Tracking which types are principally known in OCaml

Samuel Vivien

Cambium - INRIA & PSL

January 23, 2025

What is principality ?

```
> ocaml --help
Usage: ocaml <options> <files>
Options are:
    ...
    -principal  Check principality of type inference
    -no-principal  Do not check principality of type inference (default)
    ...
```

What is principality ?

```
> ocaml --help
Usage: ocaml <options> <files>
Options are:
    ...
    -principal  Check principality of type inference
    -no-principal  Do not check principality of type inference (default)
    ...
```

A principal typing in S for a term M is a typing for M which somehow represents all other possible typings in S for M

J. B. Wells

An example of principal type

```
let id = fun x → x
```

An example of principal type

```
let id = fun x → x
```

When seing this function we could infer different types for it :

- `int → int`
- `unit → unit`

An example of principal type

```
let id = fun x → x
```

When seing this function we could infer different types for it :

- `int → int`
- `unit → unit`
- `'a → 'a`

What could be a non principal type in OCaml ?

```
let f x (y : < m : 'a. 'a → 'a >) =
    ignore (
        (x = y),
        x#m 3
    )
```

What could be a non principal type in OCaml ?

```
let f x (y : < m : 'a. 'a → 'a >) =
    ignore (
        (x = y),
        x#m 3
    )
```

Top first	Bottom first

What could be a non principal type in OCaml ?

```
let f x (y : < m : 'a. 'a → 'a >) =
    ignore (
        (x = y),
        x#m 3
    )
```

Top first	Bottom first
`x = y`	

What could be a non principal type in OCaml ?

```
let f x (y : < m : 'a. 'a → 'a >) =
    ignore (
        (x = y),
        x#m 3
    )
```

Top first	Bottom first
x = y ⇒ x : <m : 'a. 'a -> 'a>	

What could be a non principal type in OCaml ?

```
let f x (y : < m : 'a. 'a → 'a >) =
    ignore (
        (x = y),
        x#m 3
    )
```

Top first	Bottom first
x = y ⇒ x : <m : 'a. 'a -> 'a> x#m 3	

What could be a non principal type in OCaml ?

```
let f x (y : < m : 'a. 'a → 'a >) =
    ignore (
        (x = y),
        x#m 3
    )
```

Top first	Bottom first
`x = y` ⇒ `x : <m : 'a. 'a -> 'a>` `x#m 3` ⇒ is valid	

What could be a non principal type in OCaml ?

```
let f x (y : < m : 'a. 'a → 'a >) =
    ignore (
        (x = y),
        x#m 3
    )
```

Top first		Bottom first
x = y	⇒ x : <m : 'a. 'a -> 'a>	x#m 3
x#m 3	⇒ is valid	

What could be a non principal type in OCaml ?

```
let f x (y : < m : 'a. 'a → 'a >) =
    ignore (
        (x = y),
        x#m 3
    )
```

Top first	Bottom first
x = y ⇒ x : <m : 'a. 'a -> 'a> x#m 3 ⇒ is valid	x#m 3 ⇒ x : <m : int -> 'b>

What could be a non principal type in OCaml ?

```
let f x (y : < m : 'a. 'a → 'a >) =
    ignore (
        (x = y),
        x#m 3
    )
```

Top first		Bottom first	
`x = y`	⇒ `x : <m : 'a. 'a -> 'a>`	`x#m 3`	⇒ `x : <m : int -> 'b>`
`x#m 3`	⇒ is valid	`x = y`	

What could be a non principal type in OCaml ?

```
let f x (y : < m : 'a. 'a → 'a >) =
    ignore (
        (x = y),
        x#m 3
    )
```

Top first		Bottom first	
`x = y`	⇒ `x : <m : 'a. 'a -> 'a>`	`x#m 3`	⇒ `x : <m : int -> 'b>`
`x#m 3`	⇒ is valid	`x = y`	⇒ Fails

What could be a non principal type in OCaml ?

```
let f x (y : < m : 'a. 'a → 'a >) =
    ignore (
        (x = y),
        x#m 3
    )
```

Top first		Bottom first	
`x = y`	⇒ `x : <m : 'a. 'a -> 'a>`	`x#m 3`	⇒ `x : <m : int -> 'b>`
`x#m 3`	⇒ principality warning	`x = y`	⇒ Fails

The type of `x` was not principal when typing `x#m 3`.

