Modularity in programming languages, the example of OCaml

A powerful module system over a strongly typed functional language

Clément Blaudeau, Cambium team
November 28, 2022
1. Languages and language research
Overview

1. Languages and language research

2. Typing
Overview

1. Languages and language research

2. Typing

3. Typing and modularity
1. Languages and language research

2. Typing

3. Typing and modularity

4. OCAML modules
Languages and language research
A diversity of languages

<table>
<thead>
<tr>
<th>Python</th>
<th>JS</th>
<th>F</th>
<th>Kotlin</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>TS</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td>OCaml</td>
<td>C</td>
<td>Perl</td>
</tr>
<tr>
<td>PHP</td>
<td>Scala</td>
<td>Swift</td>
<td>elixir</td>
</tr>
<tr>
<td>Julia</td>
<td>R</td>
<td>GO</td>
<td></td>
</tr>
</tbody>
</table>
Program properties

Oups, integer overflow error

Hello?

Wait I'm updating Windows

Oups, stack overflow

Should we steer right?

Yes, by 2 × 10^15

Hard

• Absence of error
• Termination
• Time execution bounds
• Space (memory) bounds
• Concurrency
• Logical specification
...
Program properties

Oups, integer overflow error

Should we steer left?

Hard
• Absence of error
• Termination
• Time execution bounds
• Space (memory) bounds
• Concurrency
• Logical specification
...
Program properties

- Absence of error
- Absence of error
- Absence of error
- Termination
Absence of error
Termination
Program properties

- Absence of error
- Termination
Program properties

- Absence of error
- Termination
- Time execution bounds

Wait I'm updating Windows

Should we steer left?
Program properties

- Absence of error
- Termination
- Time execution bounds

Oups, stack overflow
Should we steer left?

4(9)/14
Program properties

- Absence of error
- Termination
- Time execution bounds
- Space (memory) bounds
Program properties

- Absence of error
- Termination
- Time execution bounds
- Space (memory) bounds
Program properties

- Absence of error
- Termination
- Time execution bounds
- Space (memory) bounds
**Program properties**

- Absence of error
- Termination
- Time execution bounds
- Space (memory) bounds
- Concurrency

Should we steer right? Yes

Should we steer left? Yes

4(13)/14
Program properties

- Absence of error
- Termination
- Time execution bounds
- Space (memory) bounds
- Concurrency

Should we steer left?

Yes, by $2.3 \times 10^{15}$
Program properties

- Absence of error
- Termination
- Time execution bounds
- Space (memory) bounds
- Concurrency
- Logical specification
Program properties

- Absence of error
- Termination
- Time execution bounds
- Space (memory) bounds
- Concurrency
- Logical specification
- ...

...
Program properties

- Absence of error
- Termination
- Time execution bounds
- Space (memory) bounds
- Concurrency
- Logical specification
- ...

4(17)/14
Program properties

- Absence of error
- Termination
- Time execution bounds
- Space (memory) bounds
- Concurrency
- Logical specification
- ...

Hard
Program properties vs Language guarantees

- Absence of error
- Termination
- Time execution bounds
- Space (memory) bounds
- Concurrency
- Logical specification
- ...

Hard
What is a language?

What you say
What is a language?

What you say

What you mean

The moon eats the cow

Is it meaningful?
What is a language?

What you say

What you mean

Syntax

✓ The moon eats the cow

✓ Is it meaningful?

Typing System

Typechecker

5/3/14
What is a language?

- What you say: Syntax
- What you mean: Semantics

Typing System
Typechecker

 Grammar
Parser
Compiler
VM

The moon eats the cow

Is it meaningful?

5(4)/14
What is a language?

What you **say**

Syntax

What you **mean**

Semantics

The moon eats the cow

Is it meaningful?

Typing

Typing System

Typechecker

5 / 14
What is a language?

What you **say**

- Syntax
- Grammar

What you **mean**

- Semantics

The moon eats the cow

Is it meaningful?

Typing

Typing System

Typechecker
What is a language?

What you *say*

- **Syntax**
  - Grammar
  - Parser

What you *mean*

- **Semantics**
What is a language?

What you **say**

- Syntax
  - Grammar
  - Parser

What you **mean**

- Semantics
  - Model

The moon eats the cow

Is it meaningful?
What is a language?

What you *say*

- Syntax
  - Grammar
  - Parser

What you *mean*

- Semantics
  - Model
  - Compiler

The moon eats the cow
Is it meaningful?

Typing
Typing System
Typechecker
What is a language?

