Retrofiting OCaml modules

Clément Blaudeau, Cambium team, Inria

Ocaml meetup - *11/05/23*

PhD advisors : Didier Rémy, Gabriel Radanne

Modularity

Modularity

Reusable modules / Structural modules

Modularity

Reusable modules / Structural modules

Interfaces / Abstraction

The power of ML-modularity

Basic modularity: modules, signatures and abstraction

As a module developer

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

As a module user

```
1
2
3
4
5
6
7
8
```

```
1
2
3
4
5
```

Basic modularity: modules, signatures and abstraction

As a module developer

```
1  module Complex  = struct
2
3
4
5
6
7  end
8
9
10
11
12
13
14
15
```

As a module user

```
1
2
3
4
5
6
7
8
```

```
1
2
3
4
5
```

Basic modularity: modules, signatures and abstraction

As a module developer

```
1  module Complex  = struct
2    type t = float * float
3
4
5
6
7  end
8
9
10
11
12
13
14
15
```

As a module user

```
1
2
3
4
5
6
7
8
```

```
1
2
3
4
5
```

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex  = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end
```

As a module user

```

```

```

```

Basic modularity: modules, signatures and abstraction

As a module developer

```
1  module Complex  = struct
2    type t = float * float
3    let zero = (0., 0.)
4    let one = (1., 0.)
5    let add = ...
6    let mult = ...
7  end
8
9  module type Ring = sig
10
11
12
13
14
15 end
```

As a module user

```
1
2
3
4
5
6
7
8
```

```
1
2
3
4
5
```

Basic modularity: modules, signatures and abstraction

As a module developer

```
1  module Complex  = struct
2    type t = float * float
3    let zero = (0., 0.)
4    let one = (1., 0.)
5    let add = ...
6    let mult = ...
7  end
8
9  module type Ring = sig
10   type t
11
12
13
14
15 end
```

As a module user

```
1
2
3
4
5
6
7
8
```

```
1
2
3
4
5
```

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex  = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end

module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mult : t -> t -> t
end
```

As a module user

```


```

```


```

Basic modularity: modules, signatures and abstraction

As a module developer

```
1  module Complex : Ring = struct
2    type t = float * float
3    let zero = (0., 0.)
4    let one = (1., 0.)
5    let add = ...
6    let mult = ...
7  end
8
9  module type Ring = sig
10   type t
11   val zero : t
12   val one : t
13   val add : t -> t -> t
14   val mult : t -> t -> t
15 end
```

As a module user

```
1
2
3
4
5
6
7
8
```

```
1
2
3
4
5
```

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex : Ring = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end

module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mult : t -> t -> t
end
```

As a module user

```
module Polynomials =




end
```

```


```

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex : Ring = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end

module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mult : t -> t -> t
end
```

As a module user

```
module Polynomials =
  functor (R: Ring) -> struct



end
```

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex : Ring = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end

module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mult : t -> t -> t
end
```

As a module user

```
module Polynomials =
  functor (R: Ring) -> struct
    type t = R.t list
    let zero = []
    let one = [R.one]
    let add = ...
    let mult = ...
end
```

```

```

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex : Ring = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end

module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mult : t -> t -> t
end
```

As a module user

```
module Polynomials =
  functor (R: Ring) -> struct
    type t = R.t list
    let zero = []
    let one = [R.one]
    let add = ...
    let mult = ...
end
```

```
module CX =
```

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex : Ring = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end

module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mult : t -> t -> t
end
```

As a module user

```
module Polynomials =
  functor (R: Ring) -> struct
    type t = R.t list
    let zero = []
    let one = [R.one]
    let add = ...
    let mult = ...
end
```

```
module CX =
  Polynomials(Complex)
```

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex : Ring = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end

module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mult : t -> t -> t
end
```

As a module user

```
module Polynomials =
  functor (R: Ring) -> struct
    type t = R.t list
    let zero = []
    let one = [R.one]
    let add = ...
    let mult = ...
end
```

```
module CX =
  Polynomials(Complex)

module CXY =

```

Basic modularity: modules, signatures and abstraction

As a module developer

```
module Complex : Ring = struct
  type t = float * float
  let zero = (0., 0.)
  let one = (1., 0.)
  let add = ...
  let mult = ...
end

module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mult : t -> t -> t
end
```

As a module user

```
module Polynomials =
  functor (R: Ring) -> struct
    type t = R.t list
    let zero = []
    let one = [R.one]
    let add = ...
    let mult = ...
end
```

```
module CX =
  Polynomials(Complex)

module CXY =
  Polynomials(Polynomials(Complex))
```

Basic modularity: modules, signatures and abstraction

As a module developer

```
1  module Complex : Ring = struct
2    type t = float * float
3    let zero = (0., 0.)
4    let one = (1., 0.)
5    let add = ...
6    let mult = ...
7  end
8
9  module type Ring = sig
10   type t
11   val zero : t
12   val one : t
13   val add : t -> t -> t
14   val mult : t -> t -> t
15 end
```

✓ Interface control

As a module user

```
1  module Polynomials =
2    functor (R: Ring) -> struct
3      type t = R.t list
4      let zero = []
5      let one = [R.one]
6      let add = ...
7      let mult = ...
8  end
```

```
1  module CX =
2    Polynomials(Complex)
3
4  module CXY =
5    Polynomials(Polynomials(Complex))
```

Basic modularity: modules, signatures and abstraction

As a module developer

```
1  module Complex : Ring = struct
2    type t = float * float
3    let zero = (0., 0.)
4    let one = (1., 0.)
5    let add = ...
6    let mult = ...
7  end
8
9  module type Ring = sig
10   type t
11   val zero : t
12   val one : t
13   val add : t -> t -> t
14   val mult : t -> t -> t
15 end
```

✓ Interface control

✓ Abstraction

As a module user

```
1  module Polynomials =
2    functor (R: Ring) -> struct
3      type t = R.t list
4      let zero = []
5      let one = [R.one]
6      let add = ...
7      let mult = ...
8  end
```

```
1  module CX =
2    Polynomials(Complex)
3
4  module CXY =
5    Polynomials(Polynomials(Complex))
```

Basic modularity: modules, signatures and abstraction

As a module developer

```
1  module Complex : Ring = struct
2    type t = float * float
3    let zero = (0., 0.)
4    let one = (1., 0.)
5    let add = ...
6    let mult = ...
7  end
8
9  module type Ring = sig
10   type t
11   val zero : t
12   val one : t
13   val add : t -> t -> t
14   val mult : t -> t -> t
15 end
```

✓ Interface control

✓ Abstraction

As a module user

```
1  module Polynomials =
2    functor (R: Ring) -> struct
3      type t = R.t list
4      let zero = []
5      let one = [R.one]
6      let add = ...
7      let mult = ...
8  end
```

```
1  module CX =
2    Polynomials(Complex)
3
4  module CXY =
5    Polynomials(Polynomials(Complex))
```

✓ Polymorphism

Basic modularity: modules, signatures and abstraction

As a module developer

```
1  module Complex : Ring = struct
2    type t = float * float
3    let zero = (0., 0.)
4    let one = (1., 0.)
5    let add = ...
6    let mult = ...
7  end
8
9  module type Ring = sig
10   type t
11   val zero : t
12   val one : t
13   val add : t -> t -> t
14   val mult : t -> t -> t
15 end
```

✓ Interface control

✓ Abstraction

As a module user

```
1  module Polynomials =
2    functor (R: Ring) -> struct
3      type t = R.t list
4      let zero = []
5      let one = [R.one]
6      let add = ...
7      let mult = ...
8  end
```

```
1  module CX =
2    Polynomials(Complex)
3
4  module CXY =
5    Polynomials(Polynomials(Complex))
```

✓ Polymorphism

✓ Composition

Abstraction and dependencies - functors

Abstraction and dependencies - functors

Generative

Abstraction and dependencies - functors

Generative

Applicative

Abstraction and dependencies - functors

Generative

Functors as parameterized *sub-programs*

Applicative

Abstraction and dependencies - functors

Generative

Functors as parameterized *sub-programs*

- Internal state / effects

Applicative

Abstraction and dependencies - functors

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- *Strong* abstraction barrier

Applicative

Abstraction and dependencies - functors

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- *Strong* abstraction barrier

```
1 module Tokens () =
2
3
4
5
6
7
8
9
```

Applicative

Abstraction and dependencies - functors

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- *Strong* abstraction barrier

```
1  module Tokens () = struct
2    type t = int
3    let x = ref 0
4    ...
5  end
6
7
8
9
```

Applicative

Abstraction and dependencies - functors

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- *Strong* abstraction barrier

```
1  module Tokens () = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7
8
9
```

Applicative

Abstraction and dependencies - functors

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- *Strong* abstraction barrier

```
1  module Tokens () = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module T1 = Tokens() (* Public tokens *)
8  module T2 = Tokens() (* Private tokens *)
9
```

Applicative

Abstraction and dependencies - functors

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- *Strong* abstraction barrier

```
1  module Tokens () = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module T1 = Tokens() (* Public tokens *)
8  module T2 = Tokens() (* Private tokens *)
9  (* T1.t ≠ T2.t *) ×
```

Applicative

Abstraction and dependencies - functors

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- *Strong* abstraction barrier

```
1  module Tokens () = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module T1 = Tokens() (* Public tokens *)
8  module T2 = Tokens() (* Private tokens *)
9  (* T1.t ≠ T2.t *) ×
```

Applicative

Functors as parameterized *libraries*

Abstraction and dependencies - functors

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- *Strong* abstraction barrier

```
1  module Tokens () = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module T1 = Tokens() (* Public tokens *)
8  module T2 = Tokens() (* Private tokens *)
9  (* T1.t ≠ T2.t *) ×
```

Applicative

Functors as parameterized *libraries*

- Purity

Abstraction and dependencies - functors

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- *Strong* abstraction barrier

```
1  module Tokens () = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module T1 = Tokens() (* Public tokens *)
8  module T2 = Tokens() (* Private tokens *)
9  (* T1.t ≠ T2.t *) ×
```

Applicative

Functors as parameterized *libraries*

- Purity
- *Weak* abstraction barrier

Abstraction and dependencies - functors

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- *Strong* abstraction barrier

```
1  module Tokens () = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module T1 = Tokens() (* Public tokens *)
8  module T2 = Tokens() (* Private tokens *)
9  (* T1.t ≠ T2.t *) ×
```

Applicative

Functors as parameterized *libraries*

- Purity
- *Weak* abstraction barrier

```
1  module OrderedSet (E:OrderedType) = struct
2
3
4
5
6
7
8
9
```

Abstraction and dependencies - functors

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- *Strong* abstraction barrier

```
1  module Tokens () = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module T1 = Tokens() (* Public tokens *)
8  module T2 = Tokens() (* Private tokens *)
9  (* T1.t ≠ T2.t *) ×
```

Applicative

Functors as parameterized *libraries*

- Purity
- *Weak* abstraction barrier

```
1  module OrderedSet (E:OrderedType) = struct
2    type t = E.t list
3    let empty : t = []
4    ...
5  end
6
7
8
9
```

Abstraction and dependencies - functors

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- *Strong* abstraction barrier

```
1  module Tokens () = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module T1 = Tokens() (* Public tokens *)
8  module T2 = Tokens() (* Private tokens *)
9  (* T1.t ≠ T2.t *) ×
```

Applicative

Functors as parameterized *libraries*

- Purity
- *Weak* abstraction barrier

```
1  module OrderedSet (E:OrderedType) = (struct
2    type t = E.t list
3    let empty : t = []
4    ...
5  end : sig type t ... end)
6
7
8
9
```

Abstraction and dependencies - functors

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- *Strong* abstraction barrier

```
1  module Tokens () = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module T1 = Tokens() (* Public tokens *)
8  module T2 = Tokens() (* Private tokens *)
9  (* T1.t ≠ T2.t *) ×
```

Applicative

Functors as parameterized *libraries*

- Purity
- *Weak* abstraction barrier

```
1  module OrderedSet (E:OrderedType) = (struct
2    type t = E.t list
3    let empty : t = []
4    ...
5  end : sig type t ... end)
6
7  module S1 = OrderedSet(Integer)
8  module S2 = OrderedSet(Integer)
9
```

Abstraction and dependencies - functors

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- *Strong* abstraction barrier

```
1  module Tokens () = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module T1 = Tokens() (* Public tokens *)
8  module T2 = Tokens() (* Private tokens *)
9  (* T1.t ≠ T2.t *) ×
```

Applicative

Functors as parameterized *libraries*

- Purity
- *Weak* abstraction barrier

```
1  module OrderedSet (E:OrderedType) = (struct
2    type t = E.t list
3    let empty : t = []
4    ...
5  end : sig type t ... end)
6
7  module S1 = OrderedSet(Integer)
8  module S2 = OrderedSet(Integer)
9  (* S1.t = S2.t *) ✓
```

Abstraction and dependencies - functors

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- *Strong* abstraction barrier

```
1  module Tokens () = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module T1 = Tokens() (* Public tokens *)
8  module T2 = Tokens() (* Private tokens *)
9  (* T1.t ≠ T2.t *) ×
```

Applicative

Functors as parameterized *libraries*

- Purity
- *Weak* abstraction barrier

→ *same applications* produce same results

```
1  module OrderedSet (E:OrderedType) = (struct
2    type t = E.t list
3    let empty : t = []
4    ...
5  end : sig type t ... end)
6
7  module S1 = OrderedSet(Integer)
8  module S2 = OrderedSet(Integer)
9  (* S1.t = S2.t *) ✓
```

Abstraction and dependencies - functors

Generative

Functors as parameterized *sub-programs*

- Internal state / effects
- *Strong* abstraction barrier

```
1  module Tokens () = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module T1 = Tokens() (* Public tokens *)
8  module T2 = Tokens() (* Private tokens *)
9  (* T1.t ≠ T2.t *) ×
```

Applicative

Functors as parameterized *libraries*

- Purity
- *Weak* abstraction barrier

→ *same applications* produce same results

```
1  module OrderedSet (E:OrderedType) = (struct
2    type t = E.t list
3    let empty : t = []
4    ...
5  end : sig type t ... end)
6
7  module S1 = OrderedSet(Integer)
8  module S2 = OrderedSet(Integer)
9  (* S1.t = S2.t *) ✓
```

Applicativity granularity

```
1  module X1 = struct
2    type t = int
3    ...
4  end
5
6  module X2 = struct
7    type t = int
8    ...
9  end
10
11 OrderedSet(X1).t =? OrderedSet(X2).t
```

Applicativity granularity

Types only

```
1  module X1 = struct
2    type t = int
3    ...
4  end
5
6  module X2 = struct
7    type t = int
8    ...
9  end
10
11 OrderedSet(X1).t =? OrderedSet(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types

```
1  module X1 = struct
2    type t = int
3    ...
4  end
5
6  module X2 = struct
7    type t = int
8    ...
9  end
10
11 OrderedSet(X1).t =? OrderedSet(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  $\rightarrow$ assumes the functor's body only depends on types fields

```
1  module X1 = struct
2    type t = int
3    ...
4  end
5
6  module X2 = struct
7    type t = int
8    ...
9  end
10
11 OrderedSet(X1).t =? OrderedSet(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  $\rightarrow$ assumes the functor's body only depends on types fields

```
1  module X1 = struct
2    type t = int
3    let compare = (<)
4  end
5
6  module X2 = struct
7    type t = int
8    let compare = (>)
9  end
10
11 OrderedSet(X1).t =? OrderedSet(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  $\rightarrow$ assumes the functor's body only depends on types fields

Types and values

```
1  module X1 = struct
2    type t = int
3    let compare = (<)
4  end
5
6  module X2 = struct
7    type t = int
8    let compare = (>)
9  end
10
11 OrderedSet(X1).t =? OrderedSet(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  $\rightarrow$ assumes the functor's body only depends on types fields

Types and values

- *Abstraction safe*

