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

Overview

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

Overview

1. Languages and language research
2. Typing
3. Typing and modularity
4. OCaml modules

Languages and language research

A diversity of languages

Program properties

Program properties

- Absence of error

Program properties

- Absence of error

Program properties

- Absence of error
- Termination

Program properties

- Absence of error
- Termination

Program properties

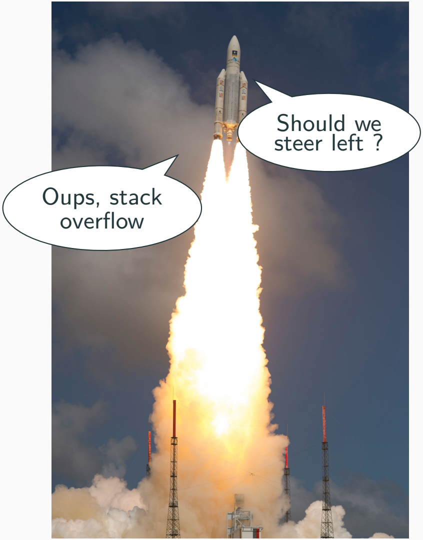
- Absence of error
- Termination

Program properties

- Absence of error
- Termination
- Time execution bounds

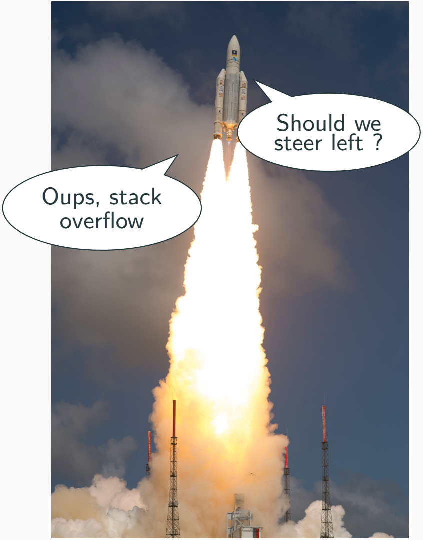
Program properties

- Absence of error
- Termination
- Time execution bounds



Program properties

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

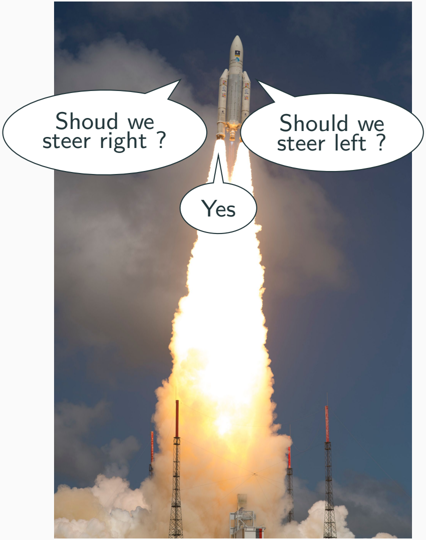


Program properties

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

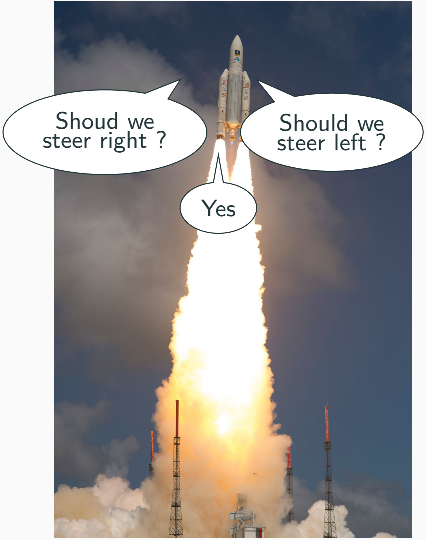
Program properties

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



Program properties

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



Program properties

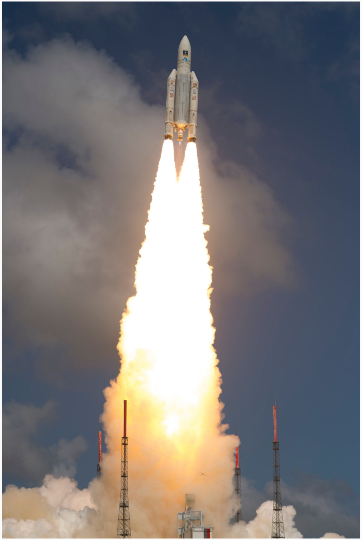
- Absence of error
- Termination
- Time execution bounds
- Space (memory) bounds
- Concurrency