What you **say**

- **Syntax**
  - Grammar
  - Parser

What you **mean**

- **Semantics**
  - Model
  - Compiler
  - VM
What is a language?

**What you *say***

- Syntax

**What you *mean***

- Semantics

The moon eats the cow

Is it meaningful?

Typing

Typing System

Typechecker
What is a language?

What you **say**

Syntax

✓

The moon eats the cow

What you **mean**

Semantics

?

Is it meaningful?

Typing

Typing System

Typechecker

5 \langle 12 \rangle / 14
What is a language?

What you say

- Syntax

What you mean

- Semantics

Is it meaningful?
What is a language?

What you **say**
- Syntax

What you **mean**
- Semantics

*Is it meaningful?*
- Typing

The moon eats the cow

✓
What is a language?

What you say

Syntax

What you mean

Semantics

Is it meaningful?

Typing

Typing System

The moon eats the cow

✓
What is a language?

What you *say*

Syntax

What you *mean*

Semantics

*Is it* meaningful?

Typing

Typing System

Typing System

Typechecker

The moon eats the cow

✓
What is a language?

What you say

- Syntax
  - Grammar
  - Parser

What you mean

- Semantics
  - Model
  - Compiler
  - VM

Is it meaningful?

Typing

- Typing System
  - Typechecker
Typing
A descriptive language overlay

\[ x = \frac{1}{2}at^2 + v_0 t + x_0 \]
A descriptive language overlay

\[ x = \frac{1}{2}at^2 + v_0 t + x_0 \]

\[ [x] = [a][T]^2 + [v_0][T] + [x_0] \]
A descriptive language overlay

\[ x = \frac{1}{3}at^2 + v_0 t + x_0 \]

\[ [x] = [a][T]^2 + [v_0][T] + [x_0] \]
A descriptive language overlay

\[ x = \frac{1}{2}at^2 + v_0 t + x_0 \]

\[ [x] = [a][T]^2 + [v_0][T] + [x_0] \]

- Preventing a certain set of errors (typechecking)
A descriptive language overlay

\[ x = \frac{1}{2}at^2 + v_0 t + x_0 \]

\[ [x] = [a][T]^2 + [v_0][T] + [x_0] \]

- Preventing a certain set of errors (typechecking)
- Light description of input/output (API)
A descriptive language overlay

\[ x = \frac{1}{2}at^2 + v_0 t + x_0 \]

\[ [x] = [a][T]^2 + [v_0][T] + [x_0] \]

- Preventing a certain set of errors (typechecking)
- Light description of input/output (API)
- Describe structures (datatypes)
A descriptive language overlay

\[ x = \frac{1}{2} a t^2 + v_0 t + x_0 \]

\[ [x] = [a][T]^2 + [v_0][T] + [x_0] \]

- Preventing a certain set of errors (typechecking)
- Light description of input/output (API)
- Describe structures (datatypes)

\(6\langle 7\rangle/14\)
A descriptive language overlay

\[ x = \frac{1}{2}at^2 + v_0 t + x_0 \]

\[ [x] = [a][T]^2 + [v_0][T] + [x_0] \]

- Preventing a certain set of errors (typechecking)
- Light description of input/output (API)
- Describe structures (datatypes)

(Mark Manasse)
A descriptive language overlay

\[ x = \frac{1}{2}at^2 + v_0 t + x_0 \]

\[ [x] = [a][T]^2 + [v_0][T] + [x_0] \]

- Preventing a certain set of errors (typechecking)
- Light description of input/output (API)
- Describe structures (datatypes)

The fundamental problem addressed by a type theory is to ensure that programs have meaning.

(Mark Manasse)
A descriptive language overlay

\[ x = \frac{1}{2} at^2 + v_0 t + x_0 \]

\[ [x] = [a][T]^2 + [v_0][T] + [x_0] \]

- Preventing a certain set of errors (typechecking)
- Light description of input/output (API)
- Describe structures (datatypes)

The fundamental problem **addressed** by a type theory is to ensure that programs **have meaning**. The fundamental problem **caused** by a type theory is that meaningful programs **may not have meanings** ascribed to them.

(Mark Manasse)
A descriptive language overlay

\[ x = \frac{1}{2}at^2 + v_0 t + x_0 \]
\[ [x] = [a][T]^2 + [v_0][T] + [x_0] \]

- Preventing a certain set of errors (typechecking)
- Light description of input/output (API)
- Describe structures (datatypes)

The fundamental problem **addressed** by a type theory is to ensure that programs **have meaning**. The fundamental problem **caused** by a type theory is that meaningful programs **may not have meanings** ascribed to them. The quest for richer type systems results from this tension.