```
1  module X1 = struct
2    type t = int
3    let compare = (<)
4  end
5
6  module X2 = struct
7    type t = int
8    let compare = (>)
9  end
10
11 OrderedSet(X1).t =? OrderedSet(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  $\rightarrow$ assumes the functor's body only depends on types fields

Types and values

- *Abstraction safe*
  $\rightarrow$ tracking equality of values

```
1  module X1 = struct
2    type t = int
3    let compare = (<)
4  end
5
6  module X2 = struct
7    type t = int
8    let compare = (>)
9  end
10
11 OrderedSet(X1).t =? OrderedSet(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  $\rightarrow$ assumes the functor's body only depends on types fields

Types and values

- *Abstraction safe*
  $\rightarrow$ tracking equality of values

```
1  module X1 = struct
2    type t = int
3    let compare = (<)
4  end
5
6  module X2 = struct
7    type t = int
8    let compare = X1.compare
9  end
10
11 OrderedSet(X1).t =? OrderedSet(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  $\rightarrow$ assumes the functor's body only depends on types fields

Types and values - *fine grained*

- *Abstraction safe*
  $\rightarrow$ tracking equality of values

```
1  module X1 = struct
2    type t = int
3    let compare = (<)
4  end
5
6  module X2 = struct
7    type t = int
8    let compare = X1.compare
9  end
10
11 OrderedSet(X1).t =? OrderedSet(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  $\rightarrow$ assumes the functor's body only depends on types fields

Types and values - *fine grained*

- *Abstraction safe*
  $\rightarrow$ tracking equality of values

Modules - *coarse grained*

```
1  module X1 = struct
2    type t = int
3    let compare = (<)
4  end
5
6  module X2 = struct
7    type t = int
8    let compare = X1.compare
9  end
10
11 OrderedSet(X1).t =? OrderedSet(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  $\rightarrow$ assumes the functor's body only depends on types fields

Types and values - *fine grained*

- *Abstraction safe*
  $\rightarrow$ tracking equality of values

Modules - *coarse grained*

- Syntactic criterion

```
1  module X1 = struct
2    type t = int
3    let compare = (<)
4  end
5
6  module X2 = struct
7    type t = int
8    let compare = X1.compare
9  end
10
11 OrderedSet(X1).t =? OrderedSet(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  $\rightarrow$ assumes the functor's body only depends on types fields

Types and values - *fine grained*

- *Abstraction safe*
  $\rightarrow$ tracking equality of values

Modules - *coarse grained*

- Syntactic criterion

```
1  module X1 = struct
2    type t = int
3    let compare = (<)
4  end
5
6
7
8
9
10
11 OrderedSet(X1).t =? OrderedSet(X1).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  $\rightarrow$ assumes the functor's body only depends on types fields

Types and values - *fine grained*

- *Abstraction safe*
  $\rightarrow$ tracking equality of values

Modules - *coarse grained*

- Syntactic criterion
  $\rightarrow$ tracking of module *aliasing*

```
1  module X1 = struct
2    type t = int
3    let compare = (<)
4  end
5
6
7
8
9
10
11 OrderedSet(X1).t =? OrderedSet(X1).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  → assumes the functor's body only depends on types fields

Types and values - *fine grained*

- *Abstraction safe*
  → tracking equality of values

Modules - *coarse grained*

- Syntactic criterion
  → tracking of module *aliasing*

```
1  module X1 = struct
2    type t = int
3    let compare = (<)
4  end
5
6  module X2 = X1
7
8
9
10
11 OrderedSet(X1).t =? OrderedSet(X2).t
```

Applicativity granularity

Types only

- Sound: types only depend on types
  $\rightarrow$ assumes the functor's body only depends on types fields

Types and values - *fine grained*

- *Abstraction safe*
  $\rightarrow$ tracking equality of values

Modules - *coarse grained*

- Syntactic criterion
  $\rightarrow$ tracking of module *aliasing*

```
1  module X1 = struct
2    type t = int
3    let compare = (<)
4  end
5
6  module X2 = X1
7
8
9
10
11 OrderedSet(X1).t =? OrderedSet(X2).t
```

Who needs module aliases ?

<footer>6 ⟨1⟩/ 11</footer>

Who needs module aliases ?

```
module type VS = sig (* Vector space *)


end
```

Who needs module aliases ?

```
module type VS = sig (* Vector space *)
  module Scalar : Field

end
```

Who needs module aliases ?

```
module type VS = sig (* Vector space *)
  module Scalar : Field
  ... (* more fields *)
end
```

Who needs module aliases ?

```
module type VS = sig (* Vector space *)
  module Scalar : Field
  ... (* more fields *)
end

module LinearAlg(V:VS) = struct
  ...


end
```

Who needs module aliases ?

```
module type VS = sig (* Vector space *)
  module Scalar : Field
  ... (* more fields *)
end

module LinearAlg(V:VS) = struct
  ...
  module ScalarSet = Set(V.Scalar)

end
```

Who needs module aliases ?

```
module type VS = sig (* Vector space *)
  module Scalar : Field
  ... (* more fields *)
end

module LinearAlg(V:VS) = struct
  ...
  module ScalarSet = Set(V.Scalar)
  ...
end
```

Who needs module aliases ?

```
module type VS = sig (* Vector space *)
  module Scalar : Field
  ... (* more fields *)
end

module LinearAlg(V:VS) = struct
  ...
  module ScalarSet = Set(V.Scalar)
  ...
end
```

```
module Make3D(K:Field) = (struct


end
```

Who needs module aliases ?

```
module type VS = sig (* Vector space *)
  module Scalar : Field
  ... (* more fields *)
end

module LinearAlg(V:VS) = struct
  ...
  module ScalarSet = Set(V.Scalar)
  ...
end
```

```
module Make3D(K:Field) = (struct
  module Scalar = K

end
```

Who needs module aliases ?

```
1  module type VS = sig (* Vector space *)
2    module Scalar : Field
3    ... (* more fields *)
4  end
5
6  module LinearAlg(V:VS) = struct
7    ...
8    module ScalarSet = Set(V.Scalar)
9    ...
10 end
```

```
11 module Make3D(K:Field) = (struct
12   module Scalar = K
13   ... (* built from K *)
14 end
15
16
17
18
19
20
```

Who needs module aliases ?

```
1  module type VS = sig (* Vector space *)
2    module Scalar : Field
3    ... (* more fields *)
4  end
5
6  module LinearAlg(V:VS) = struct
7    ...
8    module ScalarSet = Set(V.Scalar)
9    ...
10 end
```

```
11 module Make3D(K:Field) = (struct
12   module Scalar = K
13   ... (* built from K *)
14 end
15
16
17
18 module Reals = ...
19
20
```

Who needs module aliases ?

```
1  module type VS = sig (* Vector space *)
2    module Scalar : Field
3    ... (* more fields *)
4  end
5
6  module LinearAlg(V:VS) = struct
7    ...
8    module ScalarSet = Set(V.Scalar)
9    ...
10 end
```

```
11 module Make3D(K:Field) = (struct
12   module Scalar = K
13   ... (* built from K *)
14 end
15
16
17
18 module Reals = ...
19 module Space3D = LinearAlg(Make3D(Reals))
20
```

Who needs module aliases ?

```
1  module type VS = sig (* Vector space *)
2    module Scalar : Field
3    ... (* more fields *)
4  end
5
6  module LinearAlg(V:VS) = struct
7    ...
8    module ScalarSet = Set(V.Scalar)
9    ...
10 end
```

```
11 module Make3D(K:Field) = (struct
12   module Scalar = K
13   ... (* built from K *)
14 end
15
16
17
18 module Reals = ...
19 module Space3D = LinearAlg(Make3D(Reals))
20 (* Space3D.ScalarSet.t =? Set(Reals).t *)
```

Who needs module aliases ?

```
1  module type VS = sig (* Vector space *)
2    module Scalar : Field
3    ... (* more fields *)
4  end
5
6  module LinearAlg(V:VS) = struct
7    ...
8    module ScalarSet = Set(V.Scalar)
9    ...
10 end
```

```
11 module Make3D(K:Field) = (struct
12   module Scalar = K
13   ... (* built from K *)
14 end
15
16
17
18 module Reals = ...
19 module Space3D = LinearAlg(Make3D(Reals))
20 (* Space3D.ScalarSet.t =? Set(Reals).t *)
```

Structural functors

Who needs module aliases ?

```
1  module type VS = sig (* Vector space *)
2    module Scalar : Field
3    ... (* more fields *)
4  end
5
6  module LinearAlg(V:VS) = struct
7    ...
8    module ScalarSet = Set(V.Scalar)
9    ...
10 end
```

```
11 module Make3D(K:Field) = (struct
12   module Scalar = K
13   ... (* built from K *)
14 end
15
16
17
18 module Reals = ...
19 module Space3D = LinearAlg(Make3D(Reals))
20 (* Space3D.ScalarSet.t =? Set(Reals).t *)
```

Structural functors

Reusable (applicative) functors

Who needs module aliases ?

```
1  module type VS = sig (* Vector space *)
2    module Scalar : Field
3    ... (* more fields *)
4  end
5
6  module LinearAlg(V:VS) = struct
7    ...
8    module ScalarSet = Set(V.Scalar)
9    ...
10 end
```

```
11 module Make3D(K:Field) = (struct
12   module Scalar = K
13   ... (* built from K *)
14 end : sig
15
16
17 end)
18 module Reals = ...
19 module Space3D = LinearAlg(Make3D(Reals))
20 (* Space3D.ScalarSet.t =? Set(Reals).t *)
```

Who needs module aliases ?

```
1  module type VS = sig (* Vector space *)
2    module Scalar : Field
3    ... (* more fields *)
4  end
5
6  module LinearAlg(V:VS) = struct
7    ...
8    module ScalarSet = Set(V.Scalar)
9    ...
10 end
```

```
11 module Make3D(K:Field) = (struct
12   module Scalar = K
13   ... (* built from K *)
14 end : sig
15   module Scalar : (= K)
16   ...
17 end)
18 module Reals = ...
19 module Space3D = LinearAlg(Make3D(Reals))
20 (* Space3D.ScalarSet.t =? Set(Reals).t *)
```

Who needs module aliases ?

```
1  module type VS = sig (* Vector space *)
2    module Scalar : Field
3    ... (* more fields *)
4  end
5
6  module LinearAlg(V:VS) = struct
7    ...
8    module ScalarSet = Set(V.Scalar)
9    ...
10 end
```

```
11 module Make3D(K:Field) = (struct
12   module Scalar = K
13   ... (* built from K *)
14 end : sig
15   module Scalar : (= K < Field)
16   ...
17 end)
18 module Reals = ...
19 module Space3D = LinearAlg(Make3D(Reals))
20 (* Space3D.ScalarSet.t =? Set(Reals).t *)
```

Who needs module aliases ?

```
1  module type VS = sig (* Vector space *)
2    module Scalar : Field
3    ... (* more fields *)
4  end
5
6  module LinearAlg(V:VS) = struct
7    ...
8    module ScalarSet = Set(V.Scalar)
9    ...
10 end
```

```
11 module Make3D(K:Field) = (struct
12   module Scalar = K
13   ... (* built from K *)
14 end : sig
15   module Scalar : (= K < Field)
16   ...
17 end)
18 module Reals = ...
19 module Space3D = LinearAlg(Make3D(Reals))
20 (* Space3D.ScalarSet.t =? Set(Reals).t *)
```

Transparent ascription

Who needs module aliases ?

```
1  module type VS = sig (* Vector space *)
2    module Scalar : Field
3    ... (* more fields *)
4  end
5
6  module LinearAlg(V:VS) = struct
7    ...
8    module ScalarSet = Set(V.Scalar)
9    ...
10 end
```