Program properties

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

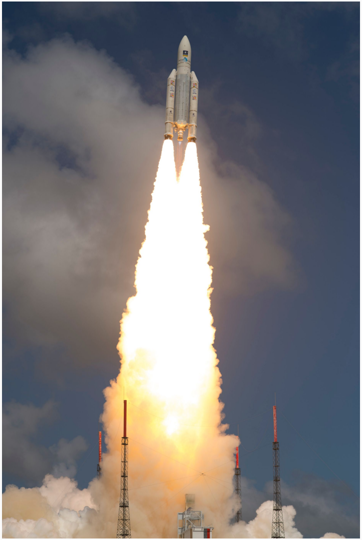
Program properties

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



Program properties

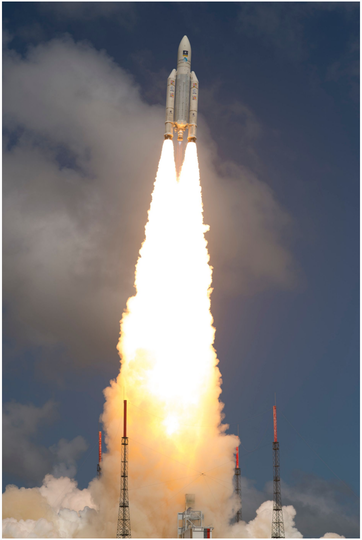
- Absence of error
- Termination
- Time execution bounds
- Space (memory) bounds
- Concurrency
- Logical specification
- ...



Program properties

Hard

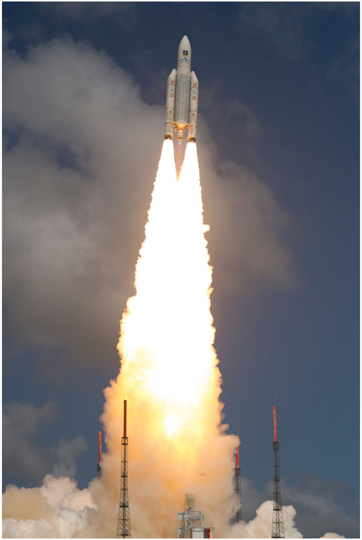
- Absence of error
- Termination
- Time execution bounds
- Space (memory) bounds
- Concurrency
- Logical specification
- ...



Program properties vs Language guarantees

Hard

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



What is a language ?

What you ***say***

What is a language ?

*What you **say***

*What you **mean***

What is a language ?

What you ***say***

What you ***mean***

Syntax

What is a language ?

*What you **say***

$\boxed{\textbf{Syntax}}$

*What you **mean***

$\boxed{\textbf{Semantics}}$

What is a language ?

*What you **say***

$\boxed{\textbf{Syntax}}$

*What you **mean***

$\boxed{\textbf{Semantics}}$

What is a language ?

*What you **say***

Syntax

Grammar

*What you **mean***

Semantics

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

What is a language ?

*What you **say***

Syntax

Grammar

Parser

*What you **mean***

Semantics

Model

Compiler

What is a language ?

*What you **say***

Syntax

Grammar

Parser

*What you **mean***

Semantics

Model

Compiler

VM

What is a language ?

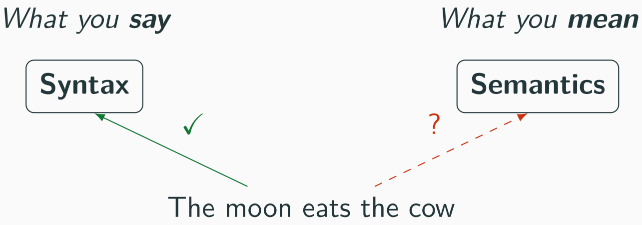
*What you **say***

$\boxed{\textbf{Syntax}}$

*What you **mean***

$\boxed{\textbf{Semantics}}$

What is a language ?



What is a language ?

*What you **say***

$\boxed{\textbf{Syntax}}$

*What you **mean***

$\boxed{\textbf{Semantics}}$

*Is it **meaningful**?*

What is a language ?

*What you **say***

$\boxed{\textbf{Syntax}}$

*What you **mean***

$\boxed{\textbf{Semantics}}$

*Is it **meaningful**?*

$\boxed{\textbf{Typing}}$

What is a language ?

*What you **say***

Syntax

*What you **mean***

Semantics

*Is it **meaningful**?*

Typing

Typing System

What is a language ?

