2$

3. Precise typing of pattern matching (II)

Boolean type connectives are needed to type pattern matching:

`match e with p1 -> e1 | p2 -> e2`

Suppose that $e : T$ and let us write $T_1 \searrow T_2$ for $T_1 \wedge \neg T_2$

- To infer the type T_1 of e_1 we need $T \wedge \{p_1\}$;

3. Precise typing of pattern matching (II)

Boolean type connectives are needed to type pattern matching:

`match e with  $p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2$`

Suppose that $e : T$ and let us write $T_1 \setminus T_2$ for $T_1 \wedge \neg T_2$

- 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 $(T \setminus \{p_1\}) \wedge \{p_2\}$;

3. Precise typing of pattern matching (II)

Boolean type connectives are needed to type pattern matching:

`match e with p1 -> e1 | p2 -> e2`

Suppose that $e : T$ and let us write $T_1 \setminus T_2$ for $T_1 \wedge \neg T_2$

- 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 $T_1 \vee T_2$.

3. Precise typing of pattern matching (II)

Boolean type connectives are needed to type pattern matching:

`match e with p1 -> e1 | p2 -> e2`

Suppose that $e : T$ and let us write $T_1 \setminus T_2$ for $T_1 \wedge \neg T_2$

- 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 $T \leq \{p_1\} \vee \{p_2\}$;

3. Precise typing of pattern matching (II)

Boolean type connectives are needed to type pattern matching:

`match e with p1 -> e1 | p2 -> e2`

Suppose that $e : T$ and let us write $T_1 \setminus T_2$ for $T_1 \wedge \neg T_2$

- 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 $T \leq \{p_1\} \vee \{p_2\}$;

3. Precise typing of pattern matching (II)

Boolean type connectives are needed to type pattern matching:

`match e with p1 -> e1 | p2 -> e2`

Suppose that $e : T$ and let us write $T_1 \searrow T_2$ for $T_1 \wedge \neg T_2$

- 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 $(T \searrow \{p_1\}) \wedge \{p_2\}$;
- 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:

[Match]

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

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

2. Precise typing of pattern matching (II)

Boolean type connectives are needed to type pattern matching:

`match e with p1 -> e1 | p2 -> e2`

Suppose that $e : T$ and let us write $T_1 \searrow T_2$ for $T_1 \wedge \neg T_2$

- 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 $(T \searrow \{p_1\}) \wedge \{p_2\}$;
- 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:

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

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

Semantic subtyping

Many dynamic languages support union and intersection types: TypeScript, Flow, Hack, Python, Julia, and Ceylon. Negation types are less common: TypeScript (with the 'Exclude' utility type) and Scala (with 'Not' type constraints).

Semantic subtyping

Many dynamic languages support union and intersection types: TypeScript, Flow, Hack, Python, Julia, and Ceylon. Negation types are less common: TypeScript (with the 'Exclude' utility type) and Scala (with 'Not' type constraints).

What differentiates our approach is the use of semantic subtyping to define the subtyping relation for union, intersection, and negation types.

Semantic subtyping

Many dynamic languages support union and intersection types: TypeScript, Flow, Hack, Python, Julia, and Ceylon. Negation types are less common: TypeScript (with the 'Exclude' utility type) and Scala (with 'Not' type constraints).

What differentiates our approach is the use of semantic subtyping to define the subtyping relation for union, intersection, and negation types.

1. Define a set-theoretic interpretation of types as sets of values

Semantic subtyping

Many dynamic languages support union and intersection types: TypeScript, Flow, Hack, Python, Julia, and Ceylon. Negation types are less common: TypeScript (with the 'Exclude' utility type) and Scala (with 'Not' type constraints).

What differentiates our approach is the use of semantic subtyping to define the subtyping relation for union, intersection, and negation types.

1. Define a set-theoretic interpretation of types as sets of values
2. Define union, intersection, and negation types as set-theoretic operations

Semantic subtyping

Many dynamic languages support union and intersection types: TypeScript, Flow, Hack, Python, Julia, and Ceylon. Negation types are less common: TypeScript (with the 'Exclude' utility type) and Scala (with 'Not' type constraints).

What differentiates our approach is the use of semantic subtyping to define the subtyping relation for union, intersection, and negation types.

1. Define a set-theoretic interpretation of types as sets of values
2. Define union, intersection, and negation types as set-theoretic operations
3. Define the subtyping relation as a set-theoretic inclusion relation

Semantic subtyping

Many dynamic languages support union and intersection types: TypeScript, Flow, Hack, Python, Julia, and Ceylon. Negation types are less common: TypeScript (with the 'Exclude' utility type) and Scala (with 'Not' type constraints).

What differentiates our approach is the use of semantic subtyping to define the subtyping relation for union, intersection, and negation types.

1. Define a set-theoretic interpretation of types as sets of values
2. Define union, intersection, and negation types as set-theoretic operations
3. Define the subtyping relation as a set-theoretic inclusion relation
4. Deduce from this how to decide the subtyping relation and compute type operators

Semantic subtyping: in practice

Semantic subtyping: in practice

1. $\llbracket \text{Int} \rrbracket = \{n \mid n \text{ is an integer}\}$
 $\llbracket \text{Bool} \rrbracket = \{\text{true}, \text{false}\}$
 $\llbracket T_1 \rightarrow T_2 \rrbracket = \{f \mid v \in \llbracket T_1 \rrbracket \implies f(v) \in \llbracket T_2 \rrbracket\}$
 $\vdots$

Semantic subtyping: in practice

- $$\begin{aligned} \llbracket \text{Int} \rrbracket &= \{n \mid n \text{ is an integer}\} \\ \llbracket \text{Bool} \rrbracket &= \{\text{true}, \text{false}\} \\ \llbracket T_1 \rightarrow T_2 \rrbracket &= \{f \mid v \in \llbracket T_1 \rrbracket \implies f(v) \in \llbracket T_2 \rrbracket\} \\ &\vdots \end{aligned}$$
- $$\llbracket T_1 \vee T_2 \rrbracket = \llbracket T_1 \rrbracket \cup \llbracket T_2 \rrbracket, \quad \llbracket T_1 \wedge T_2 \rrbracket = \llbracket T_1 \rrbracket \cap \llbracket T_2 \rrbracket, \quad \llbracket \neg T \rrbracket = \{v \mid v \notin \llbracket T \rrbracket\}$$

