Retrofiting OCaml modules

Séminaire EPFL *- 19/04/24*

Clément Blaudeau

clement.blaudeau [at] inria.fr

clement.blaudeau.net

What's up with OCaml modules ?

A powerful module system...

What's up with OCaml modules ?

A powerful module system...

- Contributed to OCaml's success

What's up with OCaml modules ?

A powerful module system...

- Contributed to OCaml's success
- Extensively used in practice

What's up with OCaml modules ?

A powerful module system...

- Contributed to OCaml's success
- Extensively used in practice
- Rich abstraction mechanisms

What's up with OCaml modules ?

A powerful module system...

- Contributed to OCaml's success
- Extensively used in practice
- Rich abstraction mechanisms

... that is unspecified...

What's up with OCaml modules ?

A powerful module system...

- Contributed to OCaml's success
- Extensively used in practice
- Rich abstraction mechanisms

... that is unspecified...

- unsupported edge-cases

What's up with OCaml modules ?

A powerful module system...

- Contributed to OCaml's success
- Extensively used in practice
- Rich abstraction mechanisms

... that is unspecified...

- unsupported edge-cases
- hard to predict

What's up with OCaml modules ?

A powerful module system...

- Contributed to OCaml's success
- Extensively used in practice
- Rich abstraction mechanisms

... that is unspecified...

- unsupported edge-cases
- hard to predict (signature avoidance)

What's up with OCaml modules ?

A powerful module system...

- Contributed to OCaml's success
- Extensively used in practice
- Rich abstraction mechanisms

... that is unspecified...

- unsupported edge-cases
- hard to predict (signature avoidance)
- hard to use

What's up with OCaml modules ?

A powerful module system...

- Contributed to OCaml's success
- Extensively used in practice
- Rich abstraction mechanisms

... that is unspecified...

- unsupported edge-cases
- hard to predict (signature avoidance)
- hard to use (recursive modules, abstract signatures)

What's up with OCaml modules ?

A powerful module system...

- Contributed to OCaml's success
- Extensively used in practice
- Rich abstraction mechanisms

... that is unspecified...

- unsupported edge-cases
- hard to predict (signature avoidance)
- hard to use (recursive modules, abstract signatures)
- hard to upgrade

What's up with OCaml modules ?

A powerful module system...

- Contributed to OCaml's success
- Extensively used in practice
- Rich abstraction mechanisms

... that is unspecified...

- unsupported edge-cases
- hard to predict (signature avoidance)
- hard to use (recursive modules, abstract signatures)
- hard to upgrade (transparent ascription, modular implicits)

What's up with OCaml modules ?

A powerful module system...

- Contributed to OCaml's success
- Extensively used in practice
- Rich abstraction mechanisms

... that is unspecified... yet !

- unsupported edge-cases
- hard to predict (signature avoidance)
- hard to use (recursive modules, abstract signatures)
- hard to upgrade (transparent ascription, modular implicits)

What's up with OCaml modules ?

A powerful module system...

- Contributed to OCaml' success
- Extensively used in practice
- Rich abstraction mechanisms

... that is unspecified... yet !

- unsupported edge-cases
- hard to predict (signature avoidance)
- hard to use (recursive modules, abstract signatures)
- hard to upgrade (transparent ascription, modular implicits)

In this presentation

In this presentation

- Tour of modularity in OCaml

In this presentation

- Tour of modularity in OCaml
  - What is *modularity* ?

In this presentation

- Tour of modularity in OCaml
  - What is *modularity* ?
  - Structures, signatures

In this presentation

- Tour of modularity in OCaml
  - What is *modularity* ?
  - Structures, signatures
  - Applicative vs Generative functors

In this presentation

- Tour of modularity in OCaml
  - What is *modularity* ?
  - Structures, signatures
  - Applicative vs Generative functors
  - Module aliases and transparent ascription

In this presentation

- Tour of modularity in OCaml
  - What is *modularity* ?
  - Structures, signatures
  - Applicative vs Generative functors
  - Module aliases and transparent ascription
- M^{ω}, a type system for OCaml

In this presentation

- Tour of modularity in OCaml
  - What is *modularity* ?
  - Structures, signatures
  - Applicative vs Generative functors
  - Module aliases and transparent ascription
- M^{ω}, a type system for OCaml
- Soundness by elaboration, with a focus on skolemization

In this presentation

- Tour of modularity in OCaml
  - What is *modularity* ?
  - Structures, signatures
  - Applicative vs Generative functors
  - Module aliases and transparent ascription
- M^{ω}, a type system for OCaml
- Soundness by elaboration, with a focus on skolemization
- Strengthening and signature avoidance

In this presentation

- Tour of modularity in OCaml
  - What is *modularity* ?
  - Structures, signatures
  - Applicative vs Generative functors
  - Module aliases and transparent ascription
- M^{ω}, a type system for OCaml
- Soundness by elaboration, with a focus on skolemization
- Strengthening and signature avoidance

In this presentation

- Tour of modularity in OCaml
  - What is *modularity* ?
  - Structures, signatures
  - Applicative vs Generative functors
  - Module aliases and transparent ascription
- M^{ω}, a type system for OCaml
- Soundness by elaboration, with a focus on skolemization
- Strengthening and signature avoidance

Disclaimer

This presentation relies on a rich history of previous work not cited here!

In this presentation

- Tour of modularity in OCaml
  - What is *modularity* ?
  - Structures, signatures
  - Applicative vs Generative functors
  - Module aliases and transparent ascription
- M^{ω}, a type system for OCaml
- Soundness by elaboration, with a focus on skolemization
- Strengthening and signature avoidance

Disclaimer

This presentation relies on a rich history of previous work not cited here!

In this presentation

- Tour of modularity in OCaml
  - What is *modularity* ?
  - Structures, signatures
  - Applicative vs Generative functors
  - Module aliases and transparent ascription
- M^{ω}, a type system for OCaml
- Soundness by elaboration, with a focus on skolemization
- Strengthening and signature avoidance

Disclaimer

This presentation relies on a rich history of previous work not cited here!
Notably, *F-ing modules* by Rossberg, Russo & Dreyer.

In this presentation

- Tour of modularity in OCaml
  - What is *modularity* ?
  - Structures, signatures
  - Applicative vs Generative functors
  - Module aliases and transparent ascription
- M^{ω}, a type system for OCaml
- Soundness by elaboration, with a focus on skolemization
- Strengthening and signature avoidance

Disclaimer

This presentation relies on a rich history of previous work not cited here! Notably, *F-ing modules* by Rossberg, Russo & Dreyer. See our OOPSLA2024 paper for more details!

The essence of modularity

The essence of modularity

Modularity for complex systems

The essence of modularity

Modularity for complex systems

- Identify and factorize common parts

The essence of modularity

Modularity for complex systems

- Identify and factorize common parts

The essence of modularity

Modularity for complex systems

- Identify and factorize common parts $\rightarrow$ *reusability*

The essence of modularity

Modularity for complex systems

- Identify and factorize common parts $\rightarrow$ *reusability*
- Gather components into *modules* and describe the interactions between modules (strong dependencies inside a module, limited dependencies between modules)

The essence of modularity

Modularity for complex systems

- Identify and factorize common parts $\rightarrow$ *reusability*
- Gather components into *modules* and describe the interactions between modules (strong dependencies inside a module, limited dependencies between modules)

The essence of modularity

Modularity for complex systems

- Identify and factorize common parts $\rightarrow$ *reusability*
- Gather components into *modules* and describe the interactions between modules (strong dependencies inside a module, limited dependencies between modules) $\rightarrow$ *structure*

The essence of modularity

Modularity for complex systems

- Identify and factorize common parts $\rightarrow$ *reusability*
- Gather components into *modules* and describe the interactions between modules (strong dependencies inside a module, limited dependencies between modules) $\rightarrow$ *structure*

The tools of modularity

The essence of modularity

Modularity for complex systems

- Identify and factorize common parts $\rightarrow$ *reusability*
- Gather components into *modules* and describe the interactions between modules (strong dependencies inside a module, limited dependencies between modules) $\rightarrow$ *structure*

The tools of modularity

- Well defined interfaces

The essence of modularity

Modularity for complex systems

- Identify and factorize common parts $\rightarrow$ *reusability*
- Gather components into *modules* and describe the interactions between modules (strong dependencies inside a module, limited dependencies between modules) $\rightarrow$ *structure*

The tools of modularity

- Well defined interfaces
- Abstraction

The essence of modularity

Modularity for complex systems

- Identify and factorize common parts $\rightarrow$ *reusability*
- Gather components into *modules* and describe the interactions between modules (strong dependencies inside a module, limited dependencies between modules) $\rightarrow$ *structure*

The tools of modularity

- Well defined interfaces
- Abstraction

The essence of modularity

Modularity for complex systems

- Identify and factorize common parts → *reusability*
- Gather components into *modules* and describe the interactions between modules (strong dependencies inside a module, limited dependencies between modules) → *structure*

The tools of modularity

- Well defined interfaces
- Abstraction

Modularity in PL

The essence of modularity

Modularity for complex systems

- Identify and factorize common parts → *reusability*
- Gather components into *modules* and describe the interactions between modules (strong dependencies inside a module, limited dependencies between modules) → *structure*

The tools of modularity

- Well defined interfaces
- Abstraction

Modularity in PL

- Functions

The essence of modularity

Modularity for complex systems

- Identify and factorize common parts → *reusability*
- Gather components into *modules* and describe the interactions between modules (strong dependencies inside a module, limited dependencies between modules) → *structure*

The tools of modularity

- Well defined interfaces
- Abstraction

Modularity in PL

- Functions
- Code/Interfaces files

The essence of modularity

Modularity for complex systems

- Identify and factorize common parts → *reusability*
- Gather components into *modules* and describe the interactions between modules (strong dependencies inside a module, limited dependencies between modules) → *structure*

The tools of modularity

- Well defined interfaces
- Abstraction

Modularity in PL

- Functions
- Code/Interfaces files
- Object oriented programming

The essence of modularity

Modularity for complex systems

- Identify and factorize common parts $\rightarrow$ *reusability*
- Gather components into *modules* and describe the interactions between modules (strong dependencies inside a module, limited dependencies between modules) $\rightarrow$ *structure*

The tools of modularity

- Well defined interfaces
- Abstraction

Modularity in PL

- Functions
- Code/Interfaces files
- Object oriented programming
- Type-classes

The essence of modularity

Modularity for complex systems

- Identify and factorize common parts $\rightarrow$ *reusability*
- Gather components into *modules* and describe the interactions between modules (strong dependencies inside a module, limited dependencies between modules) $\rightarrow$ *structure*

The tools of modularity

- Well defined interfaces
- Abstraction

Modularity in PL

- Functions
- Code/Interfaces files
- Object oriented programming
- Type-classes
- ML-modules

The essence of modularity

Modularity for complex systems

- Identify and factorize common parts → *reusability*
- Gather components into *modules* and describe the interactions between modules (strong dependencies inside a module, limited dependencies between modules) → *structure*

The tools of modularity

- Well defined interfaces
- Abstraction

Modularity in PL

- Functions
- Code/Interfaces files
- Object oriented programming
- Type-classes
- ML-modules
- Package managers

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

```
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
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    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

```
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
```

✓ Interface control

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

✓ 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

```
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
```

✓ Interface control

✓ Abstraction

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))
```

✓ 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
- ✓ Structural composition

The ML approach[1]

[1] not machine-learning

The ML approach[1]

Module terms (construction)

[1] not machine-learning

The ML approach[1]

Module terms (construction)

- Records of values, types, modules and module types

[1] not machine-learning

The ML approach[1]

Module terms (construction)

- Records of values, types, modules and module types `struct ... end`

[1] not machine-learning

The ML approach[1]

Module terms (construction)

- Records of values, types, modules and module types `struct ... end`
- Client-side abstraction : functors

[1] not machine-learning

The ML approach[1]

Module terms (construction)

- Records of values, types, modules and module types `struct ... end`
- Client-side abstraction : functors `functor (Y:S) -> M`

[1] not machine-learning

The ML approach[1]

Module terms (construction)

- Records of values, types, modules and module types `struct ... end`
- Client-side abstraction : functors `functor (Y:S) -> M`
- Developer-side abstraction : sealing

[1] not machine-learning

The ML approach[1]

Module terms (construction)

- Records of values, types, modules and module types `struct ... end`
- Client-side abstraction : functors `functor (Y:S) -> M`
- Developer-side abstraction : sealing `(M : S)`

[1] not machine-learning

The ML approach[1]

Module terms (construction)

- Records of values, types, modules and module types `struct ... end`
- Client-side abstraction : functors `functor (Y:S) -> M`
- Developer-side abstraction : sealing `(M : S)`
- A small calculus (projections, applications, recursion, etc.)

[1] not machine-learning

The ML approach[1]

Module terms (construction)

- Records of values, types, modules and module types `struct ... end`
- Client-side abstraction : functors `functor (Y:S) -> M`
- Developer-side abstraction : sealing `(M : S)`
- A small calculus (projections, applications, recursion, etc.) `M.X, F(X), ...`

[1] not machine-learning

The ML approach[1]

Module terms (construction)

- Records of values, types, modules and module types `struct ... end`
- Client-side abstraction : functors `functor (Y:S) -> M`
- Developer-side abstraction : sealing `(M : S)`
- A small calculus (projections, applications, recursion, etc.) `M.X, F(X), ...`

[1] not machine-learning

The ML approach[1]

Module terms (construction)

- Records of values, types, modules and module types `struct ... end`
- Client-side abstraction : functors `functor (Y:S) -> M`
- Developer-side abstraction : sealing `(M : S)`
- A small calculus (projections, applications, recursion, etc.) `M.X, F(X), ...`

Module types (description)

[1] not machine-learning

The ML approach[1]

Module terms (construction)

- Records of values, types, modules and module types `struct ... end`
- Client-side abstraction : functors `functor (Y:S) -> M`
- Developer-side abstraction : sealing `(M : S)`
- A small calculus (projections, applications, recursion, etc.) `M.X, F(X), ...`

Module types (description)

- User-defined *signatures*

[1] not machine-learning

The ML approach[1]

Module terms (construction)

- Records of values, types, modules and module types `struct ... end`
- Client-side abstraction : functors `functor (Y:S) -> M`
- Developer-side abstraction : sealing `(M : S)`
- A small calculus (projections, applications, recursion, etc.) `M.X, F(X), ...`

Module types (description)

- User-defined *signatures* `sig ... end, functor (Y:S) -> ...`

[1] not machine-learning

The ML approach[1]

Module terms (construction)

- Records of values, types, modules and module types `struct ... end`
- Client-side abstraction : functors `functor (Y:S) -> M`
- Developer-side abstraction : sealing `(M : S)`
- A small calculus (projections, applications, recursion, etc.) `M.X, F(X), ...`

Module types (description)

- User-defined *signatures* `sig ... end, functor (Y:S) -> ...`
- Concrete types

[1] not machine-learning

The ML approach[1]

Module terms (construction)

- Records of values, types, modules and module types `struct ... end`
- Client-side abstraction : functors `functor (Y:S) -> M`
- Developer-side abstraction : sealing `(M : S)`
- A small calculus (projections, applications, recursion, etc.) `M.X, F(X), ...`

Module types (description)

- User-defined *signatures* `sig ... end, functor (Y:S) -> ...`
- Concrete types `type t = int * bool`

[1] not machine-learning

The ML approach[1]

Module terms (construction)

- Records of values, types, modules and module types `struct ... end`
- Client-side abstraction : functors `functor (Y:S) -> M`
- Developer-side abstraction : sealing `(M : S)`
- A small calculus (projections, applications, recursion, etc.) `M.X, F(X), ...`

Module types (description)

- User-defined *signatures* `sig ... end, functor (Y:S) -> ...`
- Concrete types `type t = int * bool`
- Abstract types

[1] not machine-learning

The ML approach[1]

Module terms (construction)

- Records of values, types, modules and module types `struct ... end`
- Client-side abstraction : functors `functor (Y:S) -> M`
- Developer-side abstraction : sealing `(M : S)`
- A small calculus (projections, applications, recursion, etc.) `M.X, F(X), ...`

Module types (description)

- User-defined *signatures* `sig ... end, functor (Y:S) -> ...`
- Concrete types `type t = int * bool`
- Abstract types `type t`

[1] not machine-learning

The ML approach[1]

Module terms (construction)

- Records of values, types, modules and module types `struct ... end`
- Client-side abstraction : functors `functor (Y:S) -> M`
- Developer-side abstraction : sealing `(M : S)`
- A small calculus (projections, applications, recursion, etc.) `M.X, F(X), ...`

Module types (description)

- User-defined *signatures* `sig ... end, functor (Y:S) -> ...`
- Concrete types `type t = int * bool`
- Abstract types `type t`

[1] not machine-learning

The ML approach[1]

Module terms (construction)

- Records of values, types, modules and module types `struct ... end`
- Client-side abstraction : functors `functor (Y:S) -> M`
- Developer-side abstraction : sealing `(M : S)`
- A small calculus (projections, applications, recursion, etc.) `M.X, F(X), ...`

Module types (description)

- User-defined *signatures* `sig ... end, functor (Y:S) -> ...`
- Concrete types `type t = int * bool`
- Abstract types `type t`

modules: explicit, rich types

[1] not machine-learning

The ML approach[1]

Module terms (construction)

- Records of values, types, modules and module types — `struct ... end`
- Client-side abstraction : functors — `functor (Y:S) -> M`
- Developer-side abstraction : sealing — `(M : S)`
- A small calculus (projections, applications, recursion, etc.) — `M.X, F(X), ...`

Module types (description)

- User-defined *signatures* — `sig ... end, functor (Y:S) -> ...`
- Concrete types — `type t = int * bool`
- Abstract types — `type t`

modules: explicit, rich types $\iff$ *core*: implicit, simpler types

[1] not machine-learning

How does it relate to Scala ?

[2]Still not machine-learning

How does it relate to Scala ?

- ML[2] signatures $\sim$ Scala traits

[2]Still not machine-learning

How does it relate to Scala ?

- ML[2] signatures $\sim$ Scala traits
- ML modules $\sim$ Scala objects

[2] Still not machine-learning

How does it relate to Scala ?

- ML[2] signatures $\sim$ Scala traits
- ML modules $\sim$ Scala objects
- ML functors $\sim$ Scala functions

[2] Still not machine-learning

How does it relate to Scala ?

- ML[2] signatures $\sim$ Scala traits
- ML modules $\sim$ Scala objects
- ML functors $\sim$ Scala functions
- ML structural subtyping $\neq$ Scala nominal subtyping

[2] Still not machine-learning

How does it relate to Scala ?

- ML[2] signatures $\sim$ Scala traits
- ML modules $\sim$ Scala objects
- ML functors $\sim$ Scala functions
- ML structural subtyping $\neq$ Scala nominal subtyping
- ML non-recursive (by default) $\neq$ Scala Object system

[2] Still not machine-learning

Abstraction and functors

What flavor for you functor ?

What flavor for you functor ?

Generative

What flavor for you functor ?

Generative

Applicative

What flavor for you functor ?

Generative

Parameterized *stateful* modules

Applicative

What flavor for you functor ?

Generative

Parameterized *stateful* modules

- Internal state / effects

Applicative

What flavor for you functor ?

Generative

Parameterized *stateful* modules

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

Applicative

What flavor for you functor ?

Generative

Parameterized *stateful* modules

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

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

Applicative

What flavor for you functor ?

Generative

Parameterized *stateful* modules

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

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

Applicative

What flavor for you functor ?

Generative

Parameterized *stateful* modules

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

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

Applicative

What flavor for you functor ?

Generative

Parameterized *stateful* modules

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
1  module SymbolTable() = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module ST1 = SymboleTable() (* Private keys *)
8  module ST2 = SymboleTable() (* Public keys *)
9
```

Applicative

What flavor for you functor ?

Generative

Parameterized *stateful* modules

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
1  module SymbolTable() = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module ST1 = SymboleTable() (* Private keys *)
8  module ST2 = SymboleTable() (* Public keys *)
9  (* ST1.t ≠ ST2.t *) ×
```

Applicative

What flavor for you functor ?

Generative

Parameterized *stateful* modules

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
1  module SymbolTable() = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module ST1 = SymboleTable() (* Private keys *)
8  module ST2 = SymboleTable() (* Public keys *)
9  (* ST1.t ≠ ST2.t *) ×
```

Applicative

Parameterized *stateless* modules

What flavor for you functor ?

Generative

Parameterized *stateful* modules

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
1  module SymbolTable() = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module ST1 = SymboleTable() (* Private keys *)
8  module ST2 = SymboleTable() (* Public keys *)
9  (* ST1.t ≠ ST2.t *) ×
```

Applicative

Parameterized *stateless* modules

- Purity

What flavor for you functor ?

Generative

Parameterized *stateful* modules

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
1  module SymbolTable() = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module ST1 = SymboleTable() (* Private keys *)
8  module ST2 = SymboleTable() (* Public keys *)
9  (* ST1.t ≠ ST2.t *) ×
```

Applicative

Parameterized *stateless* modules

- Purity
- Static choice of implementation

What flavor for you functor ?

Generative

Parameterized *stateful* modules

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
1  module SymbolTable() = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module ST1 = SymboleTable() (* Private keys *)
8  module ST2 = SymboleTable() (* Public keys *)
9  (* ST1.t ≠ ST2.t *) ×
```

Applicative

Parameterized *stateless* modules

- Purity
- Static choice of implementation

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

What flavor for you functor ?

Generative

Parameterized *stateful* modules

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
1  module SymbolTable() = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module ST1 = SymboleTable() (* Private keys *)
8  module ST2 = SymboleTable() (* Public keys *)
9  (* ST1.t ≠ ST2.t *) ×
```

Applicative

Parameterized *stateless* modules

- Purity
- Static choice of implementation

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

What flavor for you functor ?

Generative

Parameterized *stateful* modules

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
1  module SymbolTable() = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module ST1 = SymboleTable() (* Private keys *)
8  module ST2 = SymboleTable() (* Public keys *)
9  (* ST1.t ≠ ST2.t *) ×
```

Applicative

Parameterized *stateless* modules

- Purity
- Static choice of implementation

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

What flavor for you functor ?

Generative

Parameterized *stateful* modules

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
1  module SymbolTable() = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module ST1 = SymboleTable() (* Private keys *)
8  module ST2 = SymboleTable() (* Public keys *)
9  (* ST1.t ≠ ST2.t *) ×
```

Applicative

Parameterized *stateless* modules

- Purity
- Static choice of implementation

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

What flavor for you functor ?

Generative

Parameterized *stateful* modules

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
1  module SymbolTable() = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module ST1 = SymboleTable() (* Private keys *)
8  module ST2 = SymboleTable() (* Public keys *)
9  (* ST1.t ≠ ST2.t *) ×
```

Applicative

Parameterized *stateless* modules

- Purity
- Static choice of implementation

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

What flavor for you functor ?

Generative

Parameterized *stateful* modules

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
1  module SymbolTable() = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module ST1 = SymboleTable() (* Private keys *)
8  module ST2 = SymboleTable() (* Public keys *)
9  (* ST1.t ≠ ST2.t *) ×
```

Applicative

Parameterized *stateless* modules

- Purity
- Static choice of implementation

→ *same applications* produce same results

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

What flavor for you functor ?

Generative

Parameterized *stateful* modules

- Internal state / effects
- Dynamic choice of implementation (via 1st class modules)

```
1  module SymbolTable() = (struct
2    type t = int
3    let x = ref 0
4    ...
5  end : sig type t ... end)
6
7  module ST1 = SymboleTable() (* Private keys *)
8  module ST2 = SymboleTable() (* Public keys *)
9  (* ST1.t ≠ ST2.t *) ×
```

Applicative

Parameterized *stateless* modules

- Purity
- Static choice of implementation

→ *same applications* produce same results

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

Applicativity granularity & Abstraction safety

```
module Set (E: Ordered) = (struct
  type t = E.t list (* ordered *)
  let rec insert (x:E.t) (s:t) =
  match x with
  | [] -> [x]
  | y::s' -> if E.compare y x
     then y::(insert x s')
     else x::y::s'
  ...
end : sig type t (* abstract *) ... end)
```

```
module X1 = struct
  type t = int
  ...
end

module X2 = struct
  type t = int
  ...
end

Set(X1).t =? Set(X2).t
```

Applicativity granularity & Abstraction safety

Types only

```
module Set (E: Ordered) = (struct
  type t = E.t list (* ordered *)
  let rec insert (x:E.t) (s:t) =
  match x with
  | [] -> [x]
  | y::s' -> if E.compare y x
     then y::(insert x s')
     else x::y::s'
  ...
end : sig type t (* abstract *) ... end)
```

```
module X1 = struct
  type t = int
  ...
end

module X2 = struct
  type t = int
  ...
end

Set(X1).t =? Set(X2).t
```

Applicativity granularity & Abstraction safety

Types only

- Sound: types only depend on types

```
module Set (E: Ordered) = (struct
  type t = E.t list (* ordered *)
  let rec insert (x:E.t) (s:t) =
  match x with
  | [] -> [x]
  | y::s' -> if E.compare y x
     then y::(insert x s')
     else x::y::s'
  ...
end : sig type t (* abstract *) ... end)
```

```
module X1 = struct
  type t = int
  ...
end

module X2 = struct
  type t = int
  ...
end

Set(X1).t =? Set(X2).t
```

Applicativity granularity & Abstraction safety

Types only

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

```
module Set (E: Ordered) = (struct
  type t = E.t list (* ordered *)
  let rec insert (x:E.t) (s:t) =
  match x with
  | [] -> [x]
  | y::s' -> if E.compare y x
     then y::(insert x s')
     else x::y::s'
  ...
end : sig type t (* abstract *) ... end)
```

```
module X1 = struct
  type t = int
  ...
end

module X2 = struct
  type t = int
  ...
end

Set(X1).t =? Set(X2).t
```

Applicativity granularity & Abstraction safety

Types only

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

```
1  module Set (E: Ordered) = (struct
2    type t = E.t list (* ordered *)
3    let rec insert (x:E.t) (s:t) =
4    match x with
5    | [] -> [x]
6    | y::s' -> if E.compare y x
7       then y::(insert x s')
8       else x::y::s'
9    ...
10 end : sig type t (* abstract *) ... end)
11
```

```
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 Set(X1).t =? Set(X2).t
```

Applicativity granularity & Abstraction safety

Types only

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

Types and values

```
module Set (E: Ordered) = (struct
  type t = E.t list (* ordered *)
  let rec insert (x:E.t) (s:t) =
  match x with
  | [] -> [x]
  | y::s' -> if E.compare y x
     then y::(insert x s')
     else x::y::s'
  ...
end : sig type t (* abstract *) ... end)
```

```
module X1 = struct
  type t = int
  let compare = (<)
end

module X2 = struct
  type t = int
  let compare = (>)
end

Set(X1).t =? Set(X2).t
```

Applicativity granularity & Abstraction safety

Types only

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

Types and values

- *Abstraction safe*: dynamically equivalent

```
1  module Set (E: Ordered) = (struct
2    type t = E.t list (* ordered *)
3    let rec insert (x:E.t) (s:t) =
4    match x with
5    | [] -> [x]
6    | y::s' -> if E.compare y x
7       then y::(insert x s')
8       else x::y::s'
9    ...
10 end : sig type t (* abstract *) ... end)
11
```

```
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 Set(X1).t =? Set(X2).t
```

Applicativity granularity & Abstraction safety

Types only

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

Types and values

- *Abstraction safe*: dynamically equivalent
  $\rightarrow$ tracking equality of values

```
1  module Set (E: Ordered) = (struct
2    type t = E.t list (* ordered *)
3    let rec insert (x:E.t) (s:t) =
4    match x with
5    | [] -> [x]
6    | y::s' -> if E.compare y x
7       then y::(insert x s')
8       else x::y::s'
9    ...
10 end : sig type t (* abstract *) ... end)
11
```