*What you **say***

Syntax

*What you **mean***

Semantics

*Is it **meaningful**?*

Typing

Typing System

Typechecker

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

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

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

A descriptive language overlay

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

- 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 $
```

Typing expressivity

Typing expressivity

Typing expressivity

$\boxed{\text{Fruits}} \longrightarrow$

Typing expressivity

$\boxed{\text{Fruits}} \longrightarrow$  $\boxed{\text{Juice}}$

Typing expressivity

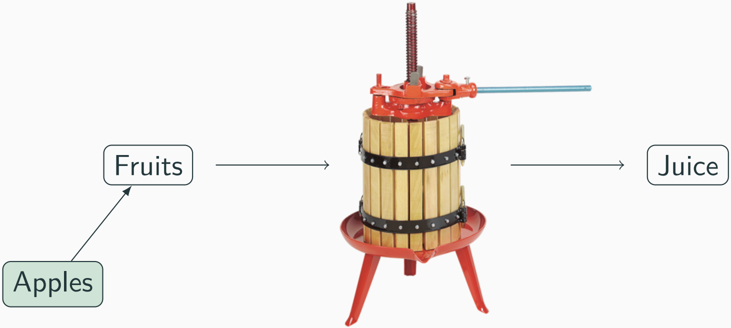
Fruits $\longrightarrow$  $\longrightarrow$ Juice

Typing expressivity

Fruits ⟶  ⟶ Juice

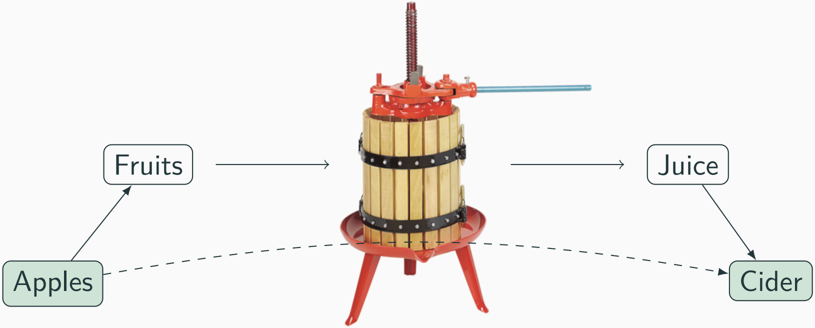
Apples ⟶ Fruits

Typing expressivity



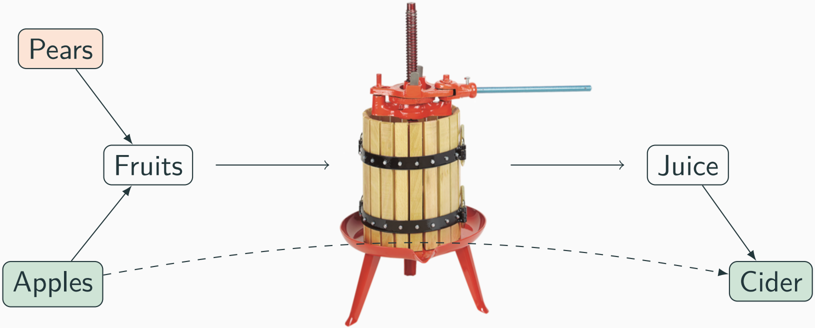
- Subtyping : Apples $<:$ Juice

Typing expressivity



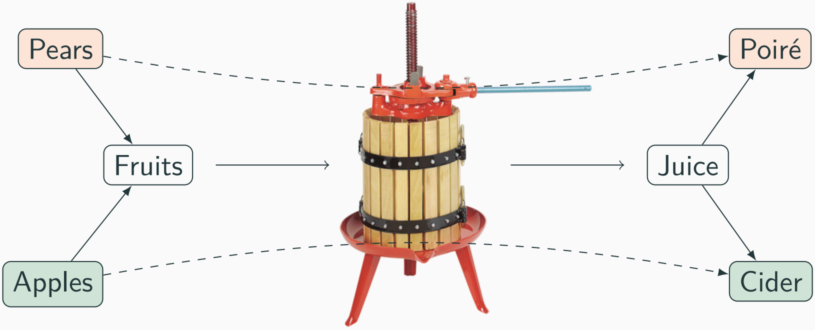
- Subtyping : Apples $<:$ Juice

Typing expressivity



- Subtyping : Apples <: Juice

Typing expressivity



- Subtyping : Apples <: Juice

Typing expressivity

Fruit (α) $\longrightarrow$  $\longrightarrow$ Juice

- Subtyping : Apples $<:$ Juice

Typing expressivity

$\boxed{\text{Fruit } (\alpha)} \longrightarrow$

$\longrightarrow \boxed{\text{Juice } (\alpha)}$

- Subtyping : Apples <: Juice

Typing expressivity

Fruit (α) $\longrightarrow$  $\longrightarrow$ Juice (α)

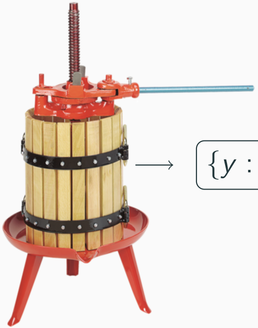
- Subtyping : Apples $<:$ Juice
- Polymorphism : $\forall \alpha. \mathrm{Fruit}(\alpha) \rightarrow \mathrm{Juice}(\alpha)$

Typing expressivity

$\boxed{\{x : \mathsf{Fruit}(\alpha)\}}$ $\longrightarrow$  $\longrightarrow$

- Subtyping : Apples $<:$ Juice
- Polymorphism : $\forall \alpha.\, \mathsf{Fruit}(\alpha) \rightarrow \mathsf{Juice}(\alpha)$

