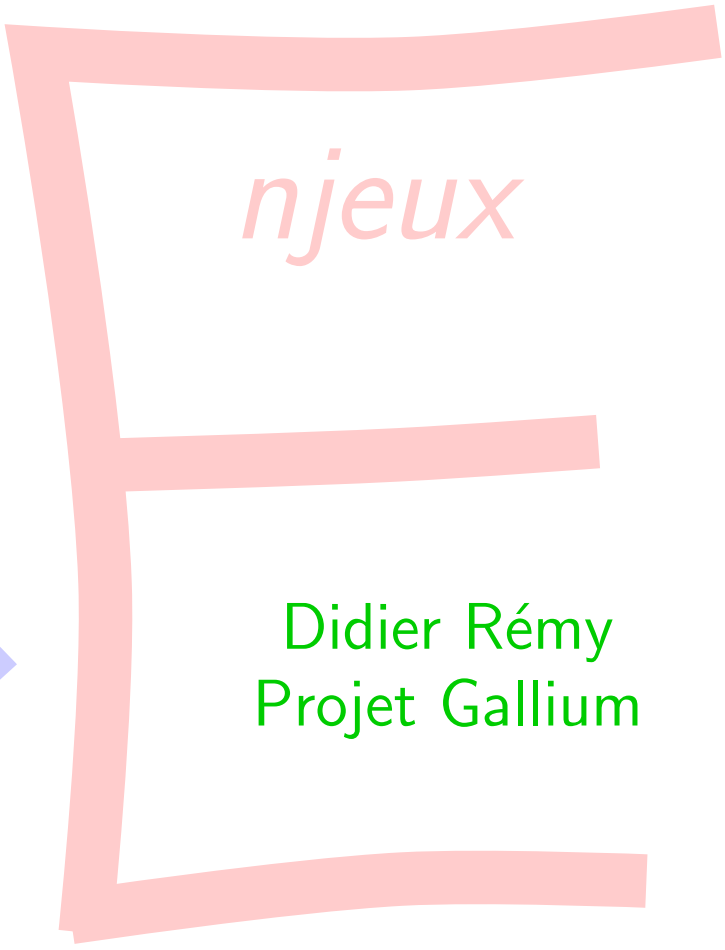
Interlude Scientifique

Inférence



Didier Rémy
Projet Gallium

Décembre 2009

Pourquoi ?

Sans typage dans OCaml, on pourrait écrire un peu n'importe quoi.

- Des programmes qui n'ont aucun sens : que fait celui-ci ?

  let p = (**fun** x y -> (y x) (y x) (**fun** u -> x u x))

- Des programmes qui bouclent (sans s'en rendre compte).

  let f x = x x **in** f f ;;

- Des typos, sans s'en rendre compte :

  let rec map f = **function** x::rest -> f x, map f rest

- Des erreurs de conception, comme mettre du grain et des cailloux dans un même sac. C'est courrir le risque d'oublier de faire le tri avant de porter le sac au moulin...

Pourquoi ?

Sans typage dans OCaml...

- On compilerait de façon moins efficace.
  - les types ne sont pas connus statiquement.
  - il faut les vérifier à l'exécution.
- Il serait plus difficile de modifier les programmes existants
  - Le typeur pointe sur les endroits où il faut ajouter de nouveaux cas, de nouveaux arguments, *etc.*

Sans types, le langage est trop libéral, donc trop dangereux.

Pour dompter le langage !

Les types permettent de rendre le langage moins sauvage, de l'apprivoiser.

- On ne peut plus écrire tout et n'importe quoi.
  - C'est bien ! Car de nombreuses erreurs de programmation rendent les programmes mal typés et sont ainsi détectées.
  - Mais c'est aussi parfois un problème : Car on perd aussi certains programmes intéressants.
- On peut structurer le programme à partir des types.
  - en définissant d'abordles structures de données... principalement leurs types
  - puis les fonctions qui opèrent sur ces structures de données
- On utilise les types pour écrire des interfaces claires pour les librairies et obliger l'utilisateur à les respecter.

Pour ou contre ?

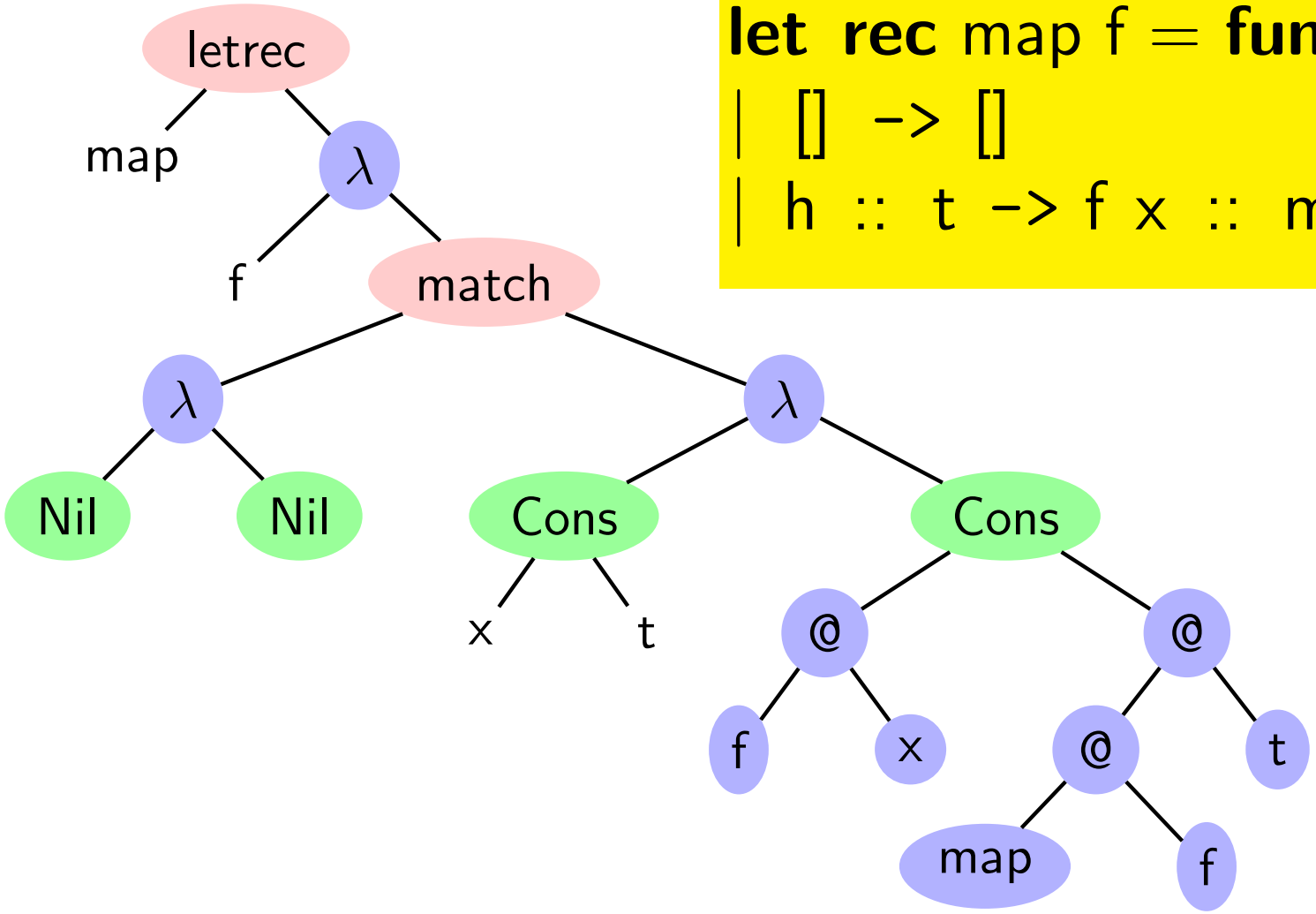
- Les bénéfices sont largement supérieurs aux inconvénients :
- Il faut conserver le typage, bien entendu, mais aussi
- rendre les systèmes de types plus puissants.... sans trop les compliquer

Les programmes

```
let rec map f = function
| [] -> []
| h :: t -> f x :: map f t
```

Les programmes

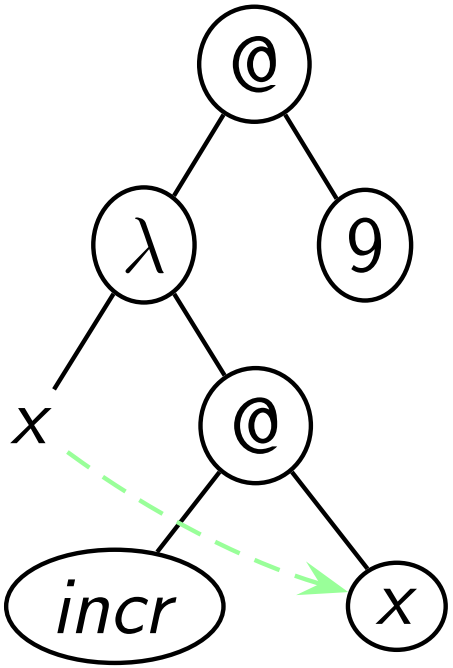
Sa syntaxe abstraite :




```
let rec map f = function
| [] -> []
| h :: t -> f x :: map f t
```

Les programmes sont modélisés par le lambda-calcul :

.