Principality warning with constructors

```
type 'a ta = C of 'a | A
type tb = C of int | B



let id x =
    let _ = C x in x
```

Principality warning with constructors

```
type 'a ta = C of 'a | A
type tb = C of int | B


let id x =
    let _ = C x in x
```

What is the infered type of `id` ?

Principality warning with constructors

```
type 'a ta = C of 'a | A
type tb = C of int | B

(* val id :  int -> int *)
let id x =
    let _ = C x in x
```

What is the infered type of `id` ?

Principality warning with constructors

```
type 'a ta = C of 'a | A
type tb = C of int | B

(* val id :  int -> int *)
let id x =
    let _ = C x in x
```

What is the infered type of `id` ?

```
type 'a ta = C of 'a | A
type tb = C of int | B

let id x =
    let _ = [A; C x] in x
```

Principality with labels

```
let foo (f : a:int → b:int → int) : int = ...

(* val bar : (a:int → b:int → int) → int *)
let bar f =
    foo f + f ~b:1 ~a:2
```

Principality with labels

```
let foo (f : a:int → b:int → int) : int = ...

(* val bar : (a:int → b:int → int) → int *)
let bar f =
    foo f + f ~b:1 ~a:2
```

- Left to right ⇒ warning
- Right to left ⇒ error

Principality with first-class modules

```
(* val foo : ((module S) → 'a) → 'a * 'a *)
let foo bar =
    (bar (module M1 : S),
     bar (module M2))
```

Principality with first-class modules

```
(* val foo : ((module S) → 'a) → 'a * 'a *)
let foo bar =
    (bar (module M1 : S),
     bar (module M2))
```

- Left to right ⇒ warning
- Right to left ⇒ error

Types have levels

```
type int = O | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```

Types have levels

int^1

```
type int = O | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```

Introducing int

Types have levels

int^1 $\quad$ $bool^2$

```
type int = O | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```

Introducing `bool`

Types have levels

int^1 $bool^2$

$?^3$

```
type int = 0 | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```

Introducing x

Types have levels

int^1 $bool^2$ $?^4$ $?^4$

$?^3$

$\_ \rightarrow^4 \_$

```
type int = O | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```

$?^5$

Introducing y and z

Types have levels

int^1 $\quad$ $bool^2$ $\quad$ $?^4$ $\quad$ $?^4$

$?^3$ $\quad$ $\_ \rightarrow^4 \_$

```
type int = O | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```

$?^5$

Typing [x; y]

Types have levels

int^1 $bool^2$ $?^4$ $?^4$

```
type int = O | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```

$?^3 = \_ \rightarrow^4 \_$

$?^5$

Typing [x; y]

Types have levels

int^1 $bool^2$ $?^3$ $?^3$

$?^3$ $\_ \to^3 \_$

```
type int = O | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```

$?^5$

Typing [x; y]

Types have levels

int^1 $bool^2$ $?^3$ $?^3$

$\_ \rightarrow^3 \_$

$?^5$

```
type int = 0 | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```

Typing [x; y]

Types have levels

int^1 $bool^2$ $?^3$ $?^3$

$\_ \rightarrow^3 \_$

$\_ \text{ list}^5$

```
type int = O | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```

$?^5$

`[x; y]`

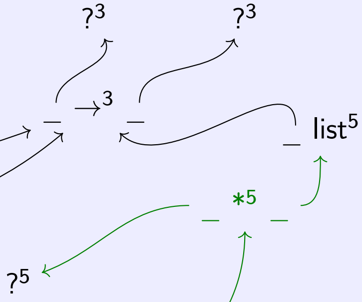
Types have levels

int^1 $bool^2$

```
type int = O | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```



(z, [x; y])

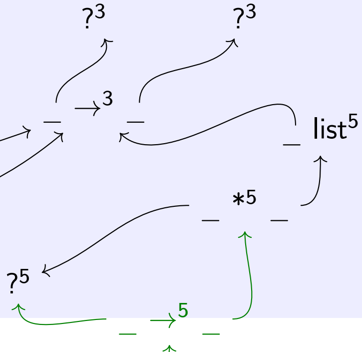
Types have levels

int^1 $bool^2$