(Mark Manasse)
Typing expressivity

```
MAIN0001* PROGRAM TO SOLVE THE QUADRATIC EQUATION
MAIN0002    READ 10,A,B,C $
MAIN0003    DISC = B*B-4*A*C $
MAIN0004    IF (DISC) NEGA,ZERO,POSI $
MAIN0005   NEGA R = 0.0 - 0.5 * B/A $
MAIN0006   AI = 0.5 * SQRTF(0.0-DISC)/A $
MAIN0007   PRINT 11,R,AI $
MAIN0008   GO TO FINISH $
MAIN0009   ZERO R = 0.0 - 0.5 * B/A $
MAIN0010   PRINT 21,R $
MAIN0011   GO TO FINISH $
MAIN0012  POSI SD = SQRTF(DISC) $
MAIN0013   R1 = 0.5*(SD-B)/A $
MAIN0014   R2 = 0.5*(0.0-(B+SD))/A $
MAIN0015   PRINT 31,R2,R1 $
MAIN0016 FINISH STOP $
MAIN0017   10 FORMAT( 3F12.5 ) $
MAIN0018   11 FORMAT( 19H TWO COMPLEX ROOTS:, F12.5,14H PLUS OR MINUS,
MAIN0019           F12.5, 2H I ) $
MAIN0020   21 FORMAT( 15H ONE REAL ROOT:, F12.5 ) $
MAIN0021   31 FORMAT( 16H TWO REAL ROOTS:, F12.5, 5H AND , F12.5 ) $
MAIN0022 END $```

7(1)/14
Typing expressivity
Typing expressivity

Fruits: Apples, Pears

Juice:
- Fruit (α)
- Juice (α)

Subtyping: Apples < Juice

Polymorphism: ∀α. Fruit(α) → Juice(α)

Dependent types: ∀α. {x: Fruit(α)} → {y: Juice(α) | y < x}

7(3)/14
Typing expressivity

Fruits

\begin{align*}
\text{Typing expressivity} & \\
\text{Fruits} & \\
\text{Subtyping: Apples} & < \text{Juice} \\
\text{Polymorphism:} & \forall \alpha. \text{Fruit}(\alpha) \rightarrow \text{Juice}(\alpha) \\
\text{Dependent types:} & \forall \alpha. \{ x : \text{Fruit}(\alpha) \} \rightarrow \{ y : \text{Juice}(\alpha) \mid y < x \}
\end{align*}
Typing expressivity

- Subtyping: Apples $<$ Juice
- Polymorphism: $\forall \alpha. \text{Fruit}(\alpha) \rightarrow \text{Juice}(\alpha)$
- Dependent types: $\forall \alpha. \{x : \text{Fruit}(\alpha)\} \rightarrow \{y : \text{Juice}(\alpha) \mid y < x\}$
Typing expressivity

Fruits \rightarrow \text{Juice}

- Subtyping: Apples \subset \text{Juice}
- Polymorphism: \forall \alpha. \text{Fruit}(\alpha) \rightarrow \text{Juice}(\alpha)
- Dependent types: \forall \alpha. \{ x : \text{Fruit}(\alpha) \} \rightarrow \{ y : \text{Juice}(\alpha) | y < x \}
Typing expressivity

Fruits (Apples) → Juice

Subtyping: Apples < Juice

Polymorphism: ∀α. Fruit(α) → Juice(α)

Dependent types: ∀α. {x: Fruit(α)} → {y: Juice(α) | y < x}
Typing expressivity

- **Subtyping** : Apples \(\subset\) Juice

\[\{x : \text{Fruit}(\alpha)\} \rightarrow \{y : \text{Juice}(\alpha) \mid y < x\}\]
Typing expressivity

- Subtyping: Apples <: Juice
Typing expressivity

- Subtyping: Apples <: Juice

Fruits

Apples

Pears

Juice

Cider
- Subtyping: Apples <: Juice
Typing expressivity

- Subtyping: Apples <: Juice
• Subtyping: Apples <: Juice

Typing expressivity
Typing expressivity

- **Subtyping**: Apples $<$: Juice
- **Polymorphism**: $\forall \alpha. \text{Fruit}(\alpha) \rightarrow \text{Juice}(\alpha)$
Typing expressivity

- **Subtyping**: Apples $\ll$ Juice
- **Polymorphism**: $\forall \alpha. \text{Fruit}(\alpha) \rightarrow \text{Juice}(\alpha)$
Typing expressivity

- **Subtyping**: Apples $\ll$ Juice
- **Polymorphism**: $\forall \alpha. \text{Fruit}(\alpha) \rightarrow \text{Juice}(\alpha)$
Typing expressivity

- **Subtyping**: Apples <: Juice
- **Polymorphism**: $\forall \alpha. \text{Fruit}(\alpha) \rightarrow \text{Juice}(\alpha)$
- **Dependent types**: $\forall \alpha. \{x : \text{Fruit}(\alpha)\} \rightarrow \{y : \text{Juice}(\alpha) | y < x\}$
Typing and modularity
Why modules in complex systems?

- Re-usability
  - Standardize basic operations
  - Factorize coding and maintenance effort
- System structure
  - Separate functionalities
  - Isolate components from each other
  - Interdependence within modules / independence across modules
- Common building blocks
  - Well defined interfaces
  - Abstraction
Why modules in complex systems?

- Re-usability
  - Standardize basic operations
  - Factorize coding and maintenance effort

- System structure
  - Separate functionalities
  - Isolate components from each other
  - Interdependence within modules / independence across modules

- Common building blocks
  - Well defined interfaces
  - Abstraction
Why modules in complex systems?

Re-usability
Why modules in complex systems?

Re-usability

- Standardize basic operations
Why modules in complex systems?

Re-usability

- Standardize basic operations
- Factorize coding and maintenance effort
Why modules in complex systems?

**Re-usability**
- Standardize basic operations
- Factorize coding and maintenance effort
- **Interchangeability**

![Image of screw threads with dimensions]
Why modules in complex systems?

Re-usability
- Standardize basic operations
- Factorize coding and maintenance effort
- Interchangeability
Why modules in complex systems?

Re-usability

- Standardize basic operations
- Factorize coding and maintenance effort
- Interchangeability

System structure

- Separate functionalities
- Isolate components from each other
- Interdependence within modules / independence across modules

Common building blocks

- Well defined interfaces
- Abstraction
Why modules in complex systems?

Re-usability

- Standardize basic operations
- Factorize coding and maintenance effort