```
11 module Make3D(K:Field) = (struct
12   module Scalar = K
13   ... (* built from K *)
14 end : sig
15   module Scalar : (= K < Field)
16   ...
17 end)
18 module Reals = ...
19 module Space3D = LinearAlg(Make3D(Reals))
20 (* Space3D.ScalarSet.t =? Set(Reals).t *)
```

Specifying OCaml modules

Position-dependent meaning of syntax

Enriched syntax

$\rightarrow$ F^{ω} quantifiers

Key mechanisms

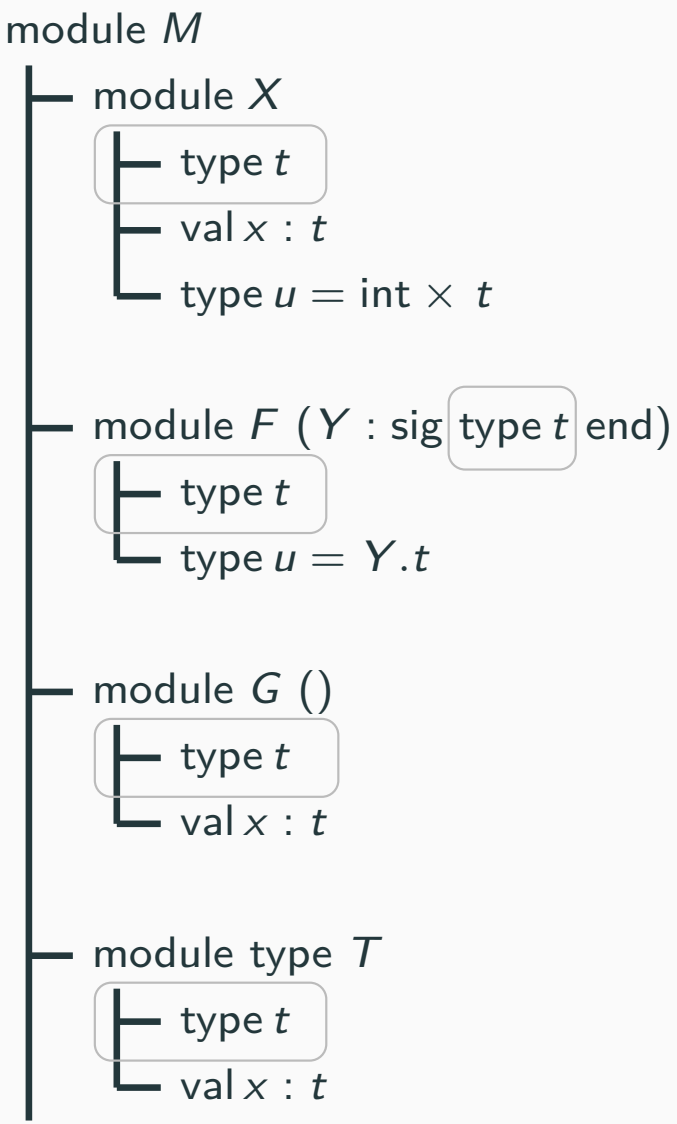
- module M
  - module X
    - type t
    - val $x : t$
    - type $u = \mathsf{int} \times t$
  - module F $(Y : \mathsf{sig}\ \mathsf{type}\ t\ \mathsf{end})$
    - type t
    - type $u = Y.t$
  - module G $()$
    - type t
    - val $x : t$
  - module type T
    - type t
    - val $x : t$

Position-dependent meaning of syntax

Enriched syntax

$\rightarrow$ F^{ω} quantifiers

Key mechanisms



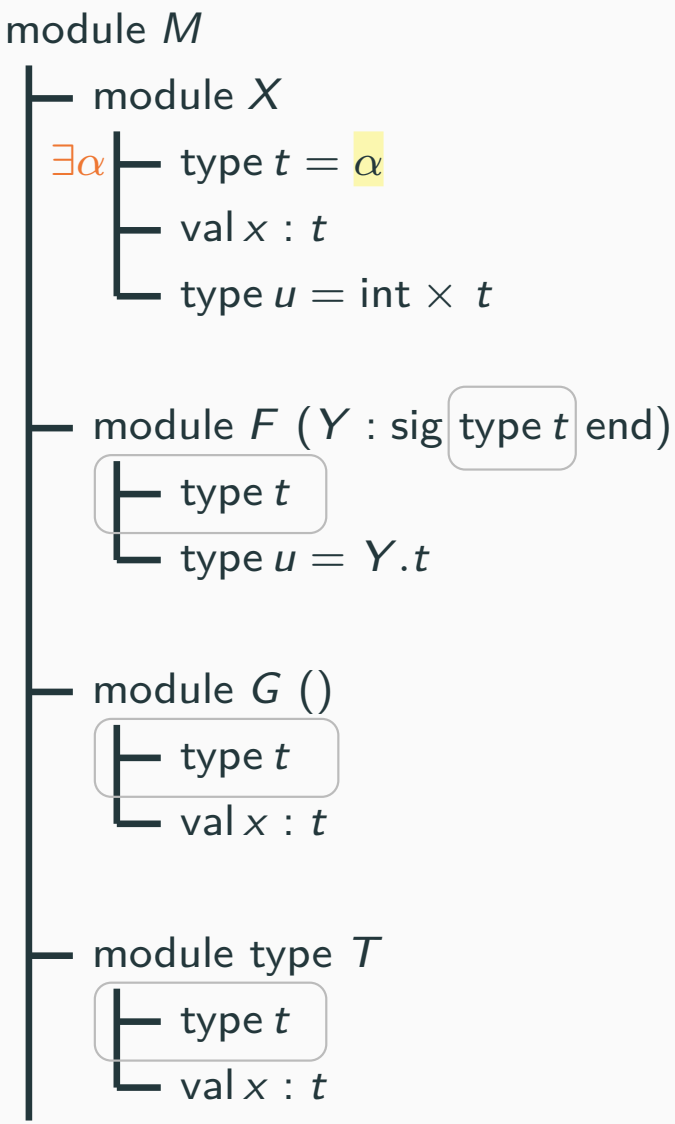
Position-dependent meaning of syntax

Enriched syntax

$\rightarrow$ F^{ω} quantifiers

- Existential for ascription

Key mechanisms



Position-dependent meaning of syntax

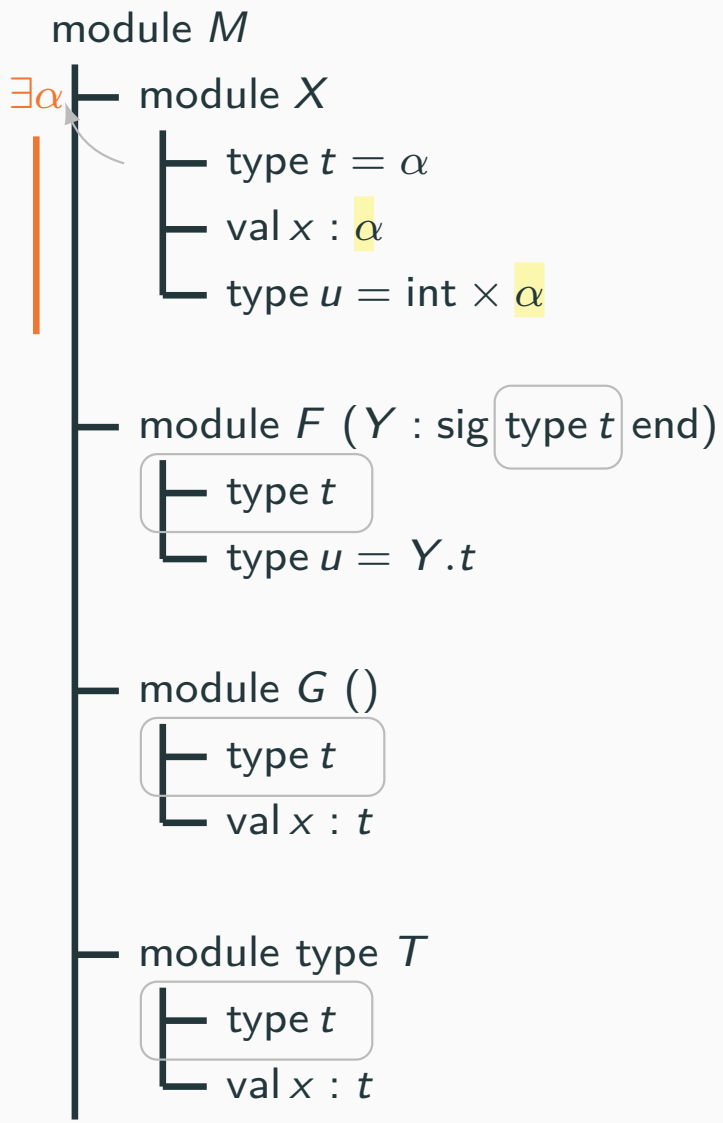
Enriched syntax

$\rightarrow$ F^{ω} quantifiers

- Existential for ascription

Key mechanisms

- Existential lifting



Position-dependent meaning of syntax

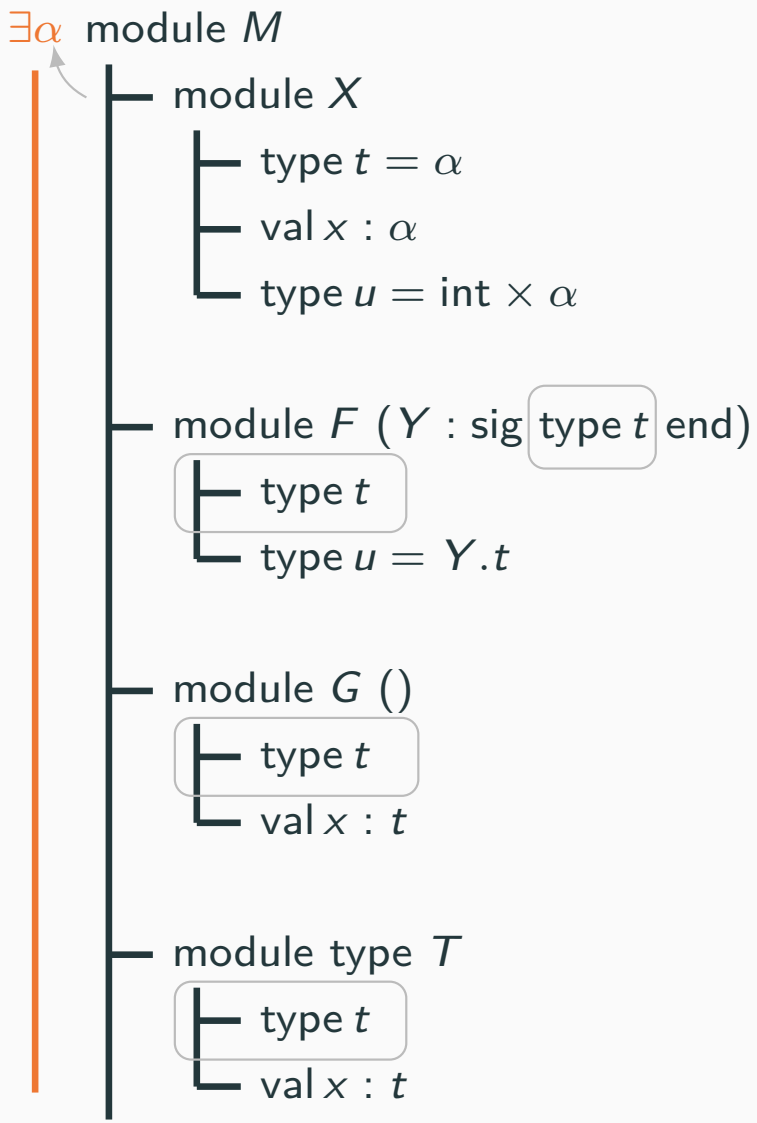
Enriched syntax

$\rightarrow$ F^{ω} quantifiers

- Existential for ascription
- Universal for functors

Key mechanisms

- Existential lifting



Position-dependent meaning of syntax

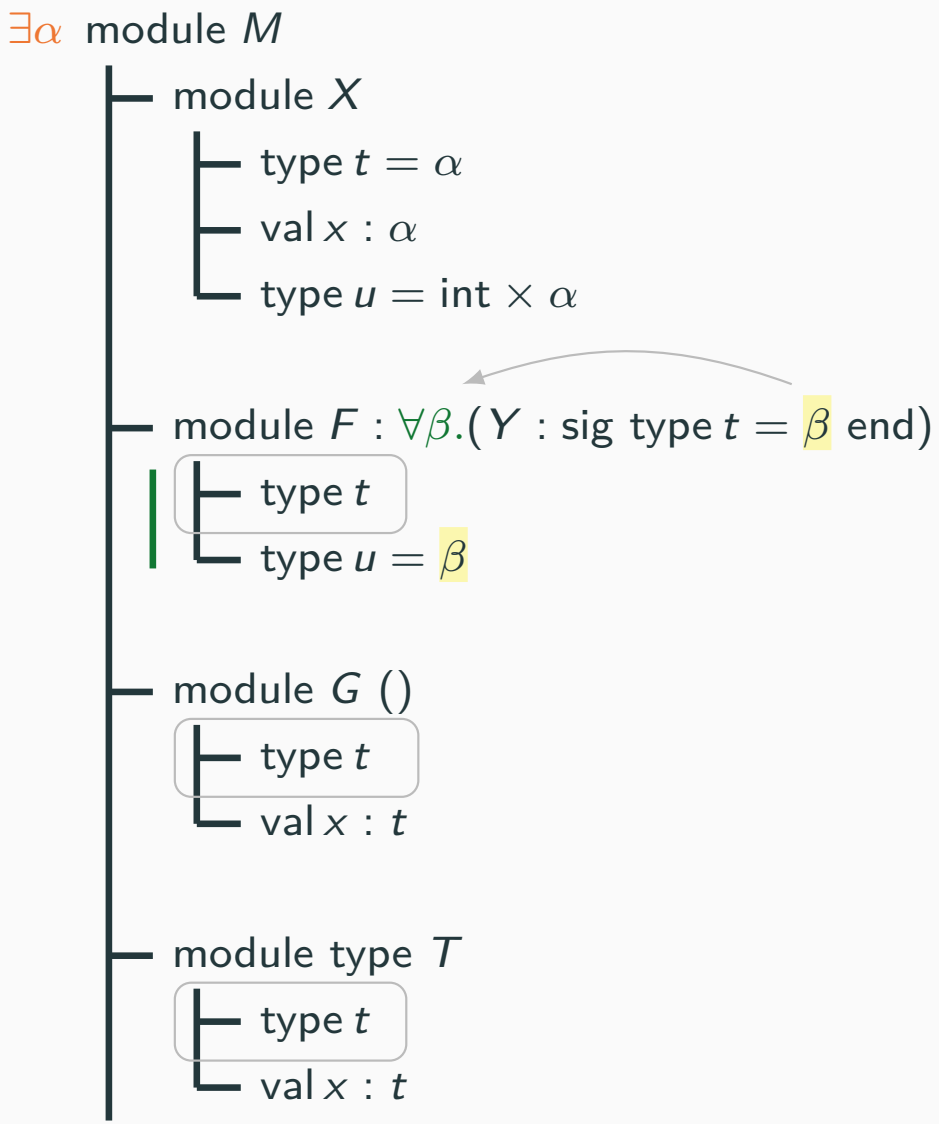
Enriched syntax

$\rightarrow$ F^{ω} quantifiers

- Existential for ascription
- Universal for functors

Key mechanisms

- Existential lifting



Position-dependent meaning of syntax

Enriched syntax

$\rightarrow$ F^{ω} quantifiers

- Existential for ascription
- Universal for functors

Key mechanisms

- Existential lifting

$\exists\alpha$ module M
- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \mathsf{int} \times \alpha$
- module $F : \forall\beta.(Y : \mathsf{sig\ type}\ t = \beta\ \mathsf{end})$
  - $\exists\gamma$ type $t = \gamma$
  - type $u = \beta$
- module G ()
  - type t
  - val $x : t$
- module type T
  - type t
  - val $x : t$

Position-dependent meaning of syntax

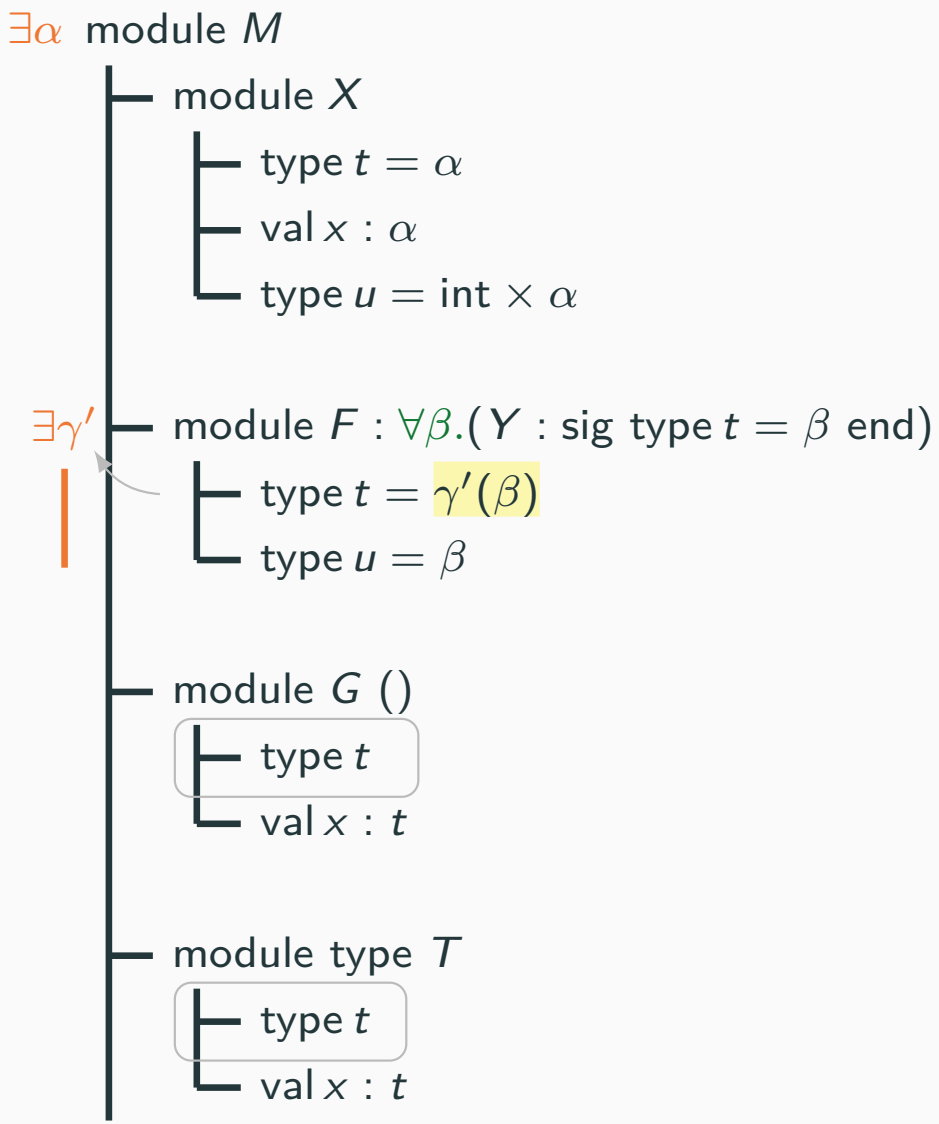
Enriched syntax

$\rightarrow$ F^{ω} quantifiers

- Existential for ascription
- Universal for functors

Key mechanisms

- Existential lifting
- Skolemization



Position-dependent meaning of syntax

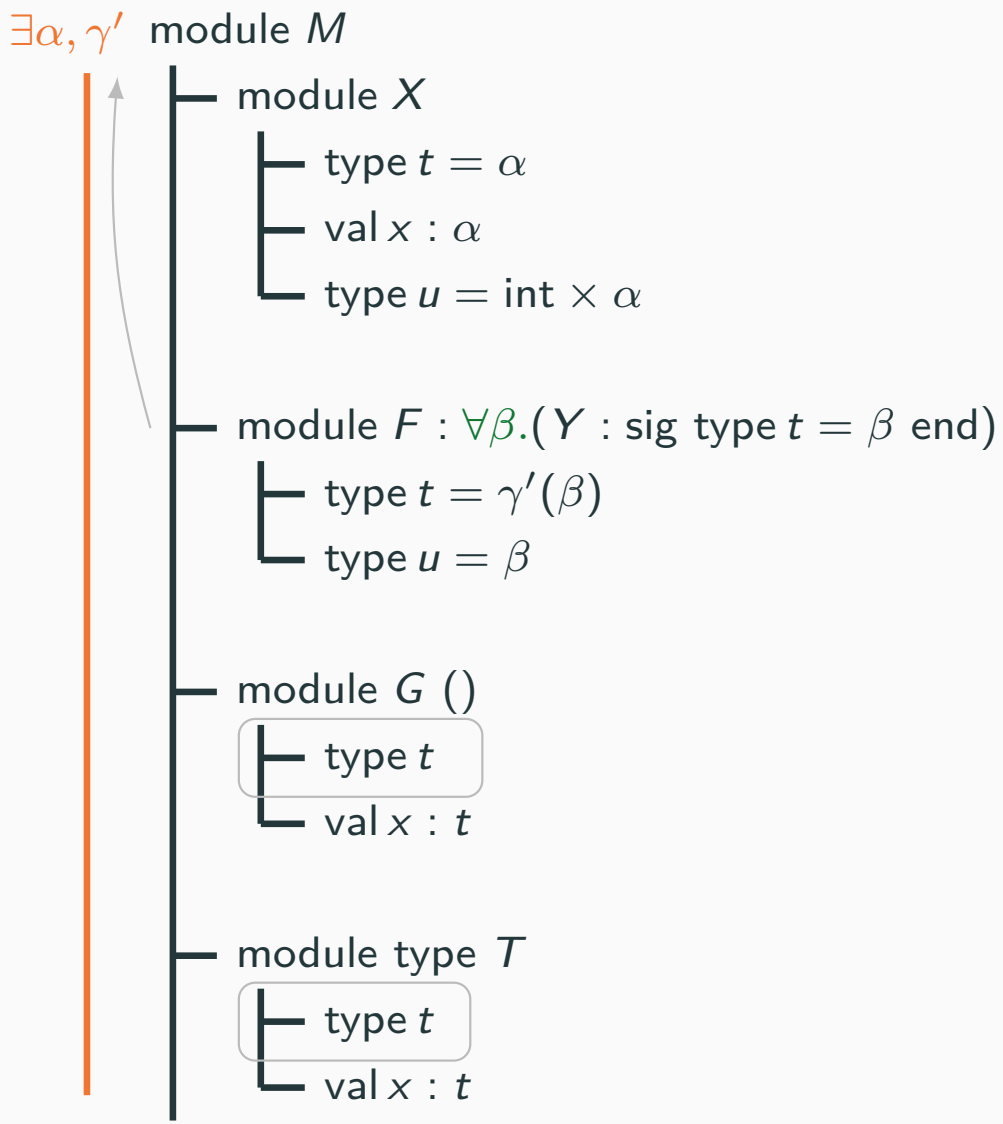
Enriched syntax

$\rightarrow$ F^{ω} quantifiers

- Existential for ascription
- Universal for functors

Key mechanisms

- Existential lifting
- Skolemization



Position-dependent meaning of syntax

Enriched syntax

$\rightarrow$ F^{ω} quantifiers

- Existential for ascription
- Universal for functors

Key mechanisms

- Existential lifting
- Skolemization

$\exists \alpha, \gamma'$ module M

- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \mathsf{int} \times \alpha$
- module $F : \forall \beta.(Y : \mathsf{sig\ type}\ t = \beta\ \mathsf{end})$
  - type $t = \gamma'(\beta)$
  - type $u = \beta$
- module G ()
  - $\exists \delta$
    - type $t = \delta$
    - val $x : t$
- module type T
  - type t
  - val $x : t$

Position-dependent meaning of syntax

Enriched syntax

$\rightarrow$ F^{ω} quantifiers

- Existential for ascription
- Universal for functors

Key mechanisms

- Existential lifting
- Skolemization

$\exists \alpha, \gamma'$ module M

- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \mathsf{int} \times \alpha$