```
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 Set(X1).t =? Set(X2).t
```

Applicativity granularity & Abstraction safety

Types only

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

Types and values

- *Abstraction safe*: dynamically equivalent
  $\rightarrow$ tracking equality of values

```
1  module Set (E: Ordered) = (struct
2    type t = E.t list (* ordered *)
3    let rec insert (x:E.t) (s:t) =
4    match x with
5    | [] -> [x]
6    | y::s' -> if E.compare y x
7       then y::(insert x s')
8       else x::y::s'
9    ...
10 end : sig type t (* abstract *) ... end)
11
```

```
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 Set(X1).t =? Set(X2).t
```

Applicativity granularity & Abstraction safety

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*: dynamically equivalent
  $\rightarrow$ tracking equality of values

```
1  module Set (E: Ordered) = (struct
2    type t = E.t list (* ordered *)
3    let rec insert (x:E.t) (s:t) =
4    match x with
5    | [] -> [x]
6    | y::s' -> if E.compare y x
7       then y::(insert x s')
8       else x::y::s'
9    ...
10 end : sig type t (* abstract *) ... end)
11
```

```
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 Set(X1).t =? Set(X2).t
```

Applicativity granularity & Abstraction safety

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*: dynamically equivalent
  $\rightarrow$ tracking equality of values

Modules - *coarse grained*

```
1  module Set (E: Ordered) = (struct
2    type t = E.t list (* ordered *)
3    let rec insert (x:E.t) (s:t) =
4    match x with
5    | [] -> [x]
6    | y::s' -> if E.compare y x
7       then y::(insert x s')
8       else x::y::s'
9    ...
10 end : sig type t (* abstract *) ... end)
11
```

```
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 Set(X1).t =? Set(X2).t
```

Applicativity granularity & Abstraction safety

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*: dynamically equivalent
  $\rightarrow$ tracking equality of values

Modules - *coarse grained*

- Tracking equality of modules

```
1  module Set (E: Ordered) = (struct
2    type t = E.t list (* ordered *)
3    let rec insert (x:E.t) (s:t) =
4    match x with
5    | [] -> [x]
6    | y::s' -> if E.compare y x
7       then y::(insert x s')
8       else x::y::s'
9    ...
10 end : sig type t (* abstract *) ... end)
11
```

```
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 Set(X1).t =? Set(X2).t
```

Applicativity granularity & Abstraction safety

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*: dynamically equivalent
  $\rightarrow$ tracking equality of values

Modules - *coarse grained*

- Tracking equality of modules
  $\rightarrow$ syntactic criterion

```
1  module Set (E: Ordered) = (struct
2    type t = E.t list (* ordered *)
3    let rec insert (x:E.t) (s:t) =
4    match x with
5    | [] -> [x]
6    | y::s' -> if E.compare y x
7       then y::(insert x s')
8       else x::y::s'
9    ...
10 end : sig type t (* abstract *) ... end)
11
```

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

Applicativity granularity & Abstraction safety

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*: dynamically equivalent
  $\rightarrow$ tracking equality of values

Modules - *coarse grained*

- Tracking equality of modules
  $\rightarrow$ syntactic criterion

```
1  module Set (E: Ordered) = (struct
2    type t = E.t list (* ordered *)
3    let rec insert (x:E.t) (s:t) =
4    match x with
5    | [] -> [x]
6    | y::s' -> if E.compare y x
7       then y::(insert x s')
8       else x::y::s'
9    ...
10 end : sig type t (* abstract *) ... end)
11
```

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

Applicativity granularity & Abstraction safety

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*: dynamically equivalent
  → tracking equality of values

Modules - *coarse grained*

- Tracking equality of modules
  → syntactic criterion
  → module *aliasing*

```
1  module Set (E: Ordered) = (struct
2    type t = E.t list (* ordered *)
3    let rec insert (x:E.t) (s:t) =
4    match x with
5    | [] -> [x]
6    | y::s' -> if E.compare y x
7       then y::(insert x s')
8       else x::y::s'
9    ...
10 end : sig type t (* abstract *) ... end)
11
```

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

Applicativity granularity & Abstraction safety

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*: dynamically equivalent
  → tracking equality of values

Modules - *coarse grained*

- Tracking equality of modules
  → syntactic criterion
  → module *aliasing*

```
1  module Set (E: Ordered) = (struct
2    type t = E.t list (* ordered *)
3    let rec insert (x:E.t) (s:t) =
4    match x with
5    | [] -> [x]
6    | y::s' -> if E.compare y x
7       then y::(insert x s')
8       else x::y::s'
9    ...
10 end : sig type t (* abstract *) ... end)
11
```

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

Module aliases and compilation schemes

ex.ml (code)

```
1 module X : T = struct ... end
2
3
4
5
6
7
```

ex.mli (types)

```
1
2
3
4
5
6
```

Module aliases and compilation schemes

ex.ml (code)

```
1 module X : T = struct ... end
2
3
4
5
6
7
```

ex.mli (types)

```
1
2
3
4
5
6
```

Module aliases and compilation schemes

ex.ml (code)

```
1 module X : T = struct ... end
2 module F = functor (Y:S) -> struct
3
4
5
6
7
```

ex.mli (types)

```
1
2
3
4
5
6
```

Module aliases and compilation schemes

ex.ml (code)

```
1 module X : T = struct ... end
2 module F = functor (Y:S) -> struct
3   ...
4
5
6
7
```

ex.mli (types)

```
1
2
3
4
5
6
```

Module aliases and compilation schemes

ex.ml (code)

```
1 module X : T = struct ... end
2 module F = functor (Y:S) -> struct
3   ...
4   module Arg = Y (* re-export *)
5
6
7
```

ex.mli (types)

```
1
2
3
4
5
6
```

Module aliases and compilation schemes

ex.ml (code)

```
1 module X : T = struct ... end
2 module F = functor (Y:S) -> struct
3   ...
4   module Arg = Y (* re-export *)
5 end
6
7
```

ex.mli (types)

```
1
2
3
4
5
6
```

Module aliases and compilation schemes

ex.ml (code)

```
1 module X : T = struct ... end
2 module F = functor (Y:S) -> struct
3   ...
4   module Arg = Y (* re-export *)
5 end
6
7 module X' = F(X).Arg
```

ex.mli (types)

```
1
2
3
4
5
6
```

Module aliases and compilation schemes

ex.ml (code)

```
1 module X : T = struct ... end
2 module F = functor (Y:S) -> struct
3   ...
4   module Arg = Y (* re-export *)
5 end
6
7 module X' = F(X).Arg
```

ex.mli (types)

```
1 module X : T
2 module F : functor (Y:S) -> sig
3   ...
4   module Arg : ? (* re-export *)
5 end
6
7 module X' : ?
```

Module aliases and compilation schemes

Aliases and functors

- Intuitively, `module X' = X`

ex.ml (code)

```
1 module X : T = struct ... end
2 module F = functor (Y:S) -> struct
3   ...
4   module Arg = Y (* re-export *)
5 end
6
7 module X' = F(X).Arg
```

ex.mli (types)

```
1 module X : T
2 module F : functor (Y:S) -> sig
3   ...
4   module Arg : ? (* re-export *)
5 end
6
7 module X' : (= X)
```

Module aliases and compilation schemes

Aliases and functors

- Intuitively, `module X' = X`
- *F* should re-export *whole* of its argument

ex.ml (code)

```
1 module X : T = struct ... end
2 module F = functor (Y:S) -> struct
3   ...
4   module Arg = Y (* re-export *)
5 end
6
7 module X' = F(X).Arg
```

ex.mli (types)

```
1 module X : T
2 module F : functor (Y:S) -> sig
3   ...
4   module Arg : (= Y) (* re-export *)
5 end
6
7 module X' : (= X)
```

Module aliases and compilation schemes

Aliases and functors

- Intuitively, `module X' = X`
- *F* should re-export *whole* of its argument
- The *static* and *dynamic* view of the signature's domain might be different (more fields, different order)

ex.ml (code)

```
1 module X : T = struct ... end
2 module F = functor (Y:S) -> struct
3   ...
4   module Arg = Y (* re-export *)
5 end
6
7 module X' = F(X).Arg
```

ex.mli (types)

```
1 module X : T
2 module F : functor (Y:S) -> sig
3   ...
4   module Arg : (= Y) (* re-export *)
5 end
6
7 module X' : (= X)
```

Module aliases and compilation schemes

Aliases and functors

- Intuitively, `module X' = X`
- *F* should re-export *whole* of its argument
- The *static* and *dynamic* view of the signature's domain might be different (more fields, different order)

→ Would force a *dynamic dispatch* compilation scheme

ex.ml (code)

```
1 module X : T = struct ... end
2 module F = functor (Y:S) -> struct
3   ...
4   module Arg = Y (* re-export *)
5 end
6
7 module X' = F(X).Arg
```

ex.mli (types)

```
1 module X : T
2 module F : functor (Y:S) -> sig
3   ...
4   module Arg : (= Y) (* re-export *)
5 end
6
7 module X' : (= X)
```

Module aliases and compilation schemes

Aliases and functors

- Intuitively, `module X' = X`
- *F* should re-export *whole* of its argument
- The *static* and *dynamic* view of the signature's domain might be different (more fields, different order)

→ Would force a *dynamic dispatch* compilation scheme

- OCaml (and SML) use *static dispatch*: fast access, slow subtyping

ex.ml (code)

```
1 module X : T = struct ... end
2 module F = functor (Y:S) -> struct
3   ...
4   module Arg = Y (* re-export *)
5 end
6
7 module X' = F(X).Arg
```

ex.mli (types)

```
1 module X : T
2 module F : functor (Y:S) -> sig
3   ...
4   module Arg : (= Y) (* re-export *)
5 end
6
7 module X' : (= X)
```

Module aliases and compilation schemes

Aliases and functors

- Intuitively, `module X' = X`
- *F* should re-export *whole* of its argument
- The *static* and *dynamic* view of the signature's domain might be different (more fields, different order)

→ Would force a *dynamic dispatch* compilation scheme

- OCaml (and SML) use *static dispatch*: fast access, slow subtyping

ex.ml (code)

```
1 module X : T = struct ... end
2 module F = functor (Y:S) -> struct
3   ...
4   module Arg = Y (* re-export *)
5 end
6
7 module X' = F(X).Arg
```

ex.mli (types)

```
1 module X : T
2 module F : functor (Y:S) -> sig
3   ...
4   module Arg : (= Y) (* re-export *)
5 end
6
7 module X' : (= X)
```

Module aliases and compilation schemes

Aliases and functors

- Intuitively, `module X' = X`
- *F* should re-export *whole* of its argument
- The *static* and *dynamic* view of the signature's domain might be different (more fields, different order)

→ Would force a *dynamic dispatch* compilation scheme

- OCaml (and SML) use *static dispatch*: fast access, slow subtyping

Transparent ascription - `(= X < S)`

ex.ml (code)

```
1 module X : T = struct ... end
2 module F = functor (Y:S) -> struct
3   ...
4   module Arg = Y (* re-export *)
5 end
6
7 module X' = F(X).Arg
```

ex.mli (types)

```
1 module X : T
2 module F : functor (Y:S) -> sig
3   ...
4   module Arg : (= Y<S) (* re-export *)
5 end
6
7 module X' : (= X<S)
```

Module aliases and compilation schemes

Aliases and functors

- Intuitively, `module X' = X`
- *F* should re-export *whole* of its argument
- The *static* and *dynamic* view of the signature's domain might be different (more fields, different order)

→ Would force a *dynamic dispatch* compilation scheme

- OCaml (and SML) use *static dispatch*: fast access, slow subtyping

Transparent ascription - `(= X < S)`

- Aliasing *and* structural information

ex.ml (code)

```
1 module X : T = struct ... end
2 module F = functor (Y:S) -> struct
3   ...
4   module Arg = Y (* re-export *)
5 end
6
7 module X' = F(X).Arg
```

ex.mli (types)

```
1 module X : T
2 module F : functor (Y:S) -> sig
3   ...
4   module Arg : (= Y<S) (* re-export *)
5 end
6
7 module X' : (= X<S)
```

Module aliases and compilation schemes

Aliases and functors

- Intuitively, `module X' = X`
- *F* should re-export *whole* of its argument
- The *static* and *dynamic* view of the signature's domain might be different (more fields, different order)

→ Would force a *dynamic dispatch* compilation scheme

- OCaml (and SML) use *static dispatch*: fast access, slow subtyping

Transparent ascription - (= X < S)

- Aliasing *and* structural information
- Compatible with static dispatch

ex.ml (code)

```
1 module X : T = struct ... end
2 module F = functor (Y:S) -> struct
3   ...
4   module Arg = Y (* re-export *)
5 end
6
7 module X' = F(X).Arg
```

ex.mli (types)

```
1 module X : T
2 module F : functor (Y:S) -> sig
3   ...
4   module Arg : (= Y<S) (* re-export *)
5 end
6
7 module X' : (= X<S)
```

Who needs module aliases ?

<footer>11 ⟨1⟩/ 27</footer>

Who needs module aliases ?

```
1  module type VS = sig (* Vector space *)
2
3
4  end
5
6
7
8
9
10
```

```
11
12
13
14
15
16
17
18
19
20
```

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 ?