Semantic subtyping: in practice

- $$\begin{aligned} \llbracket Int \rrbracket &= \{n \mid n \text{ is an integer}\} \\ \llbracket Bool \rrbracket &= \{\text{true}, \text{false}\} \\ \llbracket T_1 \rightarrow T_2 \rrbracket &= \{f \mid v \in \llbracket T_1 \rrbracket \implies f(v) \in \llbracket T_2 \rrbracket\} \\ &\vdots \end{aligned}$$
- $$\llbracket T_1 \vee T_2 \rrbracket = \llbracket T_1 \rrbracket \cup \llbracket T_2 \rrbracket, \quad \llbracket T_1 \wedge T_2 \rrbracket = \llbracket T_1 \rrbracket \cap \llbracket T_2 \rrbracket, \quad \llbracket \neg T \rrbracket = \{v \mid v \notin \llbracket T \rrbracket\}$$
- $$T_1 \leq T_2 \stackrel{def}{\iff} \llbracket T_1 \rrbracket \subseteq \llbracket T_2 \rrbracket \iff \llbracket T_1 \rrbracket \cap \llbracket \neg T_2 \rrbracket = \emptyset \iff T_1 \wedge \neg T_2 \leq \mathbb{0}$$

Semantic subtyping: in practice

- $$\begin{aligned} \llbracket \text{Int} \rrbracket &= \{n \mid n \text{ is an integer}\} \\ \llbracket \text{Bool} \rrbracket &= \{\text{true}, \text{false}\} \\ \llbracket T_1 \rightarrow T_2 \rrbracket &= \{f \mid v \in \llbracket T_1 \rrbracket \implies f(v) \in \llbracket T_2 \rrbracket\} \\ &\vdots \end{aligned}$$
- $$\llbracket T_1 \vee T_2 \rrbracket = \llbracket T_1 \rrbracket \cup \llbracket T_2 \rrbracket, \quad \llbracket T_1 \wedge T_2 \rrbracket = \llbracket T_1 \rrbracket \cap \llbracket T_2 \rrbracket, \quad \llbracket \neg T \rrbracket = \{v \mid v \notin \llbracket T \rrbracket\}$$
- $$T_1 \leq T_2 \stackrel{\text{def}}{\iff} \llbracket T_1 \rrbracket \subseteq \llbracket T_2 \rrbracket \iff \llbracket T_1 \rrbracket \cap \llbracket \neg T_2 \rrbracket = \emptyset \iff T_1 \wedge \neg T_2 \leq \mathbb{0}$$
- $$\bigwedge_{i \in I} t_i \times s_i \leq \bigvee_{i \in J} t_i \times s_i \iff \forall J' \subset J. \left(\bigwedge_{i \in I} t_i \leq \bigvee_{i \in J'} t_i \right) \text{ or } \left(\bigwedge_{i \in I} s_i \leq \bigvee_{i \in J \setminus J'} s_i \right)$$

Semantic subtyping: in practice

- $$\begin{aligned}\llbracket \text{Int} \rrbracket &= \{n \mid n \text{ is an integer}\} \\ \llbracket \text{Bool} \rrbracket &= \{\text{true}, \text{false}\} \\ \llbracket T_1 \rightarrow T_2 \rrbracket &= \{f \mid v \in \llbracket T_1 \rrbracket \implies f(v) \in \llbracket T_2 \rrbracket\} \\ &\vdots\end{aligned}$$
- $$\llbracket T_1 \vee T_2 \rrbracket = \llbracket T_1 \rrbracket \cup \llbracket T_2 \rrbracket, \quad \llbracket T_1 \wedge T_2 \rrbracket = \llbracket T_1 \rrbracket \cap \llbracket T_2 \rrbracket, \quad \llbracket \neg T \rrbracket = \{v \mid v \notin \llbracket T \rrbracket\}$$
- $$T_1 \leq T_2 \stackrel{\text{def}}{\iff} \llbracket T_1 \rrbracket \subseteq \llbracket T_2 \rrbracket \iff \llbracket T_1 \rrbracket \cap \llbracket \neg T_2 \rrbracket = \emptyset \iff T_1 \wedge \neg T_2 \leq \mathbb{0}$$
- $$\bigwedge_{i \in I} t_i \times s_i \leq \bigvee_{i \in J} t_i \times s_i \iff \forall J' \subset J. \left(\bigwedge_{i \in I} t_i \leq \bigvee_{i \in J'} t_i \right) \text{ or } \left(\bigwedge_{i \in I} s_i \leq \bigvee_{i \in J \setminus J'} s_i \right)$$

Paying the price

- ✓ Unions, intersections, and negations are really set-theoretic
- ✗ Simply extending syntactic subtyping is not an option

Theoretical foundations

- Semantic Subtyping for Set-Theoretic Types (JACM 2008)

Theoretical foundations

- Semantic Subtyping for Set-Theoretic Types (JACM 2008)
- Polymorphic Types and Subtyping (ICFP 2011)

Theoretical foundations

- Semantic Subtyping for Set-Theoretic Types (JACM 2008)
- Polymorphic Types and Subtyping (ICFP 2011)
- Typing of Polymorphic Functions and Local Type Inference (POPL 2014, POPL 2015)

Theoretical foundations

- Semantic Subtyping for Set-Theoretic Types (JACM 2008)
- Polymorphic Types and Subtyping (ICFP 2011)
- Typing of Polymorphic Functions and Local Type Inference (POPL 2014, POPL 2015)
- Gradual Typing (POPL 2019)

Theoretical foundations

- Semantic Subtyping for Set-Theoretic Types (JACM 2008)
- Polymorphic Types and Subtyping (ICFP 2011)
- Typing of Polymorphic Functions and Local Type Inference (POPL 2014, POPL 2015)
- Gradual Typing (POPL 2019)
- Monomorphic Type Reconstruction (POPL 2022)

Theoretical foundations

- Semantic Subtyping for Set-Theoretic Types (JACM 2008)
- Polymorphic Types and Subtyping (ICFP 2011)
- Typing of Polymorphic Functions and Local Type Inference (POPL 2014, POPL 2015)
- Gradual Typing (POPL 2019)
- Monomorphic Type Reconstruction (POPL 2022)
- Uniform Typing for Records and Dictionaries (ICFP 2023)