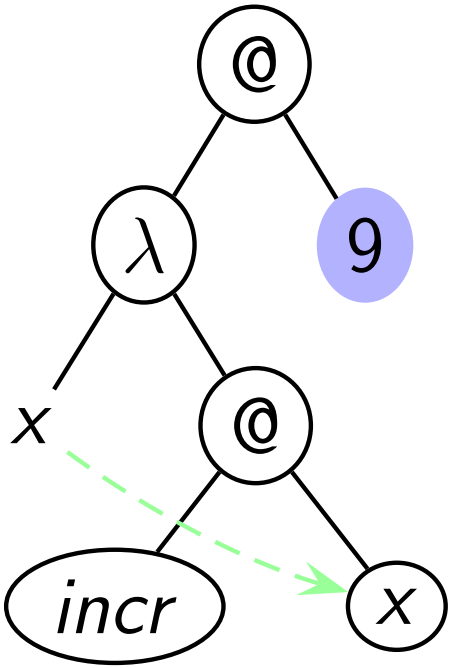


Expression

a

Les programmes sont modélisés par le lambda-calcul :

.

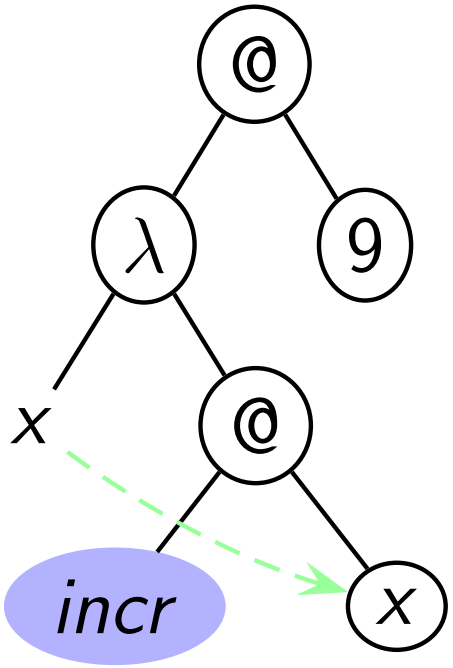


Constante passive (Constructeur)

c

Les programmes sont modélisés par le lambda-calcul :

.

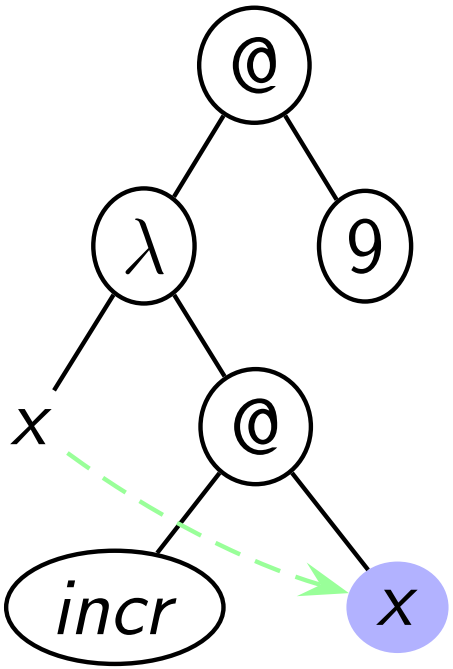


Constante active (destructeur ou primitive)

c

Les programmes sont modélisés par le lambda-calcul :

.

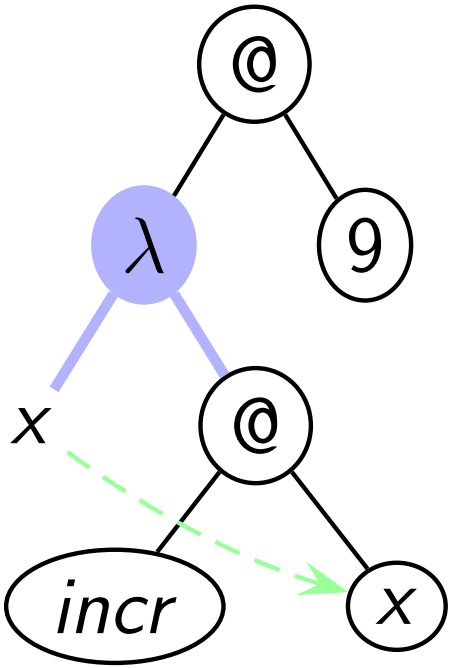


Variable

x

Les programmes sont modélisés par le lambda-calcul :

.

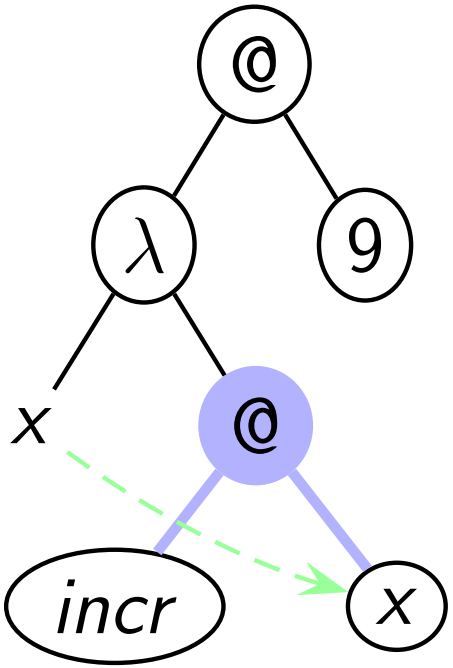


Fonction

$\lambda(x)\, a$

Les programmes sont modélisés par le lambda-calcul :

.

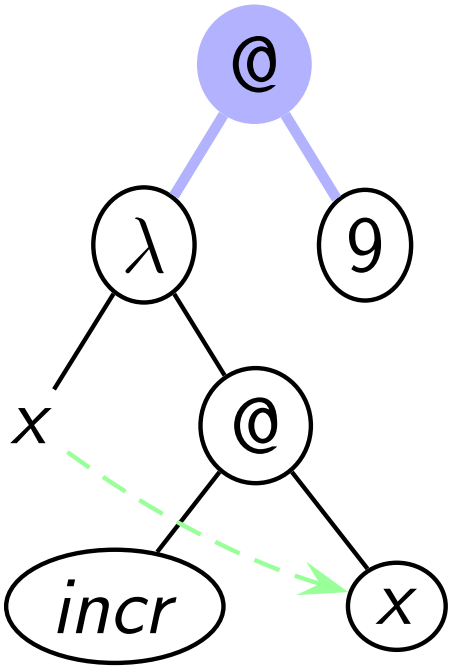


Application (de primitive)

$a_1\,(a_2)$

Les programmes sont modélisés par le lambda-calcul :

.



