# Modular Implicits

Thomas Refis

Inria & Tarides

15 octobre 2021

Modular Implicits : une brève introduction

Modular Explicits

Interlude : du typage avec niveaux

Et maintenant ?

# Du polymorphisme ad-hoc

« *Modular implicits* », 2014, Bour, White & Yallop

Actuellement en OCaml :

```
val (+) : int -> int -> int
```

```
42 + 1337;;
3.14 + 2.71;; (* error *)
```

En Haskell :

```
(+) :: Num a => a -> a -> a
```

# Du polymorphisme ad-hoc

« *Modular implicits* », 2014, Bour, White & Yallop

Actuellement en OCaml :

```
val (+) : int -> int -> int
```

```
42 + 1337;;
3.14 + 2.71;; (* error *)
```

En Haskell :

```
(+) :: Num a => a -> a -> a
```

Avec modular implicits :

```
module type Num = sig
  type t
  val add : t -> t -> t
  ...
end

val (+) : {N : Num} -> N.t -> N.t -> N.t
```

# Du polymorphisme ad-hoc *modulaire*

Fonctionnement de la recherche :

- Haskell : environnement global dans lequel se trouvent toutes les instances
- OCaml : instances définies dans des modules, les règles usuelles concernant les portées s'appliquent (*open*, ...)

# Du polymorphisme ad-hoc *modulaire*

Fonctionnement de la recherche :

- Haskell : environnement global dans lequel se trouvent toutes les instances
- OCaml : instances définies dans des modules, les règles usuelles concernant les portées s'appliquent (*open*, ...)

Implications :

- Haskell : pour un type donnée, une seule instance *canonique* d'une classe donnée
- OCaml : plusieurs modules peuvent implémenter une même interface $\Longrightarrow$ ambiguités potentielles.

# Du polymorphisme ad-hoc *modulaire*

Fonctionnement de la recherche :

- Haskell : environnement global dans lequel se trouvent toutes les instances
- OCaml : instances définies dans des modules, les règles usuelles concernant les portées s'appliquent (*open*, ...)

Implications :

- Haskell : pour un type donnée, une seule instance *canonique* d'une classe donnée
- OCaml : plusieurs modules peuvent implémenter une même interface $\Longrightarrow$ ambiguités potentielles.
  - nécessaire de bien tracer les égalités de modules (ascriptions transparents, ...)
  - désambiguation explicite

# Ambiguités et applications explicites

```
implicit module INum = Int
implicit module Int_mod_4 = struct
  type t = int
  let add x y = (Int.add x y) mod 4
  ...
end
```

```
# 42 + 1337;;
     ^

Error: Ambiguous implicit N: Int_mod_4 and
       INum are both solutions.
```

# Ambiguités et applications explicites

```
implicit module INum = Int
implicit module Int_mod_4 = struct
  type t = int
  let add x y = (Int.add x y) mod 4
  ...
end
```

Applications explicites :

```
(+) {INum} 42 1337;;
- : INum.t = 1379

(+) {Int_mod_4} 42 1337;;
- : Int_mod_4.t = 3
```

Modular Implicits : une brève introduction

Modular Explicits

Interlude : du typage avec niveaux

Et maintenant ?

# Modular Explicits

- modular explicits (MX) =
  extension d'OCaml avec des fonctions qui dépendent de modules

# Modular Explicits

- modular explicits (MX) =
  extension d'OCaml avec des fonctions qui dépendent de modules
- modular implicits =
  modular explicits + implicit argument resolution

# Modular Explicits

- modular explicits (MX) =
  extension d'OCaml avec des fonctions qui dépendent de modules
- modular implicits =
  modular explicits + implicit argument resolution

Une bonne première étape :

- sur le chemin critique pour modular implicits
- une construction directement utile

# MX : un gain en expressivité par rapport à OCaml

- Polymorphisme de rang N :

```
module type Type = sig type t end

let dp {A:Type} {B:Type} (f : {C:Type} -> C.t -> C.t) (x, y) =
  f {A} x, f {B} y
```

Comparé à la version de base :

```
val dp : ('a -> 'b) -> 'a * 'a -> 'b * 'b
```

- Polymorphisme higher-kinded

```
module type Monad = sig
  type 'a t
  val return : 'a -> 'a t
  val bind : ('a -> 'b M.t) -> 'a M.t -> 'b M.t
end

val map : {M:Monad} -> ('a -> 'b) -> 'a M.t -> 'b M.t
```