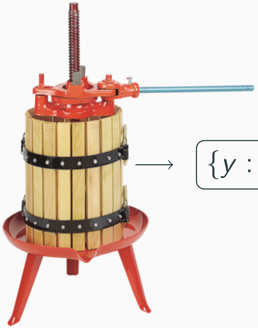
Typing expressivity

$\boxed{\{x : \text{Fruit}(\alpha)\}}$ $\longrightarrow$  $\longrightarrow$ $\boxed{\{y : \text{Juice of}\,(\alpha) \mid y < x\}}$

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

Typing expressivity

$\boxed{\{x : \mathrm{Fruit}(\alpha)\}} \longrightarrow$

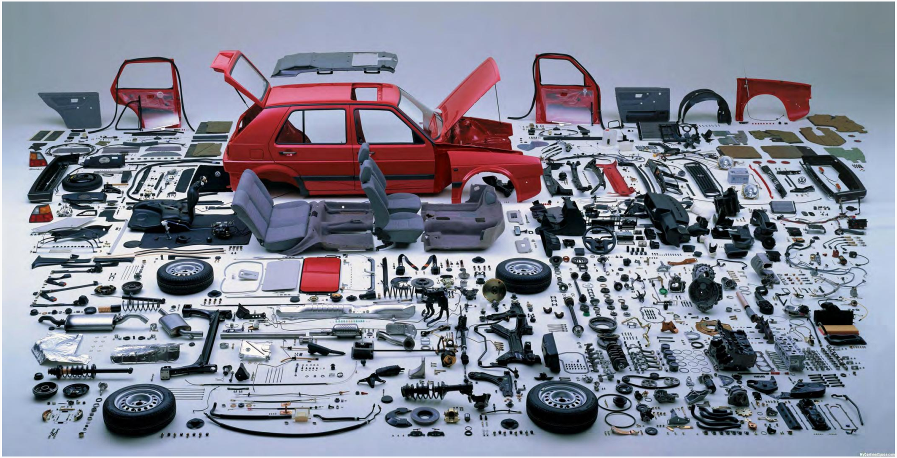


$\longrightarrow \boxed{\{y : \mathrm{Juice\ of}\,(\alpha) \mid y < x\}}$

- Subtyping : Apples $<:$ Juice
- Polymorphism : $\forall\alpha.\, \mathrm{Fruit}(\alpha) \to \mathrm{Juice}(\alpha)$
- Dependent types : $\forall\alpha.\, \{x : \mathrm{Fruit}(\alpha)\} \to \{y : \mathrm{Juice}(\alpha) \mid y < x\}$

Typing and modularity

Why modules in complex systems ?



Why modules in complex systems ?

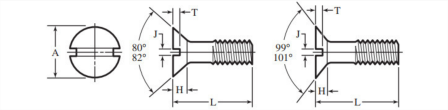
Why modules in complex systems ?

Re-usability

Why modules in complex systems ?

Re-usability

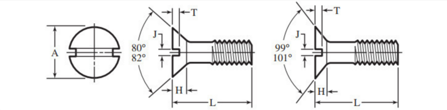
- Standardize basic operations



Why modules in complex systems ?

Re-usability

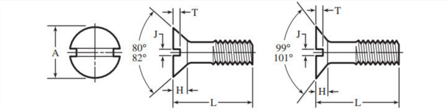
- Standardize basic operations
- Factorize coding and maintenance effort



Why modules in complex systems ?

Re-usability

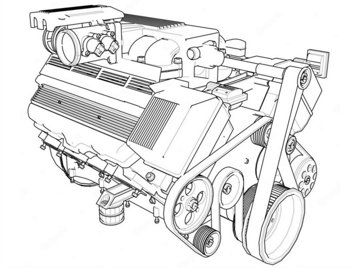
- Standardize basic operations
- Factorize coding and maintenance effort
- Interchangeability



Why modules in complex systems ?

Re-usability

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

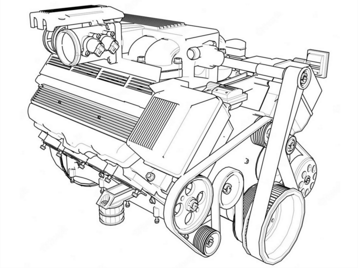


Why modules in complex systems ?

Re-usability

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

System structure



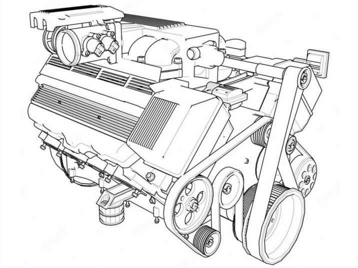
Why modules in complex systems ?

Re-usability

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

System structure

- Separate functionalities