- Interchangeability

System structure

- Separate functionalities
Why modules in complex systems?

**Re-usability**
- Standardize basic operations
- Factorize coding and maintenance effort
- Interchangeability

**System structure**
- Separate functionalities
- Isolate components from each-other
Why modules in complex systems?

Re-usability
- Standardize basic operations
- Factorize coding and maintenance effort
- Interchangeability

System structure
- Separate functionalities
- Isolate components from each-other
  - Interdependence within modules / independence across modules
Why modules in complex systems?

**Re-usability**
- Standardize basic operations
- Factorize coding and maintenance effort
- Interchangeability

**System structure**
- Separate functionalities
- Isolate components from each-other
- Interdependence within modules / independence across modules
Why modules in complex systems?

Re-usability
- Standardize basic operations
- Factorize coding and maintenance effort
- Interchangeability

System structure
- Separate functionalities
- Isolate components from each-other
- *Interdependence within* modules / *independence across* modules

Common building blocks
Why modules in complex systems?

Re-usability
- Standardize basic operations
- Factorize coding and maintenance effort
- Interchangeability

System structure
- Separate functionalities
- Isolate components from each-other
- Interdependence within modules / independence across modules

Common building blocks
- Well defined interfaces
Why modules in complex systems?

Re-usability
- Standardize basic operations
- Factorize coding and maintenance effort
- Interchangeability

System structure
- Separate functionalities
- Isolate components from each-other
- Interdependence within modules / independence across modules

Common building blocks
- Well defined interfaces
- Abstraction
Why modules in complex systems?

Re-usability
- Standardize basic operations
- Factorize coding and maintenance effort
- Interchangeability

System structure
- Separate functionalities
- Isolate components from each-other
- *Interdependence within* modules / *independence across* modules

Common building blocks
- Well defined interfaces
- Abstraction
Why modules in complex systems?

Re-usability
- Standardize basic operations
- Factorize coding and maintenance effort
- Interchangeability

System structure
- Separate functionalities
- Isolate components from each-other
- Interdependence within modules / independence across modules

Common building blocks
- Well defined interfaces
- Abstraction
Modularity in PL

Basic modularity

• Functions
• Libraries

Object oriented programming

• Abstract objects
• Public/Private fields
• Inheritance

Type-classes/Traits

• A method-focused approach
Modularity in PL

Basic modularity
Modularity in PL

Basic modularity

- Functions
Basic modularity

- Functions

Too small granularity
Basic modularity

- Functions
- Libraries

Too small granularity
Basic modularity

- Functions
  
- Libraries

Too small granularity

Not inter-changeable
Modularity in PL

Basic modularity

- Functions
  - Too small granularity
- Libraries
  - Not inter-changeable
Modularity in PL

Basic modularity

- Functions
  - Too small granularity

- Libraries
  - Not inter-changeable

Object oriented programming
Modularity in PL

Basic modularity
- Functions
- Libraries

Object oriented programming
- Abstract objects

Too small granularity
Not inter-changeable
Modularity in PL

Basic modularity
- Functions
- Libraries

Object oriented programming
- Abstract objects
- Public / Private fields

Too small granularity
Not inter-changeable
Basic modularity

- Functions
- Libraries

Object oriented programming

- Abstract objects
- Public / Private fields

Too small granularity
Not inter-changeable

Binary, no partial access
Modularity in PL

Basic modularity

- Functions
- Libraries

Object oriented programming

- Abstract objects
- Public / Private fields
- Inheritance

Too small granularity
Not inter-changeable

Binary, no partial access
Modularity in PL

Basic modularity
- Functions
- Libraries

Object oriented programming
- Abstract objects
- Public / Private fields
- Inheritance

Too small granularity
Not inter-changeable

Binary, no partial access
Modularity in PL

Basic modularity
- Functions
  Too small granularity
- Libraries
  Not inter-changeable

Object oriented programming
- Abstract objects
- Public / Private fields
  Binary, no partial access
- Inheritance

Type-classes / Traits
Modularity in PL

Basic modularity
- Functions
- Libraries

Object oriented programming
- Abstract objects
- Public / Private fields
- Inheritance

Type-classes / Traits
- A method-focused approach

Too small granularity
Not inter-changeable

Binary, no partial access
OCaml modules
module type

ComplexNumbers =

sig

  type t

  val make : float -> float -> t

  val add : t -> t -> t

  val modulus : t -> float

  val real_part : t -> float

  val imag_part : t -> float

...
The module language in OCaml

Generic interface