Application (de fonction)

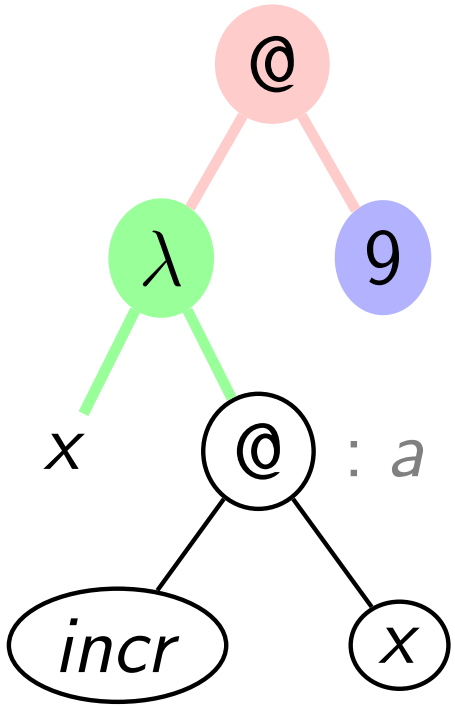
$a_1\,(a_2)$

Le calcul est modélisé par la réduction pas à pas

$$(\lambda(x)\, \mathit{incr}\, (x))\, (9)$$

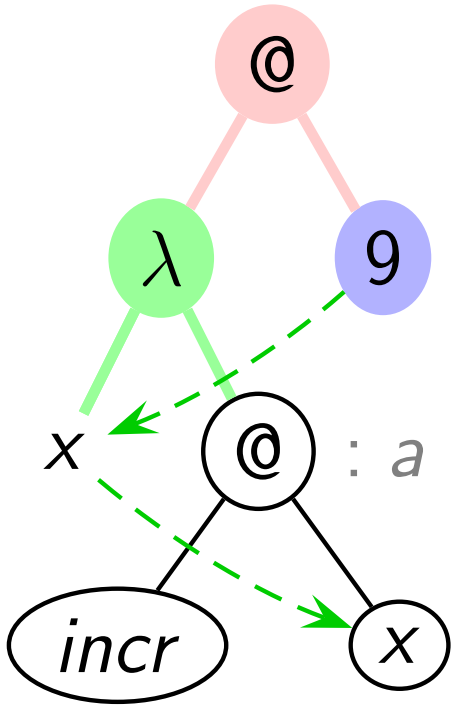
Le calcul est modélisé par la réduction (β) pas à pas

$$(\lambda(x)\, \mathit{incr}\, (x))\, (9)$$



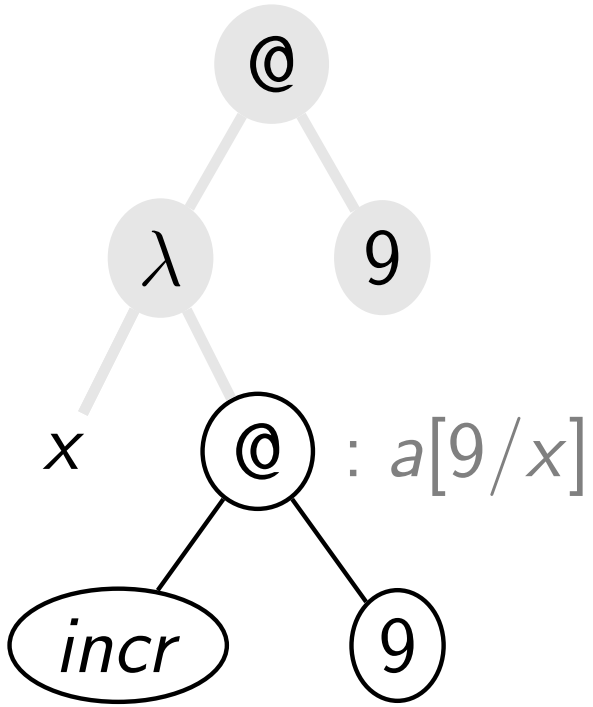
Le calcul est modélisé par la réduction (β) pas à pas

$$(\lambda(x)\, \mathit{incr}\, (x))\, (9)$$



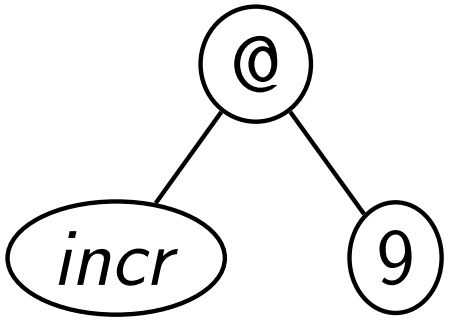
Le calcul est modélisé par la réduction pas à pas

$$(\lambda(x)\, incr\, (x))\, (9)$$



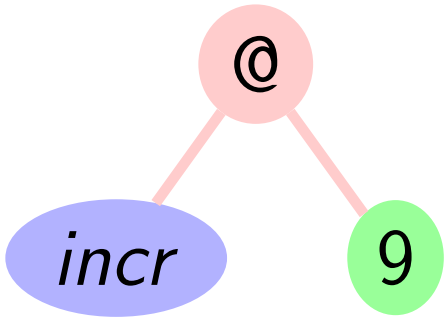
Le calcul est modélisé par la réduction pas à pas

$$(\lambda(x)\, \mathit{incr}\, (x))\, (9) \longrightarrow \mathit{incr}\, (9)$$



Le calcul est modélisé par la réduction (δ) pas à pas

$$(\lambda(x)\, \mathit{incr}\, (x))\, (9) \longrightarrow \mathit{incr}\, (9)$$



Le calcul est modélisé par la réduction pas à pas

$$(\lambda(x)\, \mathit{incr}\,(x))\,(9) \longrightarrow \mathit{incr}\,(9) \longrightarrow 10$$

10

Le calcul est modélisé par la réduction

$$(\lambda(x)\, \mathit{incr}\,(x))\,(9) \longrightarrow \mathit{incr}\,(9) \longrightarrow 10 \not\longrightarrow$$

$$10 \in \mathit{valeur}$$

Le calcul est modélisé par la réduction

$$(\lambda(x)\, \mathit{incr}\,(x))\,(9) \longrightarrow \mathit{incr}\,(9) \longrightarrow 10 \not\longrightarrow$$

$$10 \in \mathit{valeur}$$

Les réductions infinies sont permises (terminaison indécidable)

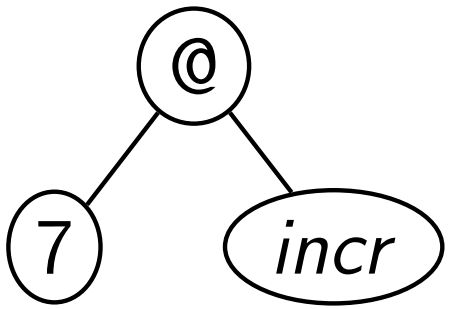
Le calcul est modélisé par la réduction

$$(\lambda(x)\, \mathit{incr}\,(x))\,(9) \longrightarrow \mathit{incr}\,(9) \longrightarrow 10 \not\longrightarrow$$

$$10 \in \mathit{valeur}$$

Les réductions infinies sont permises (terminaison indécidable)

Les erreurs sont modélisées par la réduction bloquée



$$7\,(\mathit{incr}) \begin{cases} \not\longrightarrow \\ \notin \mathit{valeur} \end{cases}$$

Les types

Les types de base

booléens

$true \quad : bool,$

$false \quad : bool.$

entiers

$0 \quad : int$

$1 \quad : int \qquad \ldots$

Les types flèches

primitives

$incr \quad : int \longrightarrow int$

$iszero \quad : int \longrightarrow bool$

fonctions

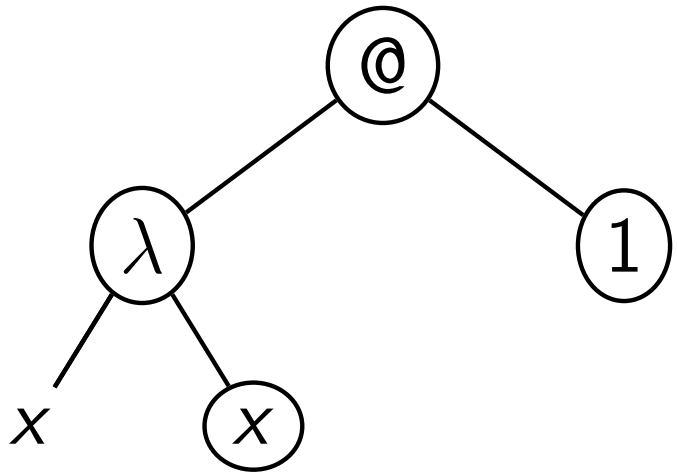
$\lambda(x)\, incr\,(x) \quad : int \longrightarrow int$

$\lambda(f)\, iszero\,(f\,(0)) \quad : (int \longrightarrow int) \longrightarrow bool$