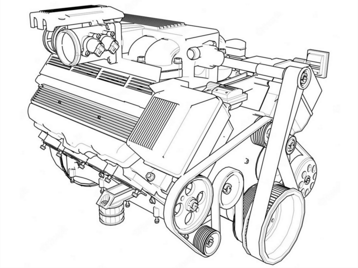
Why modules in complex systems ?

Re-usability

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

System structure

- Separate functionalities
- Isolate components from each-other



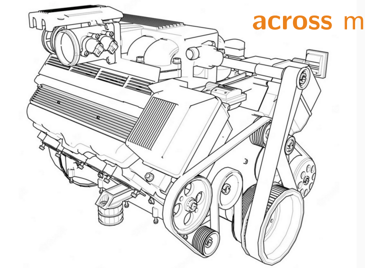
Why modules in complex systems ?

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



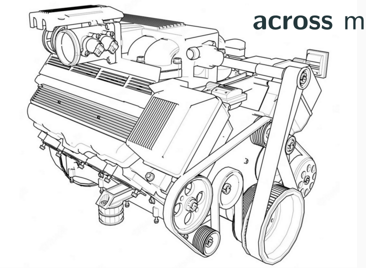
Why modules in complex systems ?

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

Modularity in PL

Basic modularity

Basic modularity

- Functions

Modularity in PL

Basic modularity

- Functions *Too small granularity*

Modularity in PL

Basic modularity

- Functions *Too small granularity*
- Libraries

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*

Modularity in PL

Basic modularity

- Functions *Too small granularity*
- Libraries *Not inter-changeable*

Object oriented programming

Modularity in PL

Basic modularity

- Functions *Too small granularity*
- Libraries *Not inter-changeable*

Object oriented programming

- Abstract objects

Modularity in PL

Basic modularity

- Functions — *Too small granularity*
- Libraries — *Not inter-changeable*

Object oriented programming

- Abstract objects
- Public / Private fields

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*

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

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

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 *Too small granularity*
- Libraries *Not inter-changeable*

Object oriented programming

- Abstract objects
- Public / Private fields *Binary, no partial access*
- Inheritance

Type-classes / Traits

- A method-focused approach

OCaml modules

The module language in OCaml

Generic interface

The module language in OCaml

Generic interface