Theoretical foundations

- Semantic Subtyping for Set-Theoretic Types (JACM 2008)
- Polymorphic Types and Subtyping (ICFP 2011)
- Typing of Polymorphic Functions and Local Type Inference (POPL 2014, POPL 2015)
- Gradual Typing (POPL 2019)
- Monomorphic Type Reconstruction (POPL 2022)
- Uniform Typing for Records and Dictionaries (ICFP 2023)
- Polymorphic Type Reconstruction and Occurrence Typing (POPL 2024)

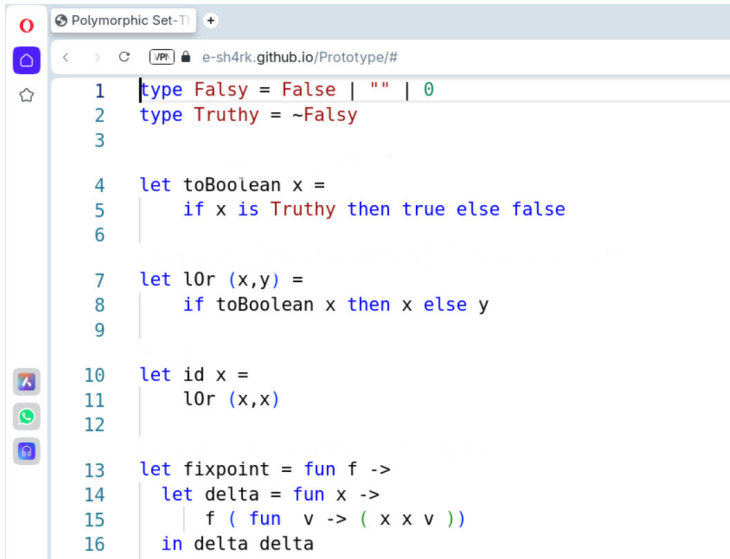
Theoretical foundations

- Semantic Subtyping for Set-Theoretic Types (JACM 2008)
- Polymorphic Types and Subtyping (ICFP 2011)
- Typing of Polymorphic Functions and Local Type Inference (POPL 2014, POPL 2015)
- Gradual Typing (POPL 2019)
- Monomorphic Type Reconstruction (POPL 2022)
- Uniform Typing for Records and Dictionaries (ICFP 2023)
- Polymorphic Type Reconstruction and Occurrence Typing (POPL 2024)
- Row Polymorphism (OOPSLA 2025)

Theoretical foundations

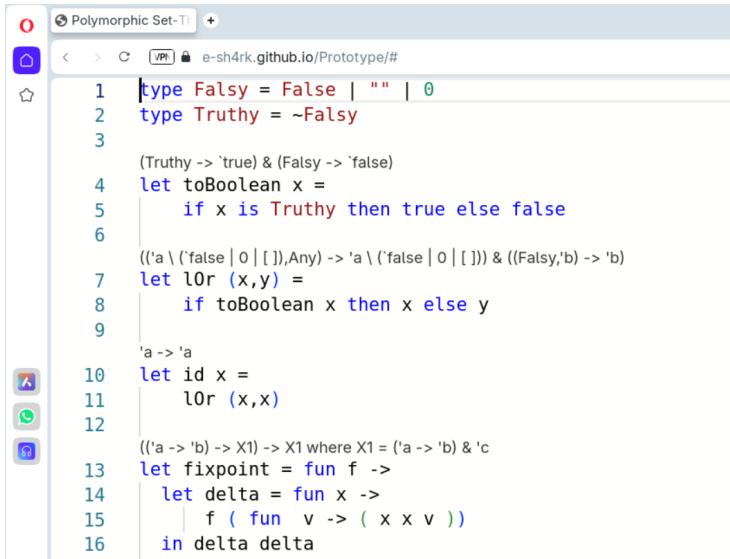
- Semantic Subtyping for Set-Theoretic Types (JACM 2008)
- Polymorphic Types and Subtyping (ICFP 2011)
- Typing of Polymorphic Functions and Local Type Inference (POPL 2014, POPL 2015)
- Gradual Typing (POPL 2019)
- Monomorphic Type Reconstruction (POPL 2022)
- Uniform Typing for Records and Dictionaries (ICFP 2023)
- Polymorphic Type Reconstruction and Occurrence Typing (POPL 2024)
- Row Polymorphism (OOPSLA 2025)
- First-class Modules (in progress)

Set-theoretic types: state of the art

A screenshot of a web browser window displaying a code editor. The browser's address bar shows the URL 'e-sh4rk.github.io/Prototype/#'. The code editor has a light blue header with the title 'Polymorphic Set-TI' and a plus sign. The code is written in a syntax-highlighted language, likely Haskell or a similar functional language. It defines types 'Falsy' and 'Truthy', a function 'toBoolean', a function 'lOr', and a fixed-point combinator 'fixpoint'. The code is numbered from 1 to 16 on the left side of the editor. The background of the code editor is white, and the text is in various colors (blue, red, green) to highlight different parts of the code.

```
1 type Falsy = False | "" | 0
2 type Truthy = ~Falsy
3
4 let toBoolean x =
5   if x is Truthy then true else false
6
7 let lOr (x,y) =
8   if toBoolean x then x else y
9
10 let id x =
11   lOr (x,x)
12
13 let fixpoint = fun f ->
14   let delta = fun x ->
15     f ( fun v -> ( x x v ) )
16   in delta delta
```

Set-theoretic types: state of the art

A screenshot of a web browser window displaying a code editor. The browser's address bar shows the URL 'e-sh4rk.github.io/Prototype/#'. The code editor contains a series of lines of code in a light blue font on a white background. The code defines a type system with 'Falsy' and 'Truthy' types, a 'toBoolean' function, a logical 'lOr' function, and a 'fixpoint' function. The code is numbered from 1 to 16 on the left side of the editor. The browser window has a light blue header bar with a red circle icon and a '+' button. On the left side of the browser window, there is a vertical sidebar with a star icon and three social media icons (Telegram, WhatsApp, and a blue circular icon).

```
1 type Falsy = False | "" | 0
2 type Truthy = ~Falsy
3
4 (Truthy -> `true) & (Falsy -> `false)
5 let toBoolean x =
6   if x is Truthy then true else false
7
8 (('a \ ('false | 0 | [ ]), Any) -> 'a \ ('false | 0 | [ ])) & ((Falsy, 'b) -> 'b)
9 let lOr (x, y) =
10   if toBoolean x then x else y
11
12 'a -> 'a
13 let id x =
14   lOr (x, x)
15
16 (('a -> 'b) -> X1) -> X1 where X1 = ('a -> 'b) & 'c
17 let fixpoint = fun f ->
18   let delta = fun x ->
19     f ( fun v -> ( x x v ) )
20   in delta delta
```