```
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```
The module language in OCaml

Generic interface

```ocaml
code language:
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```
The module language in OCaml

Generic interface

```
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
...
end
```
The module language in OCaml

Generic interface

```
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
end
```
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
end
```
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
end
```
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As a developer
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As a developer

```ocaml
module ComplexCartesian = struct
  type t = {re:float; im:float}
  let make x y = {re=x; im=y}
  let add c1 c2 = ...
  let modulus ({re:x; im:y}) =
    sqrt (x*.x +. y*.y)
  let real_part ({re:x; im:_}) = x
  let imag_part ({re:_; im:y}) = y
  ...
end
```
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As a developer

```ocaml
module ComplexCartesian = struct
  type t = {re:float; im:float}
  let make x y = {re=x; im=y}
  let add c1 c2 = ...
  let modulus ({re:x; im:y}) = sqrt (x*.x +. y*.y)
  let real_part ({re:x; im:_}) = x
  let imag_part ({re:_; im:y}) = y
  ...
end
```
The module language in OCaml

Generic interface

```
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As a developer

```
module ComplexCartesian = struct
  type t = {re:float; im:float}
  let make x y = {re=x; im=y}
end
```
The module language in OCaml

### Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

### As a developer

```ocaml
module ComplexCartesian = struct
  type t = {re:float; im:float}
  let make x y = {re=x; im=y}
  let add c1 c2 = ...
  let modulus ({re:x; im:y}) = sqrt (x*.x +. y*.y)
  let real_part ({re:x; im:_}) = x
  let imag_part ({re:_; im:y}) = y
  ...
end
```
The module language in OCaml

Generic interface

```
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As a developer

```
module ComplexCartesian = struct
  type t = {re:float; im:float}
  let make x y = {re=x; im=y}
  let add c1 c2 = ...
  let modulus ({re:x; im:y}) =
    sqrt (x*.x +. y*.y)
  let real_part ({re:_; im:y}) = y
  let imag_part ({re:x; im:_}) = x
  ...
end
```
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As a developer

```ocaml
module ComplexCartesian = struct
  type t = {re:float; im:float}
  let make x y = {re=x; im=y}
  let add c1 c2 = ...
  let modulus ({re:x; im:y}) = sqrt (x*.x +. y*.y)
  let real_part ({re:x; im:_}) = x
  let imag_part ({re:_; im:y}) = y
  ...
end
```
The module language in OCaml

Generic interface