Prémunissent contre les erreurs

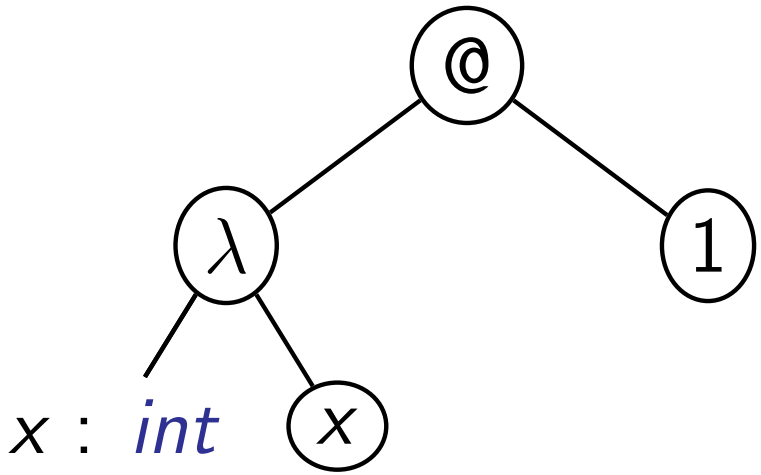
$7(incr)$ mal typé

Typage et inférence



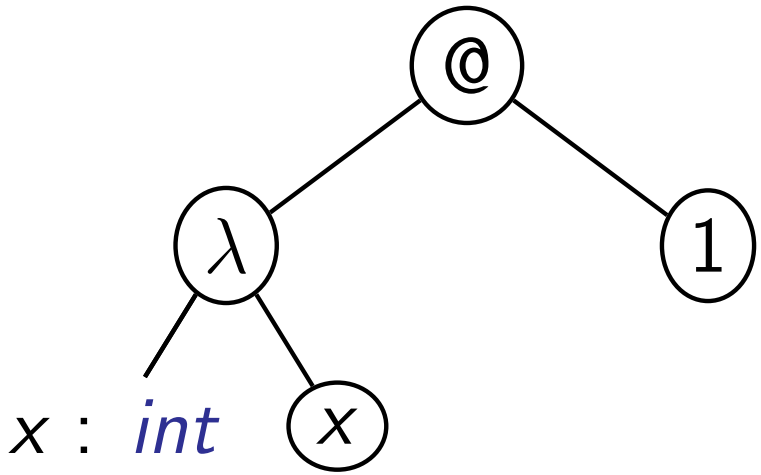
Une expression implicitement typée

Typage et inférence



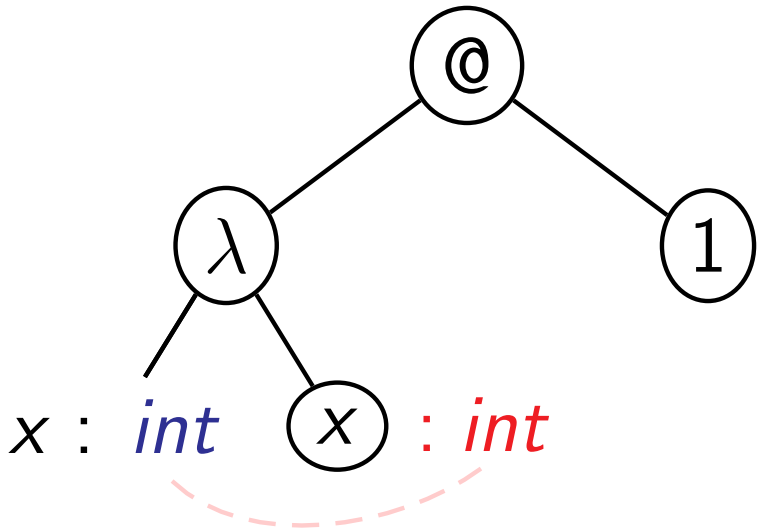
Une expression explicitement typée

Typage et inférence



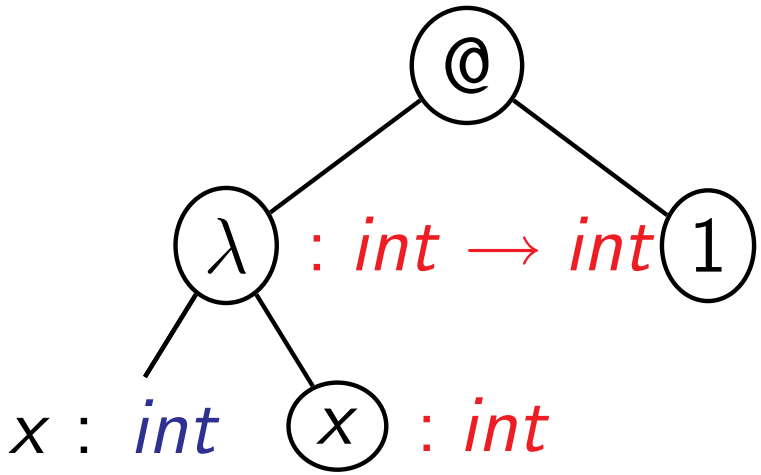
On peut calculer/vérifier les types de chaque nœud

Typage et inférence



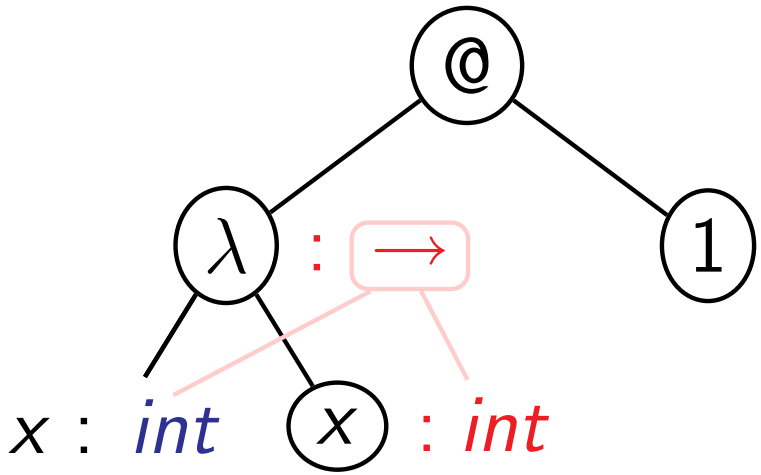
On peut calculer/vérifier les types de chaque nœud

Typage et inférence



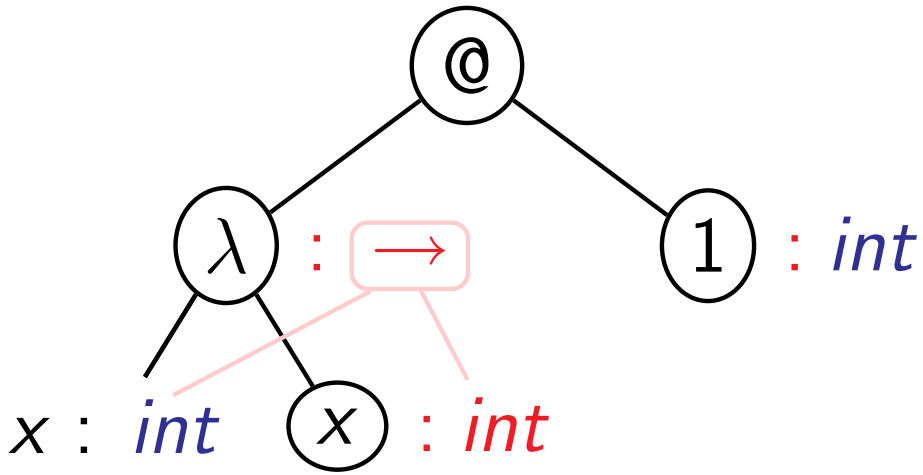
On peut calculer/vérifier les types de chaque nœud

Typage et inférence



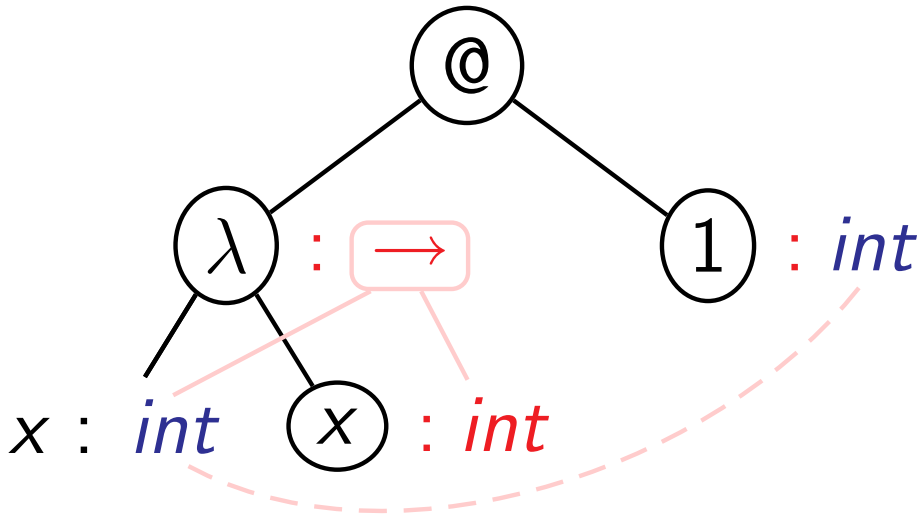
On peut calculer/vérifier les types de chaque nœud