```
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
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
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 *)
```

The M^{ω} type system

Type-sharing : path-based vs quantifiers

module M
- module X
  - type t
  - val $x : t$
  - type $u = \mathsf{int} \times t$
- module F $(Y : \mathsf{sig}\ \mathtt{type}\ t\ \mathsf{end})$
  - type t
  - type $u = Y.t$
- type $t_1 = F(X).t$
- type $t_2 = F(X).t$

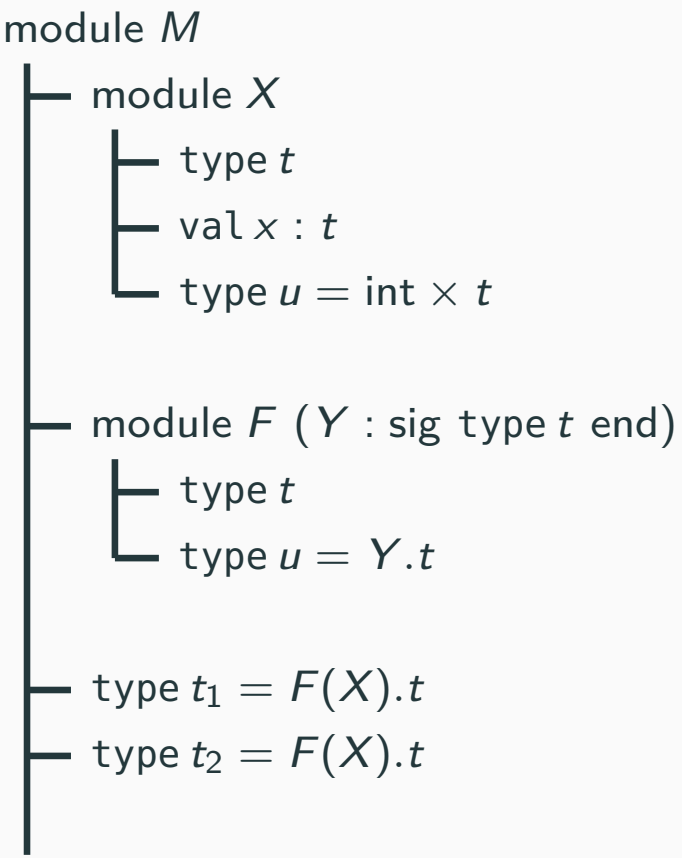
module M
- module X
  - type t
  - val $x : t$
  - type $u = \mathsf{int} \times t$
- module F $(Y : \mathsf{sig}\ \mathtt{type}\ t\ \mathsf{end})$
  - type t
  - type $u = Y.t$
- type $t_1 = F(X).t$
- type $t_2 = F(X).t$

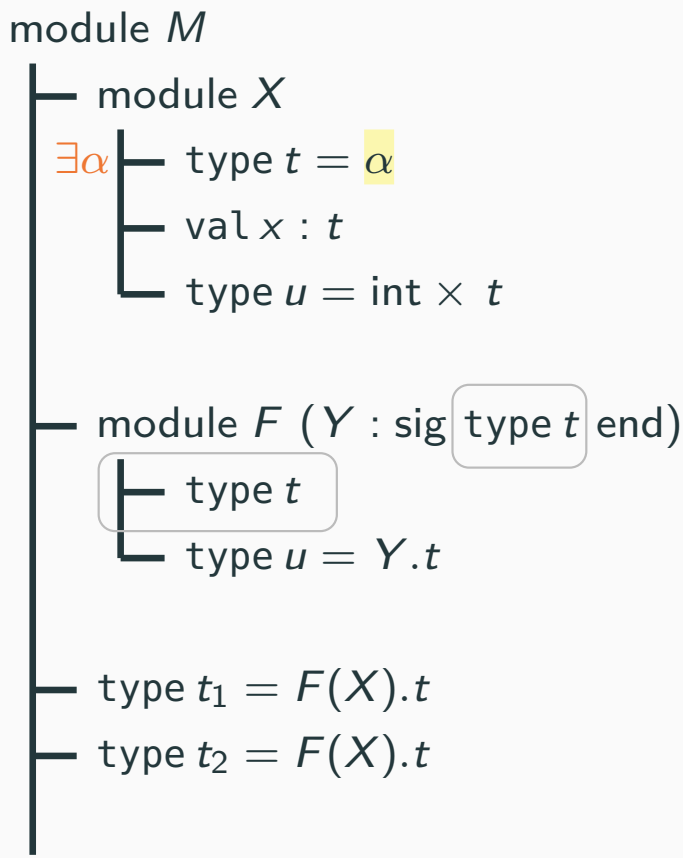
Type-sharing : path-based vs quantifiers

module M
- module X
  - type t
  - val $x : t$
  - type $u = \text{int} \times t$
- module F $(Y : \text{sig type } t \text{ end})$
  - type t
  - type $u = Y.t$
- type $t_1 = F(X).t$
- type $t_2 = F(X).t$

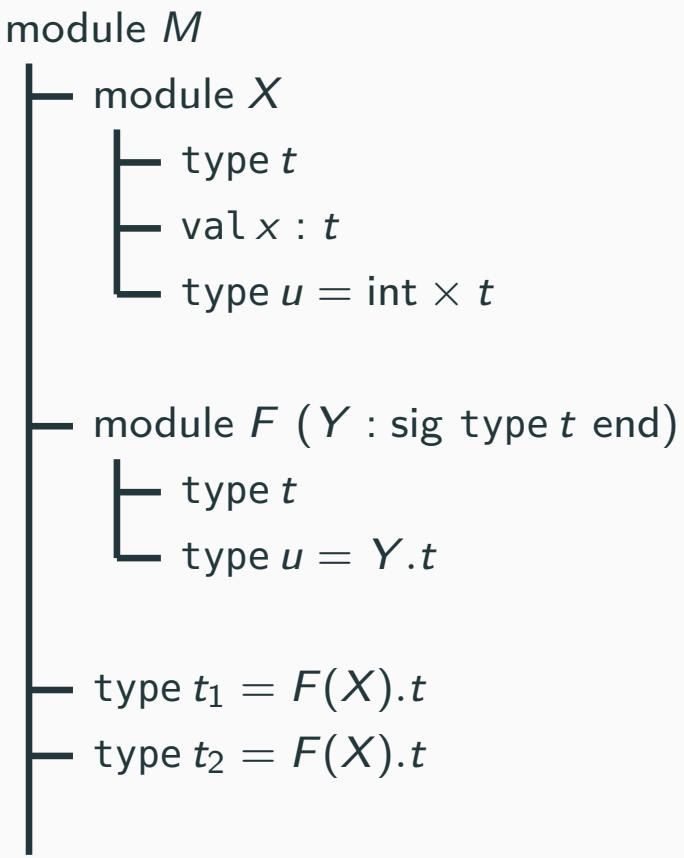
module M
- module X
  - type t
  - val $x : t$
  - type $u = \text{int} \times t$
- module F $(Y : \text{sig type } t \text{ end})$
  - type t
  - type $u = Y.t$
- type $t_1 = F(X).t$
- type $t_2 = F(X).t$

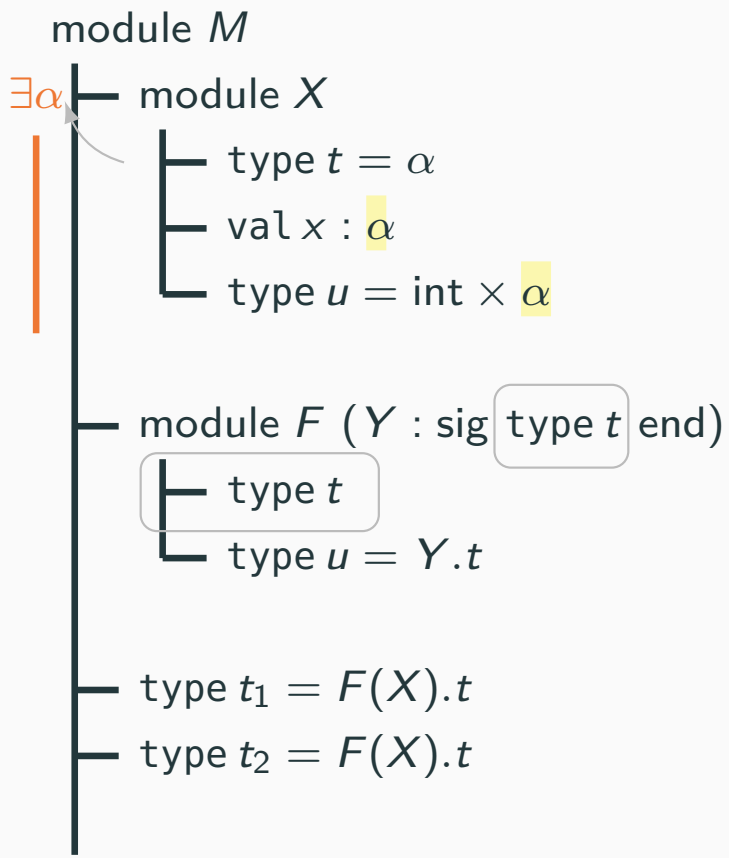
Type-sharing : path-based vs quantifiers



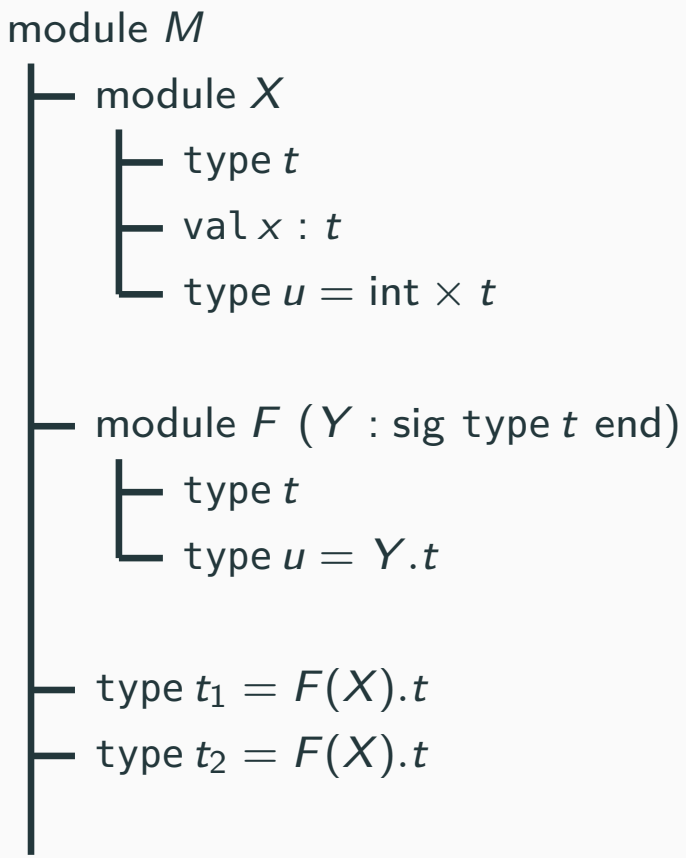


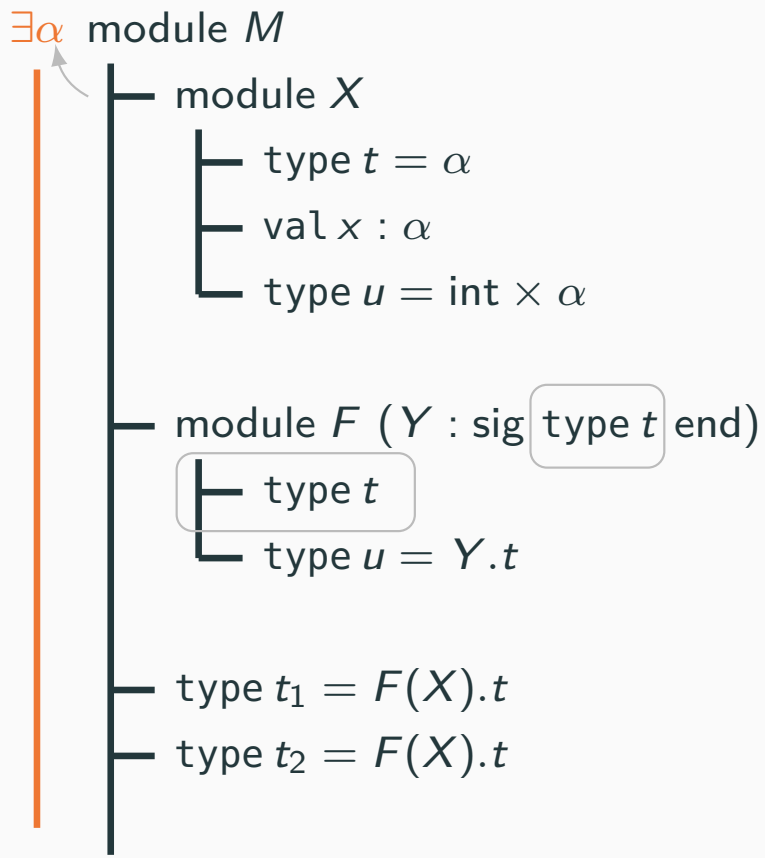
Type-sharing : path-based vs quantifiers



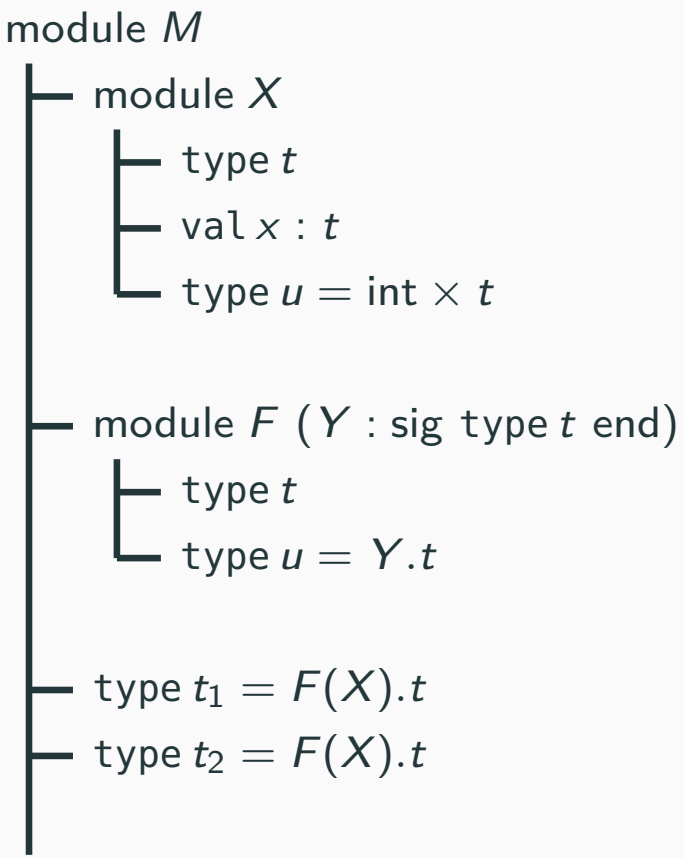


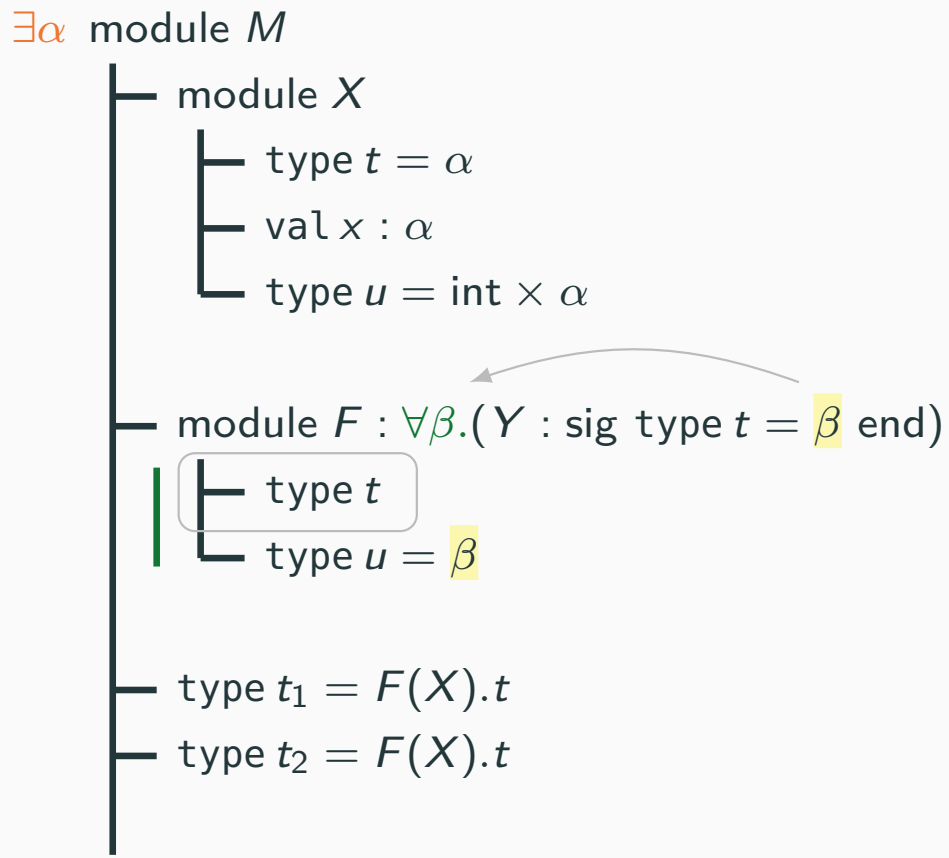
Type-sharing : path-based vs quantifiers





Type-sharing : path-based vs quantifiers



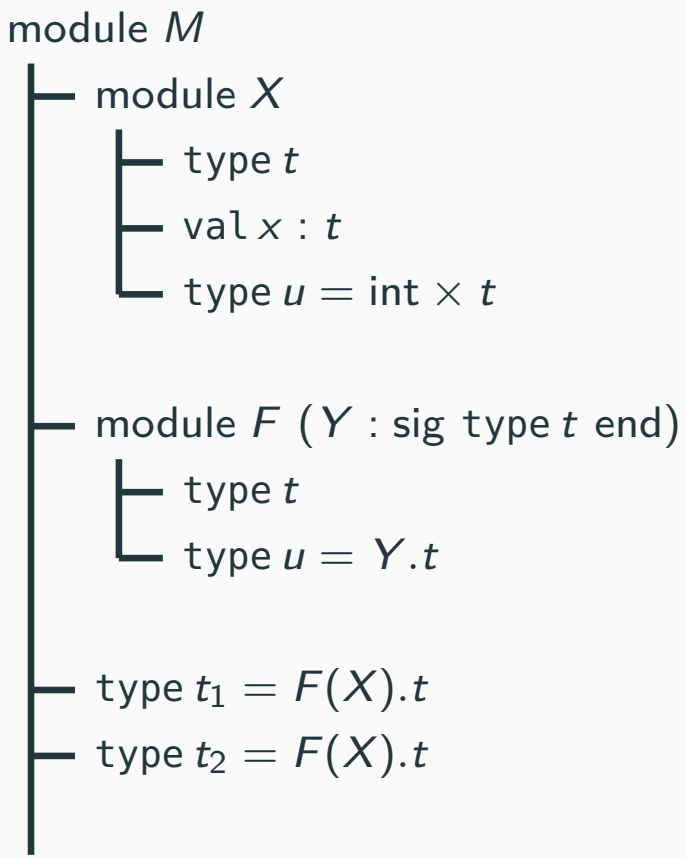


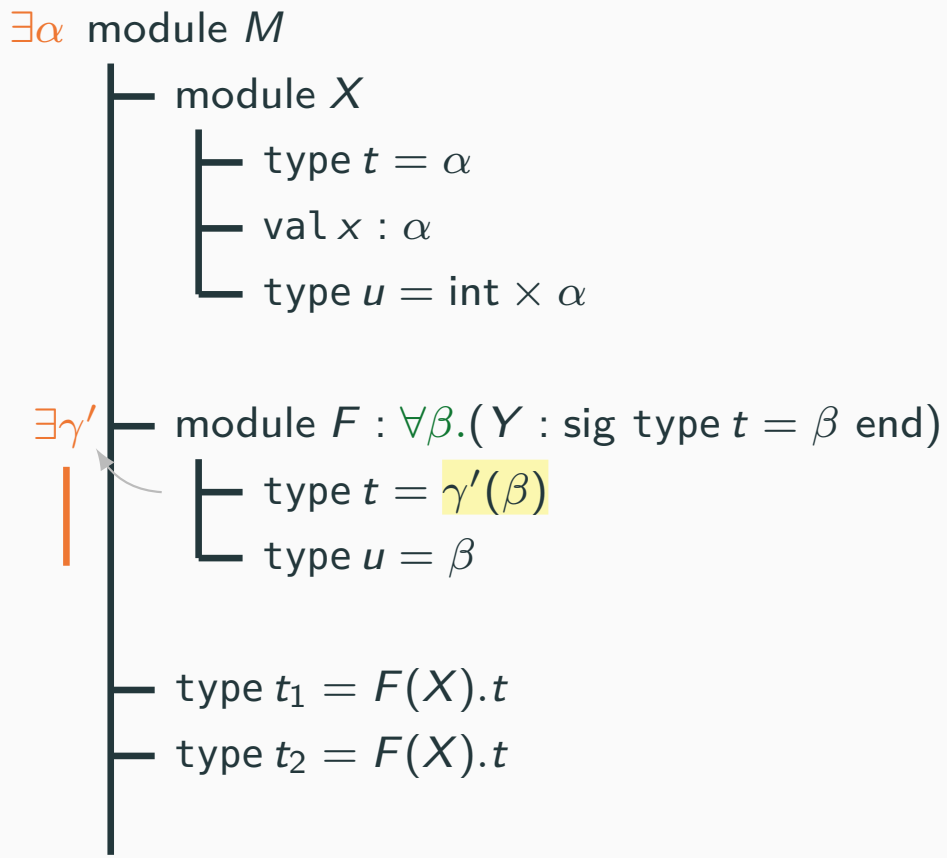
Type-sharing : path-based vs quantifiers

module M
- module X
  - type t
  - val $x : t$
  - type $u = \text{int} \times t$
- module F $(Y : \text{sig type } t \text{ end})$
  - type t
  - type $u = Y.t$
- type $t_1 = F(X).t$
- type $t_2 = F(X).t$

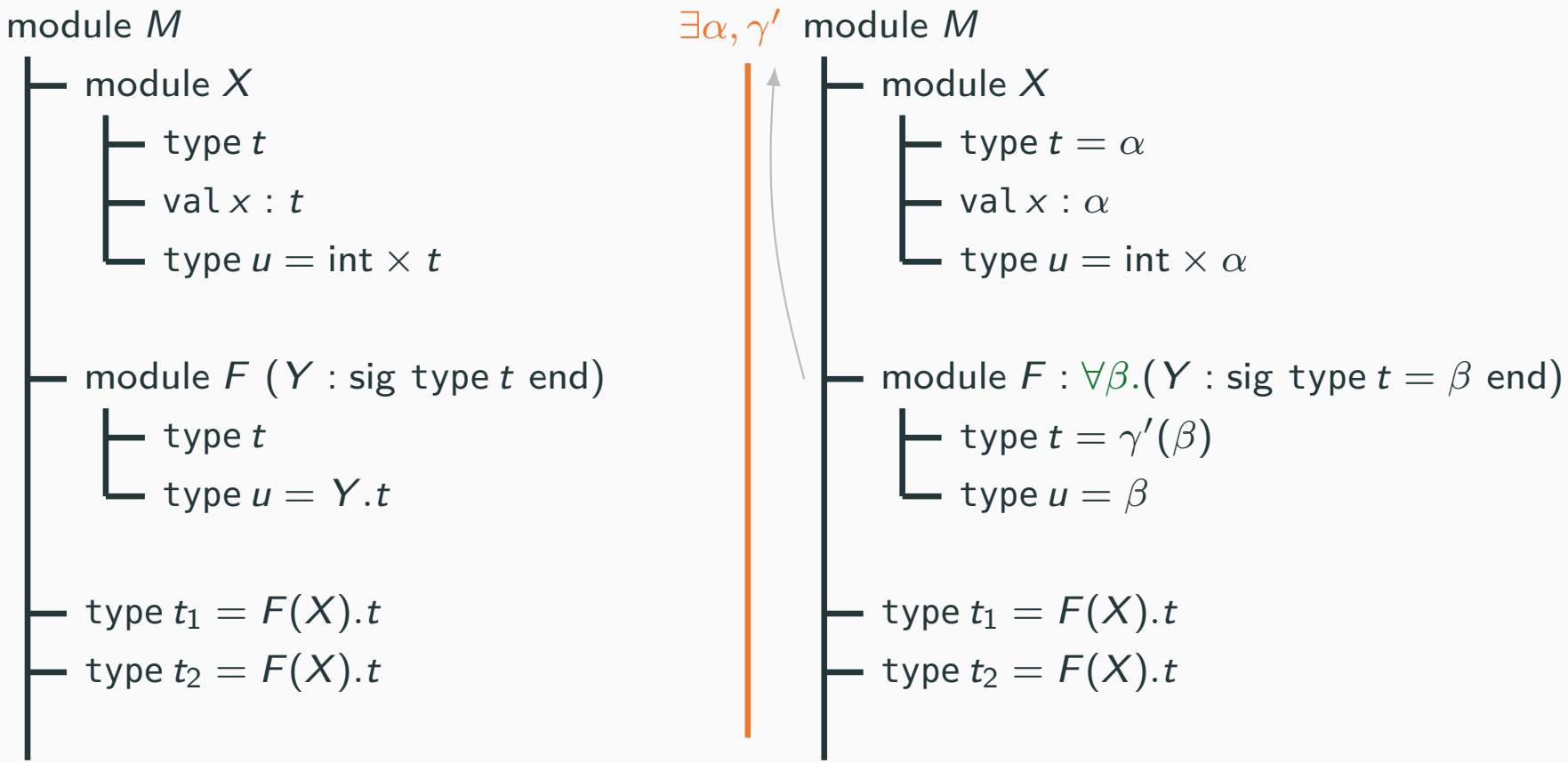
$\exists\alpha$ module M
- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \text{int} \times \alpha$
- module $F : \forall\beta.(Y : \text{sig type } t = \beta \text{ end})$
- $\exists\gamma$
  - type $t = \gamma$
  - type $u = \beta$
- type $t_1 = F(X).t$
- type $t_2 = F(X).t$

Type-sharing : path-based vs quantifiers





Type-sharing : path-based vs quantifiers



Type-sharing : path-based vs quantifiers

module M

- module X
  - type t
  - val $x : t$
  - type $u = \text{int} \times t$
- module F $(Y : \text{sig type } t \text{ end})$
  - type t
  - type $u = Y.t$
- type $t_1 = F(X).t$
- type $t_2 = F(X).t$

$\exists \alpha, \gamma'$ 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 $u = \beta$
- type $t_1 = \gamma'(\alpha)$
- type $t_2 = F(X).t$

Type-sharing : path-based vs quantifiers

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 $u = Y.t$
- type $t_1 = F(X).t$
- type $t_2 = F(X).t$

$\exists \alpha, \gamma'$ module M
- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \mathsf{int} \times \alpha$
- module $F : \forall \beta.(Y : \mathsf{sig}\ \mathsf{type}\ t = \beta\ \mathsf{end})$
  - type $t = \gamma'(\beta)$
  - type $u = \beta$
- type $t_1 = \gamma'(\alpha)$
- type $t_2 = \gamma'(\alpha)$

Type-sharing : path-based vs quantifiers

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 $u = Y.t$
- type $t_1 = F(X).t$
- type $t_2 = F(X).t$

$\exists \alpha, \gamma'$ module M
- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \mathsf{int} \times \alpha$
- module $F : \forall \beta.(Y : \mathsf{sig}\ \mathsf{type}\ t = \beta\ \mathsf{end})$
  - type $t = \gamma'(\beta)$
  - type $u = \beta$
- type $t_1 = \gamma'(\alpha)$
- type $t_2 = \gamma'(\alpha)$

- Understand signatures as F^ω *types*

Type-sharing : path-based vs quantifiers

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 $u = Y.t$
- type $t_1 = F(X).t$
- type $t_2 = F(X).t$

$\exists \alpha, \gamma'$ module M
- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \mathsf{int} \times \alpha$
- module $F : \forall \beta.(Y : \mathsf{sig}\ \mathsf{type}\ t = \beta\ \mathsf{end})$
  - type $t = \gamma'(\beta)$
  - type $u = \beta$
- type $t_1 = \gamma'(\alpha)$
- type $t_2 = \gamma'(\alpha)$

- Understand signatures as F^ω *types* $\qquad \Gamma \vdash \mathsf{S} : \exists \overline{\alpha}.\mathcal{C}$

Type-sharing : path-based vs quantifiers

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 $u = Y.t$
- type $t_1 = F(X).t$
- type $t_2 = F(X).t$

$\exists \alpha, \gamma'$ module M
- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \mathsf{int} \times \alpha$
- module $F : \forall \beta.(Y : \mathsf{sig}\ \mathsf{type}\ t = \beta\ \mathsf{end})$
  - type $t = \gamma'(\beta)$
  - type $u = \beta$
- type $t_1 = \gamma'(\alpha)$
- type $t_2 = \gamma'(\alpha)$

- Understand signatures as F^ω *types*
- Interpret modules as F^ω *terms*

$\Gamma \vdash \mathsf{S} : \exists \overline{\alpha}.\mathcal{C}$

Type-sharing : path-based vs quantifiers

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 $u = Y.t$
- type $t_1 = F(X).t$
- type $t_2 = F(X).t$

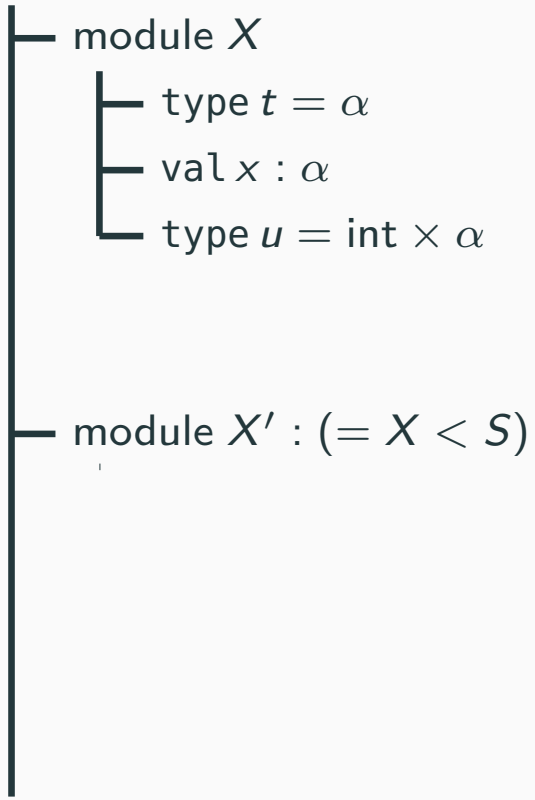
$\exists \alpha, \gamma'$ module M
- module X
  - type $t = \alpha$
  - val $x : \alpha$
  - type $u = \mathsf{int} \times \alpha$
- module $F : \forall \beta.(Y : \mathsf{sig}\ \mathsf{type}\ t = \beta\ \mathsf{end})$
  - type $t = \gamma'(\beta)$
  - type $u = \beta$
- type $t_1 = \gamma'(\alpha)$
- type $t_2 = \gamma'(\alpha)$

- Understand signatures as F^ω *types*
- Interpret modules as F^ω *terms*

$$\Gamma \vdash \mathsf{S} : \exists \overline{\alpha}.\mathcal{C}$$

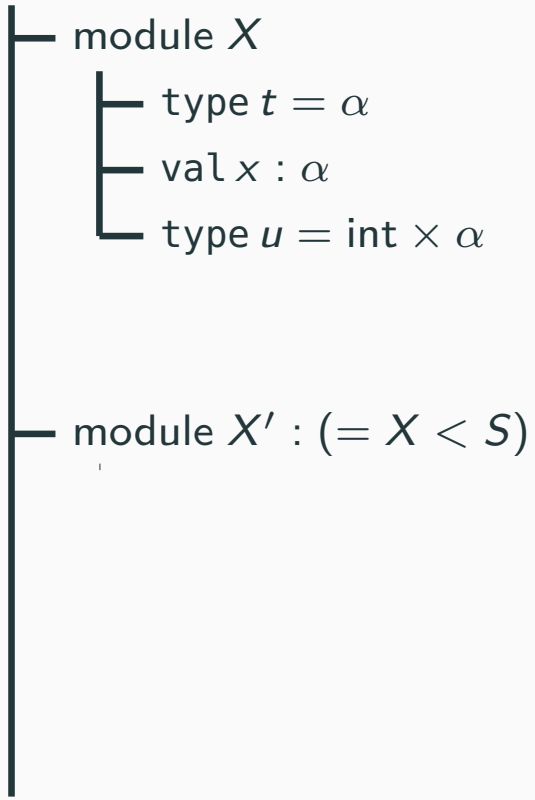
$$\Gamma \vdash \mathsf{M} : \exists \overline{\alpha}.\mathcal{C} \rightsquigarrow e$$

Module identities are abstract types



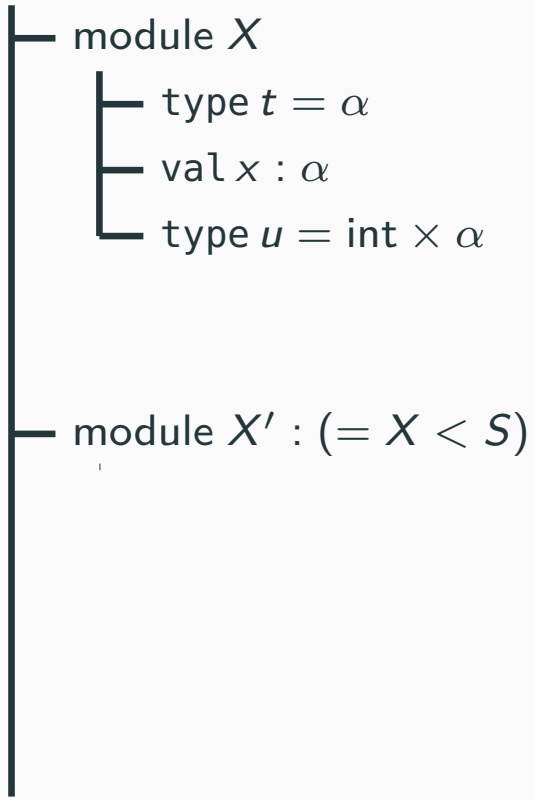
Module identities are abstract types

- Use type abstraction mechanisms to track module identities



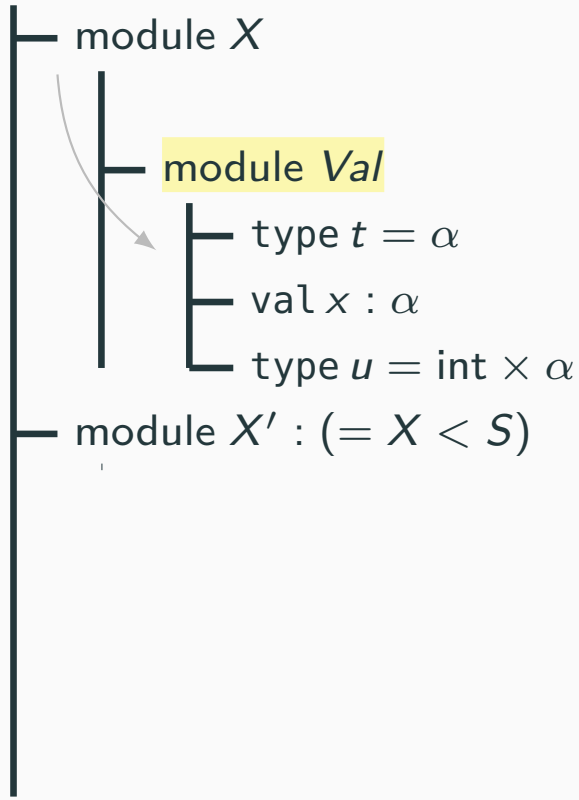
Module identities are abstract types

- Use type abstraction mechanisms to track module identities
- Add a source-to-source transformation with abstract `id` fields



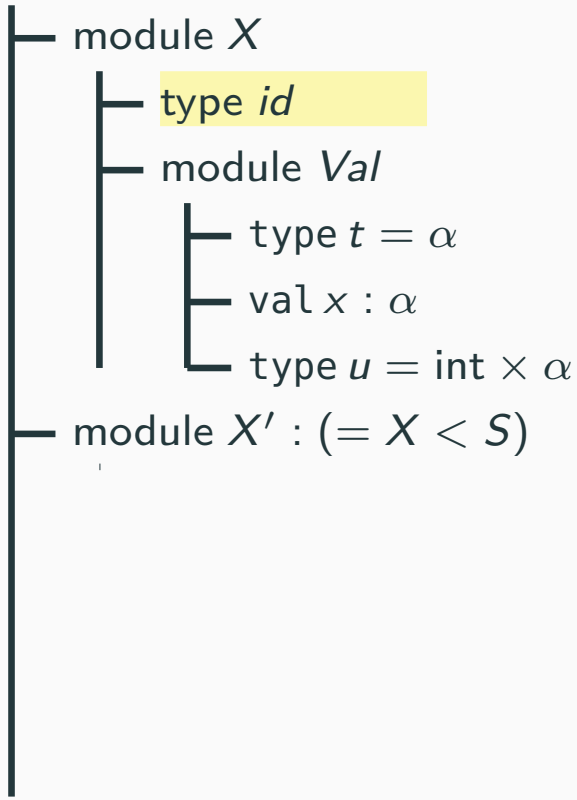
Module identities are abstract types

- Use type abstraction mechanisms to track module identities
- Add a source-to-source transformation with abstract `id` fields



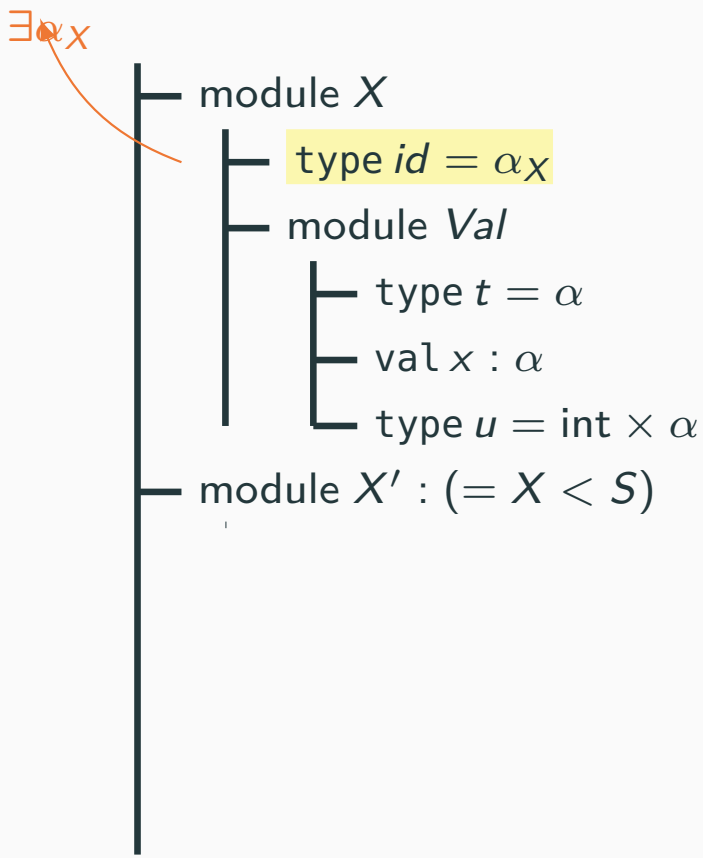
Module identities are abstract types

- Use type abstraction mechanisms to track module identities
- Add a source-to-source transformation with abstract `id` fields



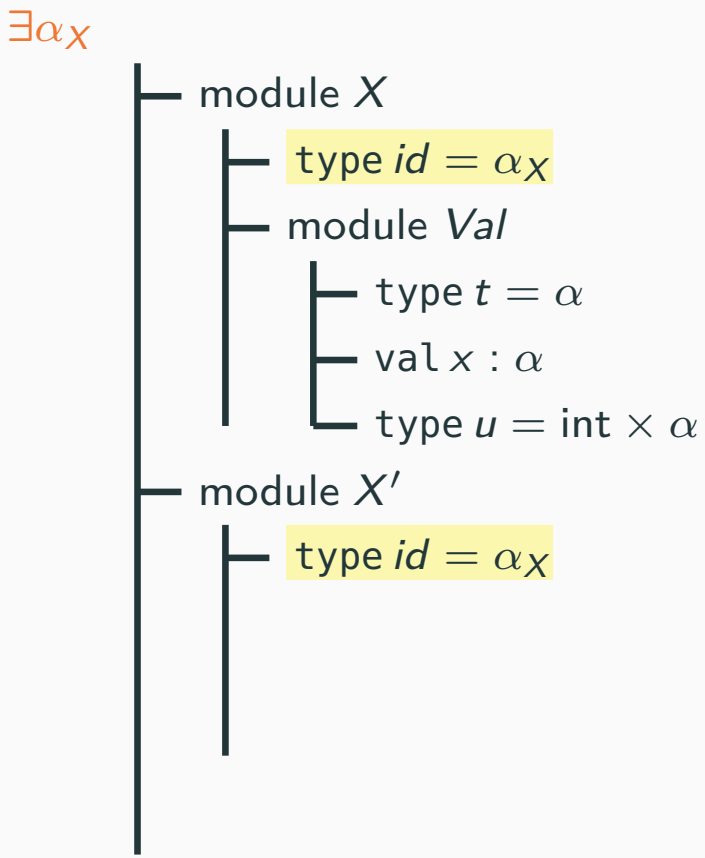
Module identities are abstract types

- Use type abstraction mechanisms to track module identities
- Add a source-to-source transformation with abstract `id` fields



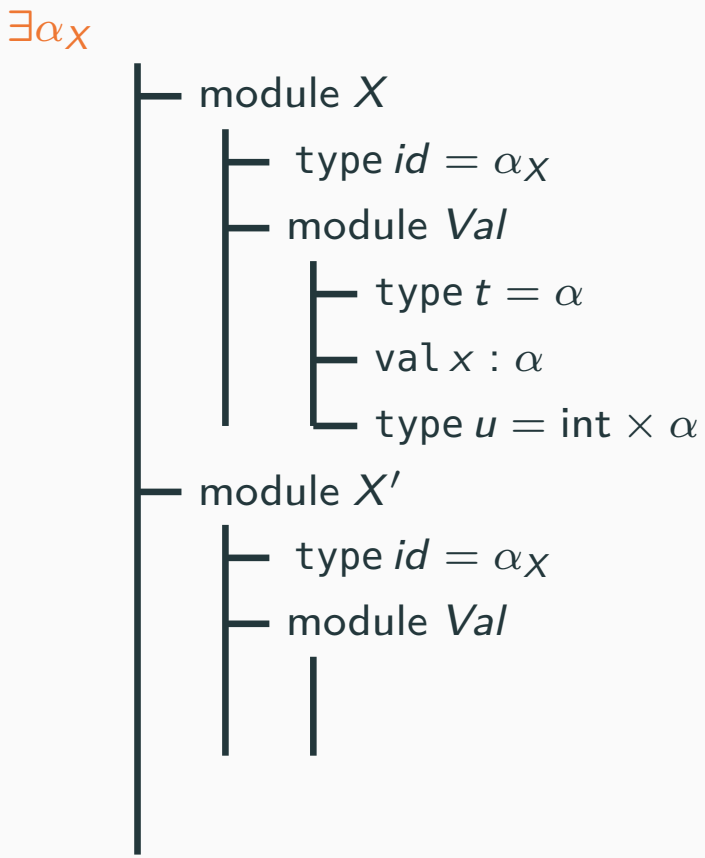
Module identities are abstract types

- Use type abstraction mechanisms to track module identities
- Add a source-to-source transformation with abstract `id` fields



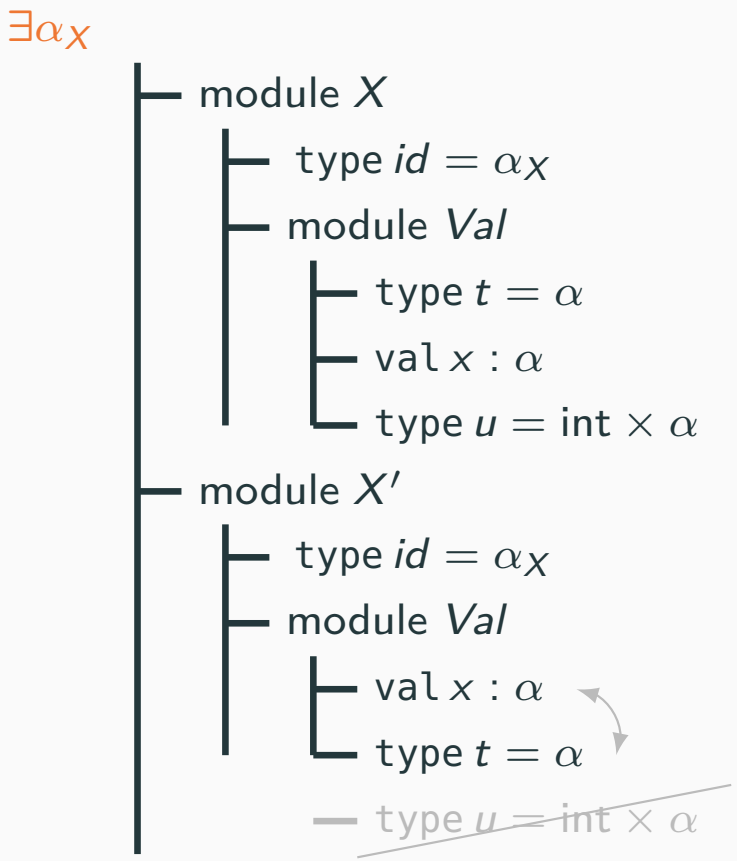
Module identities are abstract types

- Use type abstraction mechanisms to track module identities
- Add a source-to-source transformation with abstract `id` fields



Module identities are abstract types

- Use type abstraction mechanisms to track module identities
- Add a source-to-source transformation with abstract `id` fields



F^ω and canonical signatures

F^ω- with primitive records and existential types

(types)

(terms)

F^ω and canonical signatures

F^ω- with primitive records and existential types

$$\tau := \alpha \mid \tau \to \tau \mid \{\overline{\ell : \tau}\} \mid \forall\alpha.\tau \mid \exists^{\blacktriangledown}\alpha.\tau \mid \lambda\alpha.\tau \mid () \qquad \text{(types)}$$

(terms)

F^ω and canonical signatures

F^ω- with primitive records and existential types

$$\tau := \alpha \mid \tau \to \tau \mid \{\overline{\ell : \tau}\} \mid \forall\alpha.\tau \mid \exists^{\blacktriangledown}\alpha.\tau \mid \lambda\alpha.\tau \mid () \qquad \text{(types)}$$

$$e := x \mid \lambda(x : \tau).e \mid e\,e \mid \Lambda\alpha.\,e \mid e\,\tau \mid e \mathbin{@} e \mid \{\overline{\ell = e}\} \mid e.\ell \qquad \text{(terms)}$$

F^ω and canonical signatures

F^ω- with primitive records and existential types

$$\tau := \alpha \mid \tau \to \tau \mid \{\overline{\ell : \tau}\} \mid \forall\alpha.\tau \mid \exists^{\blacktriangledown}\alpha.\tau \mid \lambda\alpha.\tau \mid () \qquad \text{(types)}$$

$$e := x \mid \lambda(x : \tau).e \mid e\,e \mid \Lambda\alpha.\,e \mid e\,\tau \mid e \mathbin{@} e \mid \{\overline{\ell = e}\} \mid e.\ell$$

$$\mid \mathsf{pack}\,\langle\tau, e\rangle\,\mathsf{as}\,\exists^{\blacktriangledown}\alpha.\tau \mid \mathsf{unpack}\,\langle\alpha, x\rangle = e\,\mathsf{in}\,e \mid () \qquad \text{(terms)}$$

F^ω and canonical signatures

F^ω- with primitive records and existential types

$$\tau := \alpha \mid \tau \to \tau \mid \{\overline{\ell : \tau}\} \mid \forall\alpha.\tau \mid \exists^{\blacktriangledown}\alpha.\tau \mid \lambda\alpha.\tau \mid () \qquad \text{(types)}$$

$$e := x \mid \lambda(x : \tau).e \mid e\,e \mid \Lambda\alpha.\,e \mid e\,\tau \mid e \mathbin{@} e \mid \{\overline{\ell = e}\} \mid e.\ell$$

$$\mid \mathsf{pack}\,\langle\tau, e\rangle\,\mathsf{as}\,\exists^{\blacktriangledown}\alpha.\tau \mid \mathsf{unpack}\,\langle\alpha, x\rangle = e\,\mathsf{in}\,e \mid () \qquad \text{(terms)}$$

M^ω signatures (F^ω types)

F^ω and canonical signatures

F^ω- with primitive records and existential types

$$\tau := \alpha \mid \tau \to \tau \mid \{\overline{\ell : \tau}\} \mid \forall\alpha.\tau \mid \exists^{\blacktriangledown}\alpha.\tau \mid \lambda\alpha.\tau \mid () \qquad \text{(types)}$$

$$e := x \mid \lambda(x : \tau).e \mid e\,e \mid \Lambda\alpha.\,e \mid e\,\tau \mid e \mathbin{@} e \mid \{\overline{\ell = e}\} \mid e.\ell$$

$$\mid \mathsf{pack}\,\langle\tau, e\rangle\,\mathsf{as}\,\exists^{\blacktriangledown}\alpha.\tau \mid \mathsf{unpack}\,\langle\alpha, x\rangle = e\,\mathsf{in}\,e \mid () \qquad \text{(terms)}$$

M^ω signatures (F^ω types)

F^ω and canonical signatures

F^ω- with primitive records and existential types

$$\tau := \alpha \mid \tau \to \tau \mid \{\overline{\ell : \tau}\} \mid \forall\alpha.\tau \mid \exists^{\blacktriangledown}\alpha.\tau \mid \lambda\alpha.\tau \mid () \qquad \text{(types)}$$

$$e := x \mid \lambda(x : \tau).e \mid e\,e \mid \Lambda\alpha.\,e \mid e\,\tau \mid e \mathbin{@} e \mid \{\overline{\ell = e}\} \mid e.\ell$$

$$\mid \mathsf{pack}\,\langle\tau, e\rangle\,\mathsf{as}\,\exists^{\blacktriangledown}\alpha.\tau \mid \mathsf{unpack}\,\langle\alpha, x\rangle = e\,\mathsf{in}\,e \mid () \qquad \text{(terms)}$$

M^ω signatures (F^ω types)

$$\mathcal{C} ::= \mathsf{sig}\,\overline{\mathcal{D}}\,\mathsf{end} \mid \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \mid () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}$$

F^ω and canonical signatures

F^ω- with primitive records and existential types

$$\tau := \alpha \mid \tau \to \tau \mid \{\overline{\ell : \tau}\} \mid \forall\alpha.\tau \mid \exists^{\blacktriangledown}\alpha.\tau \mid \lambda\alpha.\tau \mid () \qquad \text{(types)}$$

$$e := x \mid \lambda(x : \tau).e \mid e\,e \mid \Lambda\alpha.\,e \mid e\,\tau \mid e \mathbin{@} e \mid \{\overline{\ell = e}\} \mid e.\ell$$

$$\mid \mathsf{pack}\,\langle\tau, e\rangle\,\mathsf{as}\,\exists^{\blacktriangledown}\alpha.\tau \mid \mathsf{unpack}\,\langle\alpha, x\rangle = e\,\mathsf{in}\,e \mid () \qquad \text{(terms)}$$

M^ω signatures (F^ω types)

$$\mathcal{C} ::= \mathsf{sig}\,\overline{\mathcal{D}}\,\mathsf{end} \mid \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \mid () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}$$

$$\begin{aligned}
\mathcal{D} ::= {} & \mathsf{val}\,x : \tau && \triangleq \ell_x : \tau \\
\mid {} & \mathsf{type}\,t = \tau && \triangleq \ell_t : \langle\!\langle\tau\rangle\!\rangle \\
\mid {} & \mathsf{module}\,X : \mathcal{C} && \triangleq \ell_X : \mathcal{C} \\
\mid {} & \mathsf{module\ type}\,T = \lambda\overline{\alpha}.\mathcal{C} && \triangleq \ell_T : \langle\!\langle\lambda\overline{\alpha}.\mathcal{C}\rangle\!\rangle
\end{aligned}$$

F^ω and canonical signatures

F^ω- with primitive records and existential types

$$\tau := \alpha \mid \tau \to \tau \mid \{\overline{\ell : \tau}\} \mid \forall\alpha.\tau \mid \exists^{\blacktriangledown}\alpha.\tau \mid \lambda\alpha.\tau \mid () \qquad \text{(types)}$$

$$e := x \mid \lambda(x : \tau).e \mid e\,e \mid \Lambda\alpha.\,e \mid e\,\tau \mid e \mathbin{@} e \mid \{\overline{\ell = e}\} \mid e.\ell$$

$$\mid \mathsf{pack}\,\langle\tau, e\rangle\,\mathsf{as}\,\exists^{\blacktriangledown}\alpha.\tau \mid \mathsf{unpack}\,\langle\alpha, x\rangle = e\,\mathsf{in}\,e \mid () \qquad \text{(terms)}$$

M^ω signatures (F^ω types)

$$\mathcal{C} ::= \mathsf{sig}\,\overline{\mathcal{D}}\,\mathsf{end} \mid \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \mid () \to \exists^{\blacktriangledown}\overline{\alpha}.\mathcal{C}$$

$$\begin{aligned}
\mathcal{D} ::= {} & \mathsf{val}\ x : \tau && \triangleq \ell_x : \tau \\
\mid {} & \mathsf{type}\ t = \tau && \triangleq \ell_t : \langle\!\langle\tau\rangle\!\rangle \\
\mid {} & \mathsf{module}\ X : \mathcal{C} && \triangleq \ell_X : \mathcal{C} \\
\mid {} & \mathsf{module\ type}\ T = \lambda\overline{\alpha}.\mathcal{C} && \triangleq \ell_T : \langle\!\langle\lambda\overline{\alpha}.\mathcal{C}\rangle\!\rangle
\end{aligned}$$

$$\langle\!\langle\tau\rangle\!\rangle \triangleq \forall\varphi.\varphi\,\tau \to \varphi\,\tau$$

The canonical system

The canonical system

Module typing
$\Gamma \vdash \mathtt{M} : \exists^{\diamond} \overline{\alpha}.\mathcal{C}$

The canonical system

Module typing
$\Gamma \vdash \mathtt{M} : \exists^{\diamond} \overline{\alpha}.\mathcal{C}$

Signature typing
$\Gamma \vdash \mathtt{S} : \lambda \overline{\alpha}.\mathcal{C}$

The canonical system

Module typing
$\Gamma \vdash \mathtt{M} : \exists^{\diamond}\overline{\alpha}.\mathcal{C}$

Signature typing
$\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}$

Subtyping
$\Gamma \vdash \mathcal{C} < \mathcal{C}'$

The canonical system

Module typing
$\Gamma \vdash \mathtt{M} : \exists^{\diamond} \overline{\alpha}.\mathcal{C}$

Signature typing
$\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}$

Subtyping
$\Gamma \vdash \mathcal{C} < \mathcal{C}'$

M-Typ-Decl-Type

$$\frac{\Gamma \vdash \mathsf{u} : \tau}{\Gamma \vdash_A (\mathsf{type}\, t = \mathsf{u}) : (\mathsf{type}\, t = \tau)}$$

The canonical system

Module typing
$\Gamma \vdash \mathsf{M} : \exists^{\diamond} \overline{\alpha}.\mathcal{C}$

Signature typing
$\Gamma \vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}$

Subtyping
$\Gamma \vdash \mathcal{C} < \mathcal{C}'$

M-Typ-Decl-Type

$$\frac{\Gamma \vdash \mathsf{u} : \tau}{\Gamma \vdash_A (\mathsf{type}\, t = \mathsf{u}) : (\mathsf{type}\, t = \tau)}$$

M-Typ-Decl-TypeAbs

$$\Gamma \vdash_A (\mathsf{type}\, t) : \lambda\alpha.(\mathsf{type}\, t = \alpha)$$

The canonical system

Module typing
$\Gamma \vdash \mathtt{M} : \exists^{\diamond} \overline{\alpha}.\mathcal{C}$

Signature typing
$\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}$

Subtyping
$\Gamma \vdash \mathcal{C} < \mathcal{C}'$

M-Typ-Decl-Type

$$\frac{\Gamma \vdash \mathsf{u} : \tau}{\Gamma \vdash_A (\mathtt{type}\, t = \mathsf{u}) : (\mathtt{type}\, t = \tau)}$$

M-Typ-Decl-TypeAbs

$$\Gamma \vdash_A (\mathtt{type}\, t) : \lambda\alpha.(\mathtt{type}\, t = \alpha)$$

M-Typ-Decl-Mod

$$\frac{\Gamma \vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module}\, X : \mathsf{S}) : \lambda\overline{\alpha}.(\mathtt{module}\, X : \mathcal{C})}$$

The canonical system

Module typing
$\Gamma \vdash \mathtt{M} : \exists^{\diamond} \overline{\alpha}.\mathcal{C}$

Signature typing
$\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}$

Subtyping
$\Gamma \vdash \mathcal{C} < \mathcal{C}'$

M-Typ-Decl-Type

$$\frac{\Gamma \vdash \mathsf{u} : \tau}{\Gamma \vdash_A (\mathsf{type}\, t = \mathsf{u}) : (\mathsf{type}\, t = \tau)}$$

M-Typ-Decl-TypeAbs

$$\Gamma \vdash_A (\mathsf{type}\, t) : \lambda\alpha.(\mathsf{type}\, t = \alpha)$$

M-Typ-Decl-Mod

$$\frac{\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash_A (\mathtt{module}\, X : \mathtt{S}) : \lambda\overline{\alpha}.(\mathsf{module}\, X : \mathcal{C})}$$

M-Typ-Sig-Ste

$$\frac{\Gamma \vdash_A \overline{\mathtt{D}} : \lambda\overline{\alpha}.\overline{\mathcal{D}} \qquad A \notin \Gamma}{\Gamma \vdash \mathtt{sig}_A\ \overline{\mathtt{D}}\ \mathsf{end} : \lambda\overline{\alpha}.\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end}}$$

The canonical system

Module typing
$\Gamma \vdash \mathsf{M} : \exists^{\diamond} \overline{\alpha}.\mathcal{C}$

Signature typing
$\Gamma \vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}$

Subtyping
$\Gamma \vdash \mathcal{C} < \mathcal{C}'$

$$\textsc{M-Typ-Sig-GenFct}\ \frac{\Gamma \vdash \mathsf{S} : \lambda\overline{\alpha}.\mathcal{C}}{\Gamma \vdash () \to \mathsf{S} : () \to \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}}$$

The canonical system

Module typing
$\Gamma \vdash \mathtt{M} : \exists^{\diamond} \overline{\alpha}.\mathcal{C}$

Signature typing
$\Gamma \vdash \mathtt{S} : \lambda \overline{\alpha}.\mathcal{C}$

Subtyping
$\Gamma \vdash \mathcal{C} < \mathcal{C}'$

M-Typ-Sig-GenFct

$$\frac{\Gamma \vdash \mathtt{S} : \lambda \overline{\alpha}.\mathcal{C}}{\Gamma \vdash () \to \mathtt{S} : () \to \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}}$$

M-Typ-Sig-AppFct

$$\frac{\Gamma \vdash \mathtt{S}_a : \lambda \overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, Y : \mathcal{C}_a \vdash \mathtt{S} : \lambda \overline{\beta}.\mathcal{C}}{\Gamma \vdash (Y : \mathtt{S}_a) \to \mathtt{S} : \lambda \overline{\beta'}.\forall \overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'}(\overline{\alpha})\right]}$$

The canonical system

Module typing
$\Gamma \vdash \mathtt{M} : \exists^{\diamond} \overline{\alpha}.\mathcal{C}$

Signature typing
$\Gamma \vdash \mathtt{S} : \lambda \overline{\alpha}.\mathcal{C}$

Subtyping
$\Gamma \vdash \mathcal{C} < \mathcal{C}'$

M-Typ-Sig-GenFct

$$\frac{\Gamma \vdash \mathtt{S} : \lambda \overline{\alpha}.\mathcal{C}}{\Gamma \vdash () \to \mathtt{S} : () \to \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}}$$

M-Typ-Sig-AppFct

$$\frac{\Gamma \vdash \mathtt{S}_a : \lambda \overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, Y : \mathcal{C}_a \vdash \mathtt{S} : \lambda \overline{\beta}.\mathcal{C}}{\Gamma \vdash (Y : \mathtt{S}_a) \to \mathtt{S} : \lambda \overline{\beta'}.\forall \overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]}$$

M-Typ-Mod-GenFct

$$\frac{\Gamma \vdash \mathtt{M} : \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}}{\Gamma \vdash () \to \mathtt{M} : () \to \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}}$$

M-Typ-Mod-AppFct

$$\frac{\Gamma \vdash \mathtt{S}_a : \lambda \overline{\alpha}.\mathcal{C}_a \qquad \Gamma, \overline{\alpha}, (Y : \mathcal{C}_a) \vdash \mathtt{M} : \exists^{\nabla} \overline{\beta}.\mathcal{C}}{\Gamma \vdash (Y : \mathtt{S}_a) \to \mathtt{M} : \exists^{\nabla} \overline{\beta'}.\forall \overline{\alpha}.\mathcal{C}_a \to \mathcal{C}\left[\overline{\beta} \mapsto \overline{\beta'(\overline{\alpha})}\right]}$$

The canonical system

Module typing
$\Gamma \vdash \mathtt{M} : \exists^{\diamond} \overline{\alpha}.\mathcal{C}$

Signature typing
$\Gamma \vdash \mathtt{S} : \lambda\overline{\alpha}.\mathcal{C}$

Subtyping
$\Gamma \vdash \mathcal{C} < \mathcal{C}'$

M-Typ-Mod-AppGen

$$\frac{\Gamma \vdash P : () \to \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}}{\Gamma \vdash P() : \exists^{\blacktriangledown} \overline{\alpha}.\mathcal{C}}$$

M-Typ-Mod-AppApp

$$\frac{\Gamma \vdash P : \forall\overline{\alpha}.\mathcal{C}_a \to \mathcal{C} \qquad \Gamma \vdash P' : \mathcal{C}' \qquad \Gamma \vdash \mathcal{C}' < \mathcal{C}_a[\overline{\alpha} \mapsto \overline{\tau}]}{\Gamma \vdash P(P') : \mathcal{C}[\overline{\alpha} \mapsto \overline{\tau}]}$$

The canonical system

Module typing
$\Gamma \vdash \mathtt{M} : \exists^{\diamondsuit} \overline{\alpha}.\mathcal{C}$

Signature typing
$\Gamma \vdash \mathtt{S} : \lambda \overline{\alpha}.\mathcal{C}$

Subtyping
$\Gamma \vdash \mathcal{C} < \mathcal{C}'$

$$\textsc{M-Typ-Mod-Proj}\quad \frac{\Gamma \vdash \mathtt{M} : \exists^{\diamondsuit} \overline{\alpha}.\,\mathsf{sig}\ \overline{\mathcal{D}}\ \mathsf{end} \qquad \mathsf{module}\ X : \mathcal{C} \in \overline{\mathcal{D}}}{\Gamma \vdash \mathtt{M}.X : \exists^{\diamondsuit} \overline{\alpha}'.\mathcal{C}}$$

The canonical system

Module typing
$\Gamma \vdash \mathtt{M} : \exists^{\diamond} \overline{\alpha}.\mathcal{C}$

Signature typing
$\Gamma \vdash \mathtt{S} : \lambda \overline{\alpha}.\mathcal{C}$

Subtyping
$\Gamma \vdash \mathcal{C} < \mathcal{C}'$

M-Typ-Mod-Ascr

$$\frac{\Gamma \vdash P : \mathcal{C} \qquad \Gamma \vdash \mathtt{S} : \lambda \overline{\alpha}.\mathcal{C}' \qquad \Gamma \vdash \mathcal{C} < \mathcal{C}'[\overline{\alpha} \mapsto \overline{\tau}]}{\Gamma \vdash (P : \mathtt{S}) : \exists^{\triangledown} \overline{\alpha}.\mathcal{C}'}$$

Soundness by elaboration

Example of encoding into F^ω

Source code