```
type int = O | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```




```
fun z -> (z, [x; y])
```

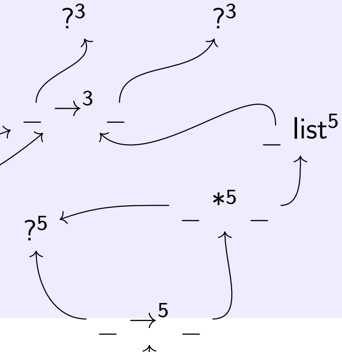
Types have levels

int^1 $bool^2$

```
type int = 0 | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```




```
fun z -> (z, [x; y])
```

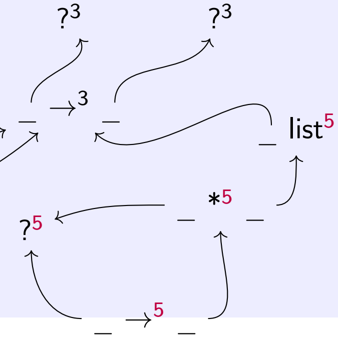
Types have levels

int^1 $bool^2$

```
type int = 0 | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```




```
fun z -> (z, [x; y])
```

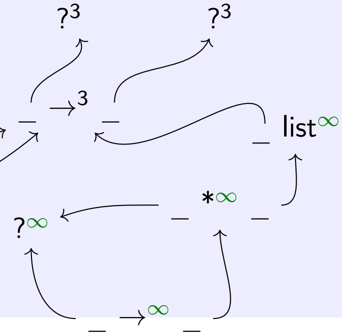
Types have levels

int^1 $bool^2$

```
type int = O | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```




```
fun z -> (z, [x; y])
```

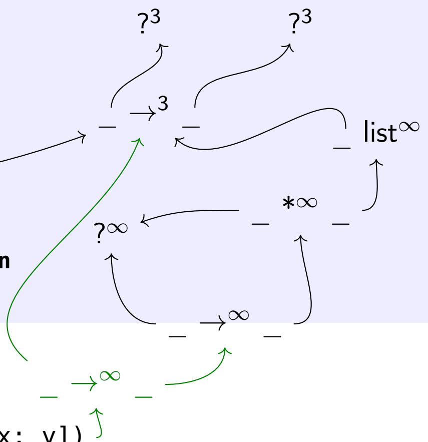
Types have levels

int^1 $bool^2$

```
type int = O | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```




```
fun y z -> (z, [x; y])
```

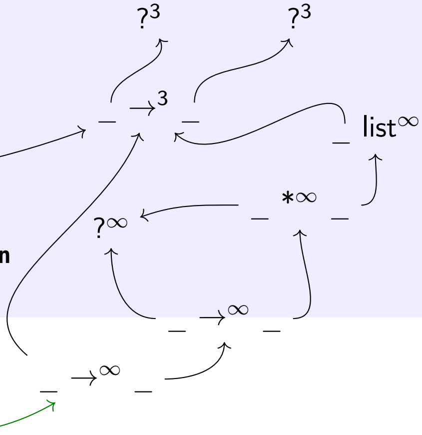
Types have levels

int^1 $bool^2$

```
type int = O | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```



Types have levels

int^1 $bool^2$ $?^3$ $?^3$

$\_ \rightarrow^3 \_$

```
type int = O | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```

$\_ \rightarrow {}'a \rightarrow {}'a * \_$ list

Types have levels

int^1 $\quad$ $bool^2$ $\quad$ $?^3$ $\quad$ $?^3$

$\_ \rightarrow^3 \_$

```
type int = O | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```

$(\_ \rightarrow \_) \rightarrow \text{'a} \rightarrow \text{'a} * \_ \text{ list}$

Rules about levels

Take away :

Rules about levels

Take away :

- Every type node has a level,

Rules about levels

Take away :

- Every type node has a level,
- Property : sub nodes always have a level older than their parents,

Rules about levels

Take away :

- Every type node has a level,
- Property : sub nodes always have a level older than their parents,
- When leaving a scope, all unification variable of that level are generalized,

Rules about levels

Take away :

- Every type node has a level,
- Property : sub nodes always have a level older than their parents,
- When leaving a scope, all unification variable of that level are generalized,

Important notice :

- Allows easy error detection/reporting :