Set-theoretic types: state of the art

```
Polymorphic Set-Ti +
e-sh4rk.github.io/Prototype/#

1 type Falsy = False | "" | 0
2 type Truthy = ~Falsy
3
4 (Truthy -> `true) & (Falsy -> `false)
4 let toBoolean x =
5   if x is Truthy then true else false
6  $\forall(\alpha \leq \text{Truthy}) \forall(\beta). (\alpha, \text{Any} \rightarrow \alpha) \wedge (\text{Falsy}, \beta \rightarrow \beta)$ 
7   (('a \ ('false | 0 | []), Any) -> 'a \ ('false | 0 | [])) & ((Falsy, 'b) -> 'b)
7 let lOr (x,y) =
8   if toBoolean x then x else y
9
10 'a -> 'a
10 let id x =
11   lOr (x,x)
12  $\forall(\alpha) \forall(\beta) \forall(\gamma). ((\alpha \rightarrow \beta) \rightarrow ((\alpha \rightarrow \beta) \wedge \gamma)) \rightarrow ((\alpha \rightarrow \beta) \wedge \gamma)$ 
13   (('a -> 'b) -> X1) -> X1 where X1 = ('a -> 'b) & 'c
13 let fixpoint = fun f ->
14   let delta = fun x ->
15     f ( fun v -> ( x x v ) )
16   in delta delta
```

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 Simple function types

2.2 Unions, intersections, negations

2.3 Polymorphism

3. Guard Analysis

3.1 Case expressions: redundancy, exhaustiveness, and narrowing

3.2 A Pinch of Formalization

4. Gradual Typing

4.1 Strong function types

4.2 Propagation of `dynamic()`

5. Integration in Elixir

Putting the theory to work: Elixir

- Elixir is a dynamic language with a strong emphasis on metaprogramming

Putting the theory to work: Elixir

- Elixir is a dynamic language with a strong emphasis on metaprogramming
- It is a functional language with a syntax similar to Ruby

Putting the theory to work: Elixir

- Elixir is a dynamic language with a strong emphasis on metaprogramming
- It is a functional language with a syntax similar to Ruby
- It is built on top of the Erlang VM, which provides a powerful runtime system for distributed and fault-tolerant applications

Putting the theory to work: Elixir

- Elixir is a dynamic language with a strong emphasis on metaprogramming
- It is a functional language with a syntax similar to Ruby
- It is built on top of the Erlang VM, which provides a powerful runtime system for distributed and fault-tolerant applications
- It has a rich ecosystem of libraries and frameworks and a strong community of nearly 100K developers (with the highest median salaries cf. 2024 Stack Overflow survey)

Putting the theory to work: Elixir

- Elixir is a dynamic language with a strong emphasis on metaprogramming
- It is a functional language with a syntax similar to Ruby
- It is built on top of the Erlang VM, which provides a powerful runtime system for distributed and fault-tolerant applications
- It has a rich ecosystem of libraries and frameworks and a strong community of nearly 100K developers (with the highest median salaries cf. 2024 Stack Overflow survey)
- It is used in production by companies like Discord, Pinterest, PepsiCo, ...

Putting the theory to work: Elixir

- Elixir is a dynamic language with a strong emphasis on metaprogramming
- It is a functional language with a syntax similar to Ruby
- It is built on top of the Erlang VM, which provides a powerful runtime system for distributed and fault-tolerant applications
- It has a rich ecosystem of libraries and frameworks and a strong community of nearly 100K developers (with the highest median salaries cf. 2024 Stack Overflow survey)
- It is used in production by companies like Discord, Pinterest, PepsiCo, ...
- Since 2021 we have been collaborating with José Valim, the creator of Elixir, and the Dashbit team to develop a type system for Elixir.

Putting the theory to work: Elixir

- Elixir is a dynamic language with a strong emphasis on metaprogramming
- It is a functional language with a syntax similar to Ruby
- It is built on top of the Erlang VM, which provides a powerful runtime system for distributed and fault-tolerant applications
- It has a rich ecosystem of libraries and frameworks and a strong community of nearly 100K developers (with the highest median salaries cf. 2024 Stack Overflow survey)
- It is used in production by companies like Discord, Pinterest, PepsiCo, ...
- Since 2021 we have been collaborating with José Valim, the creator of Elixir, and the Dashbit team to develop a type system for Elixir.
- The static type system is based on the set-theoretic types we have developed in the past years and being gradually integrated into the Elixir compiler since version 1.17 (the current version is 1.19).

Static Typing 101

A function that always errors

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

Static Typing 101

A function that always errors

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

Type error message

Type error:

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

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

Types as contracts between functions.

Types as contracts between functions.

Function A: Negate

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

Types as contracts between functions.

Function A: Negate

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

Types as contracts between functions.

Function A: Negate

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

Function B: Subtract

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

Types as contracts between functions.

Function A: Negate

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

Function B: Subtract

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

Question

- What if we modify the implementation of negate?

Union types: correct, but not precise enough.

Adding a clause for booleans

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

Union types: correct, but not precise enough.

Adding a clause for booleans

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

Union types: correct, but not precise enough.

Adding a clause for booleans

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

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

Union types: correct, but not precise enough.

Adding a clause for booleans

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

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

Type error in subtract

Type error:

```
| def subtract(a, b) when is_integer(a) and is_integer(b) do  
|   a + negate(b)  
|     ^ the operator + expects integer(), integer() as arguments,  
|       but the second argument can be integer() or boolean()
```

Type intersections for functions.

A more precise annotation

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

Type intersections for functions.

A more precise annotation

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

Type intersections for functions.

A more precise annotation

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

The function subtract type checks

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

Singleton and Negation Types

Logical negation

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

Singleton and Negation Types

Logical negation

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

Types

- Singleton types are types containing exactly one value

Singleton and Negation Types

Logical negation

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

Types

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

Polymorphism with Local Type Inference

Expressions and functions can have types containing type variables

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

Polymorphism with Local Type Inference

Expressions and functions can have types containing type variables

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

Type Variables

- Quantified using a postfix **when** with upper bounds

Polymorphism with Local Type Inference

Expressions and functions can have types containing type variables

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

Type Variables

- Quantified using a postfix **when** with upper bounds

System deduces type variable instantiation

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

Intersection and polymorphism

Logical OR

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

Intersection and polymorphism

Logical OR

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

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

Intersection and polymorphism

Logical OR

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

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:

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

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 Simple function types

2.2 Unions, intersections, negations

2.3 Polymorphism

3. Guard Analysis

3.1 Case expressions: redundancy, exhaustiveness, and narrowing

3.2 A Pinch of Formalization

4. Gradual Typing

4.1 Strong function types

4.2 Propagation of `dynamic()`

5. Integration in Elixir

Pattern Matching: Case expressions

Using a case expression

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

Using multiple function clauses

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

Both are equivalent, but the second is preferred in Elixir.