- module $F : \forall \beta.(Y : \mathsf{sig\ type}\ t = \beta\ \mathsf{end})$
  - type $t = \gamma'(\beta)$
  - type $u = \beta$
- module G () : $\exists \delta$
  - type $t = \delta$
  - val $x : \delta$
- module type T
  - type t
  - val $x : t$

Position-dependent meaning of syntax

Enriched syntax

$\rightarrow$ F^{ω} quantifiers

- Existential for ascription
- Universal for functors
- Lambda for module types

Key mechanisms

- Existential lifting
- Skolemization

$\exists \alpha, \gamma'$ module M

- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \text{int} \times \alpha$
- module $F : \forall \beta.(Y : \text{sig type } t = \beta \text{ end})$
  - type $t = \gamma'(\beta)$
  - type $u = \beta$
- module G () : $\exists \delta$
  - type $t = \delta$
  - val $x : \delta$
- module type T
  - $\lambda \rho$
    - type $t = \rho$
    - val $x : t$

Position-dependent meaning of syntax

Enriched syntax

$\rightarrow$ F^{ω} quantifiers

- Existential for ascription
- Universal for functors
- Lambda for module types

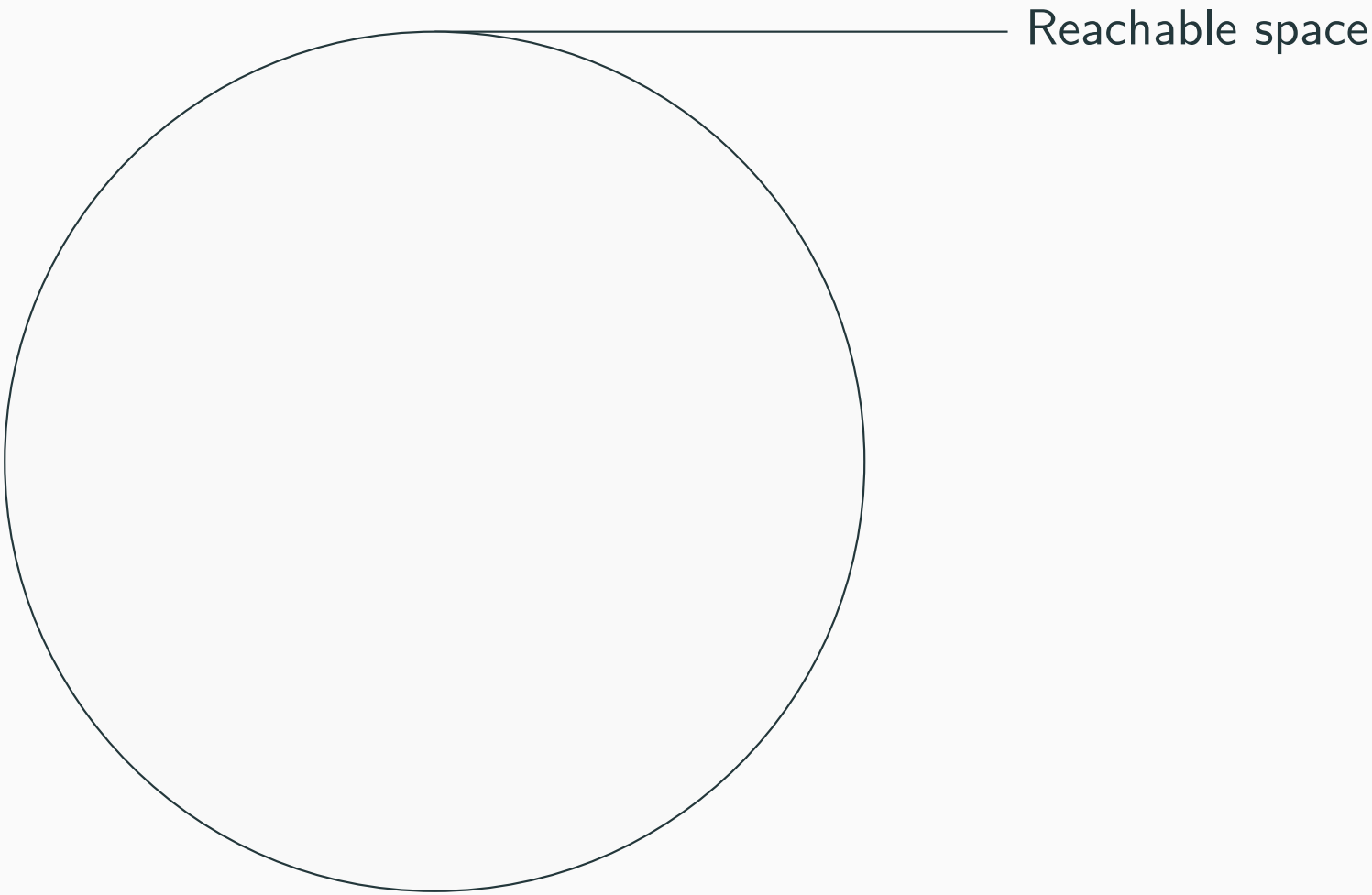
Key mechanisms

- Existential lifting
- Skolemization

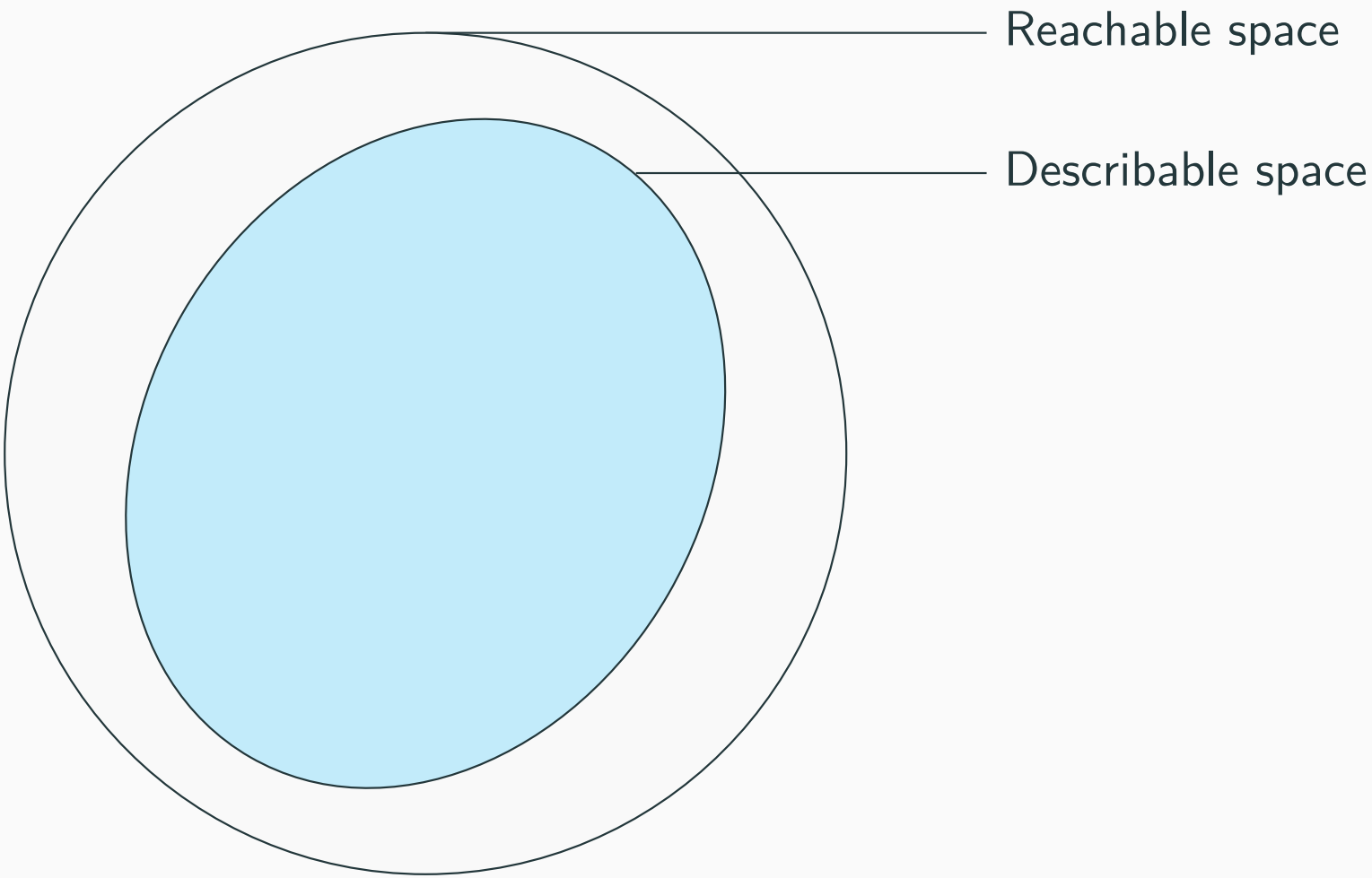
$\exists \alpha, \gamma'$ module M

- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \text{int} \times \alpha$
- module $F : \forall \beta.(Y : \text{sig type } t = \beta \text{ end})$
  - type $t = \gamma'(\beta)$
  - type $u = \beta$
- module G () : $\exists \delta$
  - type $t = \delta$
  - val $x : \delta$
- module type $T : \lambda \rho$
  - type $t = \rho$
  - val $x : \rho$

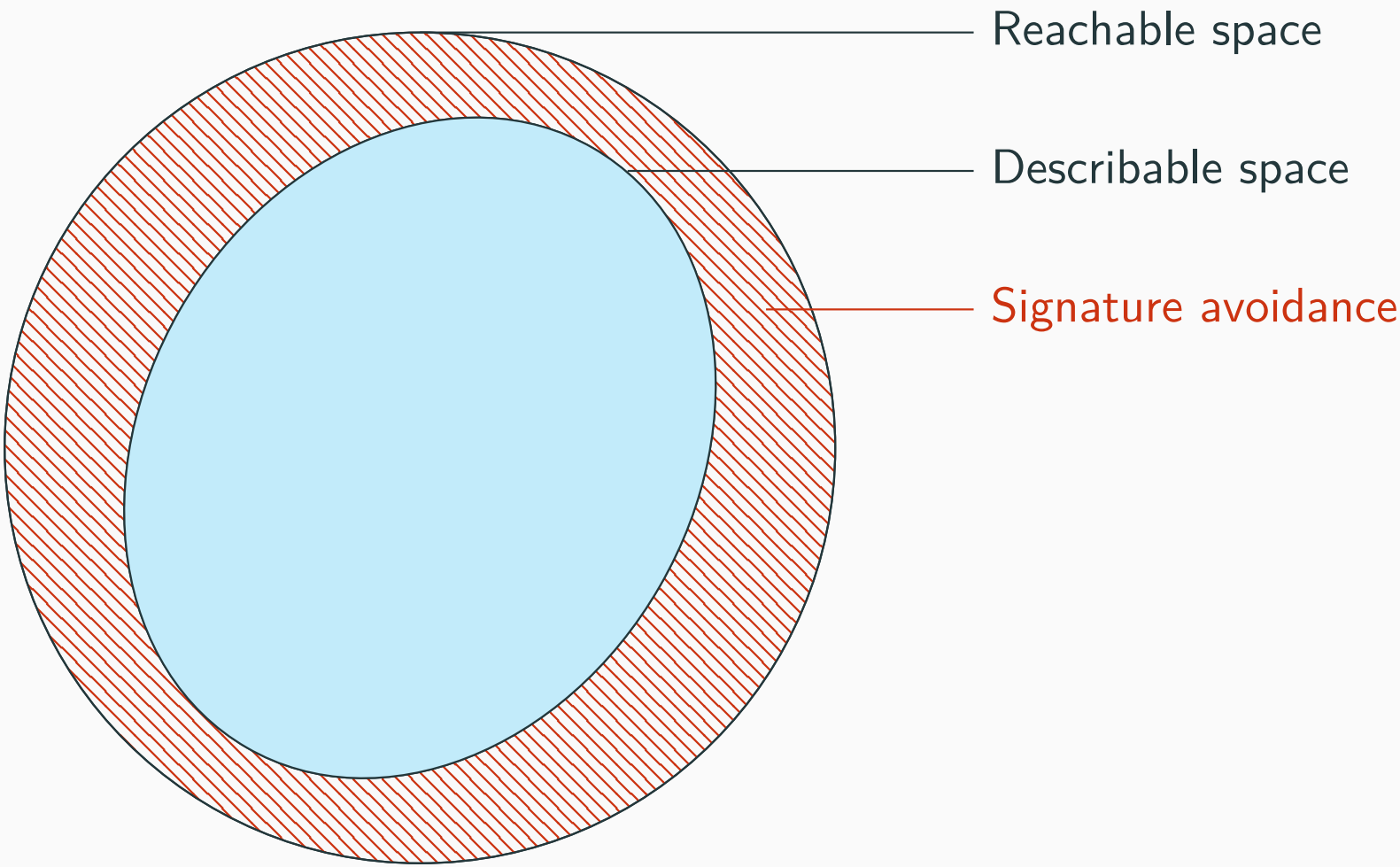
Expressivity mismatch



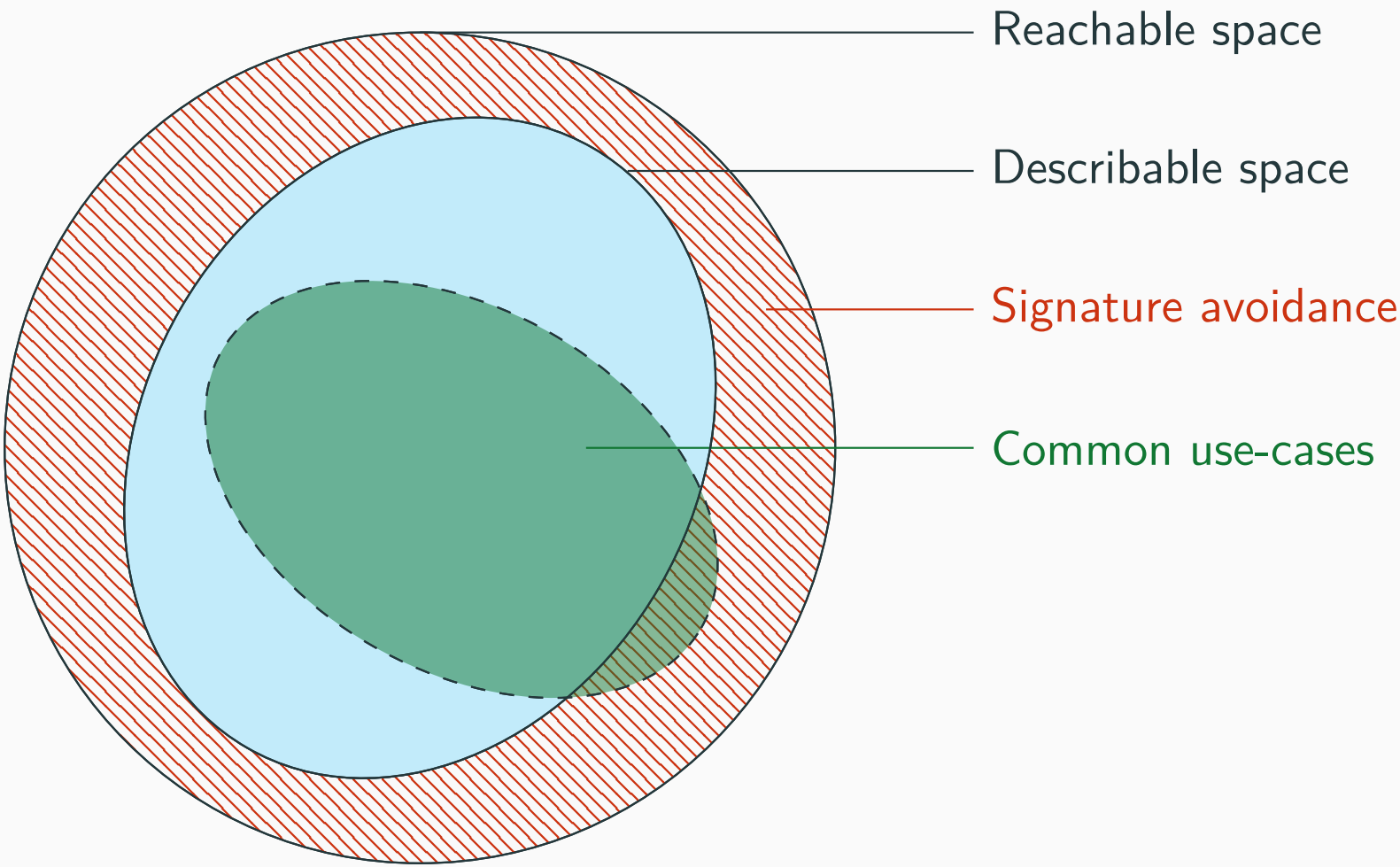
Expressivity mismatch



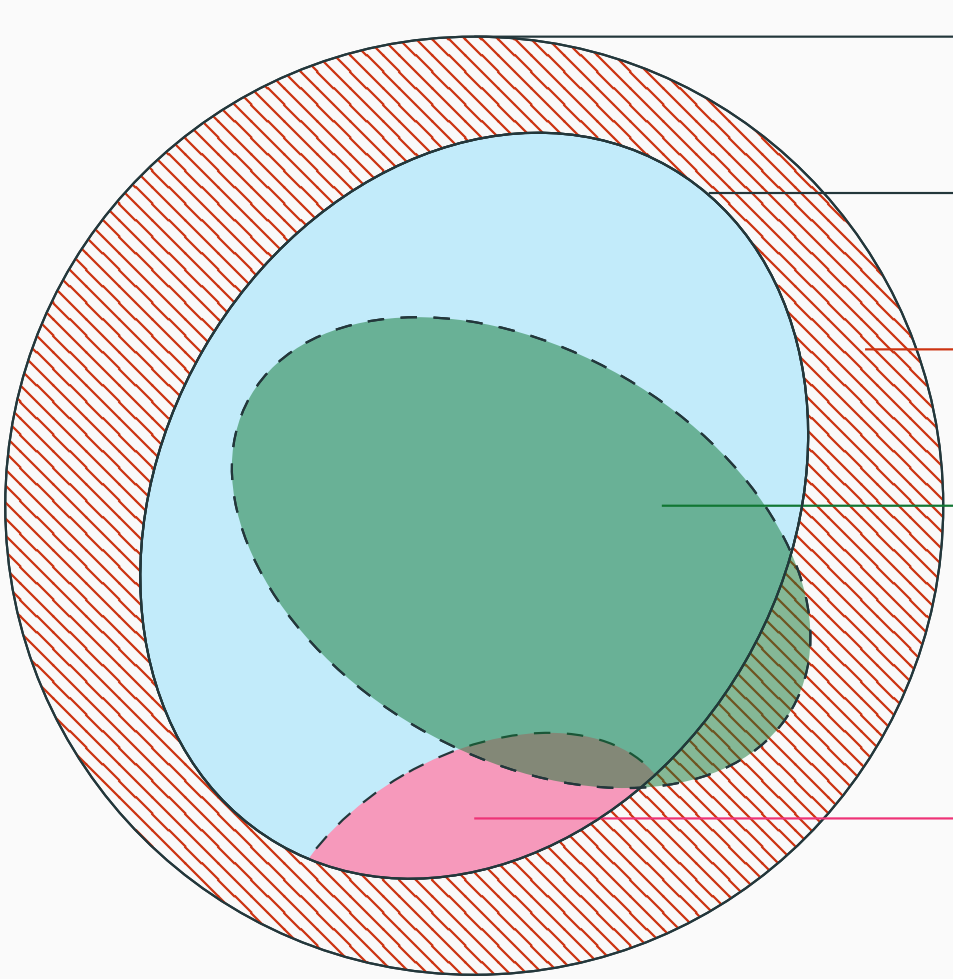
Expressivity mismatch



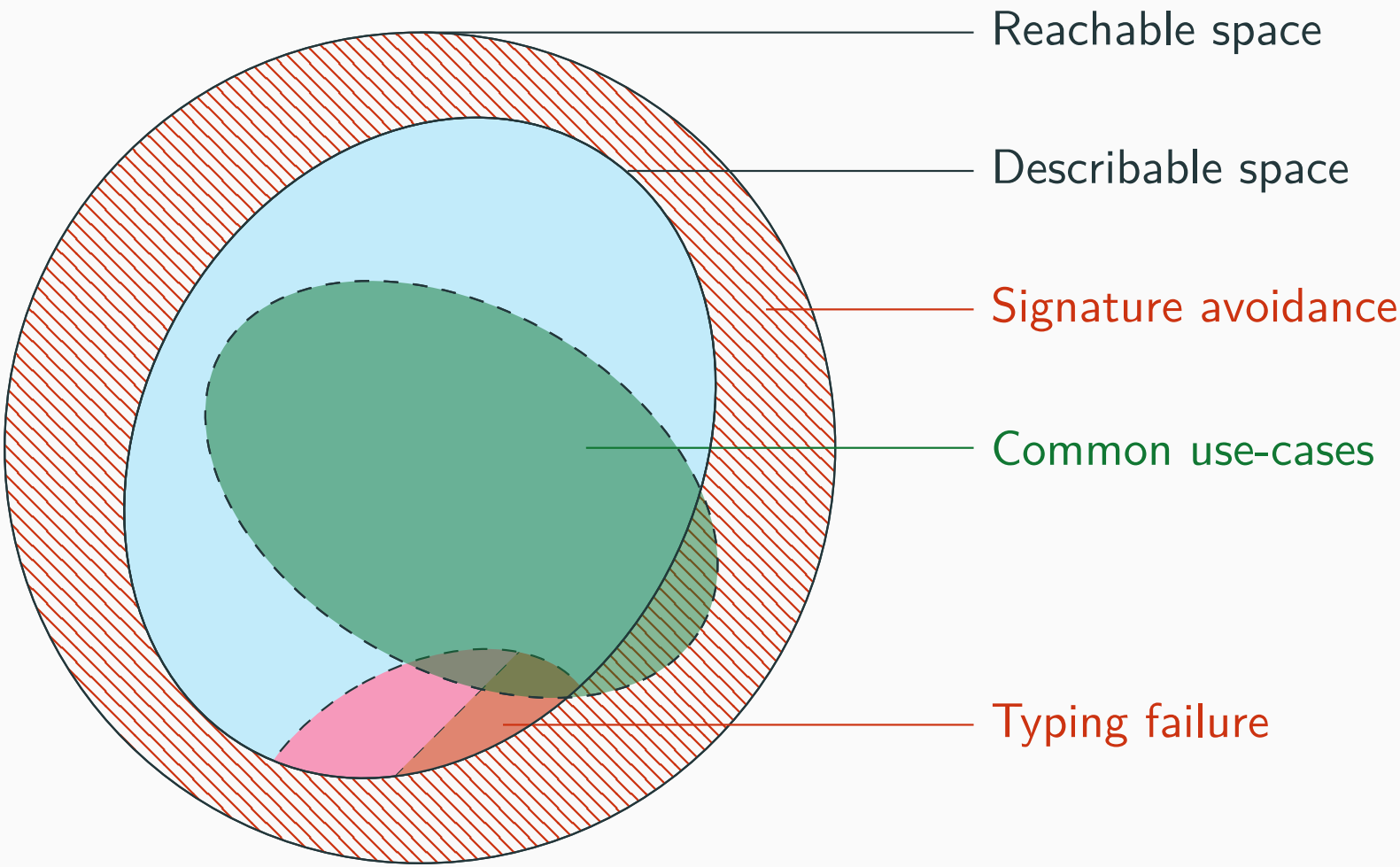
Expressivity mismatch



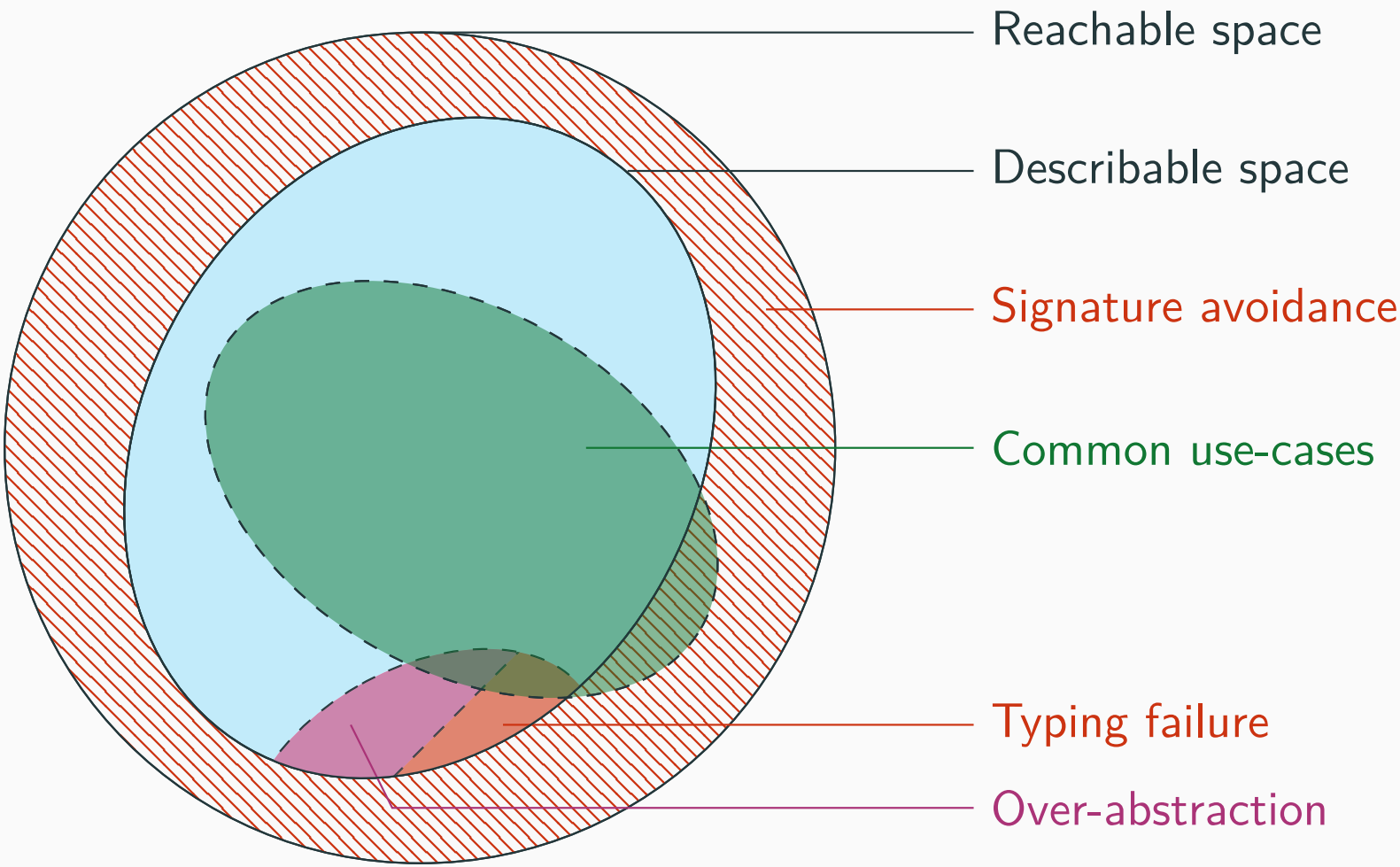
Expressivity mismatch



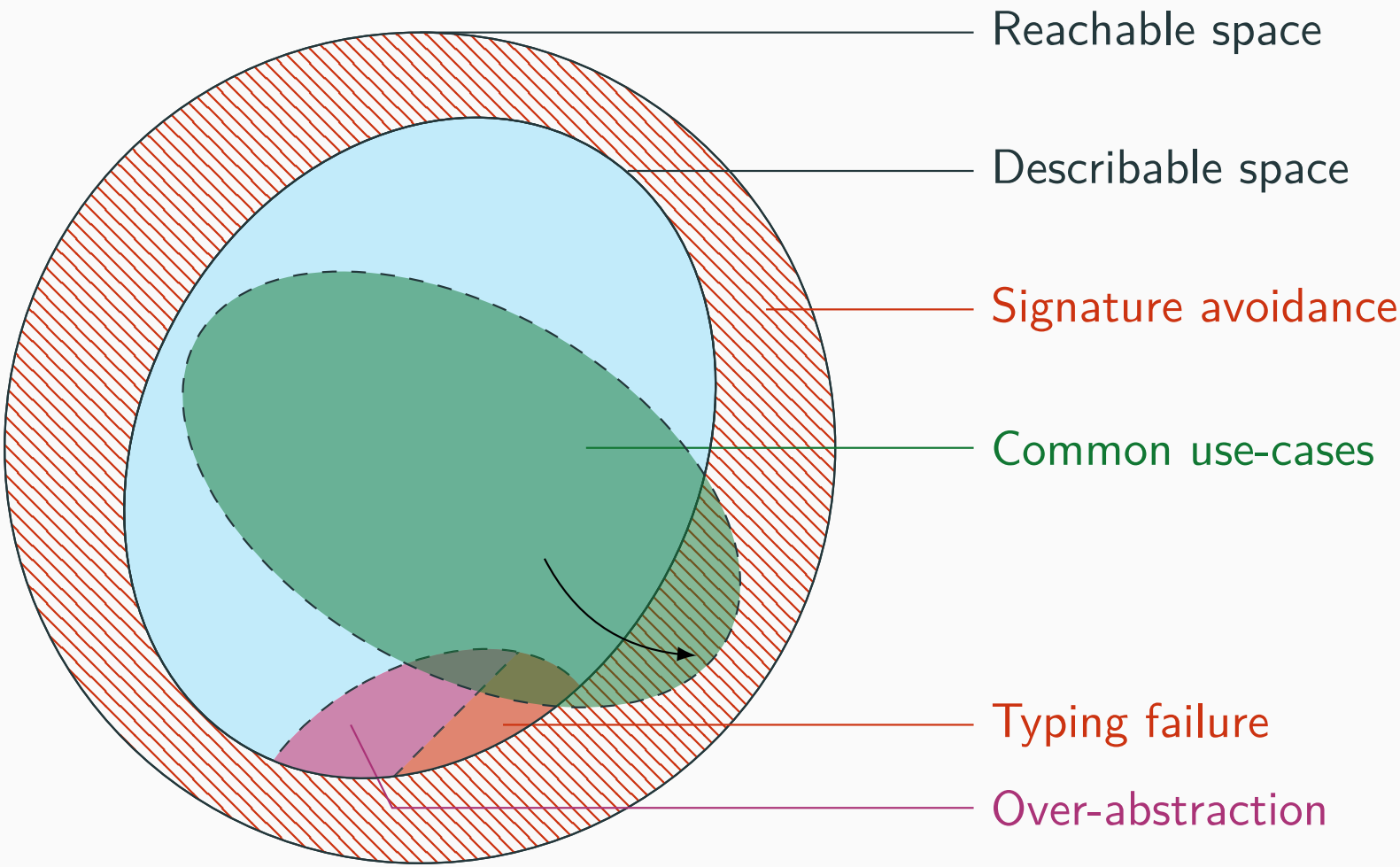
Expressivity mismatch



Expressivity mismatch



Expressivity mismatch



Conclusion

Conclusion

- Specify the OCaml module system via a translation in F^{ω}

Conclusion

- Specify the OCaml module system via a translation in F^{ω}
  - ✓ Applicative / Generative functors

Conclusion

- Specify the OCaml module system via a translation in F^{ω}
  - ✓ Applicative / Generative functors
  - ✓ Transparent ascription

Conclusion

- Specify the OCaml module system via a translation in F^{ω}
  - ✓ Applicative / Generative functors
  - ✓ Transparent ascription
  - Abstract signatures

Conclusion

- Specify the OCaml module system via a translation in F^{ω}
  - ✓ Applicative / Generative functors
  - ✓ Transparent ascription
  - Abstract signatures
  - Recursive modules

Conclusion

- Specify the OCaml module system via a translation in F^{ω}
  - ✓ Applicative / Generative functors
  - ✓ Transparent ascription
  - Abstract signatures
  - Recursive modules
- Rewrite the typechecker

Conclusion

- Specify the OCaml module system via a translation in F^{ω}
  - ✓ Applicative / Generative functors
  - ✓ Transparent ascription
  - Abstract signatures
  - Recursive modules
- Rewrite the typechecker
- Implement transparent ascription

Conclusion

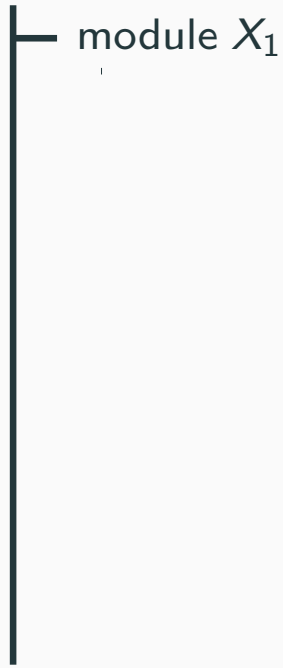
- Specify the OCaml module system via a translation in F^{ω}
  - ✓ Applicative / Generative functors
  - ✓ Transparent ascription
  - Abstract signatures
  - Recursive modules
- Rewrite the typechecker
- Implement transparent ascription
- Explore the challenges of having an hybrid syntax with quantifiers

Strengthening