```ocaml
module type ComplexNumbers = sig








end
```

The module language in OCaml

Generic interface

```
module type ComplexNumbers = sig
  type t







end
```

The module language in OCaml

Generic interface

```
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t





end
```

The module language in OCaml

Generic interface

```
module type ComplexNumbers = sig
  type t
  val make : float -> float -> t
  val add : t -> t -> t




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



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


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
```

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

```










```

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
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}







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

```
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 = ...





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)



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:x; im:_}) = x
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: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: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: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

```
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








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








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 = ...







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 = ...





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




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)


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

```
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

```
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

```
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 : ComplexNumbers)
```

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 =
```

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





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




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 = ...



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 = ...


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
```

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
```

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 =

```

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 ML[1] approach

[1] Not machine-learning

The ML[1] approach

An interface language (description)

[1] Not machine-learning

The ML[1] approach

An interface language (description)

- User-defined *signatures*

[1] Not machine-learning

The ML[1] approach

An interface language (description)

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

[1]Not machine-learning

The ML[1] approach

An interface language (description)

- User-defined *signatures* `sig ... end`
- Abstract types

[1]Not machine-learning

The ML[1] approach

An interface language (description)

- User-defined *signatures* `sig ... end`
- Abstract types `type t`

[1]Not machine-learning

The ML[1] approach

An interface language (description)

- User-defined *signatures* `sig ... end`
- Abstract types `type t`

[1]Not machine-learning

The ML[1] approach

An interface language (description)

- User-defined *signatures* `sig ... end`
- Abstract types `type t`

A module language (construction)

[1] Not machine-learning

The ML[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

[1] Not machine-learning

The ML[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`

[1] Not machine-learning

The ML[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

[1] Not machine-learning

The ML[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 `(M : S)`

[1] Not machine-learning

The ML[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 `(M : S)`
- Client-side abstraction : functors

[1] Not machine-learning

The ML[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 `(M : S)`
- Client-side abstraction : functors `functor (X:S) -> M`

[1]Not machine-learning

The ML[1] approach

An interface language (description)

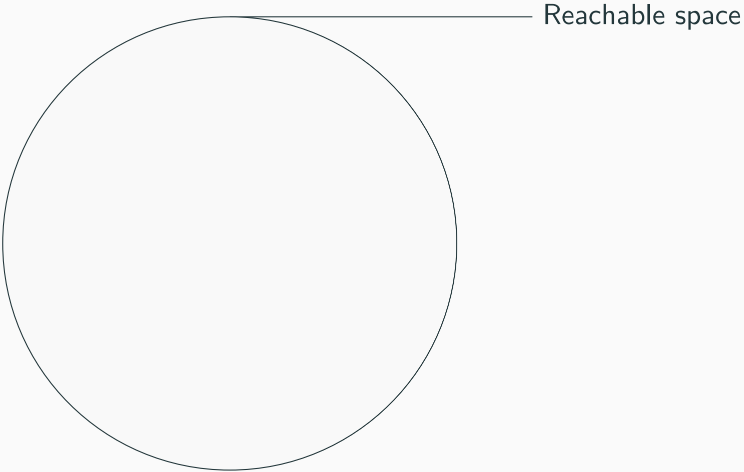
- User-defined *signatures* `sig ... end`