```
module M = struct
  module X1 : S = struct
    type t = int
    let x = 42
    type u = t -> int
  end
  module X2 = struct
    type t = X1.t * bool
    type u = X1.t * string
  end
end

```

Signature

Encoded module

Example of encoding into F^{ω}

Source code

```
1  module M = struct
2    module X1 : S = struct
3      type t = int
4      let x = 42
5      type u = t -> int
6    end
7    module X2 = struct
8      type t = X1.t * bool
9      type u = X1.t * string
10   end
11 end
12
```

Signature

$$
\begin{array}{l}
\exists \alpha.\mathsf{sig} \\
\qquad \begin{array}{lll}
\mathsf{module}\ X_1 : & \mathsf{sig}\ \mathsf{type}\ t = \alpha & \\
& \qquad \mathsf{val}\ x : \alpha & \\
& \qquad \mathsf{type}\ u = \alpha \to \mathsf{int}\ \mathsf{end} & \\
\mathsf{module}\ X_2 : & \mathsf{sig}\ \mathsf{type}\ t = \alpha \times \mathtt{bool} & \\
& \qquad \mathsf{type}\ u = \alpha \times \mathsf{string}\ \mathsf{end} &
\end{array} \\
\mathsf{end}
\end{array}
$$

Encoded module

Example of encoding into F^{ω}

Source code

```
1  module M = struct
2    module X1 : S = struct
3      type t = int
4      let x = 42
5      type u = t -> int
6    end
7    module X2 = struct
8      type t = X1.t * bool
9      type u = X1.t * string
10   end
11 end
12
```

Signature

$$\exists \alpha. \left\{ \begin{array}{l} \ell_{X_1} : \left\{ \begin{array}{l} \ell_t : \langle\!\langle \alpha \rangle\!\rangle \\ \ell_x : \alpha \\ \ell_u : \langle\!\langle \alpha \to \mathsf{int} \rangle\!\rangle \end{array} \right\} \\ \ell_{X_2} : \left\{ \begin{array}{l} \ell_t : \langle\!\langle \alpha \times \mathsf{bool} \rangle\!\rangle \\ \ell_u : \langle\!\langle \alpha \times \mathsf{string} \rangle\!\rangle \end{array} \right\} \end{array} \right\}$$

Encoded module

Example of encoding into F^{ω}

Source code

```
1  module M = struct
2    module X1 : S = struct
3      type t = int
4      let x = 42
5      type u = t -> int
6    end
7    module X2 = struct
8      type t = X1.t * bool
9      type u = X1.t * string
10   end
11 end
12
```

Signature

$$
\begin{array}{l}
\exists \alpha.\mathsf{sig} \\
\qquad \begin{array}{lll}
\mathsf{module}\ X_1 : & \mathsf{sig}\ \mathsf{type}\ t = \alpha & \\
& \qquad \mathsf{val}\ x : \alpha & \\
& \qquad \mathsf{type}\ u = \alpha \to \mathsf{int}\ \mathsf{end} & \\
\mathsf{module}\ X_2 : & \mathsf{sig}\ \mathsf{type}\ t = \alpha \times \mathtt{bool} & \\
& \qquad \mathsf{type}\ u = \alpha \times \mathsf{string}\ \mathsf{end} &
\end{array} \\
\mathsf{end}
\end{array}
$$

Encoded module

Example of encoding into F^{ω}

Source code

```
1  module M = struct
2    module X1 : S = struct
3      type t = int
4      let x = 42
5      type u = t -> int
6    end
7    module X2 = struct
8      type t = X1.t * bool
9      type u = X1.t * string
10   end
11 end
12
```

Signature

$$
\begin{array}{l}
\exists \alpha.\mathsf{sig} \\
\qquad \begin{array}{lll}
\mathsf{module}\ X_1 : & \mathsf{sig}\ \mathsf{type}\ t = \alpha & \\
& \qquad \mathsf{val}\ x : \alpha & \\
& \qquad \mathsf{type}\ u = \alpha \to \mathsf{int}\ \mathsf{end} & \\
\mathsf{module}\ X_2 : & \mathsf{sig}\ \mathsf{type}\ t = \alpha \times \mathtt{bool} & \\
& \qquad \mathsf{type}\ u = \alpha \times \mathsf{string}\ \mathsf{end} &
\end{array} \\
\mathsf{end}
\end{array}
$$

Encoded module

$$e =$$

Example of encoding into F^{ω}

Source code

```
1  module M = struct
2    module X1 : S = struct
3      type t = int
4      let x = 42
5      type u = t -> int
6    end
7    module X2 = struct
8      type t = X1.t * bool
9      type u = X1.t * string
10   end
11 end
12
```

Signature

$$\exists \alpha.\mathsf{sig}$$
$$\quad \mathsf{module}\ X_1 : \mathsf{sig}\ \mathsf{type}\ t = \alpha$$
$$\quad\quad \mathsf{val}\ x : \alpha$$
$$\quad\quad \mathsf{type}\ u = \alpha \to \mathsf{int}\ \mathsf{end}$$
$$\quad \mathsf{module}\ X_2 : \mathsf{sig}\ \mathsf{type}\ t = \alpha \times \mathtt{bool}$$
$$\quad\quad \mathsf{type}\ u = \alpha \times \mathsf{string}\ \mathsf{end}$$
$$\mathsf{end}$$

Encoded module

$$e = \{\ell_{X1} = \{\ell_t = \langle\langle \mathtt{int} \rangle\rangle, \ell_x = 42, \ell_u = \langle\langle \mathtt{int} \times \mathtt{int} \rangle\rangle\}\}$$

Example of encoding into F^{ω}

Source code