# MX : considérations sémantiques

Une *module-dependent function*, et son application :

```
let f = fun {X : S} -> (e : t)
let _ = f {M}
```

s'élaborent vers :

```
let f =
  let module F (X : S) = struct let result = e end in
  (module F : functor (X : S) -> sig val result : t end)

let _ =
  let module F = (val f) in
  let module Res = F(M) in
  Res.result
```

# MX : considérations sémantiques

Une *module-dependent function*, et son application :

```
let f = fun {X : S} -> (e : t)
let _ = f {M}
```

s'élaborent vers :

```
let f =
  let module F (X : S) = struct let result = e end in
  (module F : functor (X : S) -> sig val result : t end )
let _ =
  let module F = (val f) in
  let module Res = F(M) in
  Res.result
```

# MX : considérations sémantiques

Problème: seulement des chemins sont autorisés comme *types de paquets*.

Solution: prédéfinir le type du module ?

```
module type Res = functor (M : S) -> sig
  val result : t
end

let module F (M : S) = struct let result = e end in
(module F : Res)
```

# MX : considérations sémantiques

Problème: seulement des chemins sont autorisés comme *types de paquets*.

Solution: prédéfinir le type du module ?

```
module type Res = functor (M : S) -> sig
  val result : t
end

let module F (M : S) = struct let result = e end in
(module F : Res)
```

Problème: la signature peut dépendre de types locaux !

Par exemple :

```
let return x {M:Monad} = M.return x
val return : 'a -> {M:Monad} -> 'a M.t
```

# MX : considérations sémantiques

Problème: seulement des chemins sont autorisés comme *types de paquets*.

Solution: prédéfinir le type du module ?

```
module type Res = functor (M : S) -> sig
  val result : t
end

let module F (M : S) = struct let result = e end in
(module F : Res)
```

Problème: la signature peut dépendre de types locaux !

Par exemple :

```
let return x {M:Monad} = M.return x
val return : 'a -> {M:Monad} -> 'a M.t
```

Solution: pourquoi ne pas se donner des signatures paramétriques ?

Inspiration : ≪ *Principal Type Schemes for Modular Programs* ≫, Dreyer & Blume

# MX : considérations sémantiques

Supposons qu'on puisse écrire :

```
module type [a] Res = functor (M : Monad) -> sig
  val result : a M.t
end
```

L'exemple précédent s'élabore vers :

```
fun (x : 'a) ->
  let module F (M : Monad) = struct
    let result = M.return x
  end in
  (module F : ['a] Res)
```

# MX : considérations sémantiques

Supposons qu'on puisse écrire :

```
module type [a] Res = functor (M : Monad) -> sig
  val result : a M.t
end
```

L'exemple précédent s'élabore vers :

```
fun (x : 'a) ->
  let module F (M : Monad) = struct
    let result = M.return x
  end in
  (module F : ['a] Res)
```

Ça existe déjà (ou presque) : on peut faire du calcul sur les signatures avec `Mty with type t := type-expr`

# MX : considérations sémantiques

Supposons qu'on puisse écrire :

```
module type [a] Res = functor (M : Monad) -> sig
  val result : a M.t
end
```

L'exemple précédent s'élabore vers :

```
fun (x : 'a) ->
  let module F (M : Monad) = struct
    let result = M.return x
  end in
  (module F : ['a] Res)
```

Ça existe déjà (ou presque) : on peut faire du calcul sur les signatures avec `Mty with type t := type-expr`

Problème: ce n'est pas autorisé pour les *types de paquets*.

# MX : considérations sémantiques

Pourquoi cette restriction ?

# MX : considérations sémantiques

Pourquoi cette restriction ?

- D'autres variants de ML autorisent des signatures arbitraires (e.g. MoscowML).
- OCaml utilise les chemins seulement pour son *fast path* mais compare les signatures si les chemins diffèrent.

# MX : considérations sémantiques

En résumé :

- *module-dependent functions* : pas juste du sucre.
- l'élaboration : intuition rassurante
- restrictions syntaxiques, pour des raisons historiques.