```
let f x (type a) (y : a) = [x; y]
```

Rules about levels

Take away :

- Every type node has a level,
- Property : sub nodes always have a level older than their parents,
- When leaving a scope, all unification variable of that level are generalized,

Important notice :

- Allows easy error detection/reporting :

```
let f x (type a) (y : a) = [x; y]
```

- Also works with GADTs !

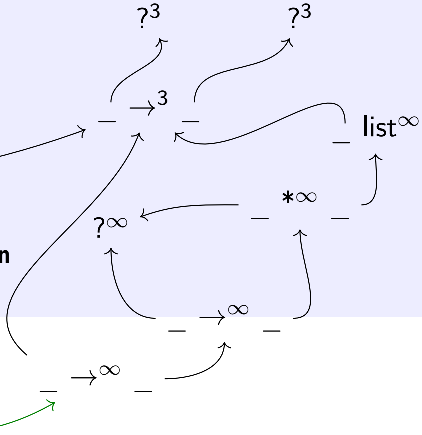
Types have levels

int^1 $bool^2$

```
type int = O | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```



Types have levels

int^1 $\quad$ $bool^2$ $\quad$ $?^3$ $\quad$ $?^3$

$\_ \rightarrow^3 \_$

```
type int = O | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```

$\_ \rightarrow 'a \rightarrow 'a * \_\ \mathsf{list}$

What could be a non principal type in OCaml ?

```
let f x (y : < m : 'a. 'a → 'a >) =
    ignore (
        (x = y),
        x#m 3
    )
```

Top first		Bottom first	
`x = y`	⇒ `x : <m : 'a. 'a -> 'a>`	`x#m 3`	⇒ `x : <m : int -> 'b>`
`x#m 3`	⇒ principality warning	`x = y`	⇒ Fails

The type of `x` was not principal when typing `x#m 3`.

What could be a non principal type in OCaml ?

```
let f x (y : < m : 'a. 'a → 'a >) =
    ignore (
        (x = y),
        x#m 3
    )
```

Top first		Bottom first	
`x = y`	⇒ `x : <m : 'a.`^{2} `'a ->`^{2} `'a>`^{2}	`x#m 3`	⇒ `x : <m : int`^{2} `->`^{2} `'b>`^{2}
`x#m 3`	⇒ principality warning	`x = y`	⇒ Fails

The type of `x` was not principal when typing `x#m 3`.

What could be a non principal type in OCaml ?

```
let f x (y : < m : 'a. 'a → 'a >) =
    ignore (
        (x = y),
        x#m 3
    )
```

Top first		Bottom first	
`x = y`	$\Rightarrow$ `x : <m : 'a.`^{2} `'a ->`^{2} `'a>`^{2}	`x#m 3`	$\Rightarrow$ `x : <m : int`^{2} `->`^{2} `'b>`^{2}
`x#m 3`	$\Rightarrow$ principality warning	`x = y`	$\Rightarrow$ Fails

The type of `x` was not principal when typing `x#m 3`, because the level of `.` is not ∞.

What about y ?

```
let f (y : <m : 'a. 'a → 'a>) =
    y#m 3
```

Does this code raise a warning ?

What about y ?