```
1  module M = struct
2    module X1 : S = struct
3      type t = int
4      let x = 42
5      type u = t -> int
6    end
7    module X2 = struct
8      type t = X1.t * bool
9      type u = X1.t * string
10   end
11 end
12
```

Signature

$$\exists \alpha.\mathsf{sig}$$
$$\quad \mathsf{module}\ X_1 : \mathsf{sig}\ \mathsf{type}\ t = \alpha$$
$$\qquad \mathsf{val}\ x : \alpha$$
$$\qquad \mathsf{type}\ u = \alpha \to \mathsf{int}\ \mathsf{end}$$
$$\quad \mathsf{module}\ X_2 : \mathsf{sig}\ \mathsf{type}\ t = \alpha \times \mathtt{bool}$$
$$\qquad \mathsf{type}\ u = \alpha \times \mathsf{string}\ \mathsf{end}$$
$$\mathsf{end}$$

Encoded module

$$e = \mathsf{pack}\langle \mathsf{int}, \{\ell_{X1} = \{\ell_t = \langle\!\langle \mathsf{int} \rangle\!\rangle, \ell_x = 42, \ell_u = \langle\!\langle \mathsf{int} \times \mathsf{int} \rangle\!\rangle\}\}\rangle$$

Example of encoding into F^ω

Source code

```
1  module M = struct
2    module X1 : S = struct
3      type t = int
4      let x = 42
5      type u = t -> int
6    end
7    module X2 = struct
8      type t = X1.t * bool
9      type u = X1.t * string
10   end
11 end
12
```

Signature

$$\exists \alpha.\text{sig}$$
$$\text{module } X_1 : \text{ sig type } t = \alpha$$
$$\text{val } x : \alpha$$
$$\text{type } u = \alpha \to \text{int end}$$
$$\text{module } X_2 : \text{ sig type } t = \alpha \times \texttt{bool}$$
$$\text{type } u = \alpha \times \text{string end}$$
$$\text{end}$$

Encoded module

$$e = \text{pack}\langle \texttt{int}, \{\ell_{X1} = \{\ell_t = \langle\!\langle \texttt{int} \rangle\!\rangle, \ell_x = 42, \ell_u = \langle\!\langle \texttt{int} \times \texttt{int} \rangle\!\rangle\}\}\rangle$$
$$\{\ell_{X2} = \{\ell_t = \langle\!\langle \alpha \times \texttt{bool} \rangle\!\rangle, \ell_u = \langle\!\langle \alpha \times \texttt{int} \rangle\!\rangle\}\}$$

Example of encoding into F^ω

Source code

```
1  module M = struct
2    module X1 : S = struct
3      type t = int
4      let x = 42
5      type u = t -> int
6    end
7    module X2 = struct
8      type t = X1.t * bool
9      type u = X1.t * string
10   end
11 end
12
```

Signature

$$\exists\alpha.\text{sig}$$
$$\text{module } X_1 : \text{ sig type } t = \alpha$$
$$\text{val } x : \alpha$$
$$\text{type } u = \alpha \to \text{int end}$$
$$\text{module } X_2 : \text{ sig type } t = \alpha \times \texttt{bool}$$
$$\text{type } u = \alpha \times \text{string end}$$
$$\text{end}$$

Encoded module

$$e = \text{unpack}\langle\alpha, y_1\rangle = \text{pack}\langle\texttt{int}, \{\ell_{X1} = \{\ell_t = \langle\!\langle\texttt{int}\rangle\!\rangle, \ell_x = 42, \ell_u = \langle\!\langle\texttt{int} \times \texttt{int}\rangle\!\rangle\}\}\rangle$$
$$\text{in} \qquad \{\ell_{X2} = \{\ell_t = \langle\!\langle\alpha \times \texttt{bool}\rangle\!\rangle, \ell_u = \langle\!\langle\alpha \times \texttt{int}\rangle\!\rangle\}\}$$

Example of encoding into F^{ω}

Source code

```
1  module M = struct
2    module X1 : S = struct
3      type t = int
4      let x = 42
5      type u = t -> int
6    end
7    module X2 = struct
8      type t = X1.t * bool
9      type u = X1.t * string
10   end
11 end
12
```

Signature

$$\exists\alpha.\text{sig}$$
$$\text{module } X_1 : \text{sig type } t = \alpha$$
$$\text{val } x : \alpha$$
$$\text{type } u = \alpha \to \text{int end}$$
$$\text{module } X_2 : \text{sig type } t = \alpha \times \texttt{bool}$$
$$\text{type } u = \alpha \times \text{string end}$$
$$\text{end}$$

Encoded module

$$e = \text{unpack}\langle\alpha, y_1\rangle = \text{pack}\langle\texttt{int}, \{\ell_{X1} = \{\ell_t = \langle\!\langle\texttt{int}\rangle\!\rangle, \ell_x = 42, \ell_u = \langle\!\langle\texttt{int} \times \texttt{int}\rangle\!\rangle\}\}\rangle$$
$$\text{in unpack}\langle\varnothing, y_2\rangle = \{\ell_{X2} = \{\ell_t = \langle\!\langle\alpha \times \texttt{bool}\rangle\!\rangle, \ell_u = \langle\!\langle\alpha \times \texttt{int}\rangle\!\rangle\}\}$$
$$\text{in}$$

Example of encoding into F^{ω}

Source code

```
1  module M = struct
2    module X1 : S = struct
3      type t = int
4      let x = 42
5      type u = t -> int
6    end
7    module X2 = struct
8      type t = X1.t * bool
9      type u = X1.t * string
10   end
11 end
12
```

Signature

$$\exists \alpha.\text{sig}$$
$$\text{module } X_1 : \text{ sig type } t = \alpha$$
$$\text{val } x : \alpha$$
$$\text{type } u = \alpha \to \text{int end}$$
$$\text{module } X_2 : \text{ sig type } t = \alpha \times \texttt{bool}$$
$$\text{type } u = \alpha \times \text{string end}$$
$$\text{end}$$

Encoded module

$$e = \text{unpack}\langle \alpha, y_1 \rangle = \text{pack}\langle \texttt{int}, \{\ell_{X1} = \{\ell_t = \langle\!\langle \texttt{int} \rangle\!\rangle, \ell_x = 42, \ell_u = \langle\!\langle \texttt{int} \times \texttt{int} \rangle\!\rangle\}\}\rangle$$
$$\text{in unpack}\langle \varnothing, y_2 \rangle = \{\ell_{X2} = \{\ell_t = \langle\!\langle \alpha \times \texttt{bool} \rangle\!\rangle, \ell_u = \langle\!\langle \alpha \times \texttt{int} \rangle\!\rangle\}\}$$
$$\text{in pack}\langle \alpha, \{\ell_{X1} = (y_1.\ell_{X1}), \ell_{X2} = (y_2.\ell_{X2})\}\rangle$$

Existentials inside applicative functors and skolemization

Lifting existentials

Existentials inside applicative functors and skolemization

Lifting existentials

✓ Through products/records types

Existentials inside applicative functors and skolemization

Lifting existentials

- ✓ Through products/records types

Existentials inside applicative functors and skolemization

Lifting existentials

✓ Through products/records types $\rightarrow$ *repacking* pattern

Existentials inside applicative functors and skolemization

Lifting existentials

✓ Through products/records types $\rightarrow$ *repacking* pattern : $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$

Existentials inside applicative functors and skolemization

Lifting existentials

- ✓ Through products/records types $\rightarrow$ *repacking* pattern : $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$
- Through arrow types

Existentials inside applicative functors and skolemization

Lifting existentials

- ✓ Through products/records types $\rightarrow$ *repacking* pattern : $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$
- Through arrow types

Existentials inside applicative functors and skolemization

Lifting existentials

- ✓ Through products/records types $\rightarrow$ *repacking* pattern : $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$
- Through arrow types : $(\tau \rightarrow \exists\alpha.\sigma) \rightsquigarrow \exists\alpha.(\tau \rightarrow \sigma)$

Existentials inside applicative functors and skolemization

Lifting existentials

- ✓ Through products/records types $\rightarrow$ *repacking* pattern : $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$
- Through arrow types : $(\tau \rightarrow \exists\alpha.\sigma) \rightsquigarrow \exists\alpha.(\tau \rightarrow \sigma)$
  $\times$ unsound in general !

Existentials inside applicative functors and skolemization

Lifting existentials

- ✓ Through products/records types $\rightarrow$ *repacking* pattern : $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$
- Through arrow types : $(\tau \rightarrow \exists\alpha.\sigma) \rightsquigarrow \exists\alpha.(\tau \rightarrow \sigma)$
  $\times$ unsound in general !

$$\vdash \lambda(b : \text{bool}).\ \text{if } b \text{ then } (\text{pack } \langle \text{int}, 42 \rangle \text{ as } \exists\alpha.\alpha)$$
$$\text{else } (\text{pack } \langle \text{string}, \text{"}\textit{EPFL}\text{"} \rangle \text{ as } \exists\alpha.\alpha)$$

Existentials inside applicative functors and skolemization

Lifting existentials

- ✓ Through products/records types $\rightarrow$ *repacking* pattern : $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$
- Through arrow types : $(\tau \rightarrow \exists\alpha.\sigma) \rightsquigarrow \exists\alpha.(\tau \rightarrow \sigma)$
  × unsound in general !

$$\vdash \lambda(b : \text{bool}).\ \text{if } b \text{ then } (\text{pack } \langle \text{int}, 42 \rangle \text{ as } \exists\alpha.\alpha) \quad : \text{bool} \rightarrow \exists\alpha.\alpha$$
$$\text{else } (\text{pack } \langle \text{string}, \text{"}\textit{EPFL}\text{"} \rangle \text{ as } \exists\alpha.\alpha)$$

Existentials inside applicative functors and skolemization

Lifting existentials

- ✓ Through products/records types $\rightarrow$ *repacking* pattern : $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$
- Through arrow types : $(\tau \rightarrow \exists\alpha.\sigma) \rightsquigarrow \exists\alpha.(\tau \rightarrow \sigma)$
  × unsound in general !

$$\vdash \lambda(b : \text{bool}).\ \text{if } b \text{ then } (\text{pack } \langle \text{int}, 42 \rangle \text{ as } \exists\alpha.\alpha) \quad : \text{bool} \rightarrow \exists\alpha.\alpha$$
$$\text{else } (\text{pack } \langle \text{string}, \text{"}\textit{EPFL}\text{"} \rangle \text{ as } \exists\alpha.\alpha)$$

wrong to give it the type : $\exists\alpha.\text{bool} \rightarrow \alpha$

Existentials inside applicative functors and skolemization

Lifting existentials

- ✓ Through products/records types $\rightarrow$ *repacking* pattern : $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$
- Through arrow types : $(\tau \rightarrow \exists\alpha.\sigma) \rightsquigarrow \exists\alpha.(\tau \rightarrow \sigma)$
  $\times$ unsound in general !

$$\vdash \lambda(b : \mathsf{bool}).\ \mathsf{if}\ b\ \mathsf{then}\ \ (\mathsf{pack}\ \langle \mathsf{int}, 42\rangle\ \mathsf{as}\ \exists\alpha.\alpha) \qquad : \mathsf{bool} \rightarrow \exists\alpha.\alpha$$
$$\mathsf{else}\ \ (\mathsf{pack}\ \langle \mathsf{string}, \text{"}\mathit{EPFL}\text{"}\rangle\ \mathsf{as}\ \exists\alpha.\alpha)$$

wrong to give it the type : $\exists\alpha.\mathsf{bool} \rightarrow \alpha$

- Through forall types :

Existentials inside applicative functors and skolemization

Lifting existentials

- ✓ Through products/records types $\rightarrow$ *repacking* pattern : $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$
- Through arrow types : $(\tau \rightarrow \exists\alpha.\sigma) \rightsquigarrow \exists\alpha.(\tau \rightarrow \sigma)$
  $\times$ unsound in general !

$$\vdash \lambda(b : \mathsf{bool}).\ \mathsf{if}\ b\ \mathsf{then}\ \ (\mathsf{pack}\ \langle \mathsf{int}, 42\rangle\ \mathsf{as}\ \exists\alpha.\alpha) \qquad : \mathsf{bool} \rightarrow \exists\alpha.\alpha$$
$$\mathsf{else}\ \ (\mathsf{pack}\ \langle \mathsf{string}, \text{"}\mathit{EPFL}\text{"}\rangle\ \mathsf{as}\ \exists\alpha.\alpha)$$

  wrong to give it the type : $\exists\alpha.\mathsf{bool} \rightarrow \alpha$

- Through forall types :

Existentials inside applicative functors and skolemization

Lifting existentials

- ✓ Through products/records types $\rightarrow$ *repacking* pattern : $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$
- Through arrow types : $(\tau \rightarrow \exists\alpha.\sigma) \rightsquigarrow \exists\alpha.(\tau \rightarrow \sigma)$
  × unsound in general !

$$\vdash \lambda(b : \mathsf{bool}).\ \mathsf{if}\ b\ \mathsf{then}\ \ (\mathsf{pack}\ \langle \mathsf{int}, 42\rangle\ \mathsf{as}\ \exists\alpha.\alpha) \qquad : \mathsf{bool} \rightarrow \exists\alpha.\alpha$$
$$\mathsf{else}\ \ (\mathsf{pack}\ \langle \mathsf{string}, \text{"}\mathit{EPFL}\text{"}\rangle\ \mathsf{as}\ \exists\alpha.\alpha)$$

wrong to give it the type : $\exists\alpha.\mathsf{bool} \rightarrow \alpha$

- Through forall types : $(\forall\beta.\exists\alpha.\tau) \rightsquigarrow \exists\alpha'.(\forall\beta.\tau[\alpha \mapsto \alpha'(\beta)])$

Existentials inside applicative functors and skolemization

Lifting existentials

- ✓ Through products/records types $\rightarrow$ *repacking* pattern : $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$
- Through arrow types : $(\tau \rightarrow \exists\alpha.\sigma) \rightsquigarrow \exists\alpha.(\tau \rightarrow \sigma)$

  $\times$ unsound in general !

$$\vdash \lambda(b : \mathsf{bool}).\ \mathsf{if}\ b\ \mathsf{then}\ \ (\mathsf{pack}\ \langle \mathsf{int}, 42\rangle\ \mathsf{as}\ \exists\alpha.\alpha) \qquad : \mathsf{bool} \rightarrow \exists\alpha.\alpha$$
$$\mathsf{else}\ \ (\mathsf{pack}\ \langle \mathsf{string}, \text{"}\mathit{EPFL}\text{"}\rangle\ \mathsf{as}\ \exists\alpha.\alpha)$$

  wrong to give it the type : $\exists\alpha.\mathsf{bool} \rightarrow \alpha$

- Through forall types : $(\forall\beta.\exists\alpha.\tau) \rightsquigarrow \exists\alpha'.(\forall\beta.\tau[\alpha \mapsto \alpha'(\beta)])$

  $\times$ not expressible in F^ω

Existentials inside applicative functors and skolemization

Lifting existentials

- ✓ Through products/records types $\rightarrow$ *repacking* pattern : $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$
- Through arrow types : $(\tau \rightarrow \exists\alpha.\sigma) \rightsquigarrow \exists\alpha.(\tau \rightarrow \sigma)$
  $\times$ unsound in general !

$$\vdash \lambda(b : \mathsf{bool}).\ \mathsf{if}\ b\ \mathsf{then}\ \ (\mathsf{pack}\ \langle \mathsf{int}, 42\rangle\ \mathsf{as}\ \exists\alpha.\alpha) \qquad : \mathsf{bool} \rightarrow \exists\alpha.\alpha$$
$$\mathsf{else}\ \ (\mathsf{pack}\ \langle \mathsf{string}, \text{"}\mathit{EPFL}\text{"}\rangle\ \mathsf{as}\ \exists\alpha.\alpha)$$

wrong to give it the type : $\exists\alpha.\mathsf{bool} \rightarrow \alpha$

- Through forall types : $(\forall\beta.\exists\alpha.\tau) \rightsquigarrow \exists\alpha'.(\forall\beta.\tau[\alpha \mapsto \alpha'(\beta)])$
  $\times$ not expressible in F^ω

The key intuition

- Existentials can be used for abstraction only, not for compatibility

Existentials inside applicative functors and skolemization

Lifting existentials

✓ Through products/records types $\rightarrow$ *repacking* pattern : $(\exists\alpha.\tau, \sigma) \rightsquigarrow \exists\alpha.(\tau, \sigma)$

- Through arrow types : $(\tau \rightarrow \exists\alpha.\sigma) \rightsquigarrow \exists\alpha.(\tau \rightarrow \sigma)$

  $\times$ unsound in general !

$$\vdash \lambda(b : \mathsf{bool}).\ \mathsf{if}\ b\ \mathsf{then}\ \ (\mathsf{pack}\ \langle \mathsf{int}, 42\rangle\ \mathsf{as}\ \exists\alpha.\alpha) \qquad : \mathsf{bool} \rightarrow \exists\alpha.\alpha$$
$$\mathsf{else}\ \ (\mathsf{pack}\ \langle \mathsf{string}, \text{"}\mathit{EPFL}\text{"}\rangle\ \mathsf{as}\ \exists\alpha.\alpha)$$

wrong to give it the type : $\exists\alpha.\mathsf{bool} \rightarrow \alpha$

- Through forall types : $(\forall\beta.\exists\alpha.\tau) \rightsquigarrow \exists\alpha'.(\forall\beta.\tau[\alpha \mapsto \alpha'(\beta)])$

  $\times$ not expressible in F^ω

The key intuition

- Existentials can be used for abstraction only, not for compatibility
- Dependencies on type parameters can be lifted via higher order types

F^{ω} with transparent existentials - Adding new constructs

$$\tau := \ \ldots \mid \exists^{\blacktriangledown}\alpha.\tau$$

$$\begin{aligned} e := \ & \ldots \mid \mathsf{pack}\ \langle \tau, e \rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\alpha.\tau \\ & \mid \mathsf{unpack}\ \langle \alpha, x \rangle = e\ \mathsf{in}\ e \end{aligned}$$

F^{ω} with transparent existentials - Adding new constructs

$$
\begin{aligned}
\tau :=\ & \ldots \mid \exists^{\blacktriangledown}\alpha.\tau \\
& \mid \exists^{\triangledown\tau}\alpha.\tau \\
e :=\ & \ldots \mid \mathsf{pack}\ \langle \tau, e \rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\alpha.\tau \\
& \mid \mathsf{unpack}\ \langle \alpha, x \rangle = e\ \mathsf{in}\ e
\end{aligned}
$$

F^{ω} with transparent existentials - Adding new constructs

$$
\begin{aligned}
\tau :=\ & \ldots \mid \exists^{\blacktriangledown}\alpha.\tau \\
& \mid \exists^{\nabla\tau}\alpha.\tau \\
e :=\ & \ldots \mid \mathsf{pack}\ \langle \tau, e \rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\alpha.\tau \\
& \mid \mathsf{unpack}\ \langle \alpha, x \rangle = e\ \mathsf{in}\ e \\
& \mid \mathsf{pack}\ e\ \mathsf{as}\ \exists^{\nabla\tau}\alpha.\sigma
\end{aligned}
$$

F^{ω} with transparent existentials - Adding new constructs

$$
\begin{aligned}
\tau := \quad & \ldots \mid \exists^{\blacktriangledown} \alpha.\tau \\
& \mid \exists^{\nabla \tau} \alpha.\tau \\
e := \quad & \ldots \mid \mathsf{pack}\ \langle \tau, e \rangle\ \mathsf{as}\ \exists^{\blacktriangledown} \alpha.\tau \\
& \mid \mathsf{unpack}\ \langle \alpha, x \rangle = e\ \mathsf{in}\ e \\
& \mid \mathsf{pack}\ e\ \mathsf{as}\ \exists^{\nabla \tau} \alpha.\sigma \\
& \mid \mathsf{repack}^{\nabla}\ \langle \alpha, x \rangle = e_1\ \mathsf{in}\ e_2
\end{aligned}
$$

F^{ω} with transparent existentials - Adding new constructs

$$
\begin{aligned}
\tau :=\ & \ldots \mid \exists^{\blacktriangledown}\alpha.\tau \\
& \mid \exists^{\nabla\tau}\alpha.\tau \\
e :=\ & \ldots \mid \mathsf{pack}\ \langle \tau, e\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\alpha.\tau \\
& \mid \mathsf{unpack}\ \langle \alpha, x\rangle = e\ \mathsf{in}\ e \\
& \mid \mathsf{pack}\ e\ \mathsf{as}\ \exists^{\nabla\tau}\alpha.\sigma \\
& \mid \mathsf{repack}^{\nabla}\ \langle \alpha, x\rangle = e_1\ \mathsf{in}\ e_2 \\
& \mid \mathsf{seal}\ e
\end{aligned}
$$

F^{ω} with transparent existentials - Adding new constructs

$$
\begin{aligned}
\tau := \ & \ldots \mid \exists^{\blacktriangledown}\alpha.\tau \\
& \mid \exists^{\nabla\tau}\alpha.\tau \\
e := \ & \ldots \mid \mathsf{pack}\ \langle \tau, e \rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\alpha.\tau \\
& \mid \mathsf{unpack}\ \langle \alpha, x \rangle = e\ \mathsf{in}\ e \\
& \mid \mathsf{pack}\ e\ \mathsf{as}\ \exists^{\nabla\tau}\alpha.\sigma \\
& \mid \mathsf{repack}^{\nabla}\ \langle \alpha, x \rangle = e_1\ \mathsf{in}\ e_2 \\
& \mid \mathsf{seal}\ e \\
& \mid \mathsf{lift}^{\rightarrow}\ e
\end{aligned}
$$

F^{ω} with transparent existentials - Adding new constructs

$$
\begin{aligned}
\tau := \ & \ldots \mid \exists^{\blacktriangledown}\alpha.\tau \\
& \mid \exists^{\nabla\tau}\alpha.\tau \\
e := \ & \ldots \mid \mathsf{pack}\ \langle \tau, e \rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\alpha.\tau \\
& \mid \mathsf{unpack}\ \langle \alpha, x \rangle = e\ \mathsf{in}\ e \\
& \mid \mathsf{pack}\ e\ \mathsf{as}\ \exists^{\nabla\tau}\alpha.\sigma \\
& \mid \mathsf{repack}^{\nabla}\ \langle \alpha, x \rangle = e_1\ \mathsf{in}\ e_2 \\
& \mid \mathsf{seal}\ e \\
& \mid \mathsf{lift}^{\rightarrow} e \\
& \mid \mathsf{lift}^{\forall} e
\end{aligned}
$$

F^{ω} with transparent existentials - Typing rules

F^ω with transparent existentials - Typing rules

F-Hide

$$\frac{\Gamma \overset{F^\omega}{\vdash} \exists^{\nabla\tau}\alpha.\sigma : \star \qquad \Gamma \overset{F^\omega}{\vdash} e : \sigma[\alpha \mapsto \tau]}{\Gamma \overset{F^\omega}{\vdash} \mathsf{pack}\ e\ \mathsf{as}\ \exists^{\nabla\tau}\alpha.\sigma : \exists^{\nabla\tau}\alpha.\sigma}$$

F^{ω} with transparent existentials - Typing rules

$$\textsc{F-Hide}\quad \frac{\Gamma \overset{F^{\omega}}{\vdash} \exists^{\nabla\tau}\alpha.\sigma : \star \qquad \Gamma \overset{F^{\omega}}{\vdash} e : \sigma[\alpha \mapsto \tau]}{\Gamma \overset{F^{\omega}}{\vdash} \mathsf{pack}\ e\ \mathsf{as}\ \exists^{\nabla\tau}\alpha.\sigma : \exists^{\nabla\tau}\alpha.\sigma}$$

$$\textsc{F-Repack}\quad \frac{\Gamma \overset{F^{\omega}}{\vdash} e_1 : \exists^{\nabla\tau}\alpha.\sigma \qquad \Gamma, \alpha, x : \sigma \overset{F^{\omega}}{\vdash} e_2 : \sigma'}{\Gamma \overset{F^{\omega}}{\vdash} \mathsf{repack}^{\nabla}\ \langle \alpha, x \rangle = e_1\ \mathsf{in}\ e_2 : \exists^{\nabla\tau}\alpha.\sigma'}$$

F^ω with transparent existentials - Typing rules

F-Hide

$$\frac{\Gamma \overset{F^\omega}{\vdash} \exists^{\nabla\tau}\alpha.\sigma : \star \qquad \Gamma \overset{F^\omega}{\vdash} e : \sigma[\alpha \mapsto \tau]}{\Gamma \overset{F^\omega}{\vdash} \mathsf{pack}\ e\ \mathsf{as}\ \exists^{\nabla\tau}\alpha.\sigma : \exists^{\nabla\tau}\alpha.\sigma}$$

F-Repack

$$\frac{\Gamma \overset{F^\omega}{\vdash} e_1 : \exists^{\nabla\tau}\alpha.\sigma \qquad \Gamma, \alpha, x : \sigma \overset{F^\omega}{\vdash} e_2 : \sigma'}{\Gamma \overset{F^\omega}{\vdash} \mathsf{repack}^{\nabla} \langle \alpha, x \rangle = e_1\ \mathsf{in}\ e_2 : \exists^{\nabla\tau}\alpha.\sigma'}$$

F-Seal

$$\frac{\Gamma \overset{F^\omega}{\vdash} e : \exists^{\nabla\tau}\alpha.\sigma}{\Gamma \overset{F^\omega}{\vdash} \mathsf{seal}\ e : \exists^{\blacktriangledown}\alpha.\sigma}$$

F^ω with transparent existentials - Typing rules

F-Hide

$$\frac{\Gamma \overset{F^\omega}{\vdash} \exists^{\nabla\tau}\alpha.\sigma : \star \qquad \Gamma \overset{F^\omega}{\vdash} e : \sigma[\alpha \mapsto \tau]}{\Gamma \overset{F^\omega}{\vdash} \mathsf{pack}\ e\ \mathsf{as}\ \exists^{\nabla\tau}\alpha.\sigma : \exists^{\nabla\tau}\alpha.\sigma}$$

F-Repack

$$\frac{\Gamma \overset{F^\omega}{\vdash} e_1 : \exists^{\nabla\tau}\alpha.\sigma \qquad \Gamma, \alpha, x : \sigma \overset{F^\omega}{\vdash} e_2 : \sigma'}{\Gamma \overset{F^\omega}{\vdash} \mathsf{repack}^{\nabla} \langle \alpha, x \rangle = e_1\ \mathsf{in}\ e_2 : \exists^{\nabla\tau}\alpha.\sigma'}$$

F-Seal

$$\frac{\Gamma \overset{F^\omega}{\vdash} e : \exists^{\nabla\tau}\alpha.\sigma}{\Gamma \overset{F^\omega}{\vdash} \mathsf{seal}\ e : \exists^{\blacktriangledown}\alpha.\sigma}$$

F-LiftArr

$$\frac{\Gamma \overset{F^\omega}{\vdash} e : \sigma_1 \to \exists^{\nabla\tau}\alpha.\sigma_2}{\Gamma \overset{F^\omega}{\vdash} \mathsf{lift}^{\to} e : \exists^{\nabla\tau}\alpha.\sigma_1 \to \sigma_2}$$

F^ω with transparent existentials - Typing rules

F-Hide

$$\frac{\Gamma \overset{F^\omega}{\vdash} \exists^{\nabla\tau}\alpha.\sigma : \star \qquad \Gamma \overset{F^\omega}{\vdash} e : \sigma[\alpha \mapsto \tau]}{\Gamma \overset{F^\omega}{\vdash} \mathsf{pack}\ e\ \mathsf{as}\ \exists^{\nabla\tau}\alpha.\sigma : \exists^{\nabla\tau}\alpha.\sigma}$$

F-Repack

$$\frac{\Gamma \overset{F^\omega}{\vdash} e_1 : \exists^{\nabla\tau}\alpha.\sigma \qquad \Gamma, \alpha, x : \sigma \overset{F^\omega}{\vdash} e_2 : \sigma'}{\Gamma \overset{F^\omega}{\vdash} \mathsf{repack}^{\nabla} \langle \alpha, x \rangle = e_1\ \mathsf{in}\ e_2 : \exists^{\nabla\tau}\alpha.\sigma'}$$

F-Seal

$$\frac{\Gamma \overset{F^\omega}{\vdash} e : \exists^{\nabla\tau}\alpha.\sigma}{\Gamma \overset{F^\omega}{\vdash} \mathsf{seal}\ e : \exists^{\blacktriangledown}\alpha.\sigma}$$

F-LiftArr

$$\frac{\Gamma \overset{F^\omega}{\vdash} e : \sigma_1 \to \exists^{\nabla\tau}\alpha.\sigma_2}{\Gamma \overset{F^\omega}{\vdash} \mathsf{lift}^{\to} e : \exists^{\nabla\tau}\alpha.\sigma_1 \to \sigma_2}$$

F-LiftAll

$$\frac{\Gamma \overset{F^\omega}{\vdash} e : \forall\alpha.\exists^{\nabla\tau}\beta.\sigma}{\Gamma \overset{F^\omega}{\vdash} \mathsf{lift}^{\forall} e : \exists^{\nabla\lambda\alpha.\tau}\beta'.\forall\alpha.\sigma[\beta \mapsto (\beta'\,\alpha)]}$$

F^{ω} with transparent existentials - is actually vanilla F^{ω}

F^{ω} with transparent existentials - is actually vanilla F^{ω}

$\rightarrow$ transparent existentials are encodable as a library

F^{ω} with transparent existentials - is actually vanilla F^{ω}

$\rightarrow$ transparent existentials are encodable as a library

$$e_0 \triangleq \left\{ \begin{array}{lcl} \mathsf{Pack} & = & \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x : \varphi\,\alpha).\,x \\ \mathsf{Seal} & = & \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x : \varphi\,\alpha).\,\mathsf{pack}\,\langle\alpha, x\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\alpha.\,\varphi\,\alpha \\ \mathsf{Repack} & = & \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x : \varphi\,\alpha).\,\Lambda\psi.\,\lambda(f : \forall\alpha.\,\varphi\,\alpha \to \psi\,\alpha).\,(f\ \alpha\ x) \\ \mathsf{Lift}^{\to} & = & \Lambda\alpha.\,\Lambda\varphi.\,\Lambda\beta.\,\lambda(f : (\beta \to \varphi\,\alpha)).\,f \\ \mathsf{Lift}^{\forall} & = & \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x : (\forall\beta.\,\varphi\,\beta\,(\alpha\,\beta))).\,x \end{array} \right\}$$

F^ω with transparent existentials - is actually vanilla F^ω

$\rightarrow$ transparent existentials are encodable as a library

$$e_0 \triangleq \left\{ \begin{array}{lcl} \mathsf{Pack} & = & \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x : \varphi\,\alpha).\,x \\ \mathsf{Seal} & = & \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x : \varphi\,\alpha).\,\mathsf{pack}\,\langle\alpha, x\rangle\ \mathsf{as}\ \exists^{\blacktriangledown}\alpha.\,\varphi\,\alpha \\ \mathsf{Repack} & = & \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x : \varphi\,\alpha).\,\Lambda\psi.\,\lambda(f : \forall\alpha.\,\varphi\,\alpha \to \psi\,\alpha).\,(f\ \alpha\ x) \\ \mathsf{Lift}^{\to} & = & \Lambda\alpha.\,\Lambda\varphi.\,\Lambda\beta.\,\lambda(f : (\beta \to \varphi\,\alpha)).\,f \\ \mathsf{Lift}^{\forall} & = & \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x : (\forall\beta.\,\varphi\,\beta\,(\alpha\,\beta))).\,x \end{array} \right\}$$

$e_{\mathbb{E}} \triangleq \mathsf{pack}\,\langle(\lambda\alpha.\,\lambda\,\varphi\,.\,\varphi\,\alpha), e_0\rangle\ \mathsf{as}$

F^{ω} with transparent existentials - is actually vanilla F^{ω}

$\rightarrow$ transparent existentials are encodable as a library

$$e_0 \triangleq \left\{ \begin{array}{lcl} \mathsf{Pack} & = & \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x : \varphi\,\alpha).\,x \\ \mathsf{Seal} & = & \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x : \varphi\,\alpha).\,\mathsf{pack}\,\langle\alpha, x\rangle\,\mathsf{as}\,\exists^{\blacktriangledown}\alpha.\,\varphi\,\alpha \\ \mathsf{Repack} & = & \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x : \varphi\,\alpha).\,\Lambda\psi.\,\lambda(f : \forall\alpha.\,\varphi\,\alpha \rightarrow \psi\,\alpha).\,(f\ \alpha\ x) \\ \mathsf{Lift}^{\rightarrow} & = & \Lambda\alpha.\,\Lambda\varphi.\,\Lambda\beta.\,\lambda(f : (\beta \rightarrow \varphi\,\alpha)).\,f \\ \mathsf{Lift}^{\forall} & = & \Lambda\alpha.\,\Lambda\varphi.\,\lambda(x : (\forall\beta.\,\varphi\,\beta\,(\alpha\,\beta))).\,x \end{array} \right\}$$

$$e_{\mathbb{E}} \triangleq \mathsf{pack}\,\langle(\lambda\alpha.\,\lambda\,\varphi\,.\,\varphi\,\alpha), e_0\rangle\,\mathsf{as}$$

$$\exists^{\blacktriangledown}\mathbb{E}\,.\left\{ \begin{array}{lcl} \mathsf{Pack} & : & \forall\alpha.\,\forall\,\varphi\,.\,\varphi\,\alpha \rightarrow \mathbb{E}\,\alpha\,\varphi \\ \mathsf{Seal} & : & \forall\alpha.\,\forall\,\varphi\,.\,\mathbb{E}\,\alpha\,\varphi \rightarrow \exists^{\blacktriangledown}\alpha.\,\varphi\,\alpha \\ \mathsf{Repack} & : & \forall\alpha.\,\forall\,\varphi\,.\,\mathbb{E}\,\alpha\,\varphi \rightarrow \forall\,\psi\,.\,(\forall\alpha.\,\varphi\,\alpha \rightarrow \psi\,\alpha) \rightarrow \mathbb{E}\,\alpha\,\psi \\ \mathsf{Lift}^{\rightarrow} & : & \forall\alpha.\,\forall\,\varphi\,.\,\forall\beta.\,(\beta \rightarrow \mathbb{E}\,\alpha\,\varphi) \rightarrow \mathbb{E}\,\alpha\,(\lambda\alpha.\,\beta \rightarrow \varphi\,\alpha) \\ \mathsf{Lift}^{\forall} & : & \forall\alpha.\,\forall\,\varphi\,.\,(\forall\beta.\,\mathbb{E}\,(\alpha\,\beta)\,(\varphi\,\beta)) \rightarrow \mathbb{E}\,\alpha\,(\lambda\alpha.\,\forall\beta.\,\varphi\,\beta\,(\alpha\,\beta)) \end{array} \right\}$$

Back to applicative functors

E-Typ-Mod-AppFct

$$\frac{\Gamma \vdash \mathsf{S} : \lambda\alpha.\mathcal{C}_a \qquad \Gamma, \alpha, Y : \mathcal{C}_a \vdash \mathtt{M} : \exists\beta.\mathcal{C} \rightsquigarrow e}{\Gamma \vdash (Y : \mathsf{S}) \to \mathtt{M} : \quad \exists\beta'. \quad \forall\alpha.\mathcal{C}_a \to \mathcal{C}[\beta \mapsto \beta'(\alpha)]}$$

$$\rightsquigarrow \qquad ?$$

Back to applicative functors

E-Typ-Mod-AppFct

$$\frac{\Gamma \vdash \mathsf{S} : \lambda\alpha.\mathcal{C}_a \qquad \Gamma, \alpha, Y : \mathcal{C}_a \vdash \mathtt{M} : \exists^{\nabla\tau}\beta.\mathcal{C} \rightsquigarrow e}{\Gamma \vdash (Y : \mathsf{S}) \to \mathtt{M} : \quad \exists\beta'. \quad \forall\alpha.\mathcal{C}_a \to \mathcal{C}[\beta \mapsto \beta'(\alpha)]}$$

$$\rightsquigarrow \qquad ?$$

$$\Gamma, \alpha, Y : \mathcal{C}_a \vdash e \quad : \qquad \exists^{\nabla\tau}\beta.\mathcal{C}$$

Back to applicative functors

E-Typ-Mod-AppFct

$$\frac{\Gamma \vdash \mathsf{S} : \lambda\alpha.\mathcal{C}_a \qquad \Gamma, \alpha, Y : \mathcal{C}_a \vdash \mathsf{M} : \exists^{\nabla\tau}\beta.\mathcal{C} \rightsquigarrow e}{\Gamma \vdash (Y : \mathsf{S}) \to \mathsf{M} : \exists^{\nabla\lambda\alpha.\tau}\beta'.\forall\alpha.\mathcal{C}_a \to \mathcal{C}[\beta \mapsto \beta'(\alpha)]}$$

$$\rightsquigarrow \qquad ?$$

$$\Gamma, \alpha, Y : \mathcal{C}_a \vdash e \quad : \qquad \exists^{\nabla\tau}\beta.\mathcal{C}$$

Back to applicative functors

E-Typ-Mod-AppFct

$$\frac{\Gamma \vdash \mathsf{S} : \lambda\alpha.\mathcal{C}_a \qquad \Gamma, \alpha, Y : \mathcal{C}_a \vdash \mathsf{M} : \exists^{\nabla\tau}\beta.\mathcal{C} \rightsquigarrow e}{\Gamma \vdash (Y : \mathsf{S}) \to \mathsf{M} : \exists^{\nabla\lambda\alpha.\tau}\beta'.\forall\alpha.\mathcal{C}_a \to \mathcal{C}[\beta \mapsto \beta'(\alpha)] \rightsquigarrow (\Lambda\alpha.\lambda(Y : \mathcal{C}_a).e)}$$

$$\Gamma, \alpha, Y : \mathcal{C}_a \vdash e \quad : \quad \exists^{\nabla\tau}\beta.\mathcal{C}$$

$$\Gamma \vdash \Lambda\alpha.\lambda(Y : \mathcal{C}_a).e \quad : \quad \forall\alpha.\mathcal{C}_a \to \exists^{\nabla\tau}\beta.\mathcal{C}$$

Back to applicative functors

E-Typ-Mod-AppFct

$$\frac{\Gamma \vdash \mathsf{S} : \lambda\alpha.\mathcal{C}_a \qquad \Gamma, \alpha, Y : \mathcal{C}_a \vdash \mathsf{M} : \exists^{\nabla\tau}\beta.\mathcal{C} \rightsquigarrow e}{\Gamma \vdash (Y : \mathsf{S}) \to \mathsf{M} : \exists^{\nabla\lambda\alpha.\tau}\beta'.\forall\alpha.\mathcal{C}_a \to \mathcal{C}[\beta \mapsto \beta'(\alpha)]}$$

$$\rightsquigarrow \quad \mathsf{lift}^{\to}\ (\Lambda\alpha.\,\lambda(Y : \mathcal{C}_a).\,e)$$

$$\Gamma, \alpha, Y : \mathcal{C}_a \vdash e \quad : \quad \exists^{\nabla\tau}\beta.\mathcal{C}$$

$$\Gamma \vdash \Lambda\alpha.\,\lambda(Y : \mathcal{C}_a).\,e \quad : \quad \forall\alpha.\mathcal{C}_a \to \exists^{\nabla\tau}\beta.\mathcal{C}$$

$$\Gamma \vdash \mathsf{lift}^{\to}(\Lambda\alpha.\,\lambda(Y : \mathcal{C}_a).\,e) \quad : \quad \forall\alpha.\exists^{\nabla\tau}\beta.\mathcal{C}_a \to \mathcal{C}$$

Back to applicative functors

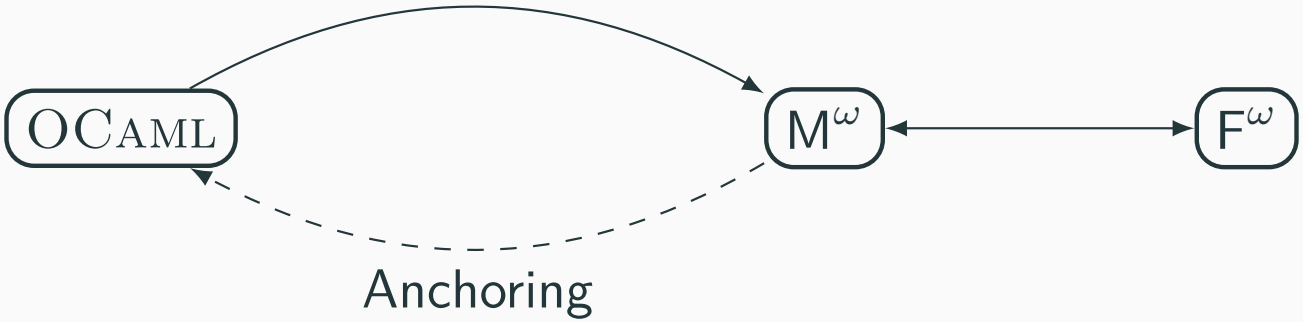
E-Typ-Mod-AppFct

$$\frac{\Gamma \vdash \mathsf{S} : \lambda\alpha.\mathcal{C}_a \qquad \Gamma, \alpha, Y : \mathcal{C}_a \vdash \mathsf{M} : \exists^{\nabla\tau}\beta.\mathcal{C} \rightsquigarrow e}{\Gamma \vdash (Y : \mathsf{S}) \to \mathsf{M} : \exists^{\nabla\lambda\alpha.\tau}\beta'.\forall\alpha.\mathcal{C}_a \to \mathcal{C}[\beta \mapsto \beta'(\alpha)]}$$

$$\rightsquigarrow \mathsf{lift}^{\forall}\ \mathsf{lift}^{\to}\ (\Lambda\alpha.\,\lambda(Y : \mathcal{C}_a).\,e)$$

$$\begin{array}{lll}
\Gamma, \alpha, Y : \mathcal{C}_a \vdash e & : & \exists^{\nabla\tau}\beta.\mathcal{C} \\
\Gamma \vdash \Lambda\alpha.\,\lambda(Y : \mathcal{C}_a).\,e & : & \forall\alpha.\mathcal{C}_a \to \exists^{\nabla\tau}\beta.\mathcal{C} \\
\Gamma \vdash \mathsf{lift}^{\to}(\Lambda\alpha.\,\lambda(Y : \mathcal{C}_a).\,e) & : & \forall\alpha.\exists^{\nabla\tau}\beta.\mathcal{C}_a \to \mathcal{C} \\
\Gamma \vdash \mathsf{lift}^{\forall}\mathsf{lift}^{\to}(\Lambda\alpha.\,\lambda(Y : \mathcal{C}_a).\,e) & : & \exists^{\nabla\lambda\alpha.\tau}\beta'.\forall\alpha.\mathcal{C}_a \to \mathcal{C}
\end{array}$$

The big picture



Conclusion and future work

Take-away

A new specification for OCaml modules - M^{ω}

- Understanding signatures with explicit F^{ω} quantifiers

Take-away

A new specification for OCaml modules - M^{ω}

- Understanding signatures with explicit F^{ω} quantifiers
- A new internal representation for the typechecker

Take-away

A new specification for OCaml modules - M^{ω}

- Understanding signatures with explicit F^{ω} quantifiers
- A new internal representation for the typechecker

Take-away

A new specification for OCaml modules - M^{ω}

- Understanding signatures with explicit F^{ω} quantifiers
- A new internal representation for the typechecker

A new approach to abstraction inside applicative functors

- Transparent existentials types

Take-away

A new specification for OCaml modules - M^{ω}

- Understanding signatures with explicit F^{ω} quantifiers
- A new internal representation for the typechecker

A new approach to abstraction inside applicative functors

- Transparent existentials types
- Skolemization *a posteriori*

Take-away

A new specification for OCaml modules - M^{ω}

- Understanding signatures with explicit F^{ω} quantifiers
- A new internal representation for the typechecker

A new approach to abstraction inside applicative functors

- Transparent existentials types
- Skolemization *a posteriori*

Take-away

A new specification for OCaml modules - M^{ω}

- Understanding signatures with explicit F^{ω} quantifiers
- A new internal representation for the typechecker

A new approach to abstraction inside applicative functors

- Transparent existentials types
- Skolemization *a posteriori*

Future work

- Support a more significant subset of OCaml (missing: recursive modules)

Take-away

A new specification for OCaml modules - M^{ω}

- Understanding signatures with explicit F^{ω} quantifiers
- A new internal representation for the typechecker

A new approach to abstraction inside applicative functors

- Transparent existentials types
- Skolemization *a posteriori*

Future work

- Support a more significant subset of OCaml (missing: recursive modules)
- See if it scales! (practical challenges)

Take-away

A new specification for OCaml modules - M^{ω}

- Understanding signatures with explicit F^{ω} quantifiers
- A new internal representation for the typechecker

A new approach to abstraction inside applicative functors

- Transparent existentials types
- Skolemization *a posteriori*

Future work

- Support a more significant subset of OCaml (missing: recursive modules)
- See if it scales! (practical challenges)
- Mechanize (Coq)

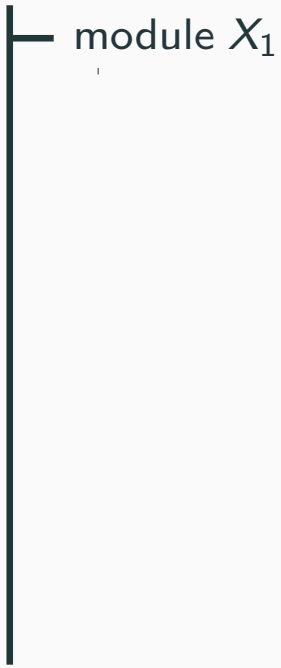
Abstraction, scopes and signatures

Strengthening

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

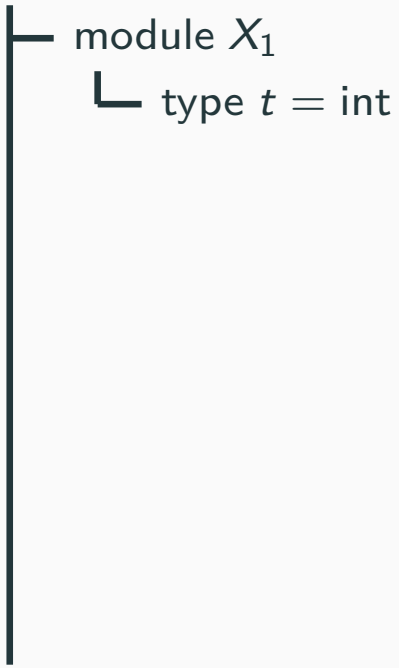
Strengthening