Exhaustiveness and redundancy checking

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

Exhaustiveness and redundancy checking

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

Exhaustiveness checking: handling all cases

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

Exhaustiveness and redundancy checking

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

Exhaustiveness checking: handling all cases

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

Exhaustiveness and redundancy checking

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

Exhaustiveness checking: handling all cases

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

Redundancy checking: detecting unused clauses

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

Exhaustiveness and redundancy checking

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

Exhaustiveness checking: handling all cases

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

Redundancy checking: detecting unused clauses

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

Narrowing: inferring more precise types

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

Narrowing: inferring more precise types

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

Automatically infer a return type

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

Narrowing: inferring more precise types

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

Automatically infer a return type

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

Narrowing: inferring more precise types

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

Automatically infer a return type

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

implies

has type :timeout

Narrowing: inferring more precise types

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

Automatically infer a return type

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

implies

has type :timeout

Narrowing: inferring more precise types

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

Automatically infer a return type

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

has_type {:~~delay~~, integer()}

Narrowing: inferring more precise types

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

Automatically infer a return type

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

Inferring a more precise type

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

Narrowing: inferring more precise types

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

Automatically infer a return type

~~\$ result() =>~~ ← just remove the annotation

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

Inferring a more precise type

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

A Pinch of Formalization

α type variables, c constants, x variables

Base types $b ::= \text{int} \mid \text{atom} \mid \text{function} \mid \text{tuple}$

$$\text{Types } t, s ::= b \mid c \mid \alpha \mid \bar{t} \rightarrow t \mid \{\bar{t}\} \mid t \vee t \mid t \wedge t \mid \neg t$$
$$\begin{aligned} \text{Expressions } e, f &::= c \mid x \mid \lambda(\bar{x}.e) \mid f(\bar{e}) \mid \{\bar{e}\} \mid \text{elem}(e, e) \\ &\mid \text{let } x:t = e \text{ in } e \mid \text{case } e \text{ do } \overline{pg \rightarrow e} \end{aligned}$$

Patterns $p ::= x \mid c \mid \{\bar{p}\}$

```

Guards       $g ::= g \text{ and } g \mid g \text{ or } g \mid \text{not } g \mid \text{is\_integer}(d)$ 
             $\mid \text{is\_atom}(d) \mid \text{is\_tuple}(d) \mid \text{is\_function}(d, d)$ 
             $\mid d == d \mid d != d \mid d < d \mid d \leq d$ 

```

Selectors $d ::= c \mid x \mid \text{elem}(d, d) \mid \text{tuple_size}(d)$

Notation: $\bar{u} = u_1, \dots, u_n$ and we may write p when g for pg

A Pinch of Formalization

α type variables, c constants, x variables

Base types $b ::= \text{int} \mid \text{atom} \mid \text{function} \mid \text{tuple}$

Types $t, s ::= b \mid c \mid \alpha \mid \bar{t} \rightarrow t \mid \{\bar{t}\} \mid t \vee t \mid t \wedge t \mid \neg t$

Expressions $e, f ::= c \mid x \mid \lambda(\bar{x}. e) \mid f(\bar{e}) \mid \{\bar{e}\} \mid \text{elem}(e, e)$
 $\mid \text{let } x : t = e \text{ in } e \mid \text{case } e \text{ do } \overline{pg \rightarrow e}$

Patterns $p ::= x \mid c \mid \{\bar{p}\}$

Guards $g ::= g \text{ and } g \mid g \text{ or } g \mid \text{not } g \mid \text{is_integer}(d)$
 $\mid \text{is_atom}(d) \mid \text{is_tuple}(d) \mid \text{is_function}(d, d)$
 $\mid d == d \mid d != d \mid d < d \mid d <= d$

Selectors $d ::= c \mid x \mid \text{elem}(d, d) \mid \text{tuple_size}(d)$

Notation: $\bar{u} = u_1, \dots, u_n$ and we may write p when g for pg

A Pinch of Formalization

α type variables, c constants, x variables

Base types $b ::= \text{int} \mid \text{atom} \mid \text{function} \mid \text{tuple}$

Types $t, s ::= b \mid c \mid \alpha \mid \bar{t} \rightarrow t \mid \{\bar{t}\} \mid t \vee t \mid t \wedge t \mid \neg t$

Expressions $e, f ::= c \mid x \mid \lambda(\bar{x}. e) \mid f(\bar{e}) \mid \{\bar{e}\} \mid \text{elem}(e, e)$
 $\mid \text{let } x : t = e \text{ in } e \mid \text{case } e \text{ do } \overline{pg \rightarrow e}$

Patterns $p ::= x \mid c \mid \{\bar{p}\}$

Guards $g ::= g \text{ and } g \mid g \text{ or } g \mid \text{not } g \mid \text{is_integer}(d)$
 $\mid \text{is_atom}(d) \mid \text{is_tuple}(d) \mid \text{is_function}(d, d)$
 $\mid d == d \mid d != d \mid d < d \mid d <= d$

Selectors $d ::= c \mid x \mid \text{elem}(d, d) \mid \text{tuple_size}(d)$

Notation: $\bar{u} = u_1, \dots, u_n$ and we may write p when g for pg

Types for patterns and guards

With guards we lose the property that the set of accepted values is a type

Types for patterns and guards

With guards we lose the property that the set of accepted values is a type

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

Surely accepted type: largest type containing only values that match $p \text{ when } g$ (noted $\{pg\}$)

Possibly accepted type: smallest type containing all values that match $p \text{ when } g$ (noted $\langle pg \rangle$)

Types for patterns and guards

With guards we lose the property that the set of accepted values is a type

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

Surely accepted type: largest type containing only values that match $p \text{ when } g$ (noted $\{pg\}$)

Possibly accepted type: smallest type containing all values that match $p \text{ when } g$ (noted $\langle pg \rangle$)