```
let f (y : <m : 'a. 'a → 'a>) =
    y#m 3
```

Does this code raise a warning ?

No, because

$$\texttt{y : <m : 'a.}^{\infty}\ \texttt{'a ->}^{\infty}\ \texttt{'a>}^{\infty}$$

Why does it work ?

Why does it work ?

In OCaml 5.3 we have only 2 ways to propagate type information :

Why does it work ?

In OCaml 5.3 we have only 2 ways to propagate type information :

f is know, x is unknown

Unification Ex : f x	

Why does it work ?

In OCaml 5.3 we have only 2 ways to propagate type information :

`f` is know, `x` is unknown

Unification Ex : `f x` Levels are propagated.	

Why does it work ?

In OCaml 5.3 we have only 2 ways to propagate type information :

`f` is know, `x` is unknown

Unification	Code infered from type
Ex : `f x`	Ex : `f (C x)`
Levels are propagated.	

Why does it work ?

In OCaml 5.3 we have only 2 ways to propagate type information :

`f` is know, `x` is unknown

Unification	Code infered from type
Ex : `f x`	Ex : `f (C x)`
Levels are propagated.	Raise a principality warning if the type was fragile.

Why does it work ?

In OCaml 5.3 we have only 2 ways to propagate type information :

`f` is know, `x` is unknown

Unification	Code infered from type
Ex : `f x`	Ex : `f (C x)`
Levels are propagated.	Raise a principality warning if the type was fragile.

Code infered from type can be :

- Labelled arguments

Why does it work ?

In OCaml 5.3 we have only 2 ways to propagate type information :

`f` is know, `x` is unknown

Unification	Code infered from type
Ex : `f x`	Ex : `f (C x)`
Levels are propagated.	Raise a principality warning if the type was fragile.

Code infered from type can be :

- Labelled arguments
- Constructor/record disambiguation

Why does it work ?

In OCaml 5.3 we have only 2 ways to propagate type information :

`f` is know, `x` is unknown

Unification	Code infered from type
Ex : `f x`	Ex : `f (C x)`
Levels are propagated.	Raise a principality warning if the type was fragile.

Code infered from type can be :

- Labelled arguments
- Constructor/record disambiguation
- First-class modules

Why does it work ?

In OCaml 5.3 we have only 2 ways to propagate type information :

`f` is know, `x` is unknown

Unification	Code infered from type
Ex : `f x`	Ex : `f (C x)`
Levels are propagated.	Raise a principality warning if the type was fragile.

Code infered from type can be :

- Labelled arguments
- Constructor/record disambiguation
- First-class modules
- Modular implicits (?)

1. Principality, definition and use in OCaml
2. Annotating types with levels
3. How to use levels for principality
4. What about modular implicits ?

What are modular implicits ?

Implicit modules arguments for functions.

What are modular implicits ?

Implicit modules arguments for functions.

```
module type Print = sig
    type t
    val print : t → unit
end
```

What are modular implicits ?

Implicit modules arguments for functions.

```
module type Print = sig
    type t
    val print : t → unit
end

let print {P : Print} (v : P.t) = P.print v
```

What are modular implicits ?

Implicit modules arguments for functions.

```
module type Print = sig
    type t
    val print : t → unit
end

let print {P : Print} (v : P.t) = P.print v

implicit module PInt = struct ... end
implicit module PString = struct ... end
implicit module PList (X : Print) = struct ... end
```

What are modular implicits ?

Implicit modules arguments for functions.

```
module type Print = sig
    type t
    val print : t → unit
end

let print {P : Print} (v : P.t) = P.print v

implicit module PInt = struct ... end
implicit module PString = struct ... end
implicit module PList (X : Print) = struct ... end

let () =
    print 3;
    print [1; 2; 3];
    print "Hello world\n"
```

How does this interact with principality ?

How does this interact with principality ?

- Code generated based on types

How does this interact with principality ?

- Code generated based on types
- Type information are never principal/robust

How does this interact with principality ?

- Code generated based on types
- Type information are never principal/robust

Current principality tracing in OCaml cannot handle such a feature.

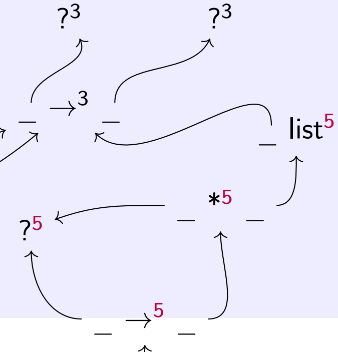
Types have levels

int^1 $bool^2$

```
type int = O | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```




```
fun z -> (z, [x; y])
```

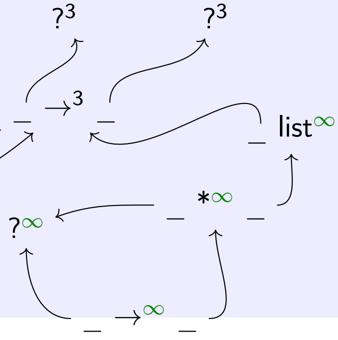
Types have levels

int^1 $bool^2$

```
type int = O | S of int

type bool = True | False

let foo x =
    let bar (y : _ → _) z = (z, [x; y]) in
    bar x
```




```
fun z -> (z, [x; y])
```