Typage et inférence



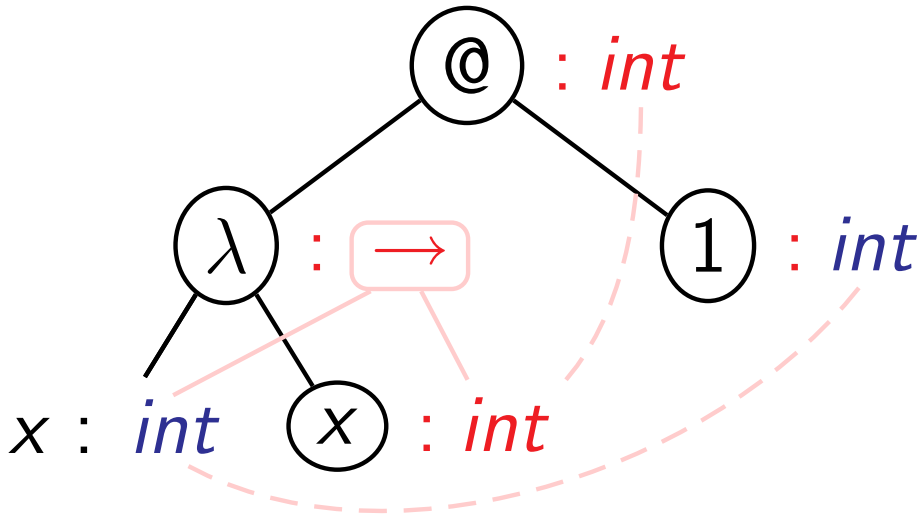
On peut calculer/vérifier les types de chaque nœud

Typage et inférence



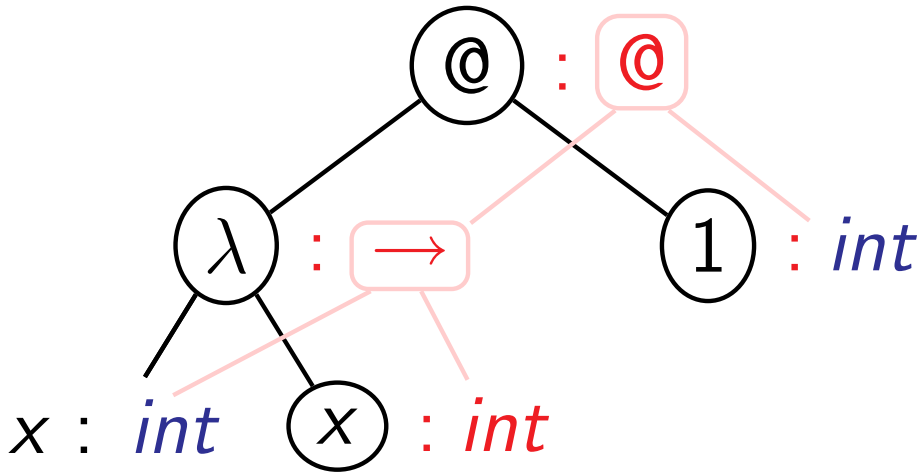
On peut calculer/vérifier les types de chaque nœud

Typage et inférence



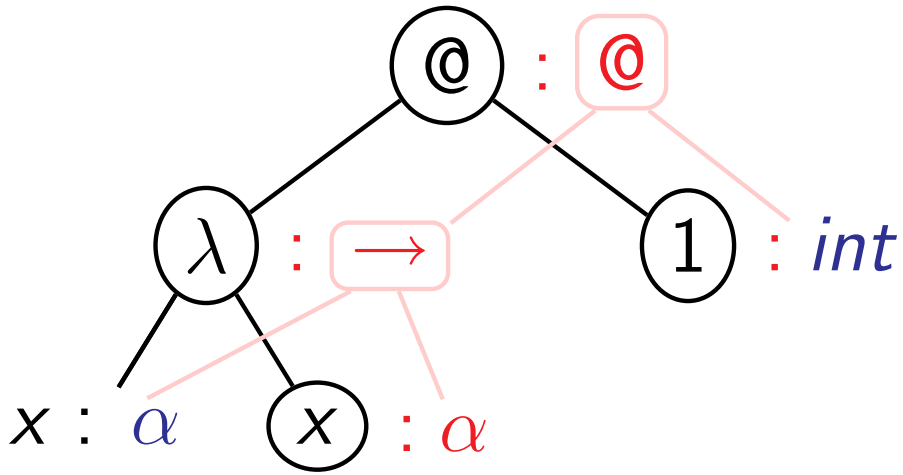
On peut calculer/vérifier les types de chaque nœud