# MX : considérations pratiques

## Overlap avec les modules de première classe

```
module type S = sig type t = int val mk_t : unit -> t end

let fcm (module X : S) = X.mk_t ()
```

# MX : considérations pratiques

## Overlap avec les modules de première classe

```
module type S = sig type t = int val mk_t : unit -> t end

let fcm (module X : S) = X.mk_t ()
let fmx {X : S} = X.mk_t ()
```

# MX : considérations pratiques

## Overlap avec les modules de première classe

```
module type S = sig type t = int val mk_t : unit -> t end

let fcm (module X : S) = X.mk_t ()
let fmx {X : S} = X.mk_t ()


module M = struct
  type t = int
  let mk_t () = 0
end

let () = fcm (module M)
let () = fmx {M}
```

# MX : considérations pratiques

## Overlap avec les modules de première classe

```
module type S = sig type t = int val mk_t : unit -> t end

let fcm (module X : S) = X.mk_t ()
let fmx {X : S} = X.mk_t ()


module M = struct
  type t = int
  let mk_t () = 0
end

let () = fcm (module M)
let () = fmx {M}


let () = List.map fcm [(module M:S); (module M:S)]
```

# MX : considérations pratiques

## Overlap avec les modules de première classe

```
module type S = sig type t (* = int *) val mk_t : unit -> t end

let fcm (module X : S) = X.mk_t ()
let fmx {X : S} = X.mk_t ()


module M = struct
  type t = int
  let mk_t () = 0
end

(* let () = fcm (module M) *)
let () = fmx {M}


(* let () = List.map fcm [(module M:S); (module M:S)] *)
```

# MX : considérations pratiques

Constats :

- constructions interchangables sur la partie où elles overlappent (coercions possible dans les deux sens)
- on peut utiliser la même syntaxe pour les deux constructions

Différents choix possibles pour la suite, pas de meilleur choix évident.

# Interlude : les niveaux pour ML

Un "détail" d'implémentation pour avoir de la généralisation efficace, enforcer les portées, etc.

# Interlude : les niveaux pour ML

Un "détail" d'implémentation pour avoir de la généralisation efficace, enforcer les portées, etc.

Comment ça marche ?

- Les insertions dans l'environnement de typage sont ordonnées
- On peut associer à chaque type sa position ($=$ niveau) dans l'environnement
- Un type ne peut référencer que des types plus anciens que lui

# Interlude : les niveaux pour ML

Généralisation : quantification de toutes les variables qui ne sont pas partagées avec l'environnement

$\Longrightarrow$ on traverse le type et l'environnement

# Interlude : les niveaux pour ML

Généralisation : quantification de toutes les variables qui ne sont pas partagées avec l'environnement
$\Longrightarrow$ on traverse le type et l'environnement

Si on annote les variables avec leur niveau, alors :
généralisation = quantification de toutes les variables dont le niveau est supérieur au niveau courant
$\Longrightarrow$ on traverse seulement le type

# Interlude : les niveaux pour ML

Généralisation : quantification de toutes les variables qui ne sont pas partagées avec l'environnement
$\Longrightarrow$ on traverse le type et l'environnement

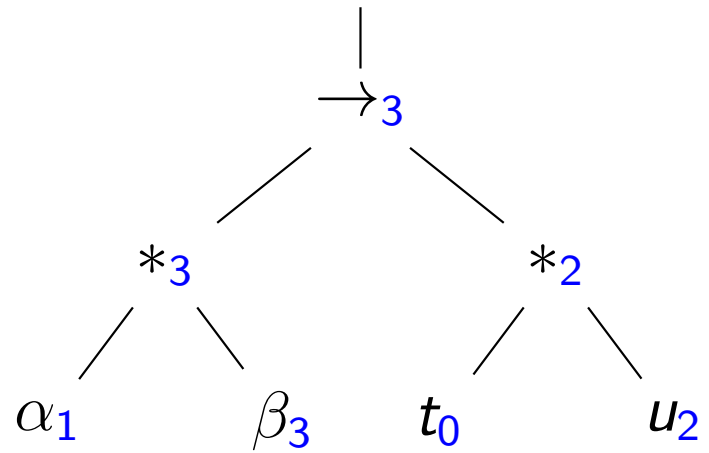
Si on annote les variables avec leur niveau, alors :
généralisation = quantification de toutes les variables dont le niveau est supérieur au niveau courant
$\Longrightarrow$ on traverse seulement le type

Optimisation : on peut annoter tous les nœuds du type avec leur niveau
$\Longrightarrow$ on peut arrêter la traversée plus tôt