```
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As a developer

```
module ComplexCartesian = struct
  type t = {re:float; im:float}
  let make x y = {re=x; im=y}
  let add c1 c2 = ...
  let modulus ({re:x; im:y}) =
    sqrt (x*.x +. y*.y)
  let real_part ({re:_; im:y}) = y
  let imag_part ({re:_; im:_}) = x
end
```
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As a developer

```ocaml
module ComplexCartesian = struct
  type t = {re:float; im:float}
  let make x y = {re=x; im=y}
  let add c1 c2 = ...
  let modulus ({re:x; im:y}) = sqrt (x*.x +. y*.y)
  let real_part ({re:x; im:_}) = x
  let imag_part ({re:_; im:y}) = y
  ...
end
```
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As a developer

```ocaml
module ComplexCartesian = struct
  type t = {re:float; im:float}
  let make x y = {re=x; im=y}
  let add c1 c2 = ...
  let modulus ({re:x; im:y}) =
    sqrt (x*.x +. y*.y)
  let real_part ({re:x; im:_}) = x
  let imag_part ({re:_; im:y}) = y
  ...
end
```
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As a developer

```ocaml
module ComplexCartesian = (struct
  type t = {re:float; im:float}
  let make x y = {re=x; im=y}
  let add c1 c2 = ...
  let modulus ({re:x; im:y}) = sqrt (x*.x +. y*.y)
  let real_part ({re:x; im:_}) = x
  let imag_part ({re:_; im:y}) = y
  ...
end : ComplexNumbers)
```
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As another developer

```ocaml
module ComplexPolar = struct
  type t = float * float
  let make x y = ...
  let add c1 c2 = ...
  let modulus (m,a) = a
  let real_part (m,a) = m *. (cos a)
  let imag_part (m,a) = ...
  ...
end
```
The module language in OCaml

Generic interface

```
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As another developer

```
module ComplexPolar = struct
  type t = float * float
  let make x y = ...
  let add c1 c2 = ...
  let modulus (m,a) = a
  let real_part (m,a) = m *. (cos a)
  let imag_part (m,a) = ...
  ...
end
```
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As another developer

```ocaml
module ComplexPolar = struct
  type t = float * float
  let make x y = ...
  let add c1 c2 = ...
  let modulus (m,a) = a
  let real_part (m,a) = m *. (cos a)
  let imag_part (m,a) = ...
  ...
end
```
Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As another developer

```ocaml
module ComplexPolar = struct
  type t = float * float
  let make x y = ...
  let add c1 c2 = ...
  let modulus (m,a) = a
  let real_part (m,a) = m *. (cos a)
  let imag_part (m,a) = ...
  ...
end
```
The module language in OCaml

Generic interface

```
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As another developer

```
module ComplexPolar = struct
  type t = float * float
  let make x y = ...
  let add c1 c2 = ...
  let modulus (m,a) = a
  let real_part (m,a) = m *. (cos a)
  let imag_part (m,a) = ...
  ...
end
```
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As another developer

```ocaml
module ComplexPolar = struct
  type t = float * float
  let make x y = ...
  let add c1 c2 = ...
  let modulus (m,a) = a
  let real_part (m,a) = m *. (cos a)
  let imag_part (m,a) = ...
  ...
end
```
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As another developer

```ocaml
module ComplexPolar = struct
  type t = float * float
  let make x y = ...
  let add c1 c2 = ...
  let modulus (m,a) = a
  let real_part (m,a) = m *. (cos a)
  let imag_part (m,a) = ...
  end
```
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As another developer

```ocaml
module ComplexPolar = struct
  type t = float * float
  let make x y = ...
  let add c1 c2 = ...
  let modulus (m,a) = a
  let real_part (m,a) =
      m *. (cos a)
  let imag_part (m,a) = ...
  ...
end
```
The module language in OCaml

### Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

### As another developer

```ocaml
module ComplexPolar = struct
  type t = float * float
  let make x y = ...
  let add c1 c2 = ...
  let modulus (m,a) = a
  let real_part (m,a) = m *. (cos a)
  let imag_part (m,a) = ...
  ...
end
```
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As a user

```ocaml
module ComplexPolynomials = 
  functor (C: ComplexNumbers) ->
    struct
      type t = C.t list
      let add p1 p2 = ...
      let degree p = ...
      let roots p = ...
      ...
    end

module M = ComplexPolynomials(ComplexPolar)
```
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As a user

```ocaml
module ComplexPolynomials = functor (C: ComplexNumbers) -> struct
  type t = C.t list
  let add p1 p2 = ...
  let degree p = ...
  let roots p = ...
  ...
end

module M = ComplexPolynomials(ComplexPolar)
```
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As a user

```ocaml
module ComplexPolynomials = functor (C: ComplexNumbers) -> struct
  type t = C.t list
  ...
end
```
The module language in OCaml