Typage et inférence



On peut calculer/vérifier les types de chaque nœud

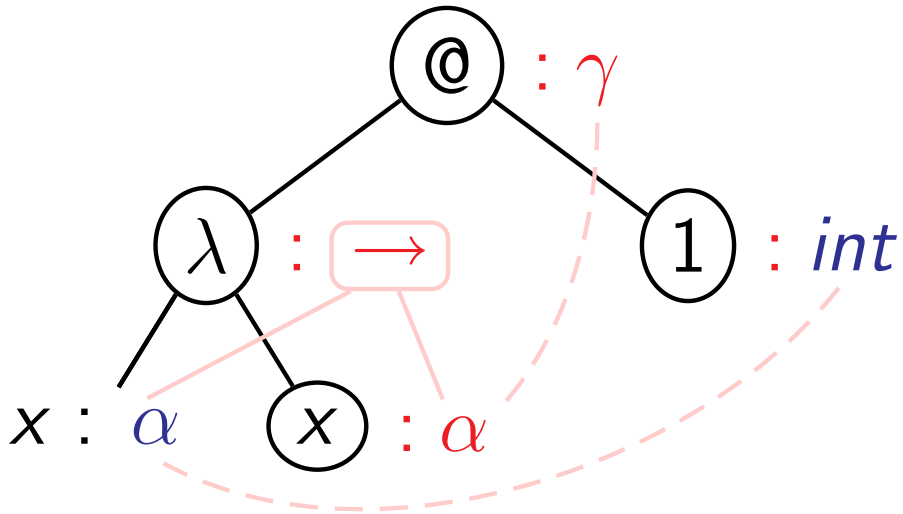
Typage et inférence



On peut aussi synthètiser les types de chaque nœud

α variable de type $=$ inconnue

Typage et inférence

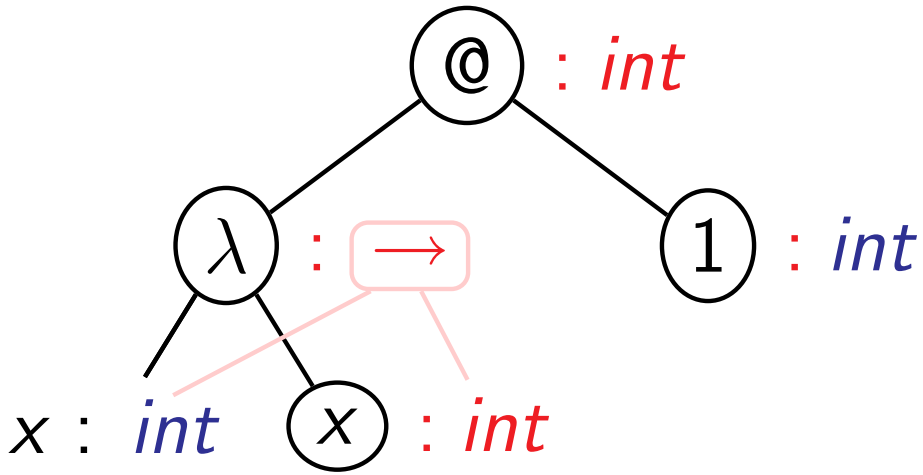


On peut aussi synthètiser les types de chaque nœud

α variable de type $=$ inconnue

en résolvant un problème d'unification de première ordre

Typage et inférence



On peut aussi synthètiser les types de chaque nœud

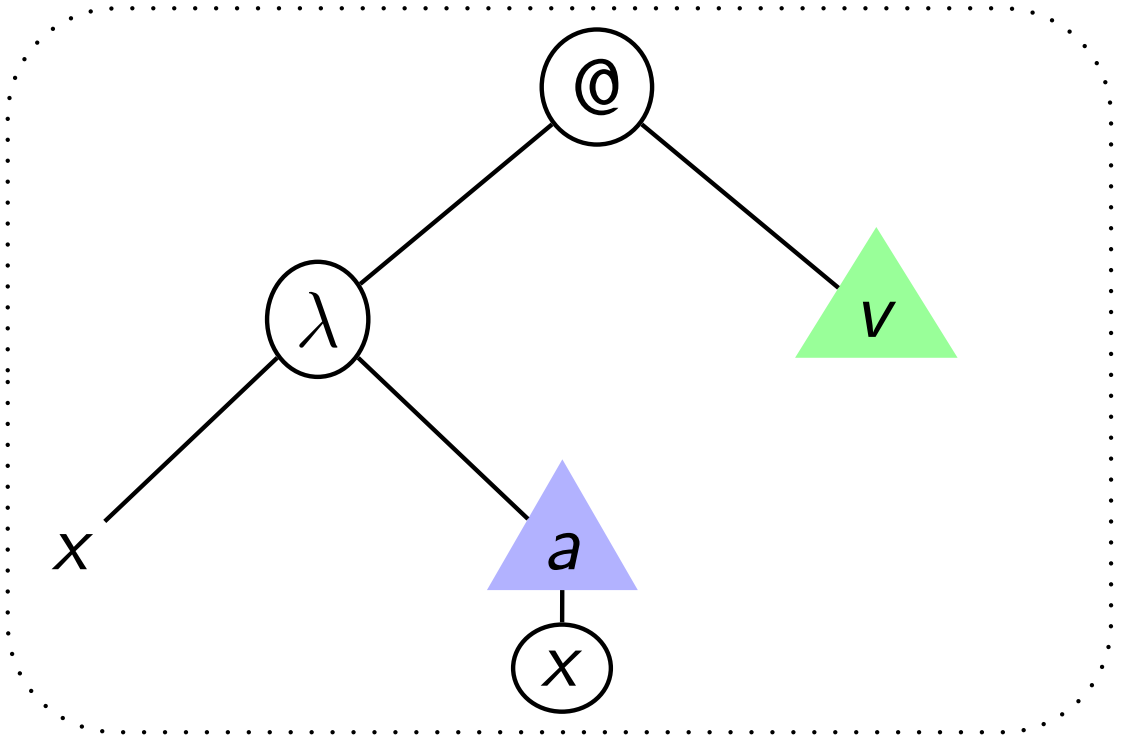
α variable de type $=$ inconnue

en résolvant un problème d'unification de première ordre

solution : $\alpha = \gamma = \mathit{int}$

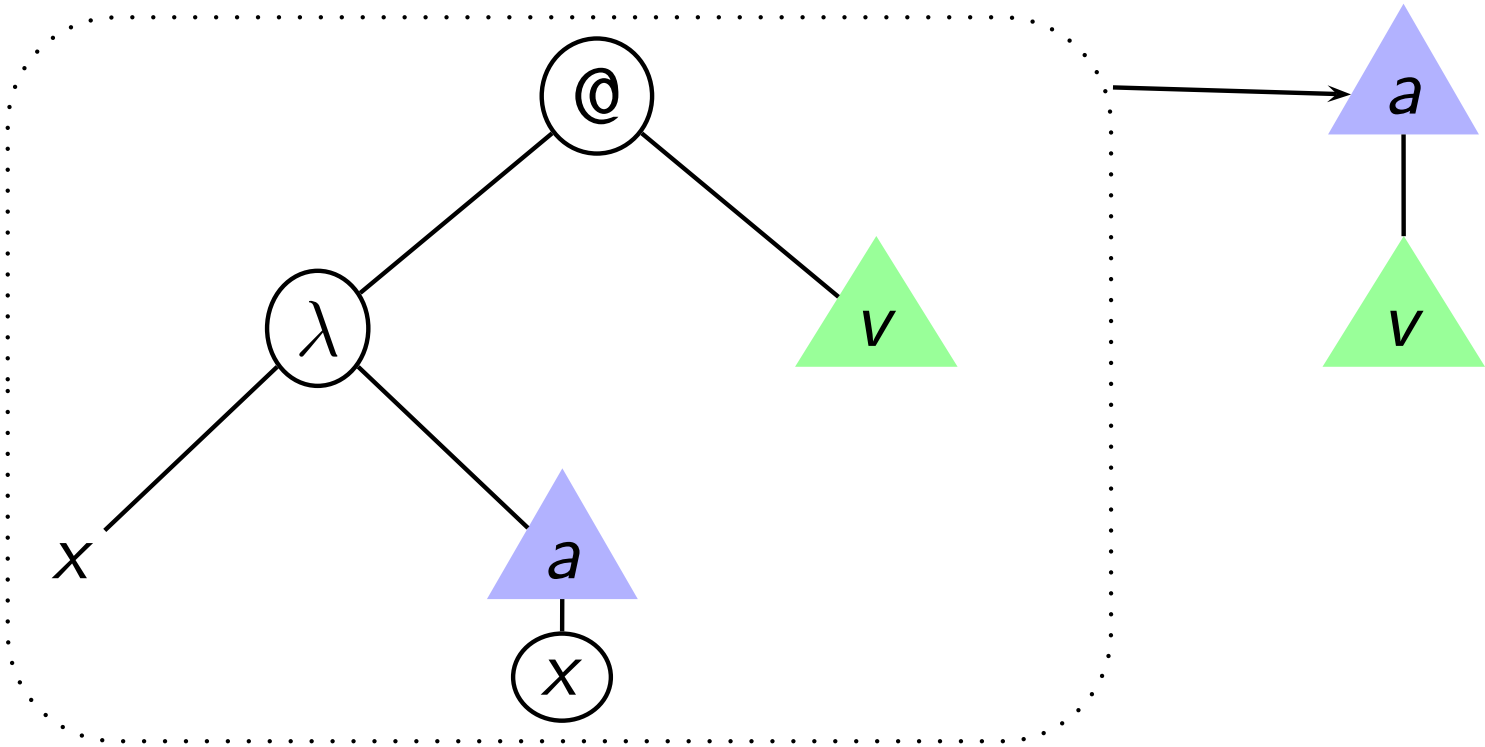
Les types, garantissent la sûreté.