What if we didn't want types to become principal

```
module type Default = sig type t val d : t end

let default {D : Default} () = D.d
```

What if we didn't want types to become principal

```
module type Default = sig type t val d : t end

let default {D : Default} () = D.d

implicit module M = struct
    type t = a:int → b:int → int
    let d = ...
end
```

What if we didn't want types to become principal

```
module type Default = sig type t val d : t end

let default {D : Default} () = D.d

implicit module M = struct
    type t = a:int → b:int → int
    let d = ...
end

(* val f : a:int → b:int → int *)
let f = default ()
```

What if we didn't want types to become principal

```
module type Default = sig type t val d : t end

let default {D : Default} () = D.d

implicit module M = struct
    type t = a:int → b:int → int
    let d = ...
end

(* val f : a:int → b:int → int *)
let f = default ()

(* val _ : int *)
let _ = f ~b:2 ~a:1
```

A fix ?

Proposal : add a boolean saying whether this type is or can become principal.

A fix ?

Proposal : add a boolean saying whether this type is or can become principal.

- True $\Rightarrow$ this type was infered in a satisfying way, thus it can be relied on.

A fix ?

Proposal : add a boolean saying whether this type is or can become principal.

- True $\Rightarrow$ this type was infered in a satisfying way, thus it can be relied on.
- False $\Rightarrow$ this type is too fragile to be relied on.

A fix ?

Proposal : add a boolean saying whether this type is or can become principal.

- True ⇒ this type was infered in a satisfying way, thus it can be relied on.
- False ⇒ this type is too fragile to be relied on.

Already exists with labels.

```
(* val id : (a:int → b:int → 'a) → (a:int → b:int → 'b) *)
let id f =
    let _ f ~a:1 ~b:2 in f
```

A fix ?

Proposal : add a boolean saying whether this type is or can become principal.

- True ⇒ this type was infered in a satisfying way, thus it can be relied on.
- False ⇒ this type is too fragile to be relied on.

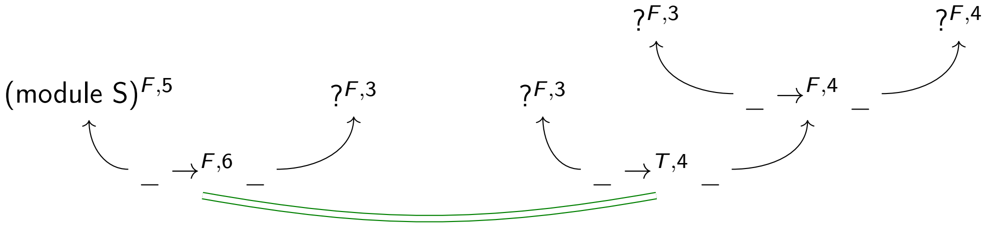
Already exists with labels.

```
(* val id : (a:int → b:int → 'a) → (a:int → b:int → 'b) *)
let id f =
    let _ f ~a:1 ~b:2 in f
```

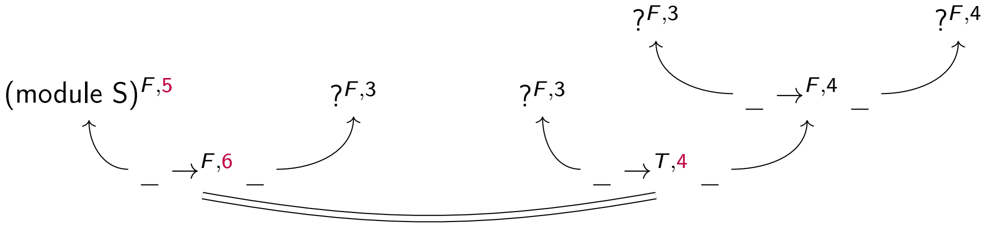
```
let fail f = id f ~b:1 ~a:1
```

```
              ^^
Error: This function is applied to arguments
       in an order different from other calls.
       This is only allowed when the real type is known.
```