Generic interface

```
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As a user

```
module ComplexPolynomials = functor (C: ComplexNumbers) -> struct
  type t = C.t list
  let add p1 p2 = ...
  let degree p = ...
  let roots p = ...
  ...
end

module M = ComplexPolynomials(ComplexPolar)
```
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As a user

```ocaml
module ComplexPolynomials = functor (C: ComplexNumbers) -> struct
  type t = C.t list
  let add p1 p2 = ...
  let degree p = ...
  ...
end

module M = ComplexPolynomials(ComplexPolar)
```
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As a user

```ocaml
module ComplexPolynomials = functor (C: ComplexNumbers) -> struct
  type t = C.t list
  let add p1 p2 = ...
  let degree p = ...
  let roots p = ...
end
```

module M = ComplexPolynomials(ComplexPolar)
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As a user

```ocaml
module ComplexPolynomials = functor
  (C: ComplexNumbers) -> struct
    type t = C.t list
    let add p1 p2 = ...
    let degree p = ...
    let roots p = ...
    ...
  end

module M = ComplexPolynomials(ComplexPolar)
```
Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As a user

```ocaml
module ComplexPolynomials = functor (C: ComplexNumbers) -> struct
  type t = C.t list
  let add p1 p2 = ...
  let degree p = ...
  let roots p = ...
  ...
end
module M = ComplexPolynomials(ComplexPolar)
```
The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t
  val modulus : t -> float
  val real_part : t -> float
  val imag_part : t -> float
  ...
end
```

As a user

```ocaml
module ComplexPolynomials = functor (C: ComplexNumbers) -> struct
  type t = C.t list
  let add p1 p2 = ...
  let degree p = ...
  let roots p = ...
  ...
end
module M = ComplexPolynomials(ComplexPolar)
```
The ML\(^1\) approach

\(^1\)Not machine-learning
The ML\(^1\) approach

An interface language (description)

\(^1\)Not machine-learning
The ML\textsuperscript{1} approach

An interface language (description)

- User-defined \textit{signatures}

\textsuperscript{1}Not machine-learning
The ML\textsuperscript{1} approach

An interface language (description)

- User-defined \textit{signatures} 

\texttt{sig ... end}

\textsuperscript{1}Not machine-learning
The ML\textsuperscript{1} approach

An interface language (description)

- User-defined *signatures*
  
  sig ... end

- Abstract types

\textsuperscript{1}Not machine-learning
The ML\textsuperscript{1} approach

An interface language (description)

- User-defined \textit{signatures}\n  \begin{verbatim}
  sig ... end
  \end{verbatim}
- Abstract types
  \begin{verbatim}
  type t
  \end{verbatim}

\textsuperscript{1}Not machine-learning
The ML$^1$ approach

An interface language (description)

- User-defined signatures
  
  ```ml
  sig ... end
  ```

- Abstract types
  
  ```ml
  type t
  ```

$^1$Not machine-learning
The ML$^1$ approach

An interface language (description)
- User-defined signatures
- Abstract types

A module language (construction)

\[ \text{sig ... end} \]
\[ \text{type } t \]

$^1$Not machine-learning
The ML\textsuperscript{1} approach

An interface language (description)
- User-defined signatures
- Abstract types

A module language (construction)
- Separate the module and core languages

\textsuperscript{1}Not machine-learning
The ML\textsuperscript{1} approach

An interface language (description)
- User-defined signatures
  \texttt{sig ... end}
- Abstract types
  \texttt{type t}

A module language (construction)
- Separate the module and core languages
  \texttt{struct ... end}

\textsuperscript{1}Not machine-learning
The ML\textsuperscript{1} approach

An interface language (description)
- User-defined *signatures*
  ```
  sig ... end
  ```
- Abstract types
  ```
  type t
  ```

A module language (construction)
- Separate the module and core languages
  ```
  struct ... end
  ```
- Developer-side abstraction : sealing

\textsuperscript{1}Not machine-learning
The ML\(^1\) approach

**An interface language (description)**

- User-defined *signatures*
  
  ```plaintext
  sig ... end
  ```

- Abstract types
  
  ```plaintext
  type t
  ```

**A module language (construction)**

- Separate the module and core languages
  
  ```plaintext
  struct ... end
  ```

- Developer-side abstraction : sealing
  
  ```plaintext
  (M : S)
  ```

\(^1\) Not machine-learning
The ML\(^1\) approach

**An interface language (description)**
- User-defined *signatures*
  ```ml
  sig ... end
  ```
- Abstract types
  ```ml
  type t
  ```