- Abstract types `type t`

A module language (construction)

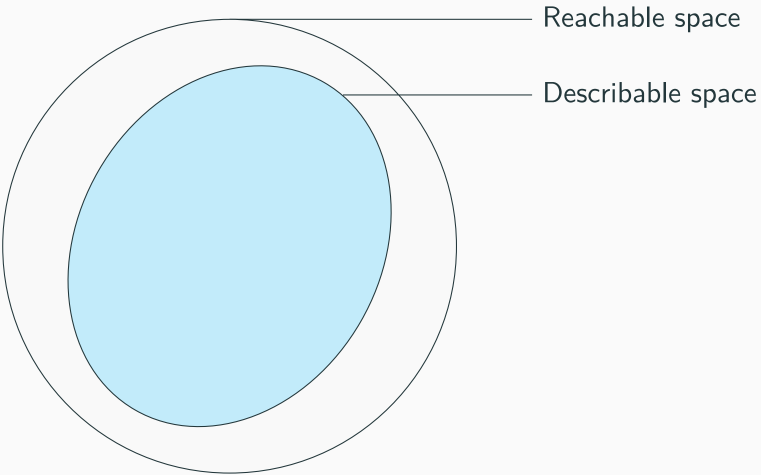
- Separate the module and core languages `struct ... end`
- Developer-side abstraction : sealing `(M : S)`
- Client-side abstraction : functors `functor (X:S) -> M`
- A real module calculus (functions, calls, projections, ...)

[1] Not machine-learning

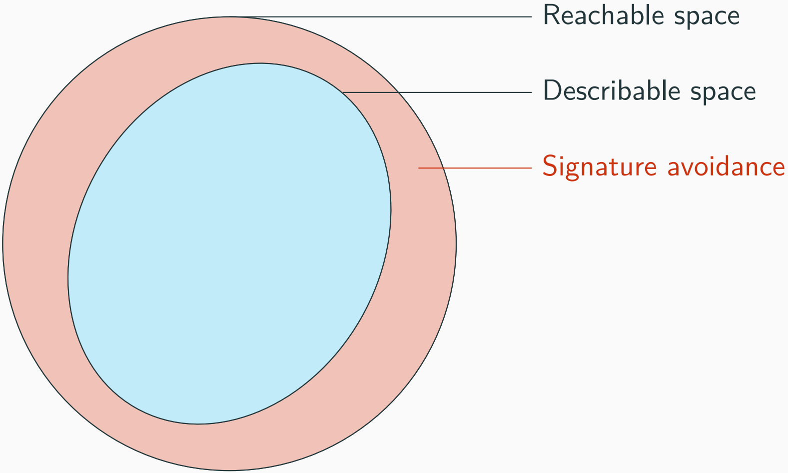
Reachable and *describable* spaces mismatch



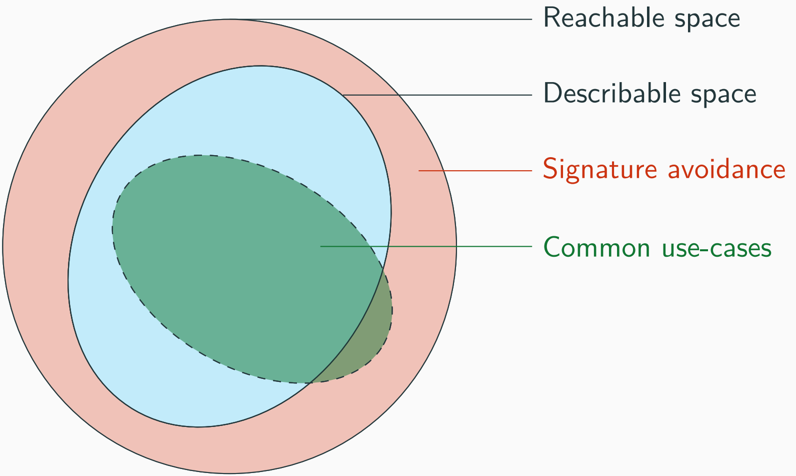
Reachable and *describable* spaces mismatch



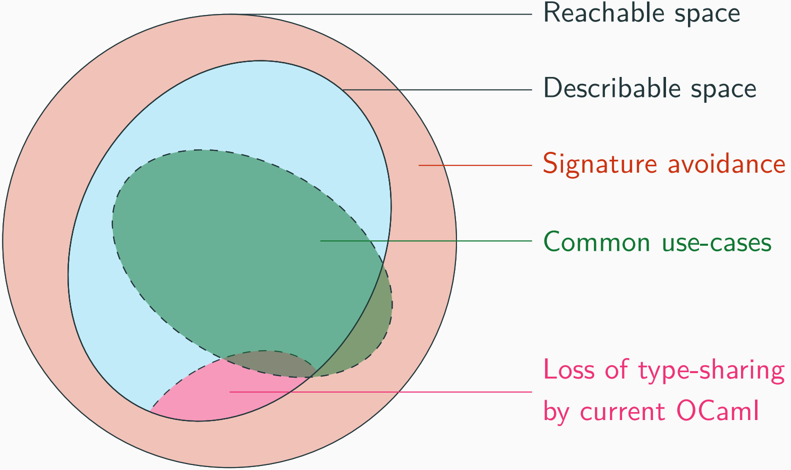
Reachable and *describable* spaces mismatch



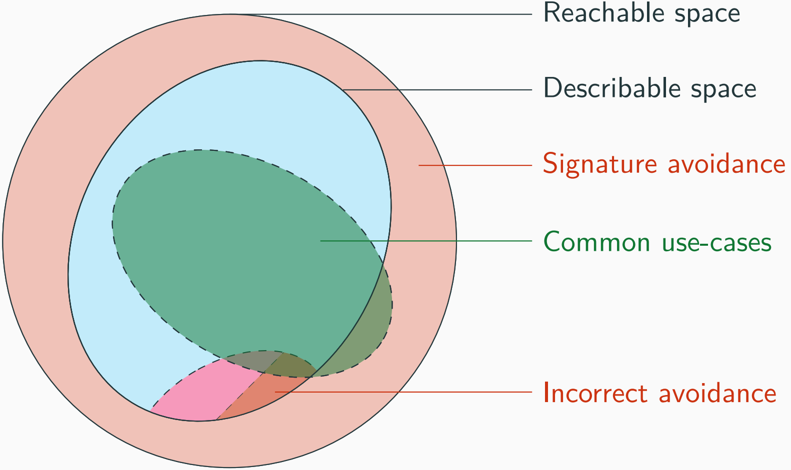
Reachable and *describable* spaces mismatch



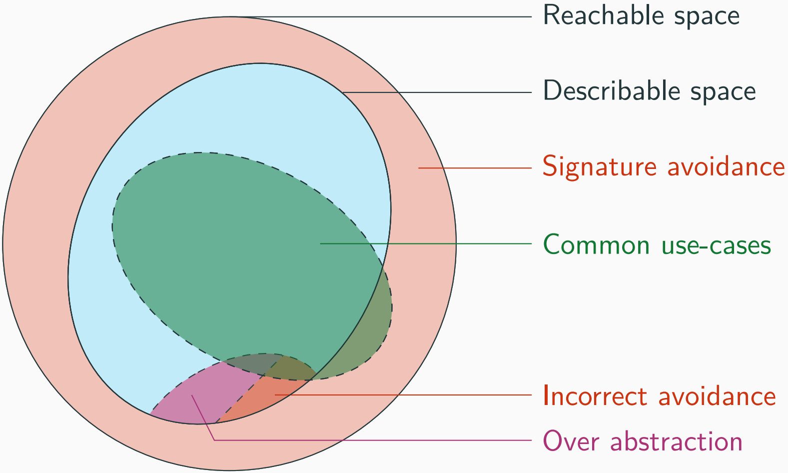
Reachable and *describable* spaces mismatch



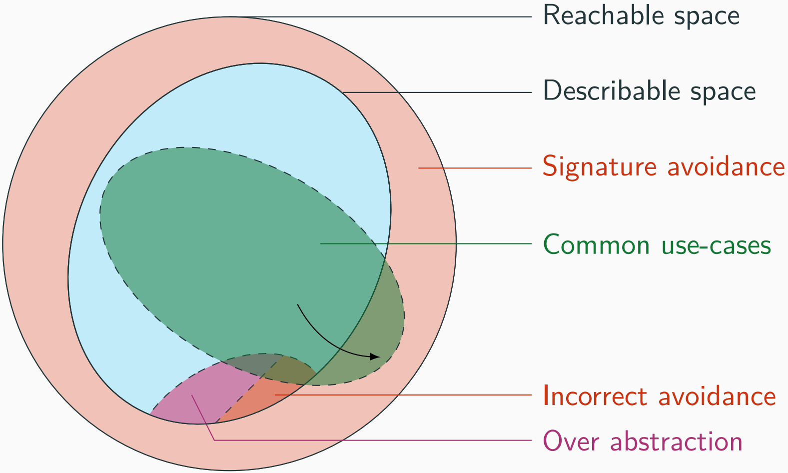
Reachable and *describable* spaces mismatch



Reachable and *describable* spaces mismatch



Reachable and *describable* spaces mismatch



A approach by *elaboration*

OCaml

A approach by *elaboration*

$\boxed{\textsc{OCaml}}$

Hard to formalize

A approach by *elaboration*

OCaml → Hard to formalize

OCaml → Ad-hoc techniques

A approach by *elaboration*

OCaml

- Hard to formalize
- Ad-hoc techniques
- Signature avoidance

A approach by *elaboration*

OCaml

- Hard to formalize
- Ad-hoc techniques
- Signature avoidance

A approach by *elaboration*

OCaml

F^{ω}

A approach by *elaboration*

Clear formalization

OCaml

F^{ω}

A approach by *elaboration*

OCaml

Clear formalization

Standard techniques $\leftarrow$ F^ω

A approach by *elaboration*

OCaml

Clear formalization

Standard techniques

Expressivity

F^ω

A approach by *elaboration*

OCaml

Clear formalization

Standard techniques

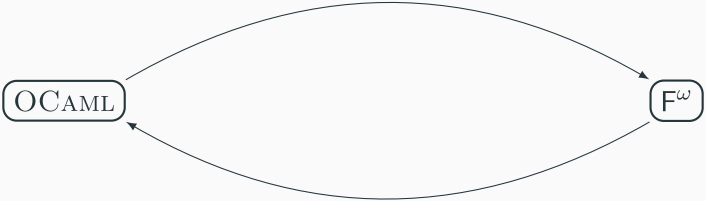
Expressivity

F^{ω}

A approach by *elaboration*

$$\begin{tikzcd} \boxed{\textsc{OCaml}} \arrow[rrrr, bend left] & & & & \boxed{\mathsf{F}^{\omega}} \end{tikzcd}$$

A approach by *elaboration*



Conclusion

- Programming languages are continuously evolving

Conclusion

- 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 ?