```
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
```



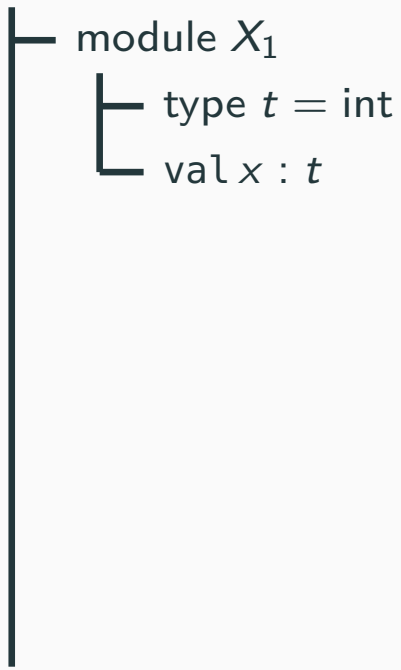
Strengthening

```
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
```



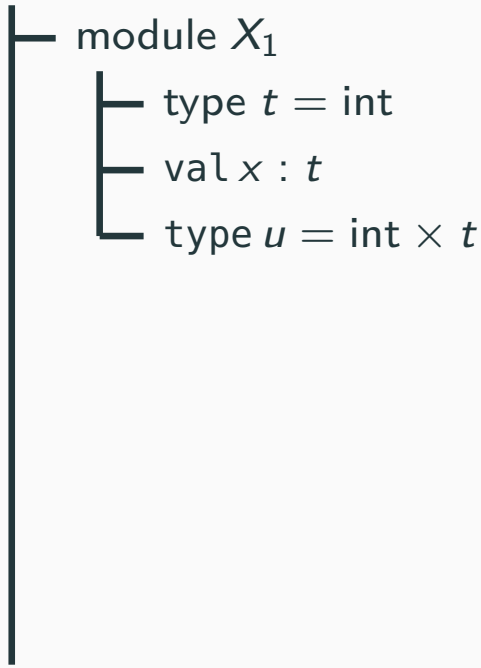
Strengthening

```
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
```



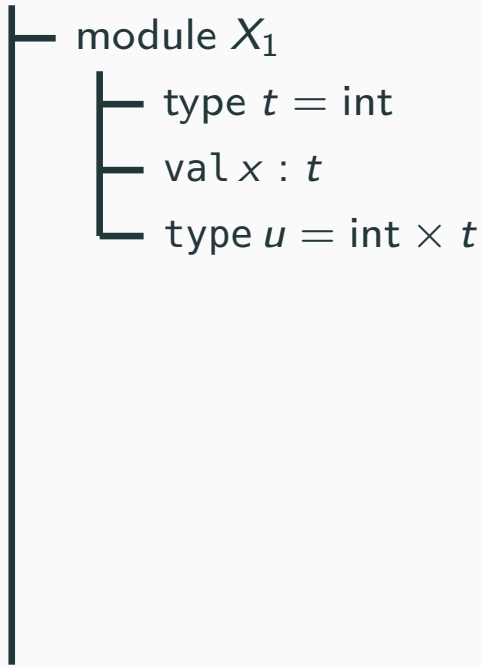
Strengthening

```
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
```



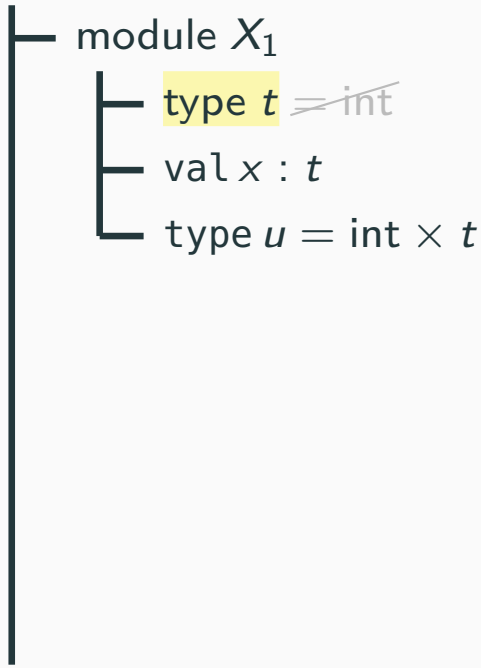
Strengthening

```
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
```



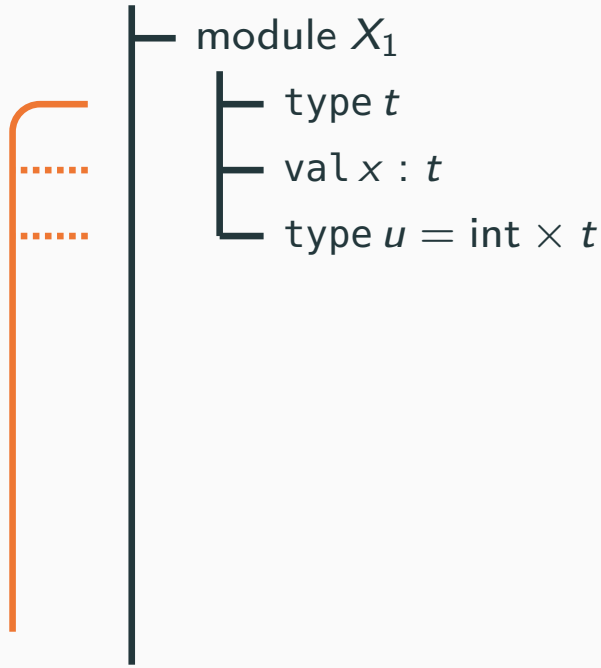
Strengthening

```
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
```



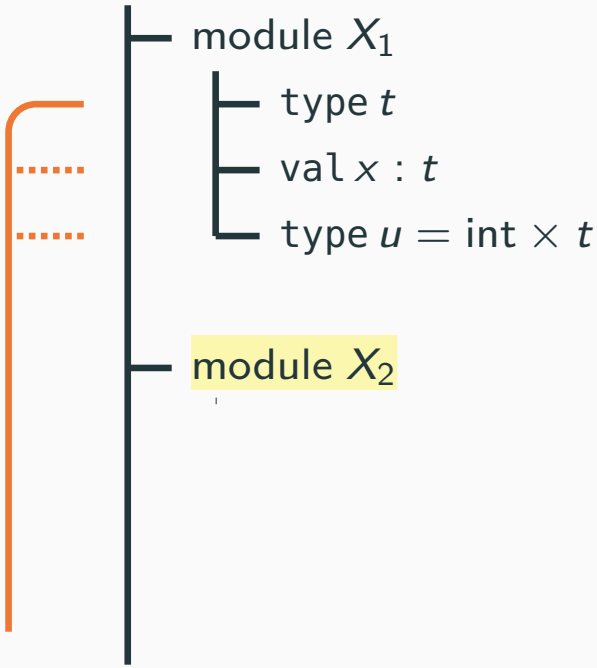
Strengthening

```
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
```



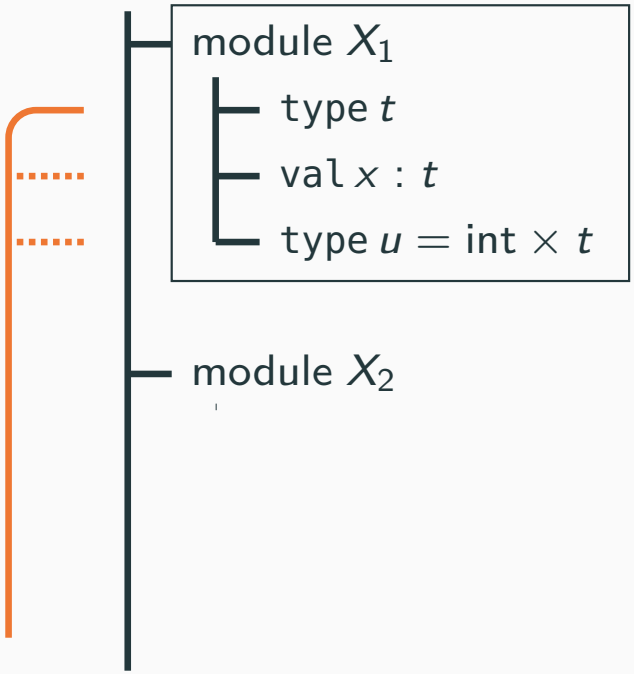
Strengthening