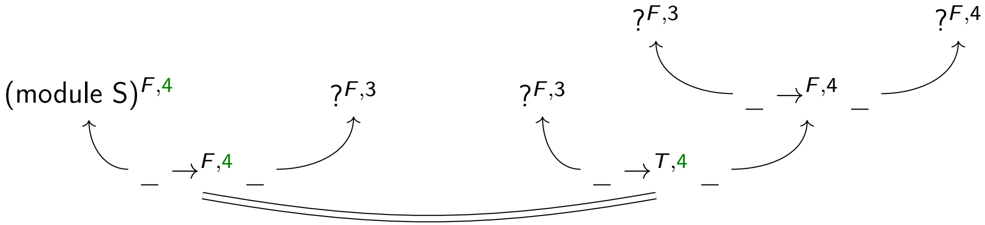
Unification with a boolean



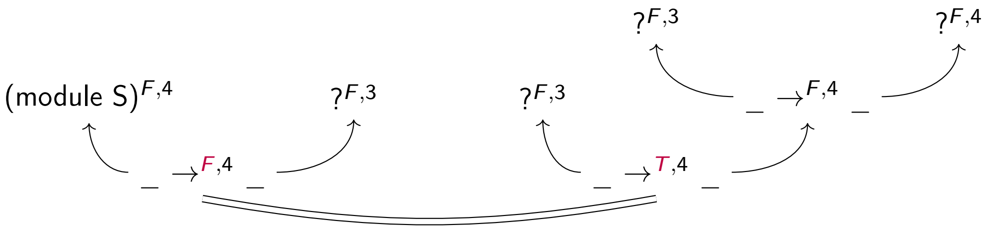
Unification with a boolean



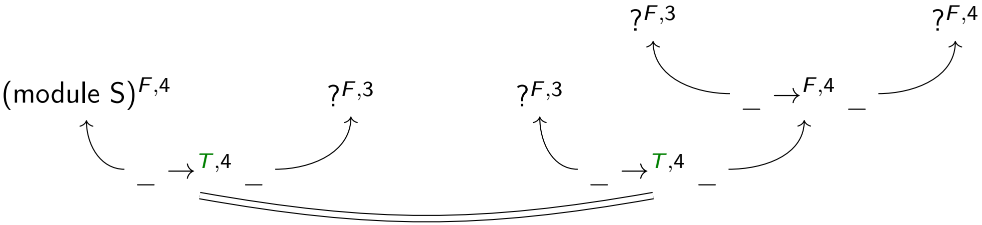
Unification with a boolean



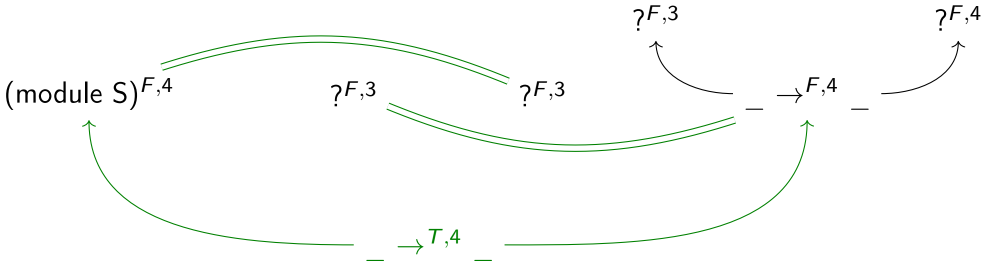
Unification with a boolean



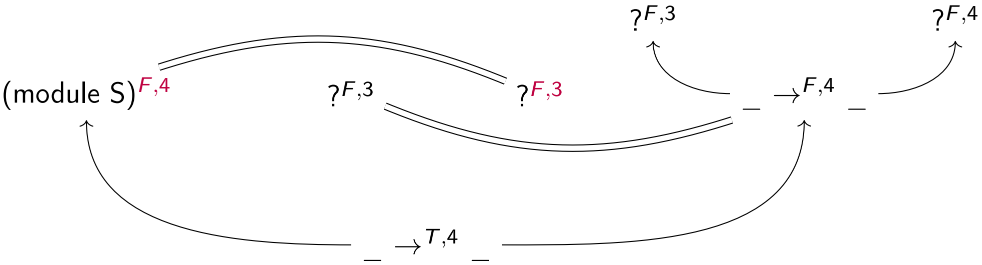
Unification with a boolean



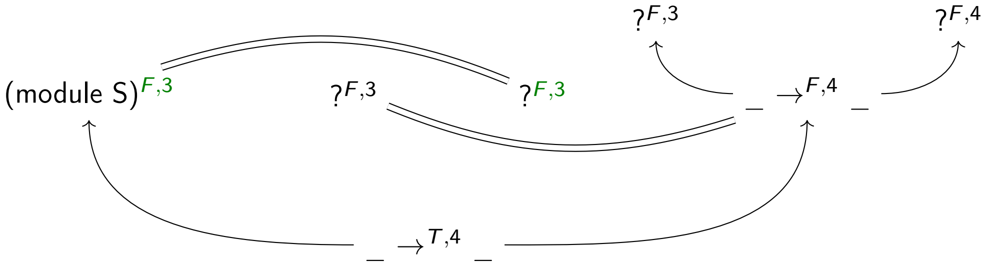
Unification with a boolean



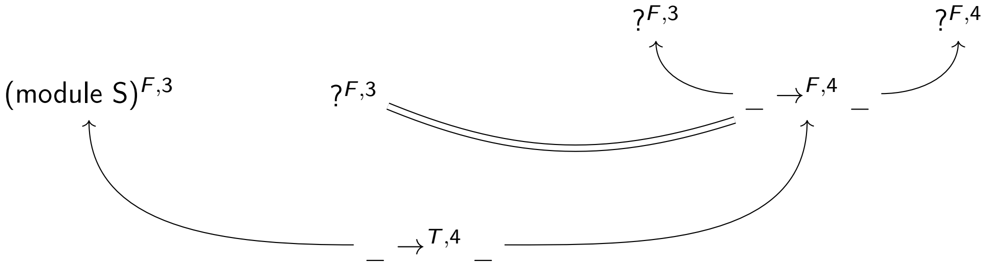
Unification with a boolean