# Interlude : les niveaux pour ML

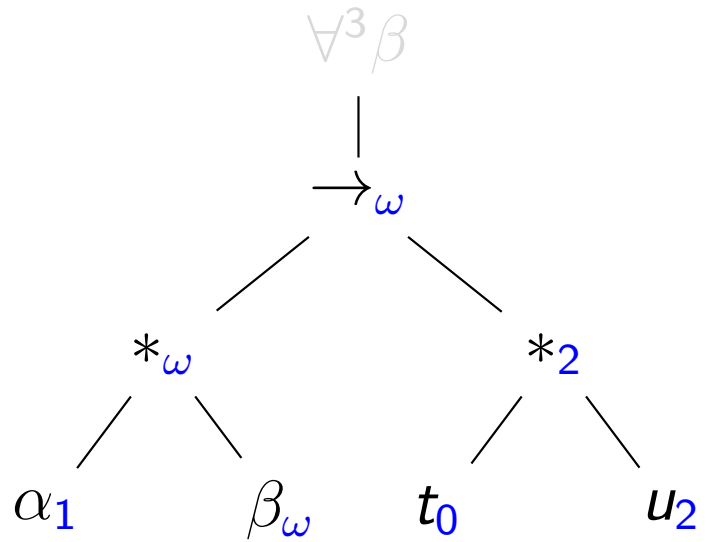
Étant donné un environnement $\Gamma = t, \alpha, u$ (de niveaux 0, 1 et 2).

Exemple de type :



# Interlude : les niveaux pour ML

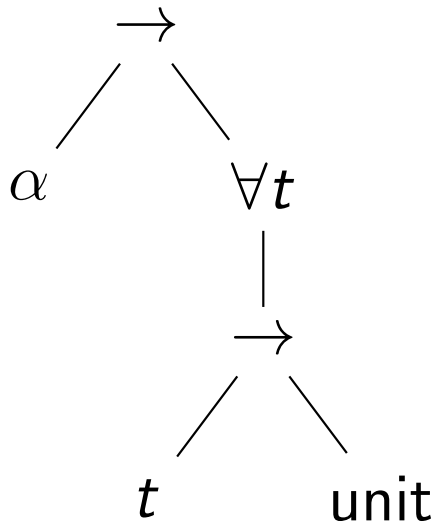
Étant donné un environnement $\Gamma = t, \alpha, u$ (de niveaux 0, 1 et 2).
Niveau d'une variable quantifiée : $\omega$
Exemple de schéma de type :



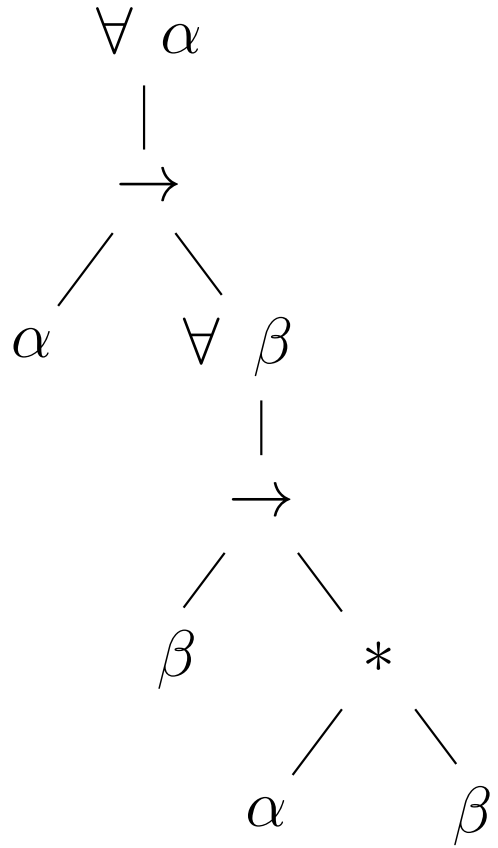
# Interlude : et modular (ex/im)plicites ?

Empêcher que les types s'échappent de leur portée :

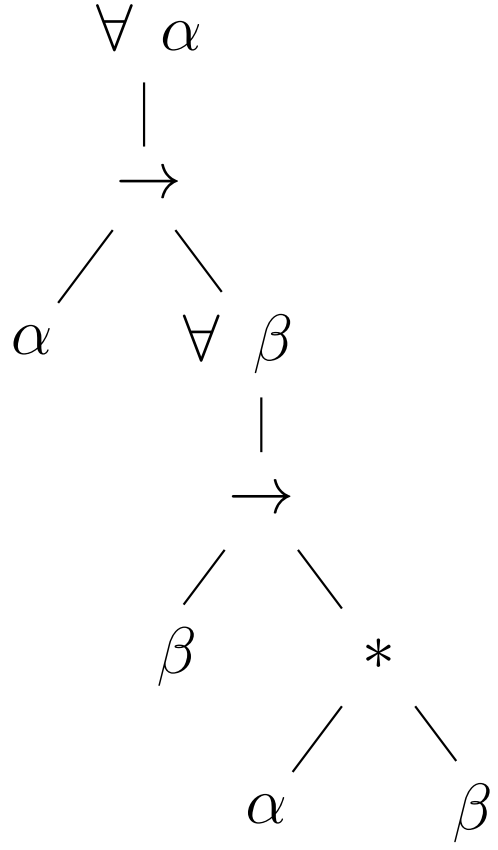
```
module type Type = sig type t end

let f r {T : Type} (x : T.t) = r := x;;
```