```
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
```



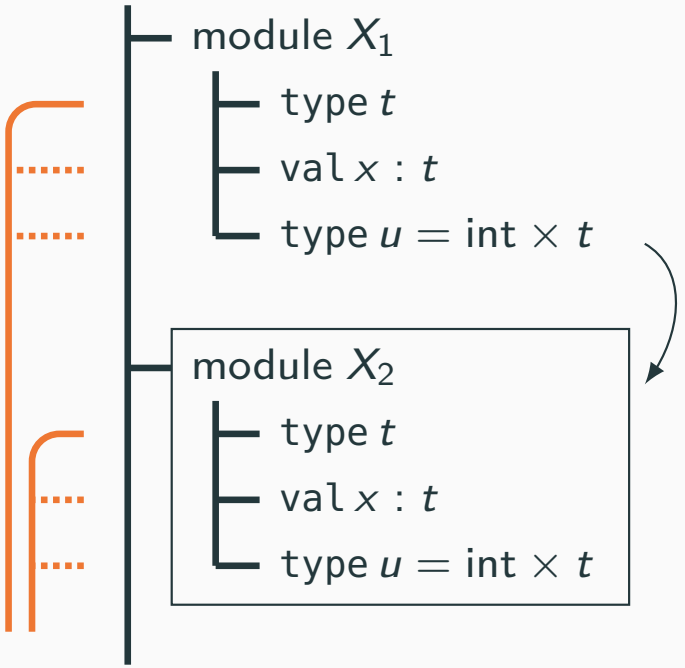
Strengthening

```
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
```



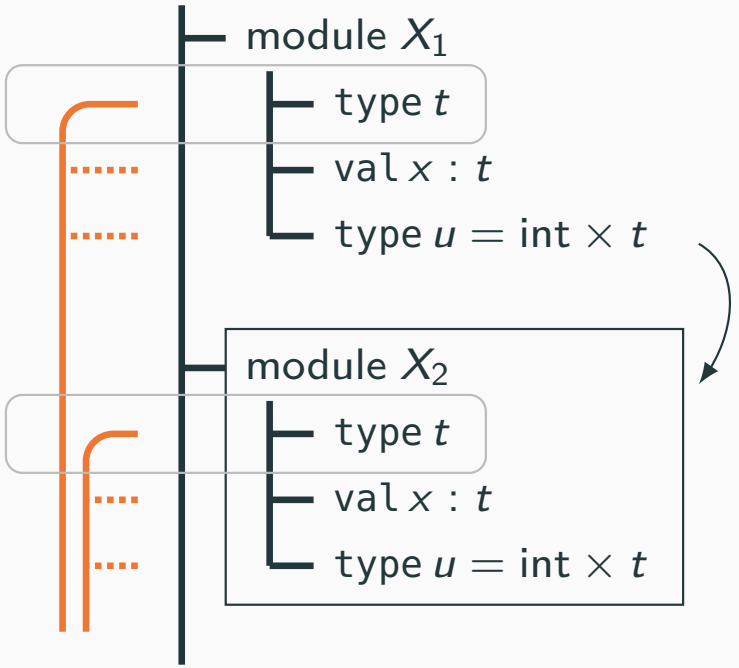
Strengthening

```
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
```



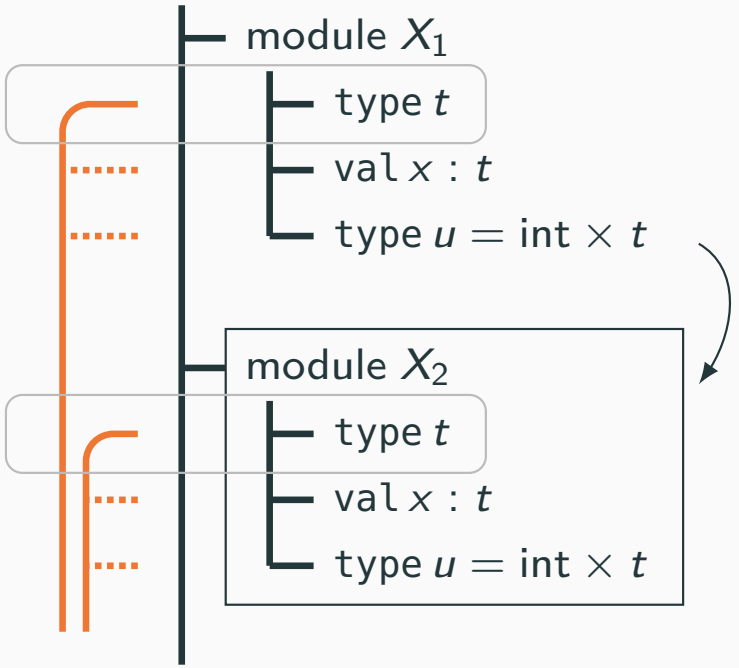
Strengthening

```
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
```



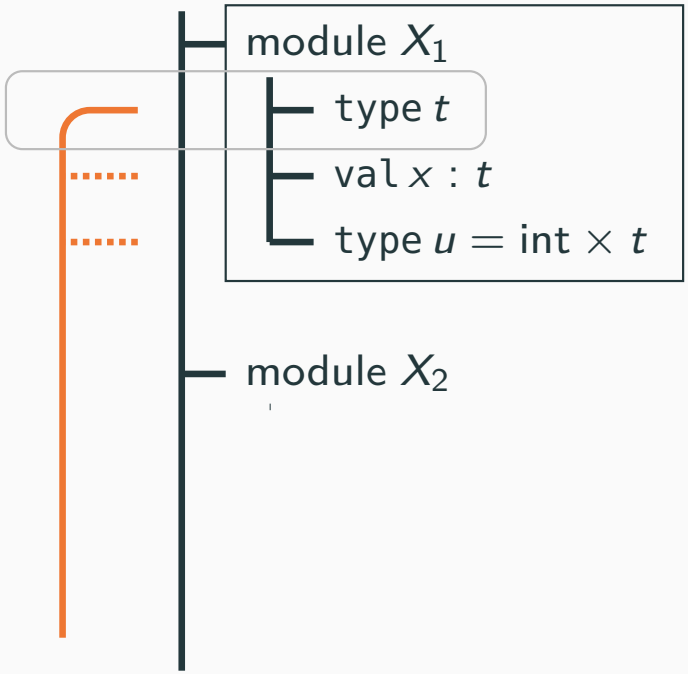
Strengthening

```
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)
```



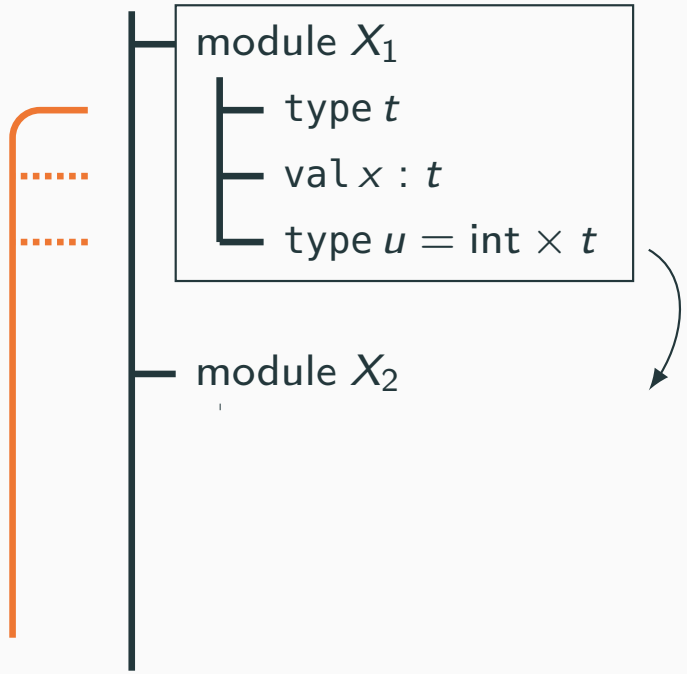
Strengthening

```
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
```



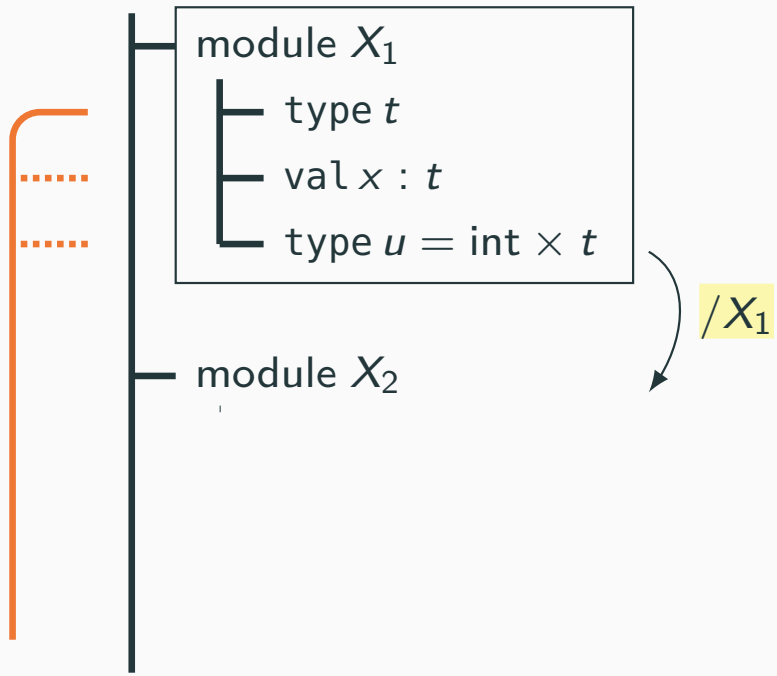
Strengthening

```
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
```



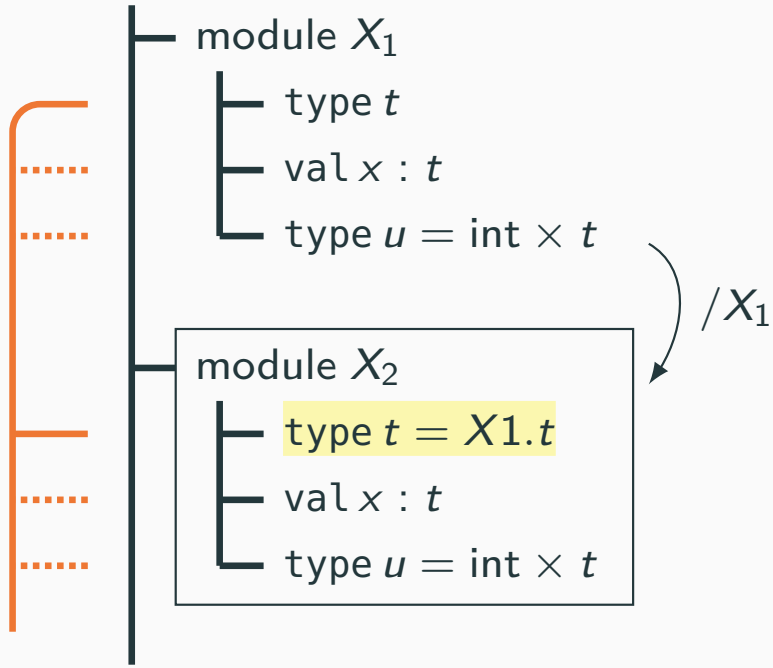
Strengthening

```
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
```



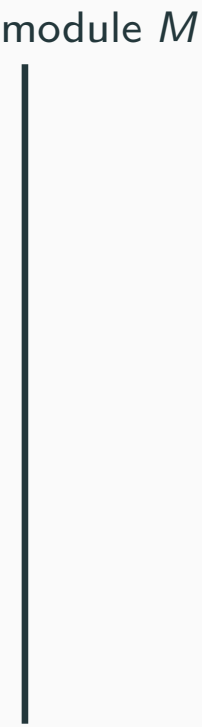
Strengthening

```
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
```



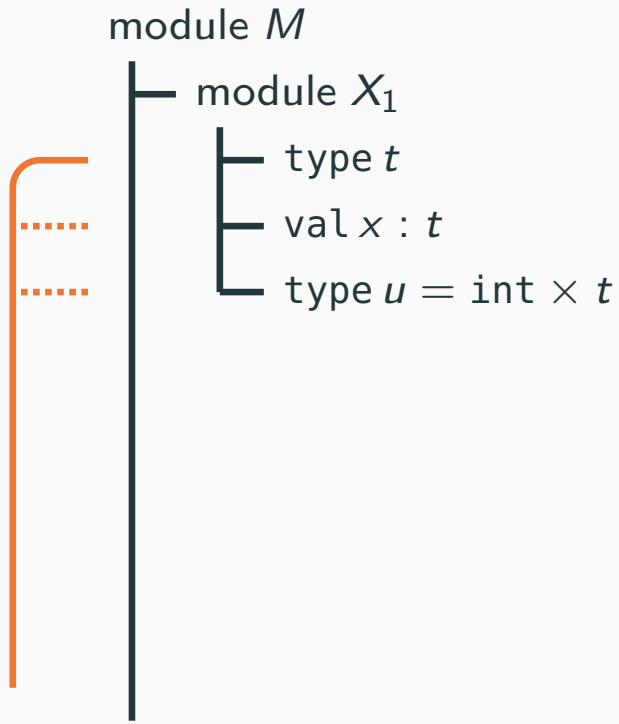
Signature avoidance

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



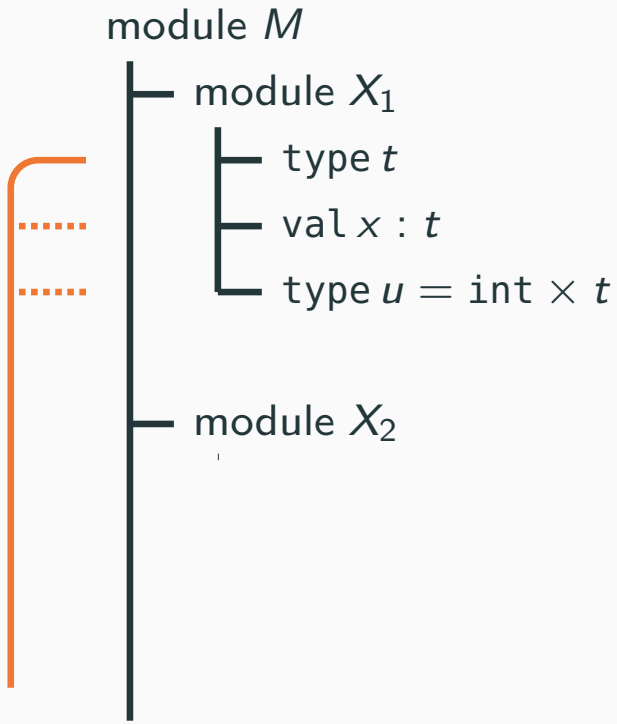
Signature avoidance

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



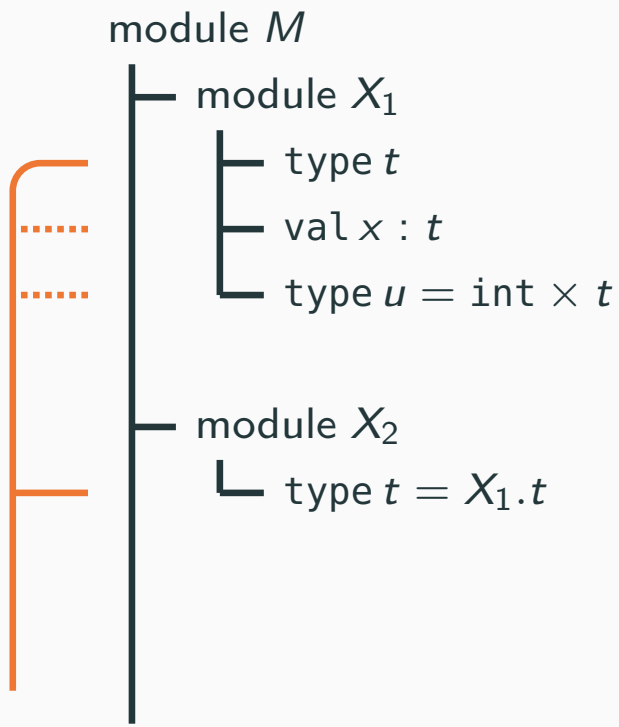
Signature avoidance

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



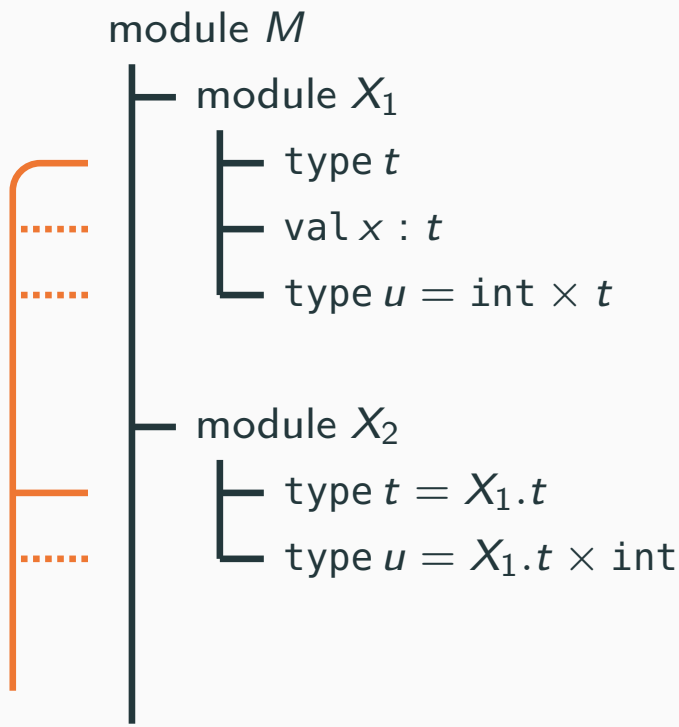
Signature avoidance

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



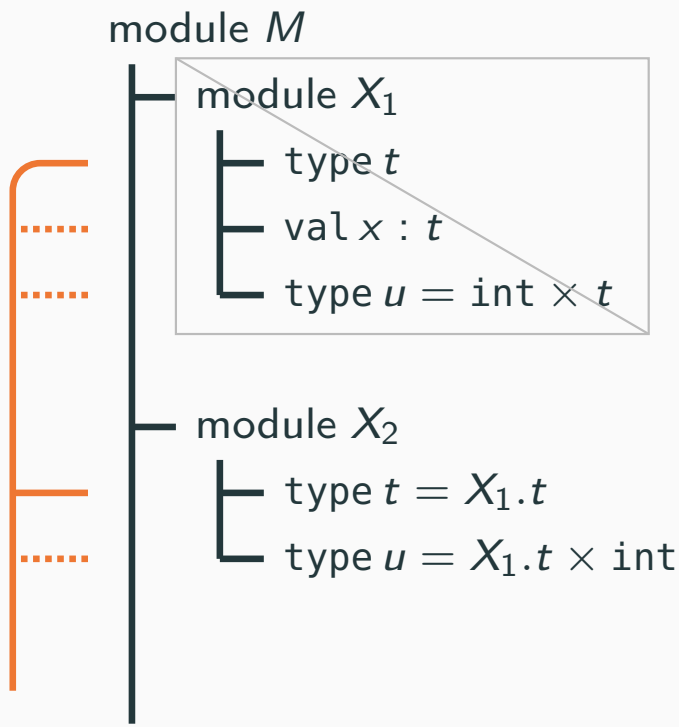
Signature avoidance

```
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
```



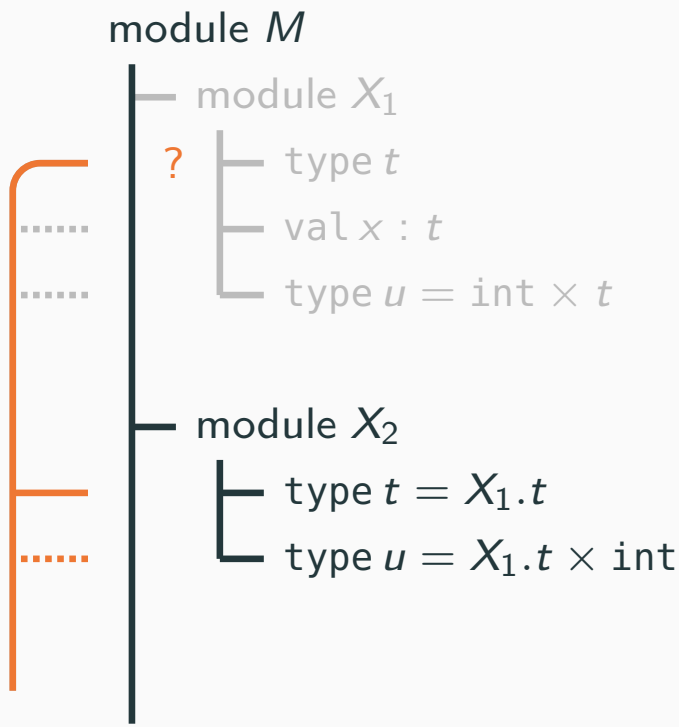
Signature avoidance

```
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
```



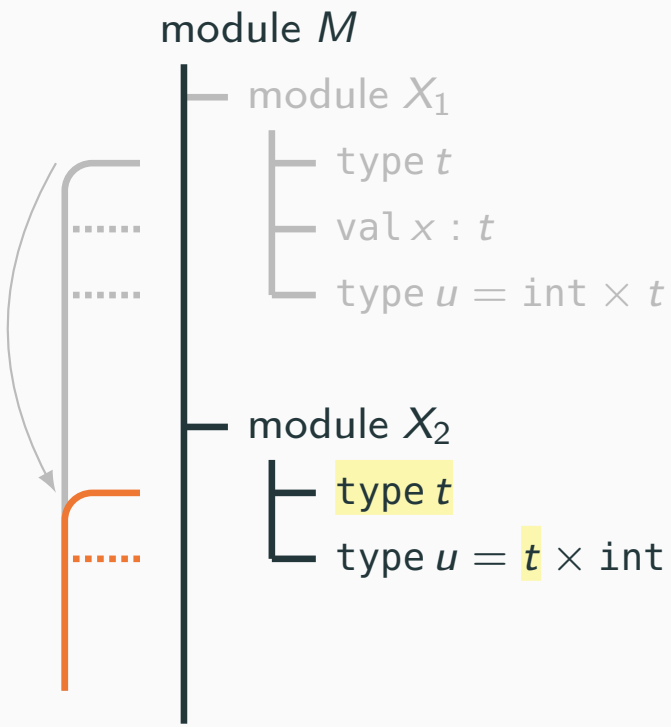
Signature avoidance

```
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
```



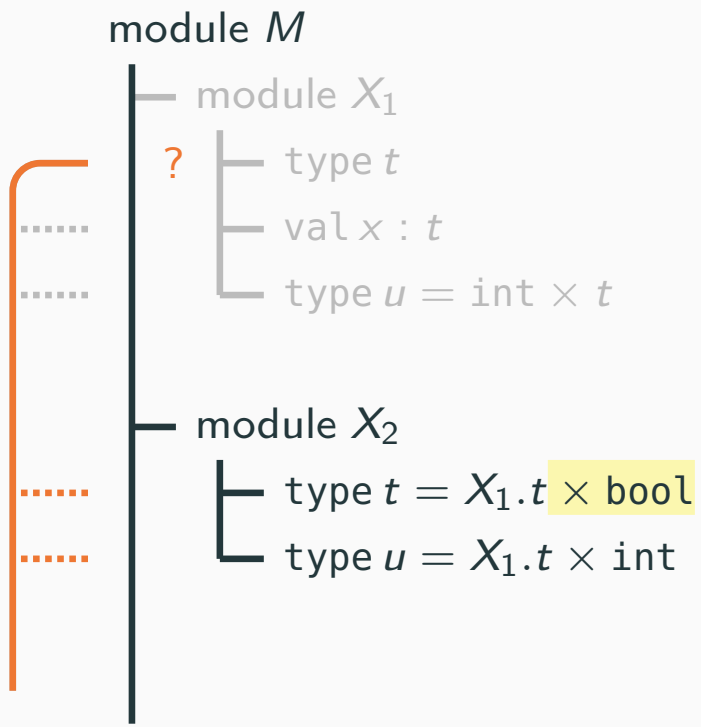
Signature avoidance

```
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
```



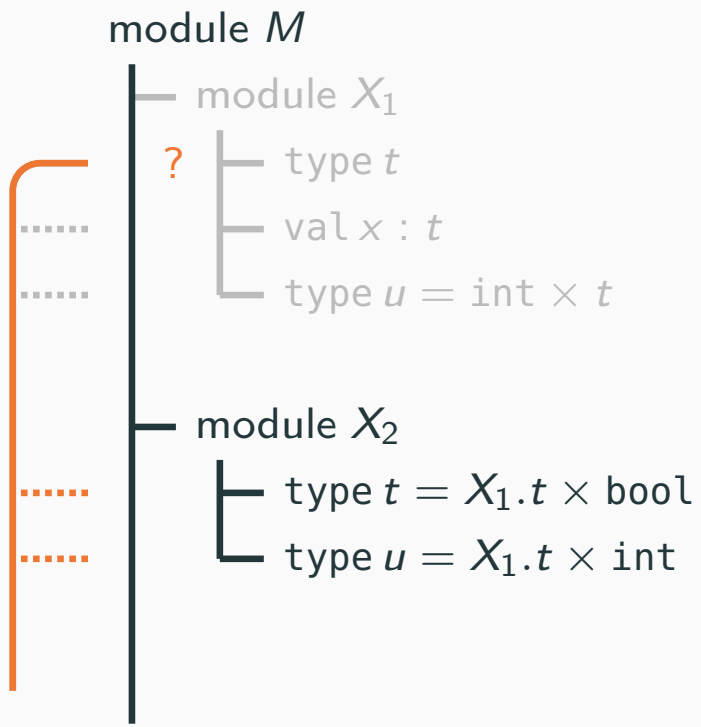
Signature avoidance

```
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
```



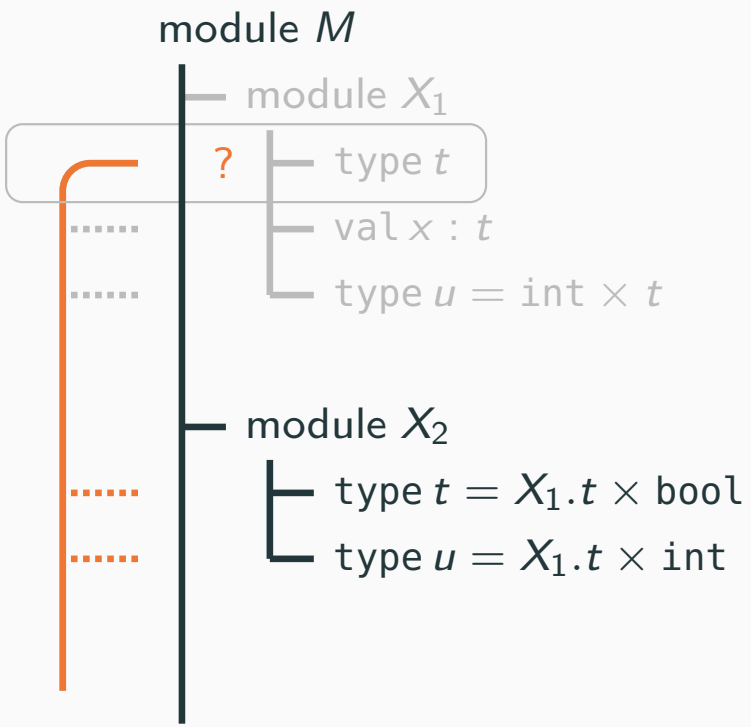
Signature avoidance