Example

Let $p \text{ when } g$ be $x \text{ when } (\text{is_map}(x) \text{ and } \text{map_size}(x) == 2) \text{ or } \text{is_list}(x)$

- $\{pg\} = \text{map}() \text{ or } \text{list}()$ (it can accept only maps or lists)
- $\langle pg \rangle = \text{list}()$ (it will surely accept all the lists)

Types for patterns and guards

With guards we lose the property that the set of accepted values is a type

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

Surely accepted type: largest type containing only values that match $p \text{ when } g$ (noted $\{pg\}$)

Possibly accepted type: smallest type containing all values that match $p \text{ when } g$ (noted $\langle pg \rangle$)

Example

Let $p \text{ when } g$ be $x \text{ when } (\text{is_map}(x) \text{ and } \text{map_size}(x) == 2) \text{ or } \text{is_list}(x)$

- $\{pg\} = \text{map}() \text{ or } \text{list}()$ (it can accept **only** maps or lists)
- $\langle pg \rangle = \text{list}()$ (it will **surely** accept all the lists)

Types for patterns and guards

With guards we lose the property that the set of accepted values is a type

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

Surely accepted type: largest type containing only values that match $p \text{ when } g$ (noted $\{pg\}$)

Possibly accepted type: smallest type containing all values that match $p \text{ when } g$ (noted $\langle pg \rangle$)

Example

Let $p \text{ when } g$ be $x \text{ when } (\text{is_map}(x) \text{ and } \text{map_size}(x) == 2) \text{ or } \text{is_list}(x)$

- $\{pg\} = \text{map}() \text{ or } \text{list}()$ (it can accept only maps or lists)
- $\langle pg \rangle = \text{list}()$ (it will surely accept all the lists)

Properties

- $\vdash v : \{pg\} \Rightarrow v \text{ matches } pg$
- $\vdash v : \langle pg \rangle \Leftarrow v \text{ matches } pg$

Types for patterns and guards

With guards we lose the property that the set of accepted values is a type

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

Surely accepted type: largest type containing only values that match $p \text{ when } g$ (noted $\{pg\}$)

Possibly accepted type: smallest type containing all values that match $p \text{ when } g$ (noted $\langle pg \rangle$)

Example

Let $p \text{ when } g$ be $x \text{ when } (\text{is_map}(x) \text{ and } \text{map_size}(x) == 2) \text{ or } \text{is_list}(x)$

- $\{pg\} = \text{map}() \text{ or } \text{list}()$ (it can accept only maps or lists)
- $\langle pg \rangle = \text{list}()$ (it will surely accept all the lists)

Properties

- $\vdash v : \{pg\} \Rightarrow v \text{ matches } pg \text{ under}$
 - $\vdash v : \langle pg \rangle \Leftarrow v \text{ matches } pg \text{ over}$
- } - approximate the set $\{v \mid v \text{ matches } pg\}$

Typing pattern matching

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

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

Typing pattern matching

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

Consider `case e do p1g1 → e1 ; ....; pngn → en` with $e : t_0$

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

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

Typing pattern matching

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

Consider `case e do p1g1 → e1 ; ... ; pngn → en` with $e : t_0$

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

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

In words: the expression e_i will process values:

Typing pattern matching

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

Consider `case e do p1g1 → e1 ; ... ; pngn → en` with $e : t_0$

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

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

In words: the expression e_i will process values:

- generated by e (i.e., in t_0)

Typing pattern matching

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

Consider `case e do p1g1 → e1 ; ....; pngn → en` with $e : t_0$

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

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

In words: the expression e_i will process values:

- generated by e (i.e., in t_0)
- **minus** those surely captured by the previous branches (i.e., in $\bigvee_{j < i} \{p_j g_j\}$)

Typing pattern matching

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

Consider `case e do p1g1 → e1 ; ....; pngn → en` with $e : t_0$

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

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

In words: the expression e_i will process values:

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

Typing pattern matching

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

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

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

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

In words: the expression e_i will process values:

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

$$\text{(case)} \quad \frac{\Gamma \vdash e : t_0 \quad (\forall i < n \quad \Gamma, (t_i/p_i) \vdash e_i : s_i)}{\Gamma \vdash \text{case } e \text{ do } \overline{p_i g_i} \rightarrow \overline{e_{i < n}} : \bigvee_{\{i \mid t_i \neq \emptyset\}} s_i} \quad t_0 \leq \bigvee_{i \leq n} \{p_i g_i\}$$

Typing pattern matching

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

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

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

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

In words: the expression e_i will process values:

derive the types of the variables in p_i from t_i

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

$$\text{(case)} \quad \frac{\Gamma \vdash e : t_0 \quad (\forall i < n \quad \Gamma, (t_i/p_i) \vdash e_i : s_i)}{\Gamma \vdash \text{case } e \text{ do } \overline{p_i g_i} \rightarrow e_{i < n} : \bigvee_{\{i | t_i \neq \emptyset\}} s_i} \quad t_0 \leq \bigvee_{i \leq n} \{p_i g_i\}$$

Typing pattern matching

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

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

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

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

In words: the expression e_i will process values:

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

derive the types of the variables in p_i from t_i

exhaustiveness condition

$$\text{(case)} \quad \frac{\Gamma \vdash e : t_0 \quad (\forall i < n \quad \Gamma, (t_i/p_i) \vdash e_i : s_i)}{\Gamma \vdash \text{case } e \text{ do } \overline{p_i g_i} \rightarrow e_{i < n} : \bigvee_{\{i | t_i \neq \emptyset\}} s_i}$$

$$t_0 \leq \bigvee_{i \leq n} \{p_i g_i\}$$

Typing pattern matching

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

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

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

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

In words: the expression e_i will process values:

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

use only the branches that may be selected (redundancy)

derive the types of the variables in p_i from t_i

exhaustiveness condition

$$\text{(case)} \quad \frac{\Gamma \vdash e : t_0 \quad (\forall i < n \quad \Gamma, (t_i/p_i) \vdash e_i : s_i)}{\Gamma \vdash \text{case } e \text{ do } \overline{p_i g_i} \rightarrow e_{i < n} : \bigvee_{\{i | t_i \neq \emptyset\}} s_i}$$
$$t_0 \leq \bigvee_{i \leq n} \{p_i g_i\}$$

Typing pattern matching

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

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

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

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

In words: the expression e_i will process values:

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

$$\text{(case)} \quad \frac{\Gamma \vdash e : t_0 \quad (\forall i < n \quad \Gamma, (t_i/p_i) \vdash e_i : s_i)}{\Gamma \vdash \text{case } e \text{ do } \overline{p_i g_i} \rightarrow \overline{e_{i < n}} : \bigvee_{\{i | t_i \neq \emptyset\}} s_i} \quad t_0 \leq \bigvee_{i \leq n} \{p_i g_i\}$$

Typing pattern matching

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

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

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

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

In words: the expression e_i will process values:

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

$$\text{(case)} \quad \frac{\Gamma \vdash e : t_0 \quad (\forall i < n \quad \Gamma, (t_{ij}/p_i) \vdash e_i : s_{ij})}{\Gamma \vdash \text{case } e \text{ do } \overline{p_i g_i} \rightarrow e_{ii < n} : \bigvee_{\{i | t_i \neq \emptyset\}} s_{ij}} \quad t_0 \leq \bigvee_{i \leq n} \{p_i g_i\} \quad (t_i = \bigvee_j t_{ij})$$

Precise Guard Analysis

For each pattern guard we deduce a list of flagged types (by decomposing the guard in OR-clauses):

$$\Gamma \vdash p_i g_i \rightsquigarrow [(s_{ij}, b_{ij})]_{j \leq m_i}$$

Precise Guard Analysis

For each pattern guard we deduce a list of flagged types (by decomposing the guard in OR-clauses):

$$\Gamma \vdash p_i g_i \rightsquigarrow [(s_{ij}, b_{ij})]_{j \leq m_i}$$

accepted type $\uparrow$
for a OR-clause

Precise Guard Analysis

For each pattern guard we deduce a list of flagged types (by decomposing the guard in OR-clauses):

$$\Gamma \vdash p_i g_i \rightsquigarrow [(s_{ij} \mid b_{ij})]_{j \leq m_i}$$

accepted type $\uparrow$ $\uparrow$ boolean indicating whether the type is exact

Precise Guard Analysis

For each pattern guard we deduce a list of flagged types (by decomposing the guard in OR-clauses):

$$\Gamma \vdash p_i g_i \rightsquigarrow [(s_{ij}, b_{ij})]_{j \leq m_i}$$

Guard Analysis

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

Precise Guard Analysis

For each pattern guard we deduce a list of flagged types (by decomposing the guard in OR-clauses):

$$\Gamma \vdash p_i g_i \rightsquigarrow [(s_{ij}, b_{ij})]_{j \leq m_i}$$

Guard Analysis

Consider again `pg = x when (is_map(x) and map_size(x) == 2) or is_list(x)`
 $\Gamma, \text{term}() \vdash [pg] \rightsquigarrow [(\text{map}(), \text{false}), (\text{list}(), \text{true})]$

Precise Guard Analysis

For each pattern guard we deduce a list of flagged types (by decomposing the guard in OR-clauses):

$$\Gamma \vdash p_i g_i \rightsquigarrow [(s_{ij}, b_{ij})]_{j \leq m_i}$$

Guard Analysis

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

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

$\{pg\} = \text{list}()$ (all types in the list with true flag)

$\{pg\} = \text{map()} \text{ or } \text{list}()$ (all types in the list with any flag)

Precise Guard Analysis

For each pattern guard we deduce a list of flagged types (by decomposing the guard in OR-clauses):

$$\Gamma \vdash p_i g_i \rightsquigarrow [(s_{ij}, b_{ij})]_{j \leq m_i}$$

Guard Analysis

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

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

$\{pg\} = \text{list}()$ (all types in the list with true flag)

$\{pg\} = \text{map}() \text{ or } \text{list}()$ (all types in the list with any flag)

Fine-grained analysis needed to infer the type of

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

Outline

1. Set-Theoretic Types and Dynamic Languages

1.1 Branching, overloading, and pattern matching

1.2 Semantic subtyping

2. Elixir basic typing

2.1 Simple function types

2.2 Unions, intersections, negations

2.3 Polymorphism

3. Guard Analysis

3.1 Case expressions: redundancy, exhaustiveness, and narrowing

3.2 A Pinch of Formalization

4. Gradual Typing

4.1 Strong function types

4.2 Propagation of `dynamic()`

5. Integration in Elixir

Gradual typing

Principles

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

The `dynamic()` type

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

Gradual typing

Principles

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

The `dynamic()` type

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

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

Gradual Typing Conundrum

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

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

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()
```

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()
```

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 types guarantee correct output or runtime type-check failure

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 types guarantee correct output or runtime type-check failure

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)

```
$ integer() -> integer()  
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)

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

Strong function types

(Weak) function type

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

PROPAGATION OF *dynamic()* !

x_dyn 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)

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

Propagation of `dynamic()`

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

```
$ (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:

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

Propagation of dynamic()

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

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

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

Propagation of dynamic()

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

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

```
def subtract(a, b) do                                     //if b is given an argument of type dynamic()  
  a + negate(b)
```

Propagation of dynamic()

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

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

```
def subtract(a, b) do                                     //if b is given an argument of type dynamic()  
  a + negate(b)  
  
#=> Type Error: + undefined for booleans
```

Propagation of `dynamic()`

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

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

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

Propagation of dynamic()

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

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

```
def subtract(a, b) do                                     //if b is given an argument of type dynamic()  
  a + negate(b)
```

Propagation of dynamic()

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

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

```
def subtract(a, b) do                                     //if b is given an argument of type dynamic()  
  a + negate(b)  
  
#=> Result type: integer()
```

Propagation of dynamic()

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

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

```
def subtract(a, b) do                                     //if b is given an argument of type dynamic()  
  a + negate(b)  
#=> Result type: integer() and dynamic()
```

Propagation of dynamic()

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

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

```
def subtract(a, b) do                                     //if b is given an argument of type dynamic()  
  a + negate(b)  
  
#=> Result type: integer() and dynamic()
```

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

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 Simple function types
- 2.2 Unions, intersections, negations
- 2.3 Polymorphism

3. Guard Analysis

- 3.1 Case expressions: redundancy, exhaustiveness, and narrowing
- 3.2 A Pinch of Formalization

4. Gradual Typing

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

5. Integration in Elixir

Status of the Integration

- Integration started in version 1.17 of Elixir, with type structures and gradual typing
- version 1.18 covered more types and type-checked function calls performing type inference for patterns and return types by refined guard analysis
- The current v1.19 covers all language constructs and includes basic, atom, tuple, list, map, and function types, as well as the typing of *protocols* (akin to type classes in Haskell). The forthcoming 1.20 (rc4) covers all the inference for functions. No type annotations and no parametric polymorphism, yet.