```
module type S = sig
  type t
  val x : t
  type u = int * t
end
```

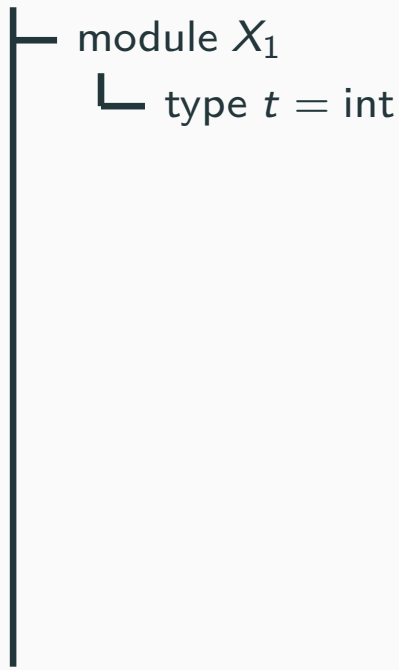
Strengthening

```
1  module type S = sig
2    type t
3    val x : t
4    type u = int * t
5  end
6
7  module X1      = struct
8
9
10
11
12
13
```



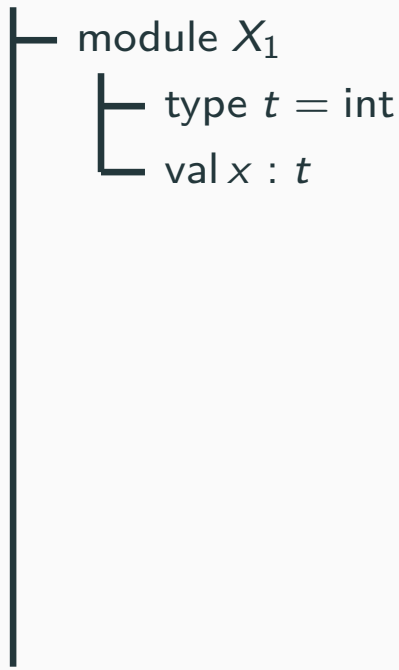
Strengthening