Les types sont préservés par la réduction



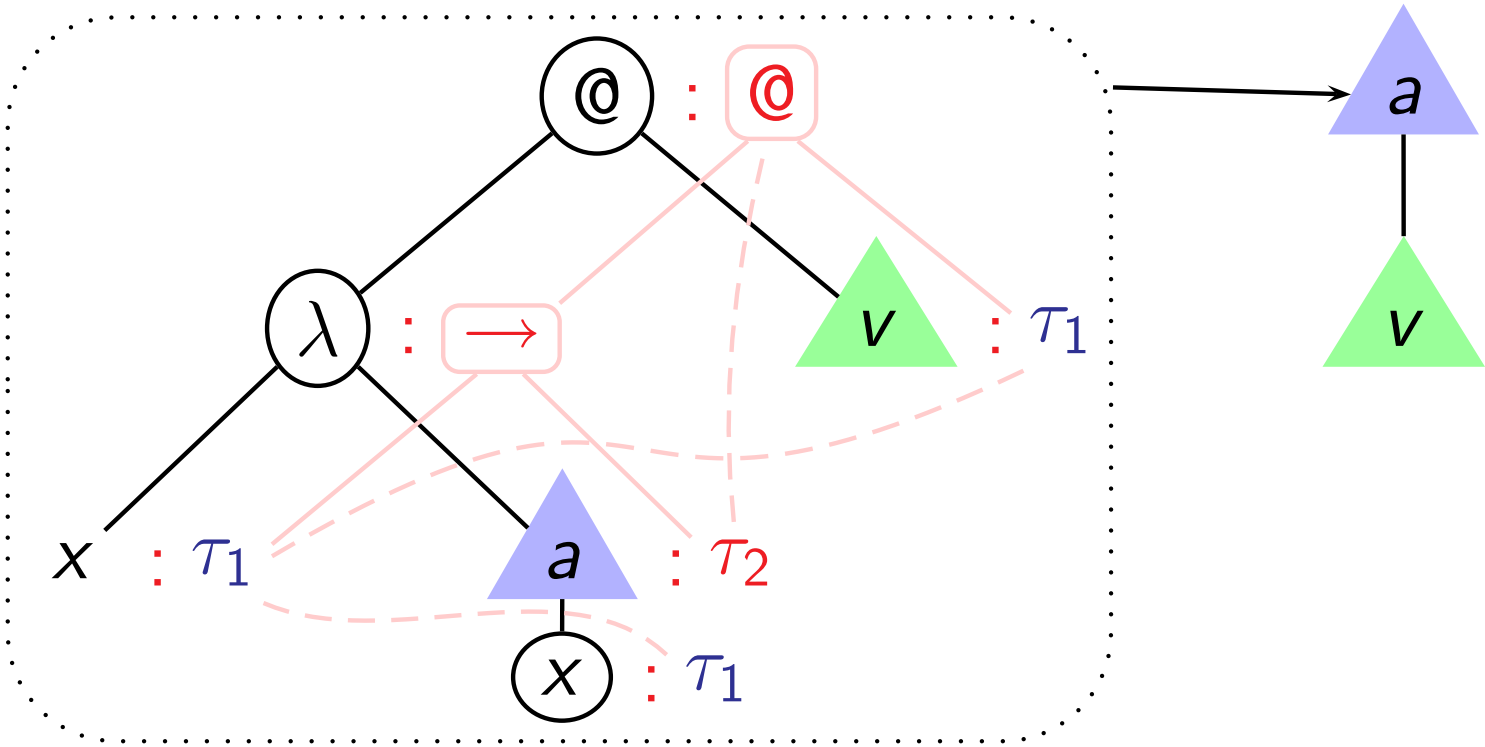
Les types, garantissent la sûreté.

Les types sont préservés par la réduction



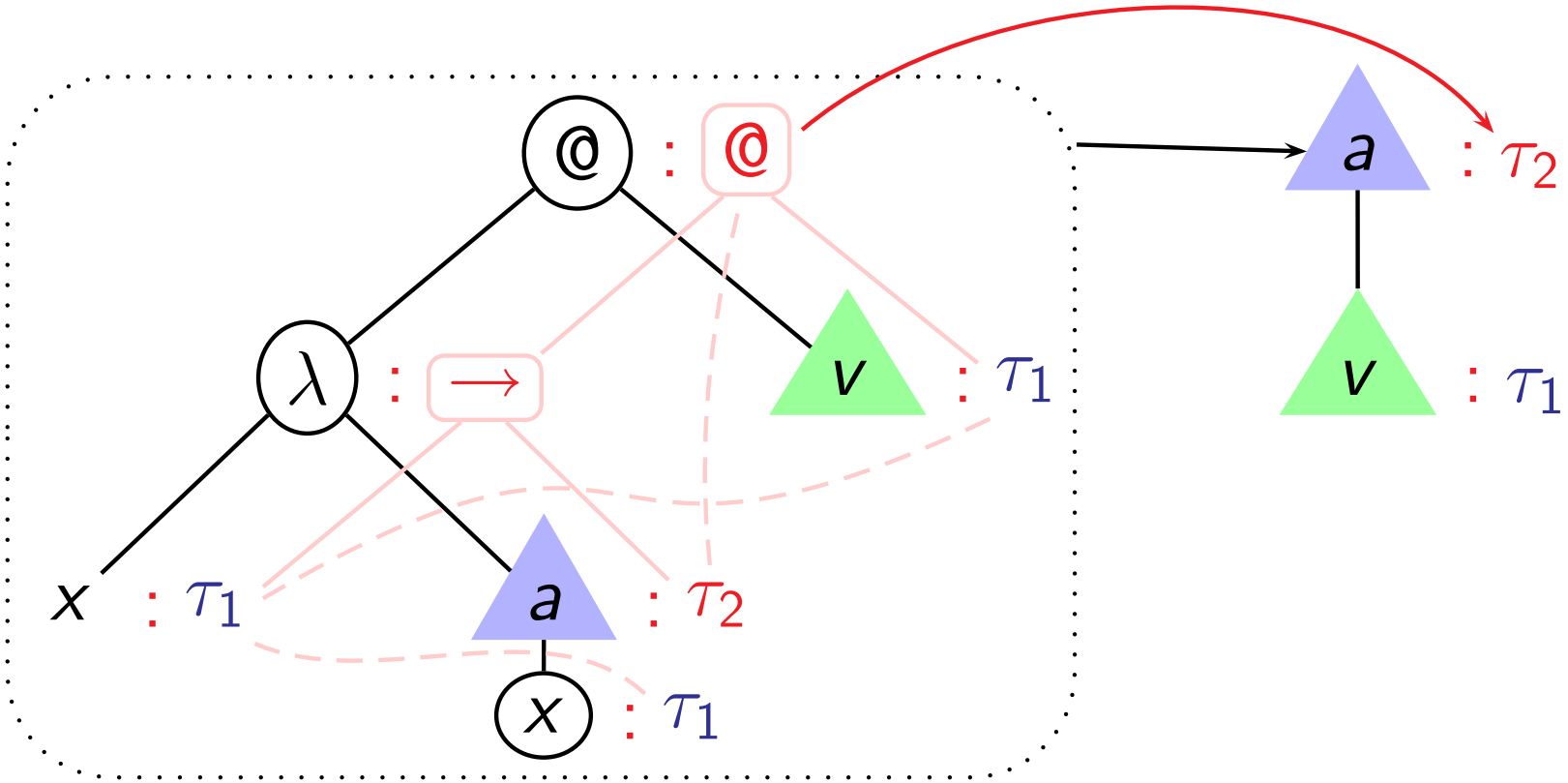
Les types, garantissent la sûreté.

Les types sont préservés par la réduction



Les types, garantissent la sûreté.

Les types sont préservés par la réduction



Les types, garantissent la sûreté.

Les types sont préservés par la réduction

Par hypothèse pour les réductions de primitives

$$incr\,(1) : int \longrightarrow 2 : int$$

Les types, garantissent la sûreté.

Les types sont préservés par la réduction

Par hypothèse pour les réductions de primitives

Les programmes bien typés irréductibles sont des valeurs

Les types, garantissent la sûreté.

Les types sont préservés par la réduction

Par hypothèse pour les réductions de primitives

Les programmes bien typés irréductibles sont des valeurs

Sûreté de l'évaluation

a_1

Les types, garantissent la sûreté.

Les types sont préservés par la réduction

Par hypothèse pour les réductions de primitives

Les programmes bien typés irréductibles sont des valeurs

Sûreté de l'évaluation

a_1

$: \tau$

Les types, garantissent la sûreté.

Les types sont préservés par la réduction

Par hypothèse pour les réductions de primitives

Les programmes bien typés irréductibles sont des valeurs

Sûreté de l'évaluation

$$a_1 \longrightarrow a_2$$

$$: \tau$$

Les types, garantissent la sûreté.

Les types sont préservés par la réduction

Par hypothèse pour les réductions de primitives

Les programmes bien typés irréductibles sont des valeurs

Sûreté de l'évaluation

$$\begin{array}{ccc} a_1 & \longrightarrow & a_2 \\ : \tau & & : \tau \end{array}$$

Les types, garantissent la sûreté.

Les types sont préservés par la réduction

Par hypothèse pour les réductions de primitives

Les programmes bien typés irréductibles sont des valeurs

Sûreté de l'évaluation

$$a_1 \longrightarrow a_2 \longrightarrow \ldots$$

$$: \tau \qquad : \tau \qquad \ldots$$

Les types, garantissent la sûreté.

Les types sont préservés par la réduction

Par hypothèse pour les réductions de primitives

Les programmes bien typés irréductibles sont des valeurs

Sûreté de l'évaluation

$$a_1 \longrightarrow a_2 \longrightarrow \ldots \longrightarrow a_n \not\longrightarrow$$

$$: \tau \qquad : \tau \qquad \ldots \qquad : \tau$$

Les types, garantissent la sûreté.