**A module language (construction)**
- Separate the module and core languages
  ```ml
  struct ... end
  ```
- Developer-side abstraction: sealing
  ```ml
  (M : S)
  ```
- Client-side abstraction: functors

\(^1\)Not machine-learning
The ML$^1$ approach

An interface language (description)
- User-defined *signatures*
  
- Abstract types

A module language (construction)
- Separate the module and core languages
- Developer-side abstraction: sealing
- Client-side abstraction: functors

---

$^1$Not machine-learning
The ML\(^1\) approach

An interface language (description)

- User-defined *signatures*  
  \[
  \text{sig} \ldots \ \text{end}
  \]
- Abstract types  
  \[
  \text{type} \ t
  \]

A module language (construction)

- Separate the module and core languages  
  \[
  \text{struct} \ldots \ \text{end}
  \]
- Developer-side abstraction: sealing  
  \[
  (M : S)
  \]
- Client-side abstraction: functors  
  \[
  \text{functor} \ (X:S) \rightarrow M
  \]
- A real module calculus (functions, calls, projections, ...)

\(^1\)Not machine-learning
Reachable and *describable* spaces mismatch

![Diagram showing the mismatch between reachable and describable spaces](image)

- **Reachable space**

**Common use-cases**
- Signature avoidance
- Incorrect avoidance
- Over abstraction
Reachable and *describable* spaces mismatch

- **Reachable space**
- **Describable space**

Common use-cases
Reachable and *describable* spaces mismatch

- **Reachable space**
- **Describable space**
- **Signature avoidance**
Reachable and describable spaces mismatch

- Reachable space
- Describable space
- Signature avoidance
- Common use-cases
Reachable and describable spaces mismatch

- **Reachable space**
- **Describable space**
- **Signature avoidance**
- **Common use-cases**
- **Loss of type-sharing by current OCaml**
Reachable and describable spaces mismatch

- Reachable space
- Describable space
- Signature avoidance
- Common use-cases
- Incorrect avoidance
Reachable and describable spaces mismatch

- Reachable space
- Describable space
- Signature avoidance
- Common use-cases
- Incorrect avoidance
- Over abstraction
Reachable and *describable* spaces mismatch

1. Signature avoidance
2. Common use-cases
3. Incorrect avoidance
4. Over abstraction
A approach by *elaboration*
A approach by *elaboration*

- OCaml
- Hard to formalize
- \( \mathbb{F} \omega \)
- Clear formalization
- Standard techniques
- Expressivity
A approach by elaboration

OCaml

Hard to formalize

Ad-hoc techniques
A approach by *elaboration*

- OCaml
- Hard to formalize
- Ad-hoc techniques
- Signature avoidance
A approach by *elaboration*

- Hard to formalize
- Ad-hoc techniques
- Signature avoidance
A approach by *elaboration*

\[
\text{OCAML} \quad \text{F}^{\omega}
\]
A approach by *elaboration*

- **OCaml**
- **Fω**

Clear formalization
A approach by *elaboration*

- OCaml
- Clear formalization
- Standard techniques

\[ F^\omega \]
A approach by *elaboration*

OCaml

Clear formalization
Standard techniques
Expressivity

\( F^\omega \)
A approach by *elaboration*

- OCaml
- Clear formalization
- Standard techniques
- Expressivity

$$\text{F}^\omega$$
A approach by *elaboration*

\[
\text{OCaml} \rightarrow \mathcal{F}^\omega
\]
A approach by *elaboration*
Conclusions

- Programming languages are continuously evolving
- Type systems help write safer and better code
- Modularity is crucial when building complex systems
- OCaml modules are a powerful approach

Questions?
Programming languages are continuously evolving

Type systems help write safer and better code
Conclusion

- Programming languages are continuously evolving
- Type systems help write safer and better code
- Modularity is crucial when building complex systems
Conclusion

- Programming languages are continuously evolving
- Type systems help write safer and better code
- Modularity is crucial when building complex systems
- OCaml modules are a powerful approach
Conclusion

- Programming languages are continuously evolving
- Type systems help write safer and better code
- Modularity is crucial when building complex systems
- OCaml modules are a powerful approach
- We work to understand it, fix it and improve it
Conclusion

- Programming languages are continuously evolving
- Type systems help write safer and better code
- Modularity is crucial when building complex systems
- OCaml modules are a powerful approach
- We work to understand it, fix it and improve it

Questions?