```
1  module type S = sig
2    type t
3    val x : t
4    type u = int * t
5  end
6
7  module X1      = struct
8    type t = int
9
10
11
12
13
```



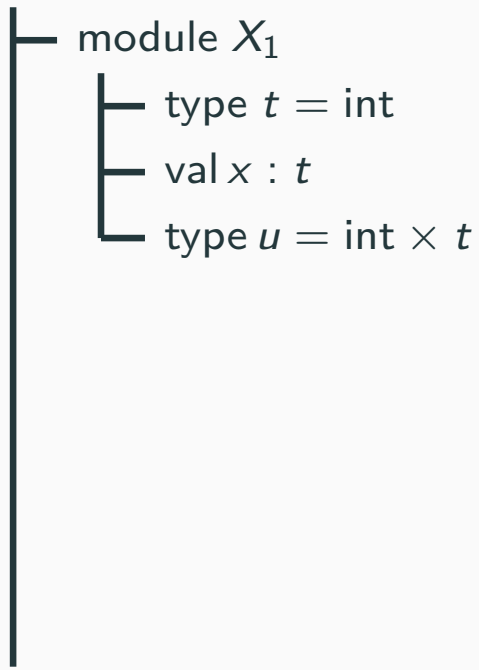
Strengthening

```
1  module type S = sig
2    type t
3    val x : t
4    type u = int * t
5  end
6
7  module X1      = struct
8    type t = int
9    let x : t = 42
10
11
12
13
```



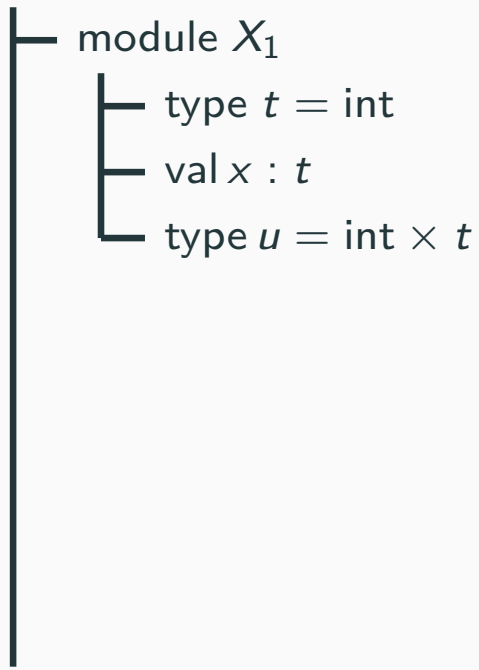
Strengthening

```
1  module type S = sig
2    type t
3    val x : t
4    type u = int * t
5  end
6
7  module X1      = struct
8    type t = int
9    let x : t = 42
10   type u = int * t
11 end
12
13
```



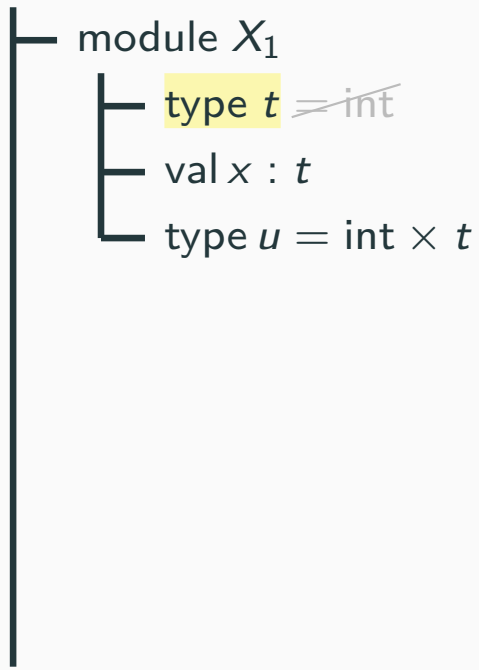
Strengthening

```
1  module type S = sig
2    type t
3    val x : t
4    type u = int * t
5  end
6
7  module X1      = struct
8    type t = int
9    let x : t = 42
10   type u = int * t
11 end
12
13
```



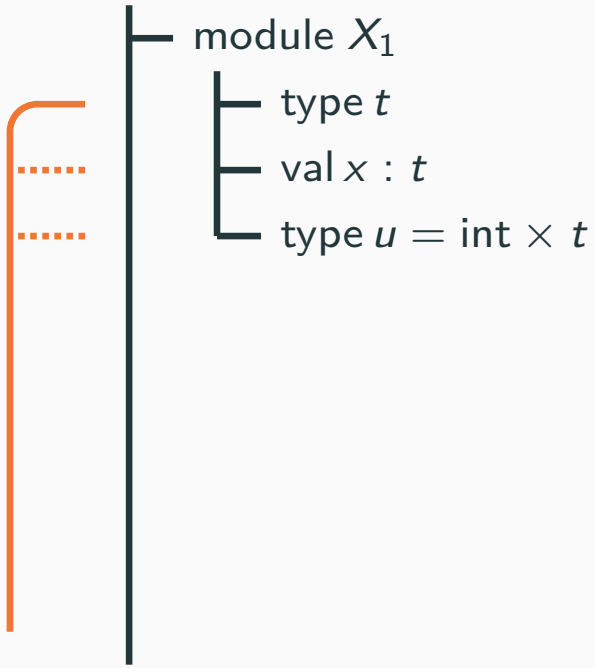
Strengthening

```
1  module type S = sig
2    type t
3    val x : t
4    type u = int * t
5  end
6
7  module X1 : S  = struct
8    type t = int
9    let x : t = 42
10   type u = int * t
11 end
12
13
```



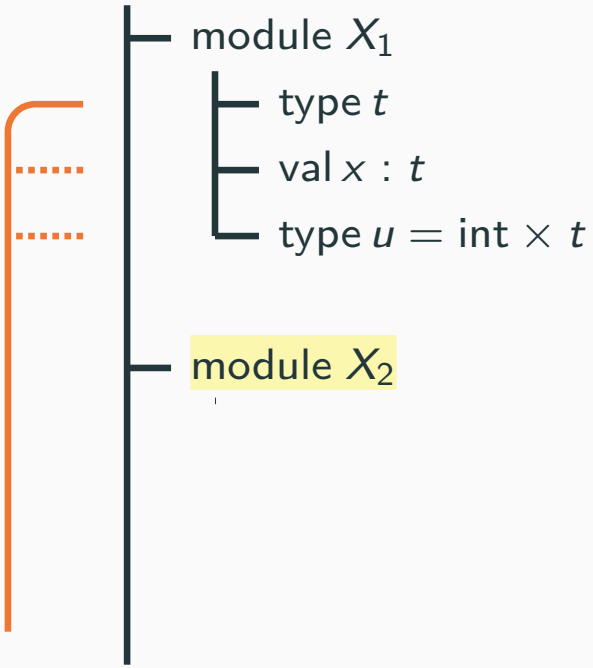
Strengthening

```
1  module type S = sig
2    type t
3    val x : t
4    type u = int * t
5  end
6
7  module X1 : S  = struct
8    type t = int
9    let x : t = 42
10   type u = int * t
11 end
12
13
```



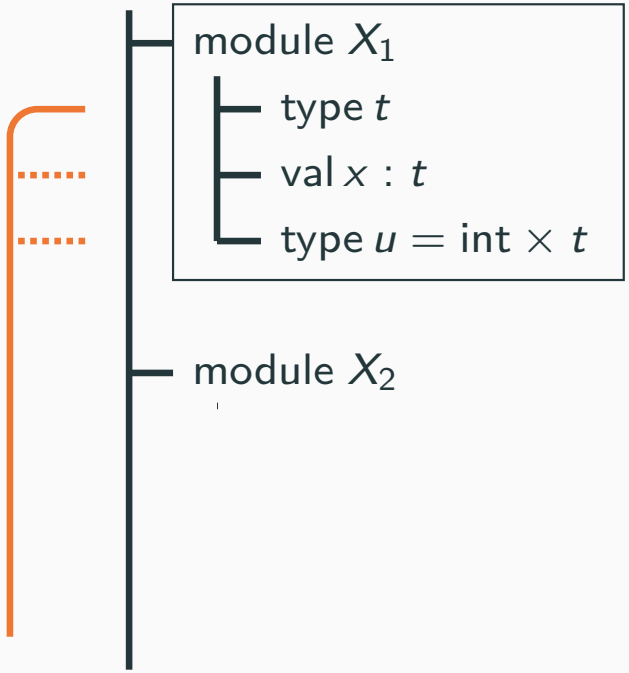
Strengthening