Status of the Integration

- Integration started in version 1.17 of Elixir, with type structures and gradual typing
- version 1.18 covered more types and type-checked function calls performing type inference for patterns and return types by refined guard analysis
- The current v1.19 covers all language constructs and includes basic, atom, tuple, list, map, and function types, as well as the typing of *protocols* (akin to type classes in Haskell). The forthcoming 1.20 (rc4) covers all the inference for functions. **No type annotations and no parametric polymorphism, yet.**

Codebase	LoC	Files	Modules	Type Checking Time	Total Compilation Time	%
Remote	>1,000,000	>10,000	18,059	11.116 s	707.598 s	1.6
Livebook	61093	254	299	0.177 s	4.112 s	4.3
Credo	29181	252	264	0.059 s	1.305 s	4.5
Phoenix	21389	71	88	0.049 s	0.525 s	9.3
Hex	15632	196	241	0.091 s	1.339 s	6.8

Status of the Integration

- Integration started in version 1.17 of Elixir, with type structures and gradual typing
- version 1.18 covered more types and type-checked function calls performing type inference for patterns and return types by refined guard analysis
- The current v1.19 covers all language constructs and includes basic, atom, tuple, list, map, and function types, as well as the typing of *protocols* (akin to type classes in Haskell). The forthcoming 1.20 (rc4) covers all the inference for functions. **No type annotations and no parametric polymorphism, yet.**

Codebase	LoC	Files	Modules	Type Checking Time	Total Compilation Time	%
Remote	>1,000,000	>10,000	18,059	11.116 s	707.598 s	1.6
Livebook	61093	254	299	0.177 s	4.112 s	4.3
Credo	29181	252	264	0.059 s	1.305 s	4.5
Phoenix	21389	71	88	0.049 s	0.525 s	9.3
Hex	15632	196	241	0.091 s	1.339 s	6.8

Discovered type errors in all of them

Status of the Integration

- Very positive feedback from developers on versions 1.17 and 1.18
- No syntactic changes required to Elixir code was particularly appreciated
- Uncovered previously hidden errors without significant overhead
- Found issues in major projects: Phoenix, Livebook, Postgrex, and Flame that persisted undetected in production, sometimes for years

Status of the Integration

- Very positive feedback from developers on versions 1.17 and 1.18
- No syntactic changes required to Elixir code was particularly appreciated
- Uncovered previously hidden errors without significant overhead
- Found issues in major projects: Phoenix, Livebook, Postgrex, and Flame that persisted undetected in production, sometimes for years

Real-world Bug Discoveries

- **Phoenix:** Critical type mismatch in `String.to_atom/1` (deployed on 14,600+ websites)
- **Postgrex:** 15 lines of dead code removed
- **Flame:** 2 dead code paths found via guard analysis
- **Live-view:** 6 lines of dead code removed
- **Remote:** detected a critical type mismatch application undetected by tests because in an error branch that is rarely executed.

Other Languages using (more or less) Semantic Subtyping:

- CDuce (and currently its spin offs: SSTT and MLsem)
- Ballerina (by WSO2) uses semantic subtyping since its Swan Lake release
- Luau (by Roblox) uses semantic subtyping when its syntactic subtyping fails.
- R (Jan Vitek's group has been developing a type system for R based on semantic subtyping)
- Julia developers affirm to have drawn inspiration from Semantic Subtyping for the type system.
- Python (ty, the high-performance Python type checker developed by astral, uses semantic subtyping and several low-level data-structures and optimizations developed for Elixir to implement semantic types)
- Ethylizer (uses semantic subtyping type reconstruction to type-check unannotated Erlang code)
- eqWAlizer (use semantic subtyping to type WhatsApp Erlang code)

Future work

[In order of state of completion]

Types for the Elixir Module System:

Currently, only the type `module()` is supported, while we want a finer-grained typing of functions manipulating modules:

```
$ (X: Stack[el=bool], sx: X.stack, Y: Stack[el=bool], sy: Y.stack) -> Y.stack
def move(X, sx, Y, sy), do: Y.push(sy, X.top(sx))
```

Record types comprehensions:

We want to type standard library functions that manipulate records with first-class keys, such as `Map.intersect/2` and `Map.reject/2`.

`intersect/2` $(\alpha, \beta) \rightarrow \% \{ \theta \in \text{keyof}(\alpha) \wedge \text{keyof}(\beta) \mid \beta[\theta] \}$ when $\alpha, \beta : \text{map}()$

`reject/2` $(\alpha, ((\text{keyof}(\alpha) \rightarrow \text{term}) \wedge (\text{keyof}(\alpha) \wedge \beta) \rightarrow \neg \text{truthy})) \rightarrow$
 $\% \{ \theta \in \text{keyof}(\alpha) \setminus \beta \mid \alpha[\theta] \}$ when $\alpha : \text{map}()$

Types for processes and message passing: Currently, only the type `pid()` is supported. We are working on typing processes with the set of messages they can receive and later, introduce Mailbox types, to ensure liveness properties.

Tooling: We are developing migration tools, IDEs, error messages, ... In particular, we are feeding LLMs with automatic translations of Typespecs into Elixir Types together with the type errors they produce to improve the quality of LLM-generated type annotations and error messages.

Implementation:

- v1.20 completion of inference for functions
- v1.21 first modification of the language: explicit type annotation for structs
- v1.22 `$`-prefixed type annotations for functions
- later: polymorphic types, row polymorphism, polymorphic inference, ...

References

On Elixir Typing:

- [1] G. Castagna, G. Duboc, and J. Valim: *The Design Principles of the Elixir Type System*. The Art, Science, and Engineering of Programming, vol.8, n.2, 2024.
- [2] G. Castagna and G. Duboc: *Guard Analysis and Safe Erasure Gradual Typing: a Type System for Elixir*, 7, 2025. Submitted for publication (available on my web page).

On Set-Theoretic types:

- [3] G. Castagna: *Programming with Union, Intersection, and Negation Types*. In *The French School of Programming*, Springer, pag. 309—378, March, 2024. ISBN 978-3-031-34517-3. Preprint at [arXiv:2111.03354](https://arxiv.org/abs/2111.03354).