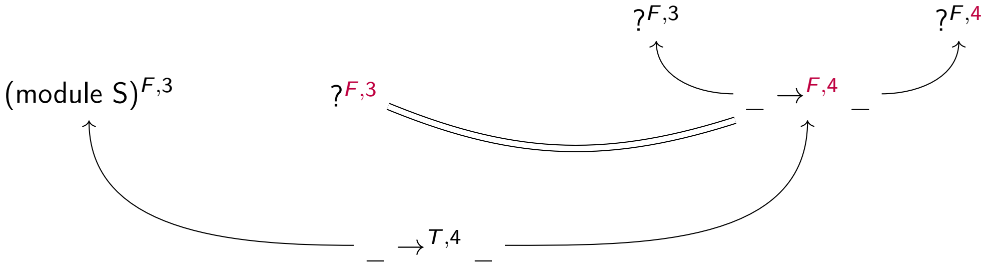
Unification with a boolean



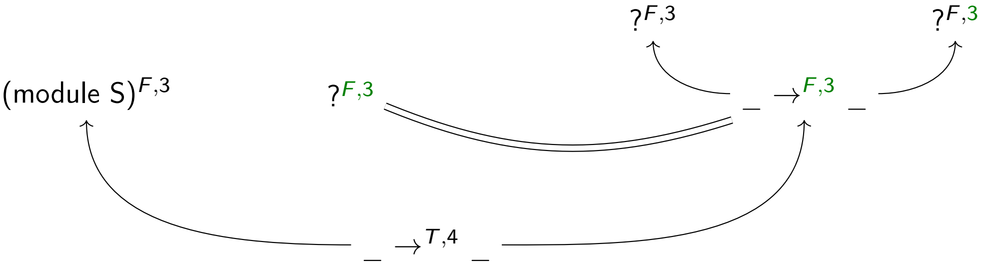
Unification with a boolean



Unification with a boolean



Unification with a boolean



Unification with a boolean

$$
\begin{tikzcd}
& & & ?^{F,3} & & ?^{F,3} \\
(\text{module S})^{F,3} & & & & \_ \to^{F,3} \_ \arrow[ul, bend left] \arrow[ur, bend right] & \\
& & \_ \to^{T,4} \_ \arrow[ull, bend left] \arrow[urr, bend right] & & &
\end{tikzcd}
$$

Questions ?

Do you have any questions ?