```
1  module type S = sig
2    type t
3    val x : t
4    type u = int * t
5  end
6
7  module X1 : S  = struct
8    type t = int
9    let x : t = 42
10   type u = int * t
11 end
12
13 module X2 = X1
```



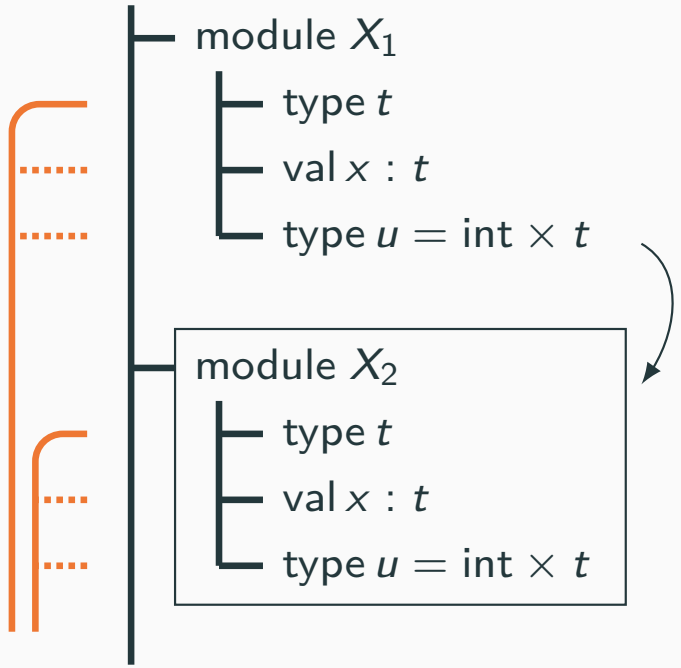
Strengthening

```
1  module type S = sig
2    type t
3    val x : t
4    type u = int * t
5  end
6
7  module X1 : S  = struct
8    type t = int
9    let x : t = 42
10   type u = int * t
11 end
12
13 module X2 = X1
```



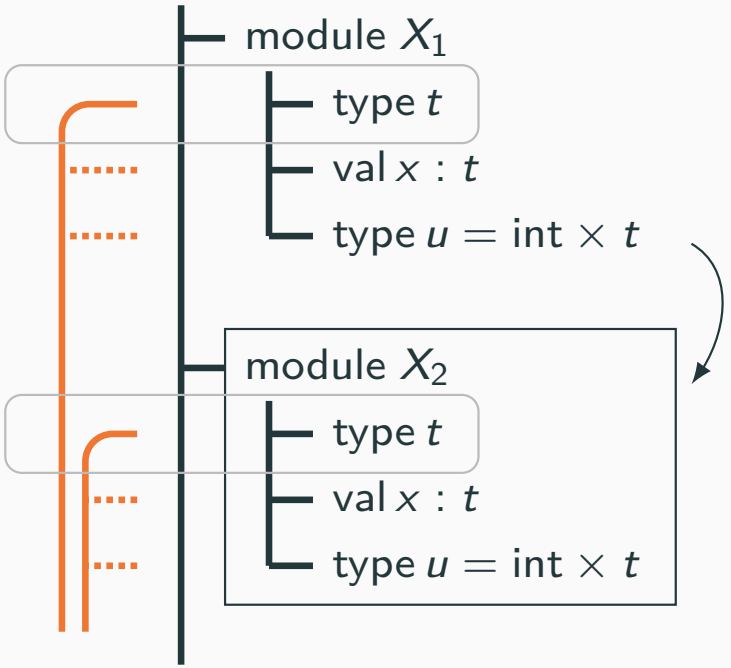
Strengthening

```
1  module type S = sig
2    type t
3    val x : t
4    type u = int * t
5  end
6
7  module X1 : S  = struct
8    type t = int
9    let x : t = 42
10   type u = int * t
11 end
12
13 module X2 = X1
```



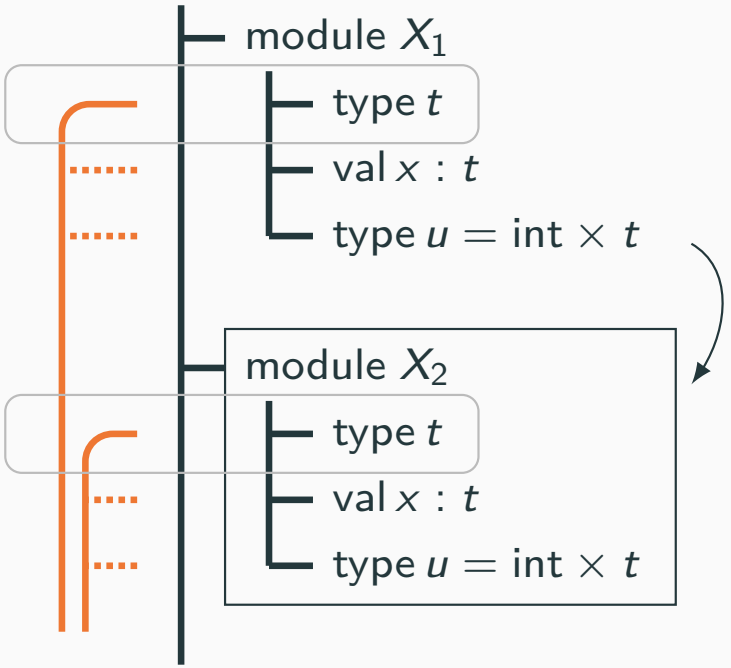
Strengthening

```
1  module type S = sig
2    type t
3    val x : t
4    type u = int * t
5  end
6
7  module X1 : S  = struct
8    type t = int
9    let x : t = 42
10   type u = int * t
11 end
12
13 module X2 = X1
```



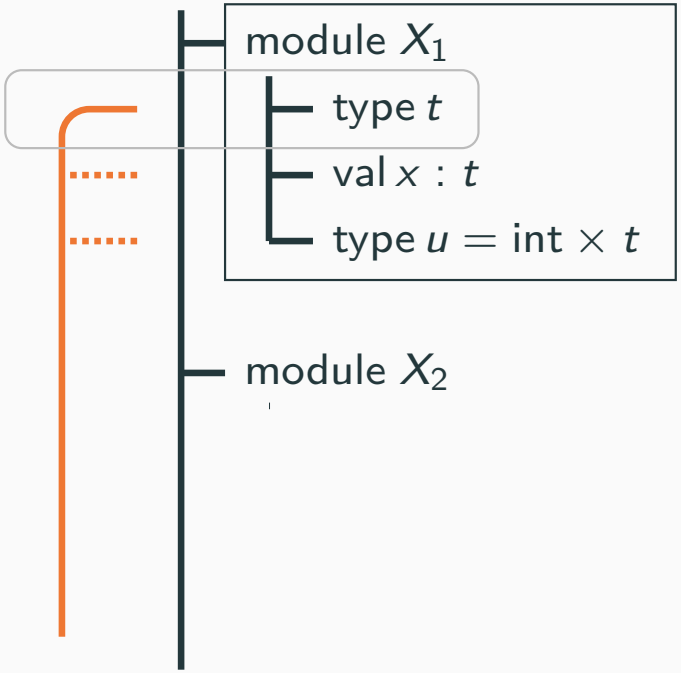
Strengthening

```
1  module type S = sig
2    type t
3    val x : t
4    type u = int * t
5  end
6
7  module X1 : S  = struct
8    type t = int
9    let x : t = 42
10   type u = int * t
11 end
12
13 module X2 = (X1:S)
```



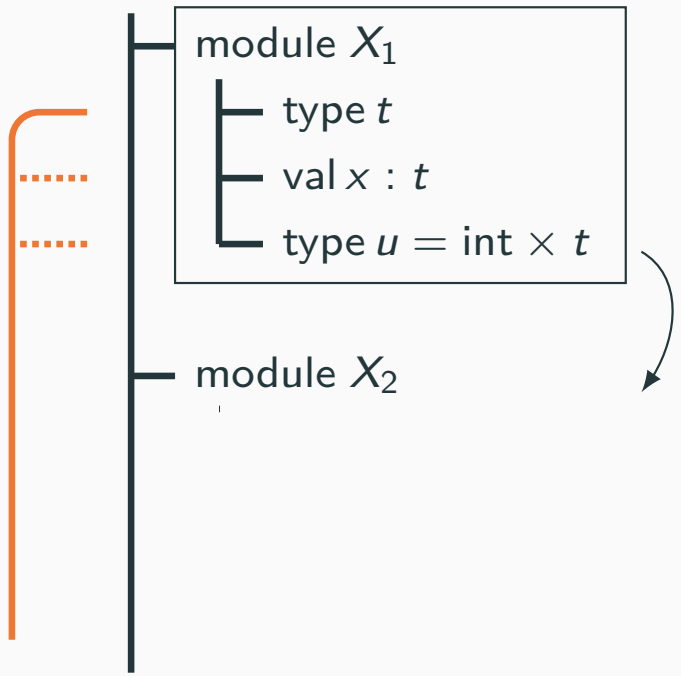
Strengthening

```
1  module type S = sig
2    type t
3    val x : t
4    type u = int * t
5  end
6
7  module X1 : S  = struct
8    type t = int
9    let x : t = 42
10   type u = int * t
11 end
12
13 module X2 = X1
```



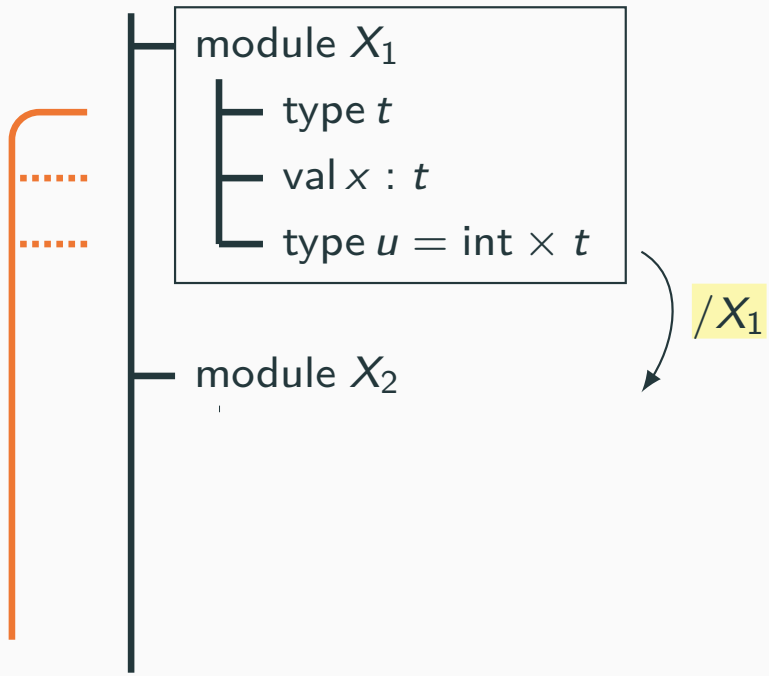
Strengthening

```
1  module type S = sig
2    type t
3    val x : t
4    type u = int * t
5  end
6
7  module X1 : S  = struct
8    type t = int
9    let x : t = 42
10   type u = int * t
11 end
12
13 module X2 = X1
```



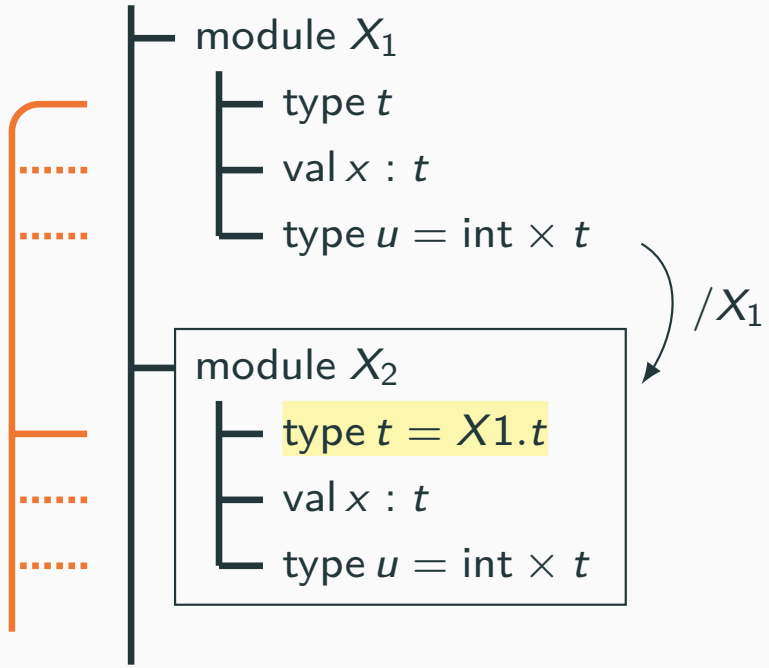
Strengthening

```
1  module type S = sig
2    type t
3    val x : t
4    type u = int * t
5  end
6
7  module X1 : S  = struct
8    type t = int
9    let x : t = 42
10   type u = int * t
11 end
12
13 module X2 = X1
```



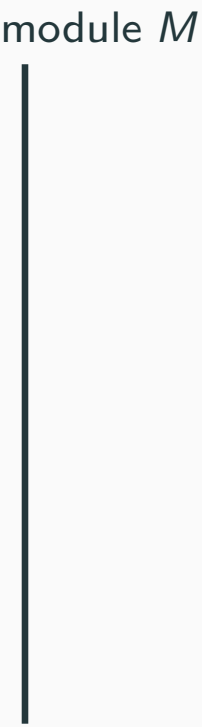
Strengthening

```
1  module type S = sig
2    type t
3    val x : t
4    type u = int * t
5  end
6
7  module X1 : S  = struct
8    type t = int
9    let x : t = 42
10   type u = int * t
11 end
12
13 module X2 = X1
```



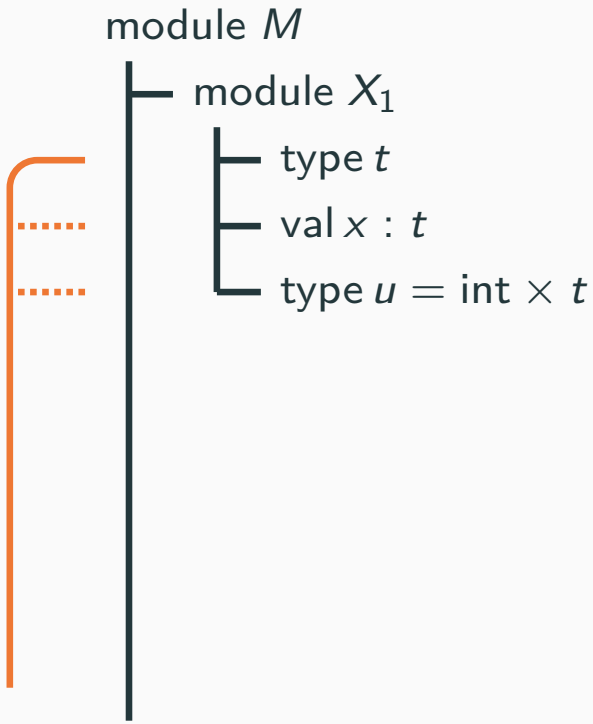
Signature avoidance