```
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
```



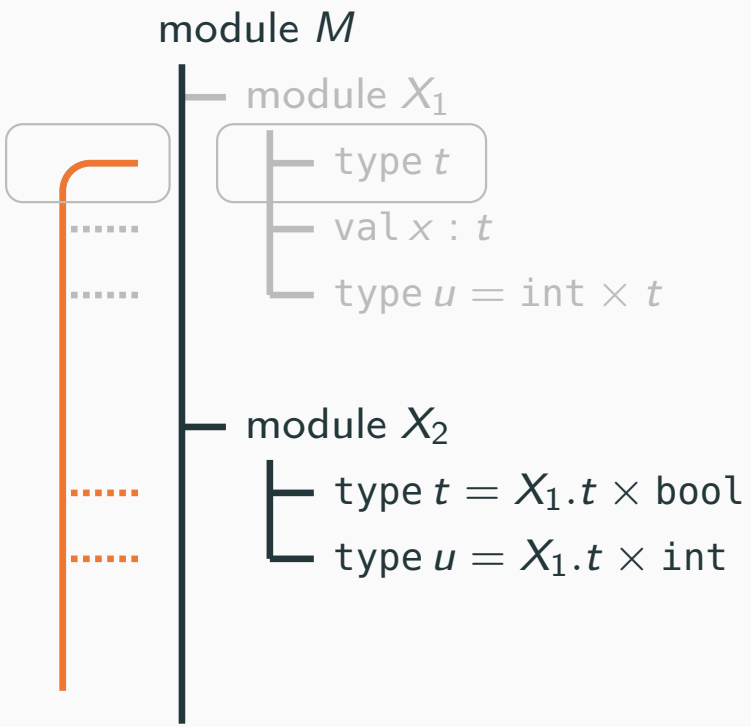
Signature avoidance

```
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
```

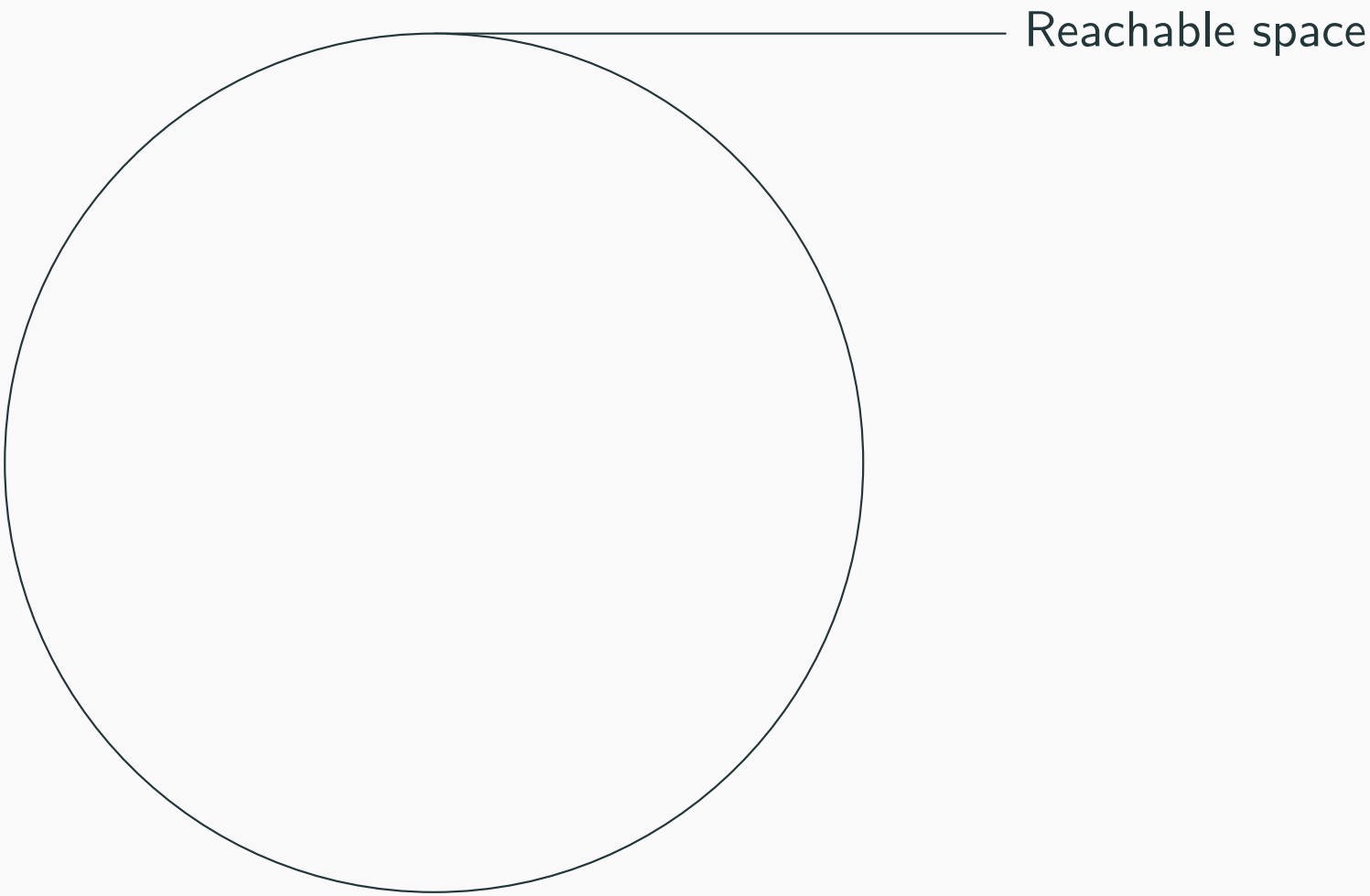


Signature avoidance

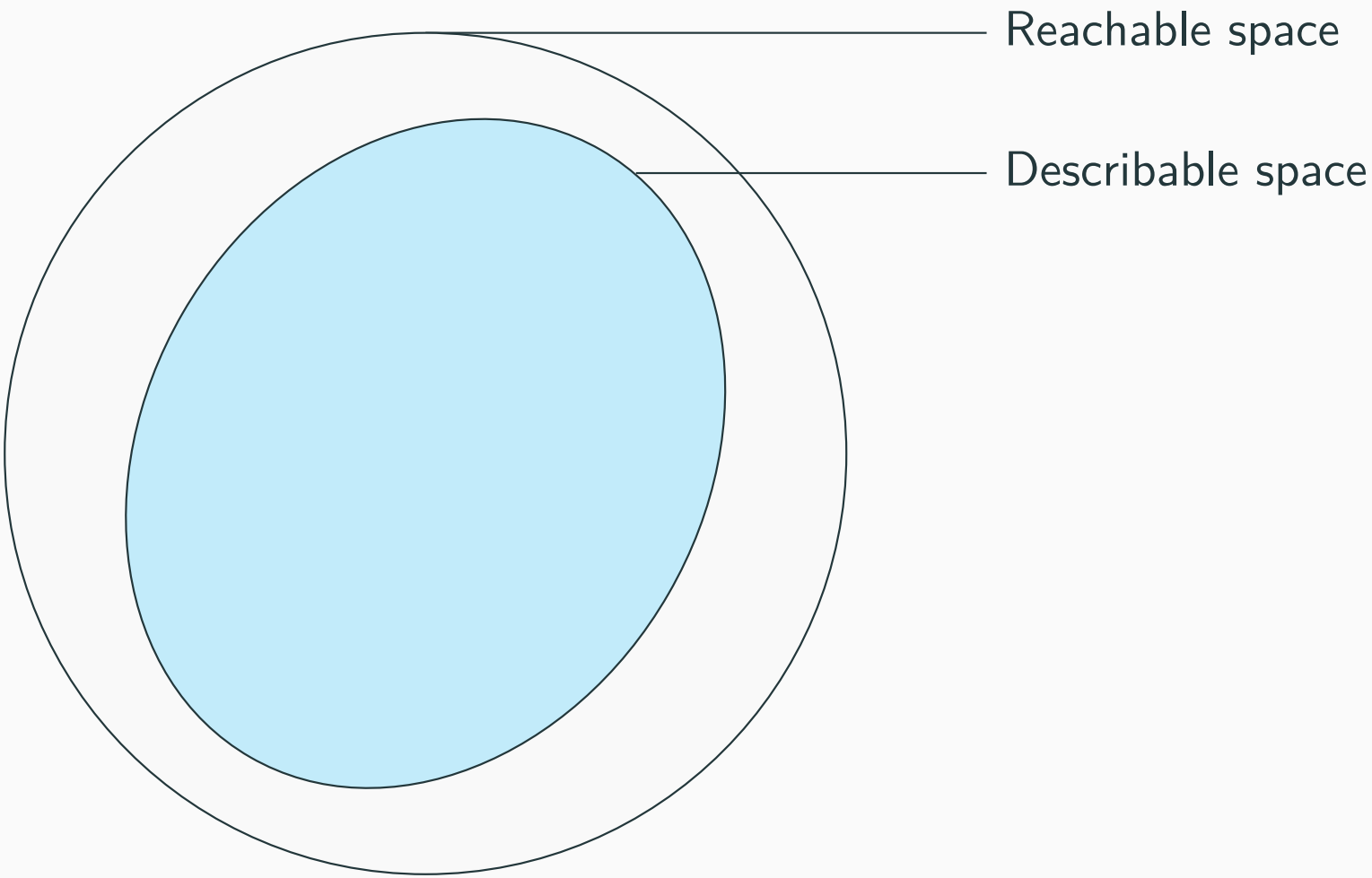
```
1  module M = (struct
2    module X1 : S = ...
3
4    module X2 = struct
5      type t = X1.t * bool
6      type u = X1.t * int
7    end
8  end).X2
```



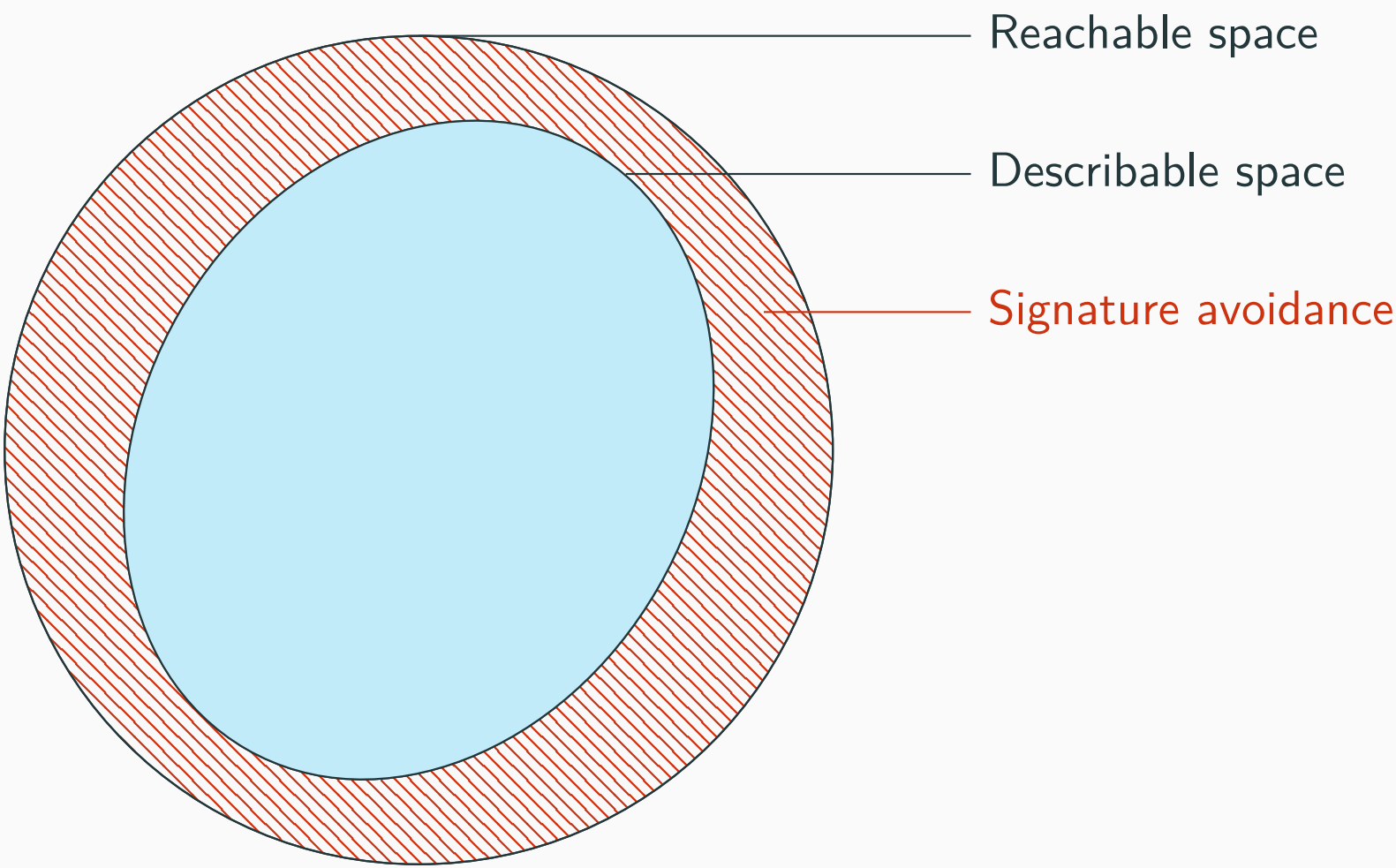
Expressivity mismatch



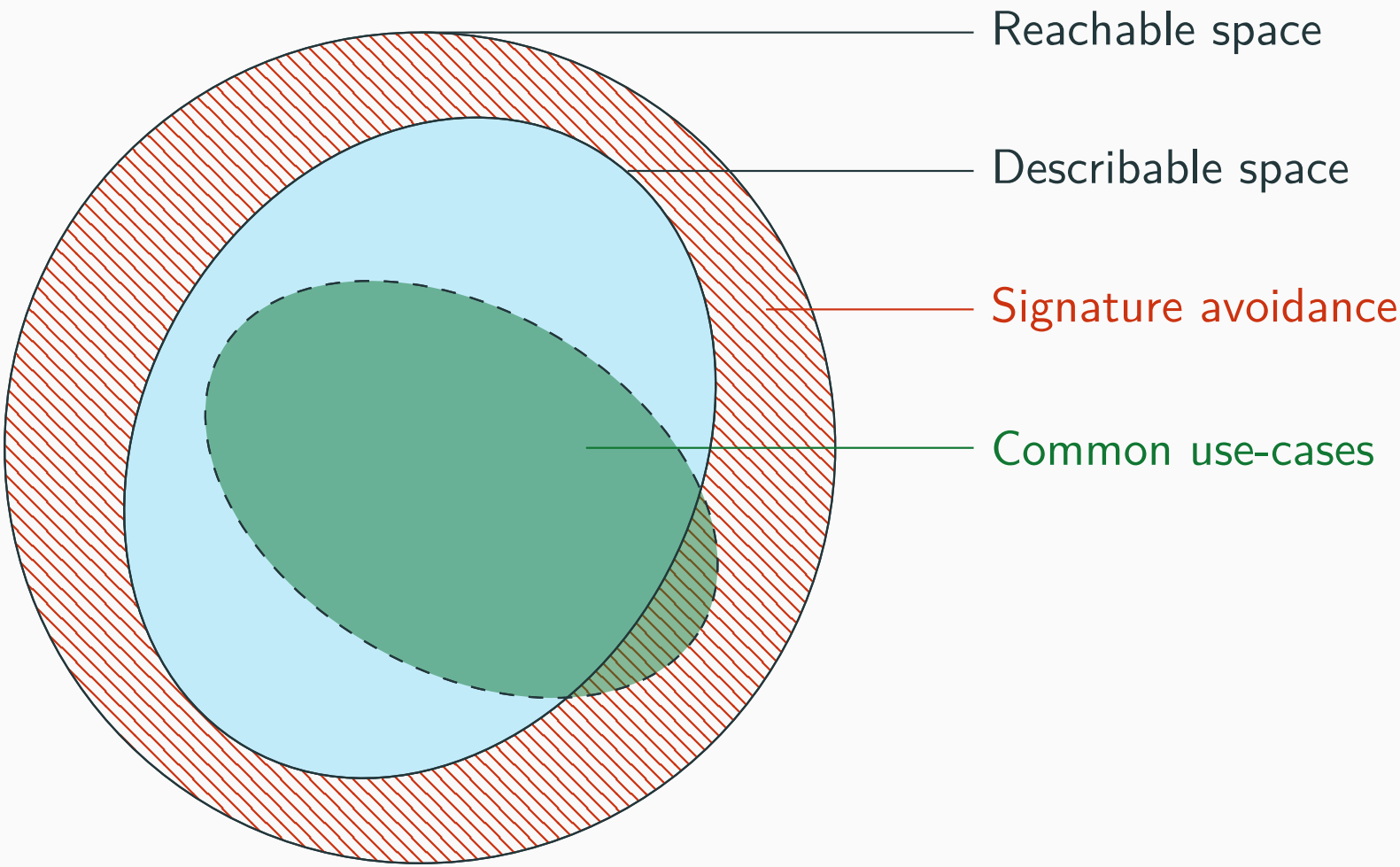
Expressivity mismatch



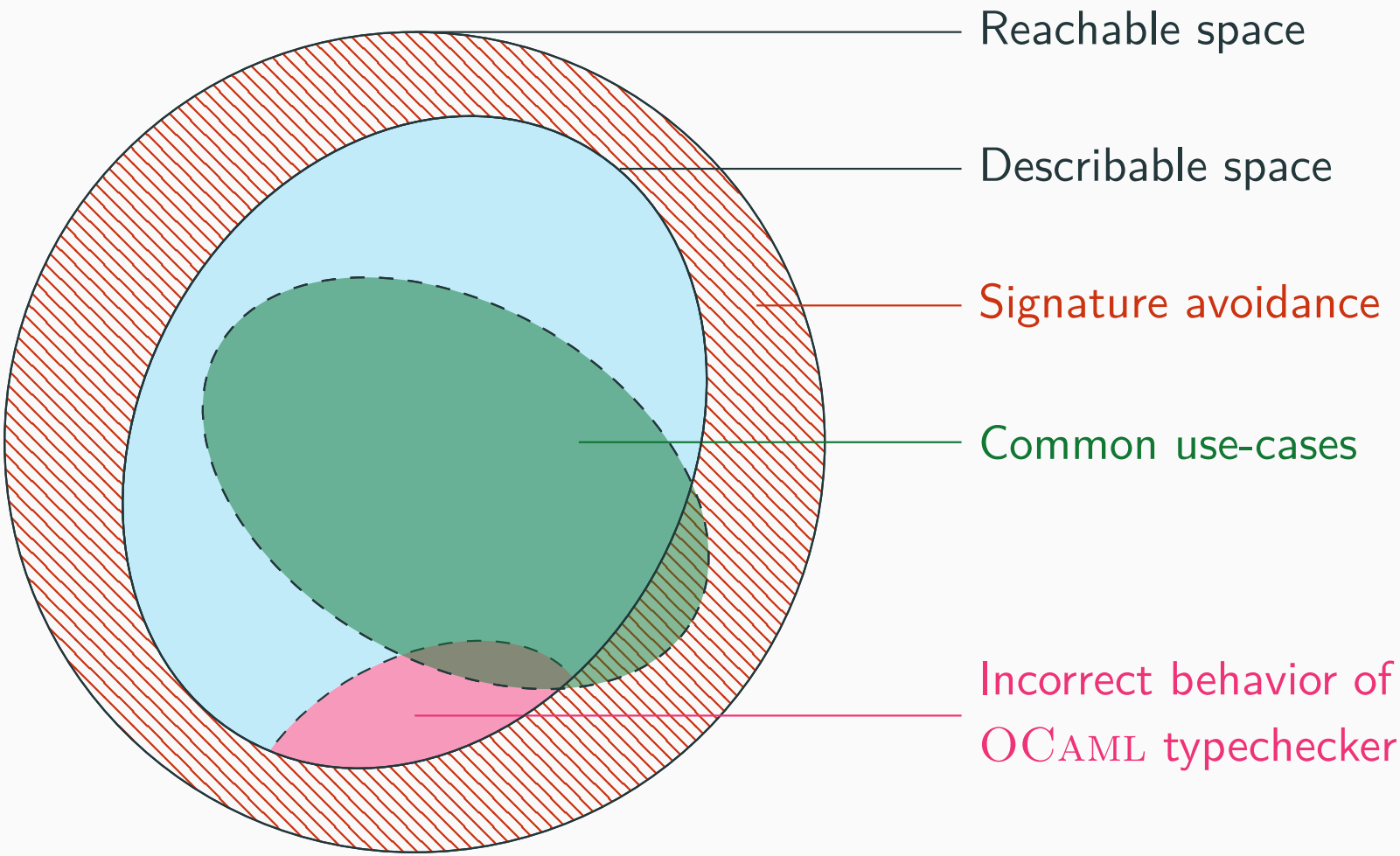
Expressivity mismatch



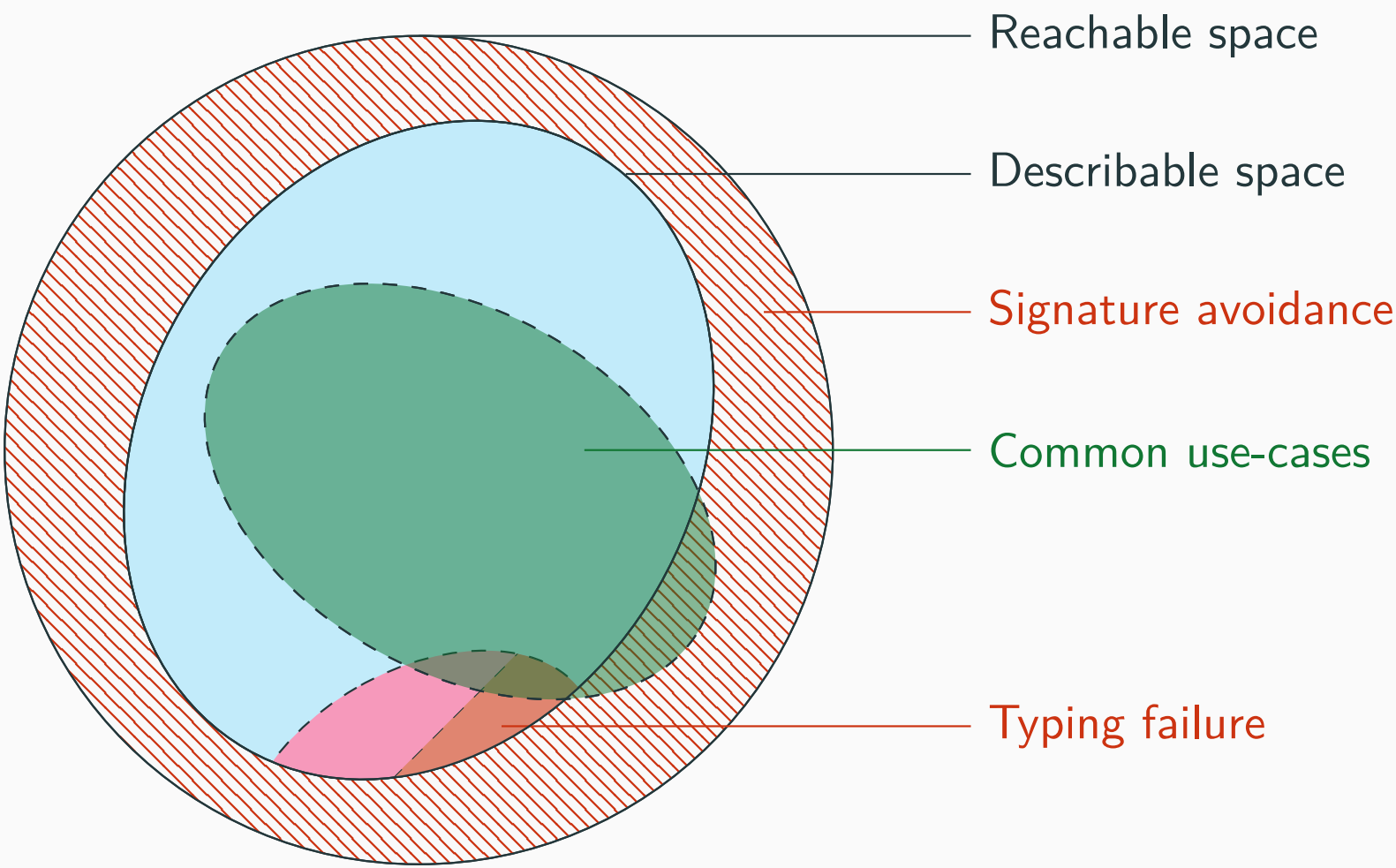
Expressivity mismatch



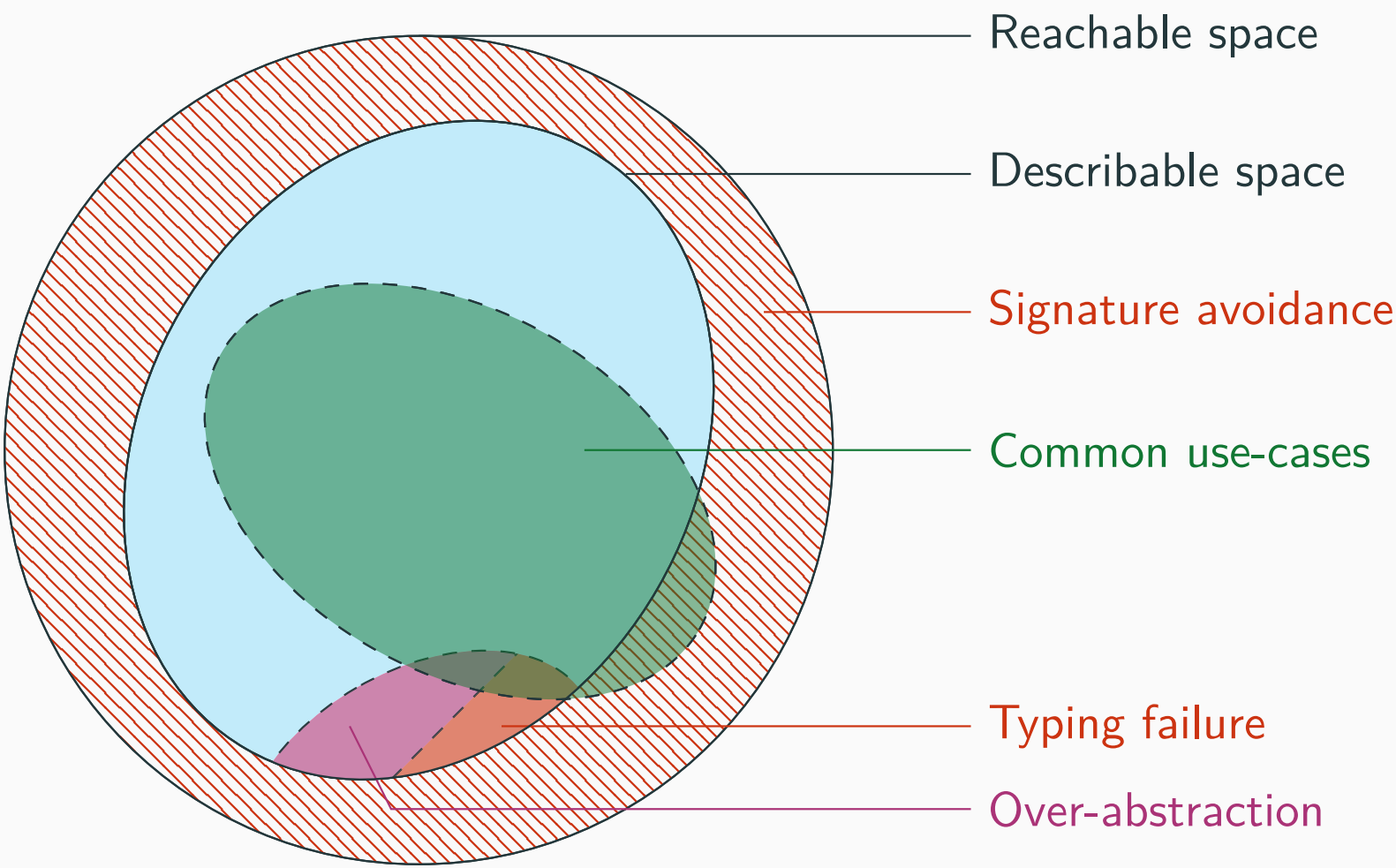
Expressivity mismatch



Expressivity mismatch



Expressivity mismatch



Expressivity mismatch