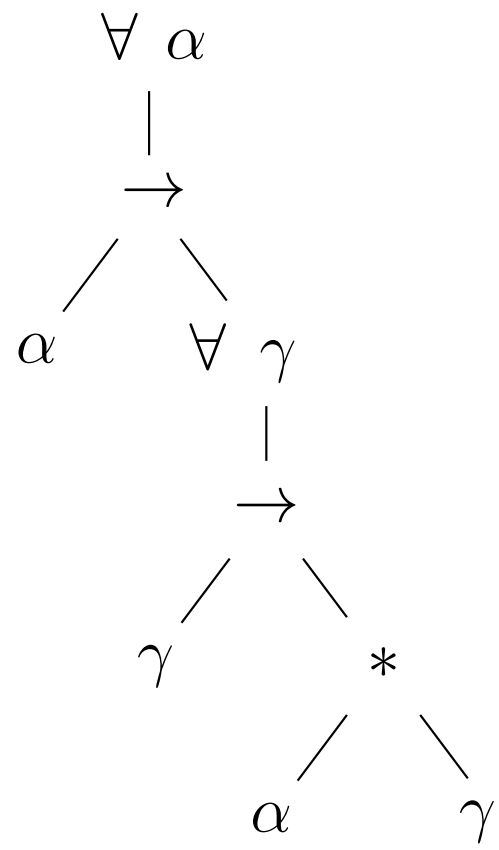


# Interlude : polymorphisme de rang N

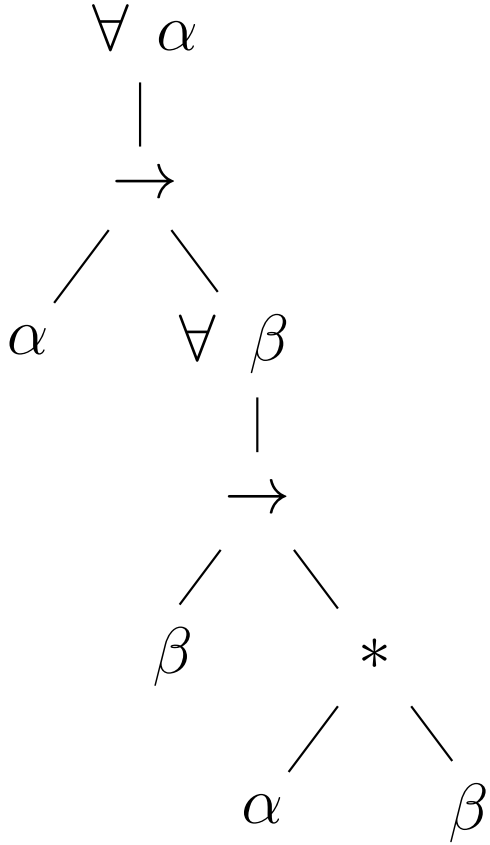


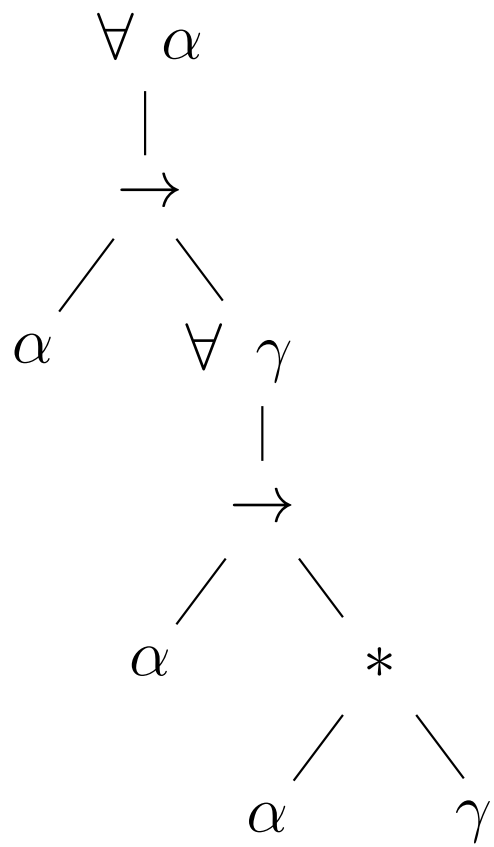
# Interlude : polymorphisme de rang N



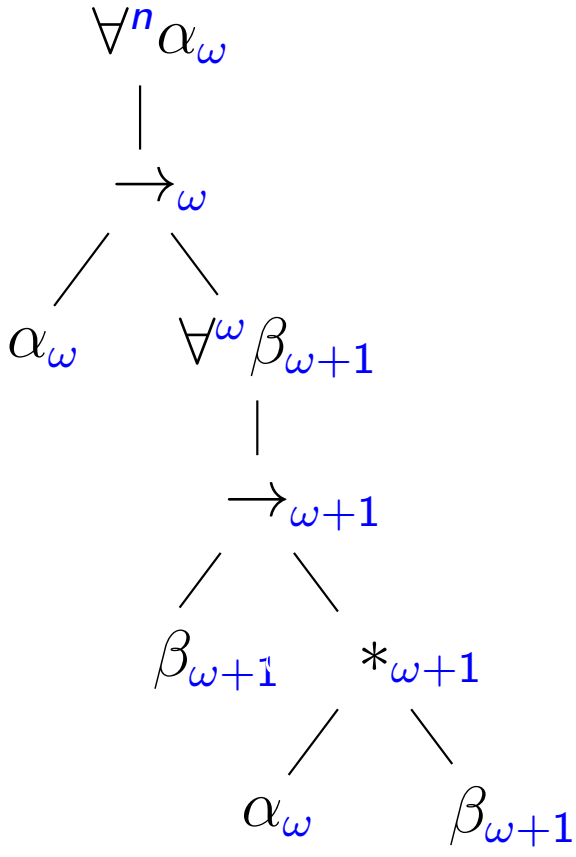



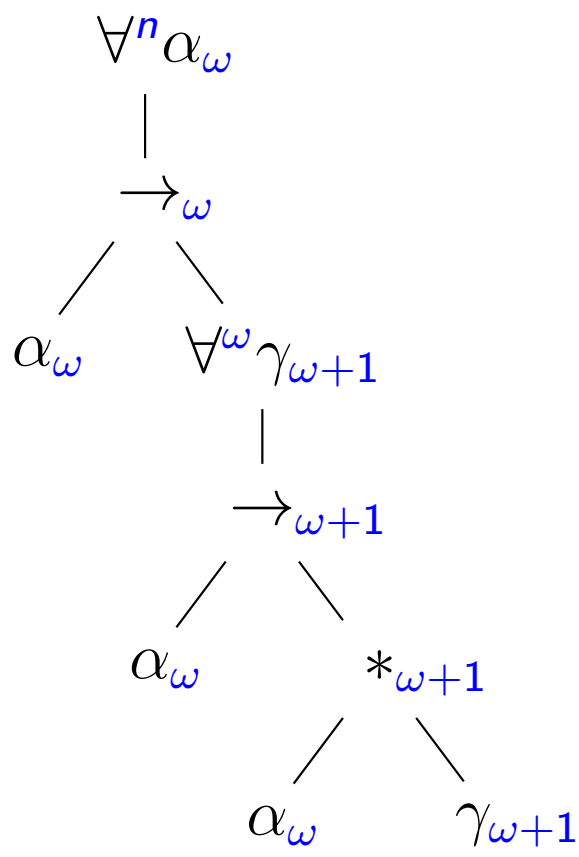
# Interlude : polymorphisme de rang N



# Interlude : polymorphisme de rang N



# Interlude : types équi-récursifs

« *Numbering matters : First-order canonical forms for second-order recursive types.* », Nadji Gauthier and François Pottier.

Modular Implicits : une brève introduction

Modular Explicits

Interlude : du typage avec niveaux

Et maintenant ?

# Et maintenant ?

- Finir le travail d'implémentation de MX et merger
- Continuer le travail sur les niveaux

# Et maintenant ?

- Finir le travail d'implémentation de MX et merger
- Continuer le travail sur les niveaux
- Se concentrer sur la partie « recherches d'arguments implicites »
  - clarifier la spécification de la proposition de 2014
  - répertorier des exemples problématiques
  - regarder ce qui est fait ailleurs : Coq, Agda, etc. (« *HO pattern unification* »)