```
1 | module M =  struct
2 |
3 |
4 |
5 |
6 |
7 |
8 | end
```



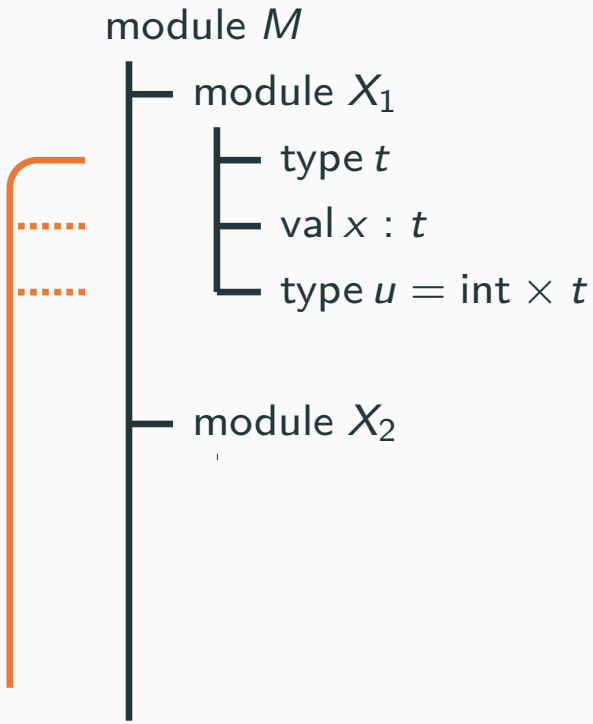
Signature avoidance

```
1  module M =  struct
2    module X1 : S = ...
3
4
5
6
7
8  end
```



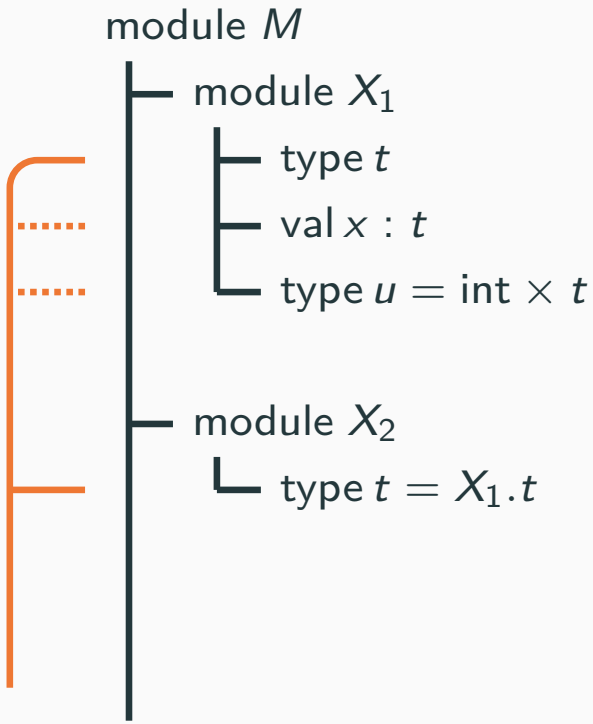
Signature avoidance

```
1  module M =  struct
2    module X1 : S = ...
3
4    module X2 = struct
5
6
7
8  end
```



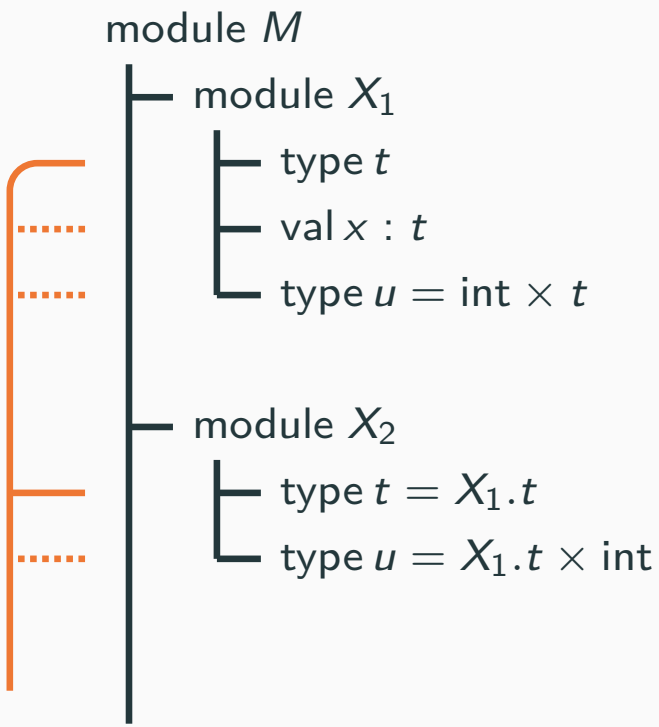
Signature avoidance

```
1  module M =  struct
2    module X1 : S = ...
3
4    module X2 = struct
5      type t = X1.t
6
7
8  end
```



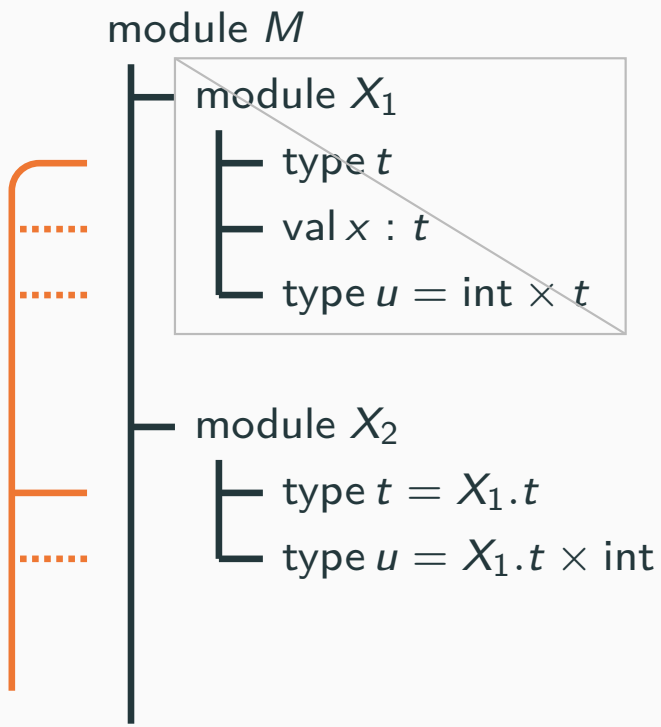
Signature avoidance

```
1  module M =  struct
2    module X1 : S = ...
3
4    module X2 = struct
5      type t = X1.t
6      type u = X1.t * int
7    end
8  end
```



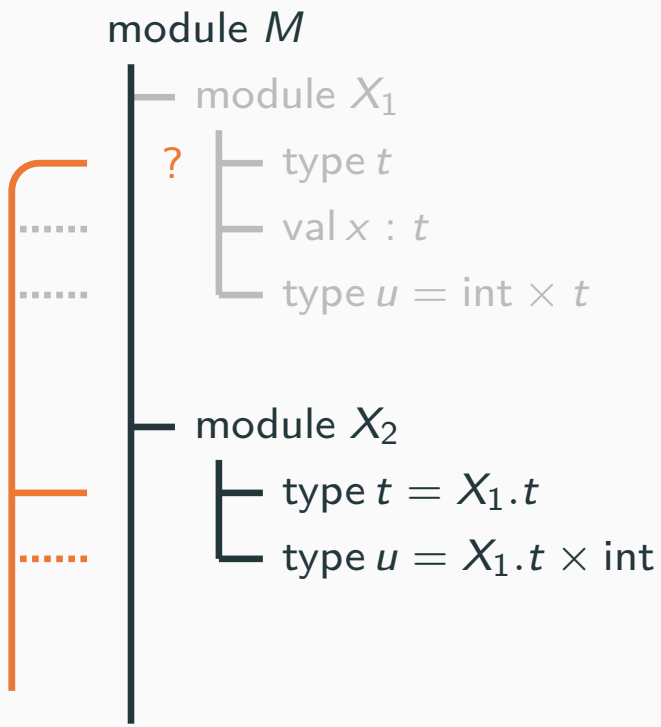
Signature avoidance

```
1  module M = (struct
2    module X1 : S = ...
3
4    module X2 = struct
5      type t = X1.t
6      type u = X1.t * int
7    end
8  end).X2
```



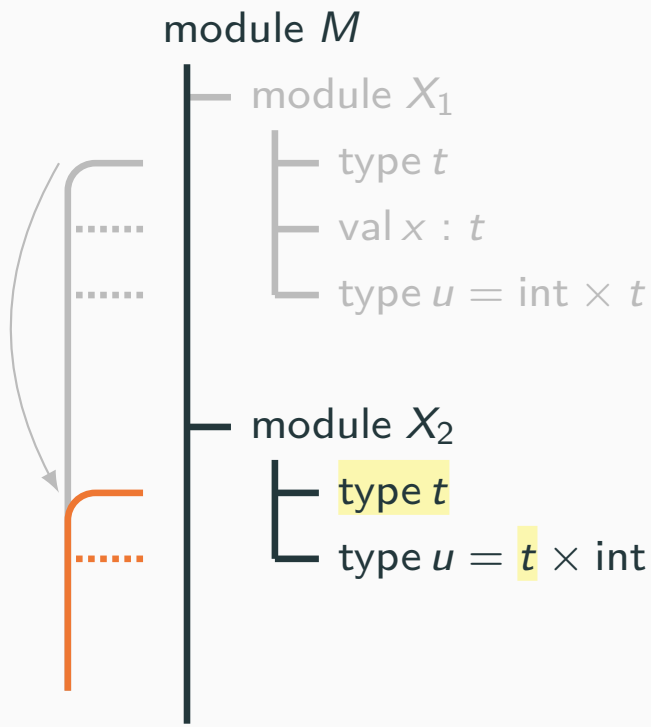
Signature avoidance

```
1 module M = (struct
2   module X1 : S = ...
3
4   module X2 = struct
5     type t = X1.t
6     type u = X1.t * int
7   end
8 end).X2
```



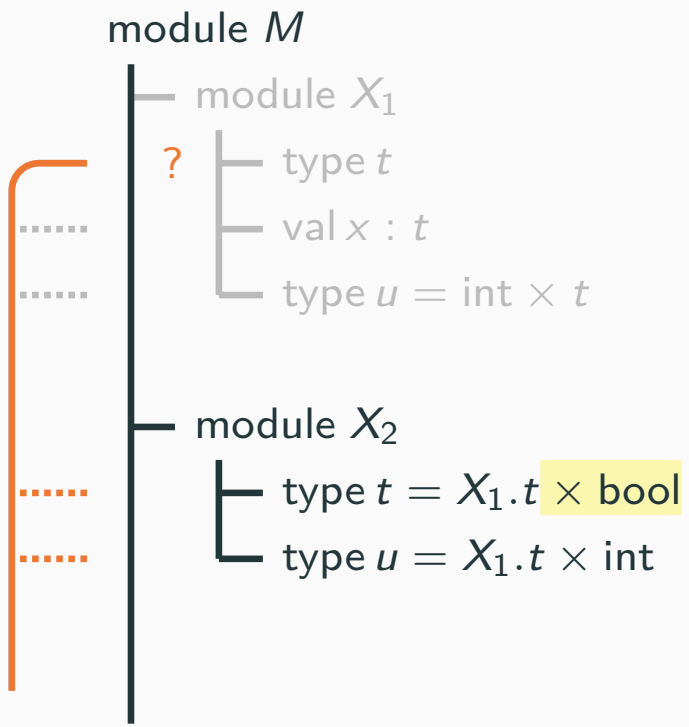
Signature avoidance

```
1  module M = (struct
2    module X1 : S = ...
3
4    module X2 = struct
5      type t = X1.t
6      type u = X1.t * int
7    end
8  end).X2
```



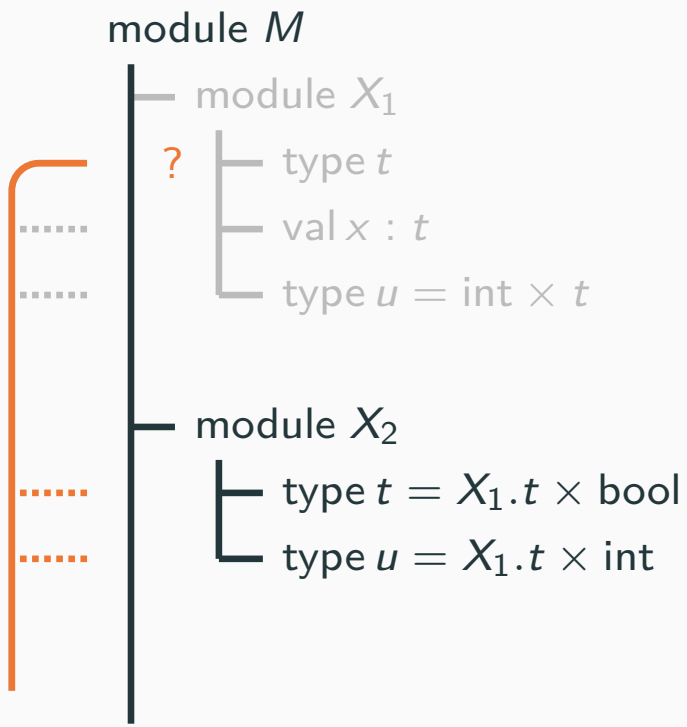
Signature avoidance

```
1  module M = (struct
2    module X1 : S = ...
3
4    module X2 = struct
5      type t = X1.t * bool
6      type u = X1.t * int
7    end
8  end).X2
```



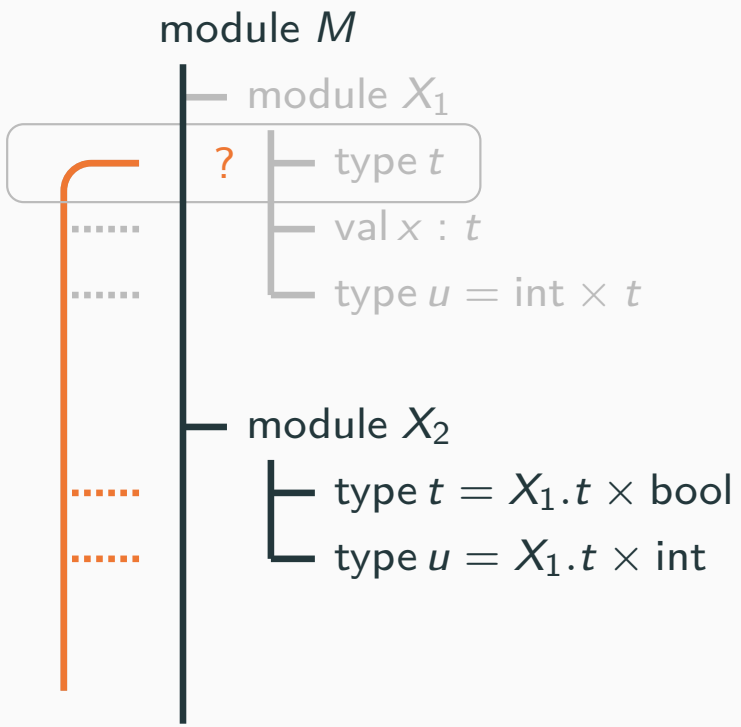
Signature avoidance

```
1  module M = (struct
2    module X1 : S = ...
3
4    module X2 = struct
5      type t = X1.t * bool
6      type u = X1.t * int
7    end
8  end).X2
```



Signature avoidance

```
1 | module M = (struct
2 |   module X1 : S = ...
3 |
4 |   module X2 = struct
5 |     type t = X1.t * bool
6 |     type u = X1.t * int
7 |   end
8 | end).X2
```



Signature avoidance

```
1  module M = (struct
2    module X1 : S = ...
3
4    module X2 = struct
5      type t = X1.t * bool
6      type u = X1.t * int
7    end
8  end).X2
```