Les types sont préservés par la réduction

Par hypothèse pour les réductions de primitives

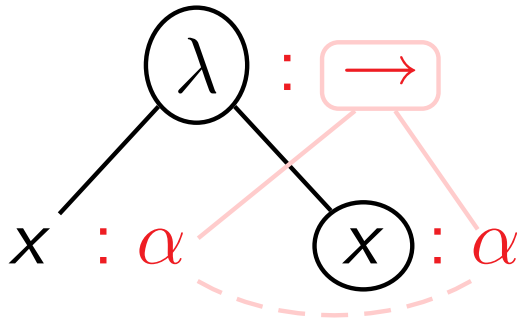
Les programmes bien typés irréductibles sont des valeurs

Sûreté de l'évaluation

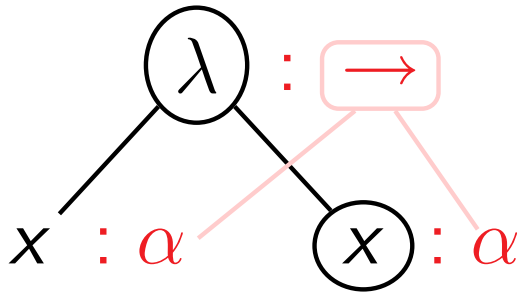
$$a_1 \longrightarrow a_2 \longrightarrow \ldots \longrightarrow a_n \in \mathit{valeurs}$$

$$: \tau \qquad : \tau \qquad \ldots \qquad : \tau$$

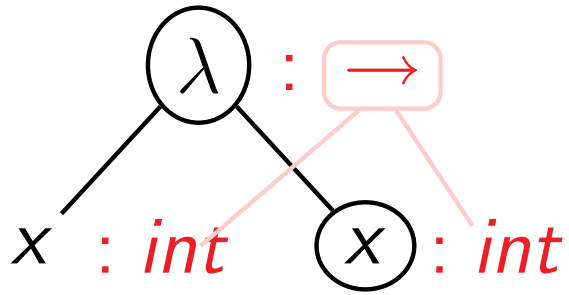
Polymorphisme : implicite



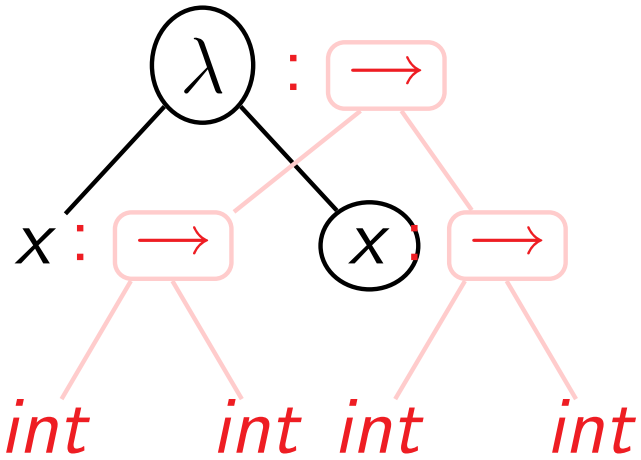
Polymorphisme : implicite



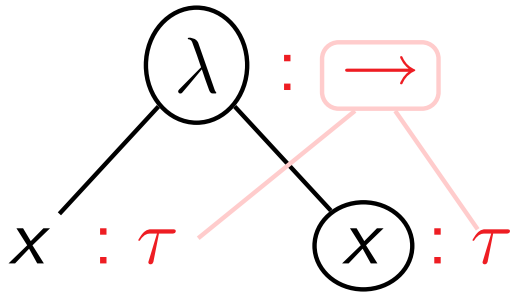
Polymorphisme : implicite



Polymorphisme : implicite

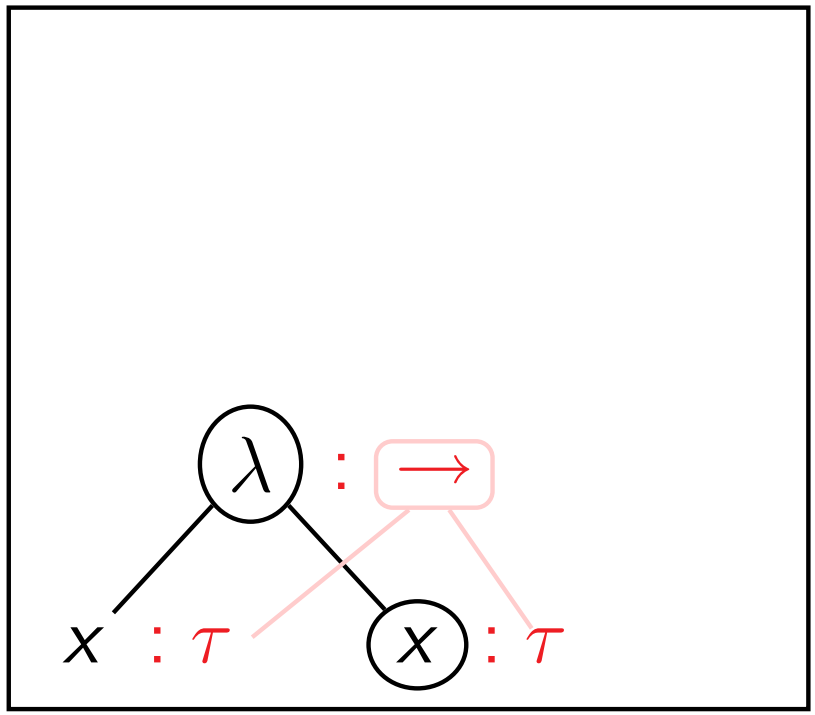


Polymorphisme : implicite



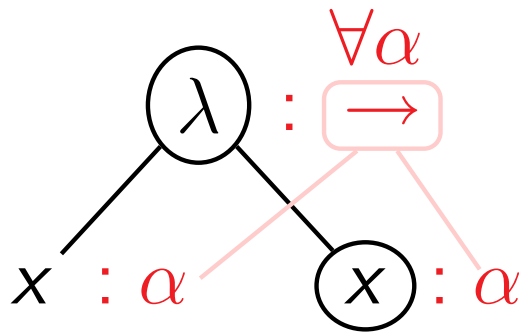
Polymorphisme : implicite

$\forall\tau$



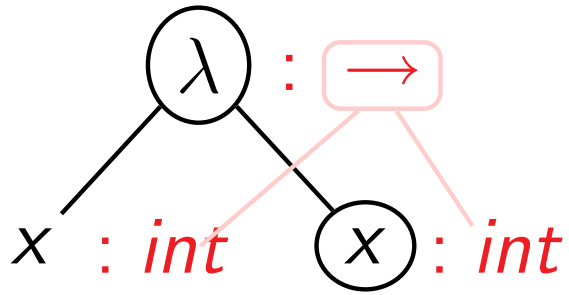
Quantification dans le méta-langage

Polymorphisme : type principaux



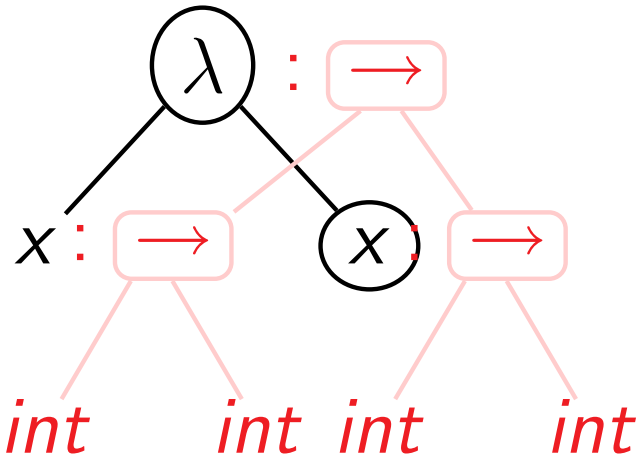
Quantification dans le langage = types polymorphes

Polymorphisme : type principaux



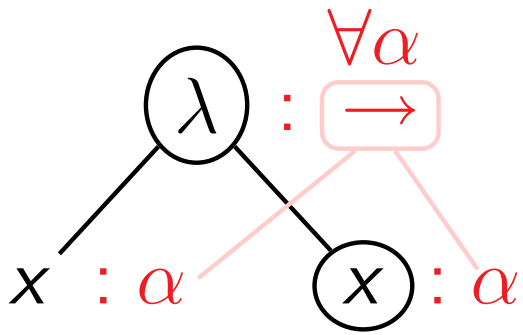
On retrouve les autres types par instantiation

Polymorphisme : type principaux

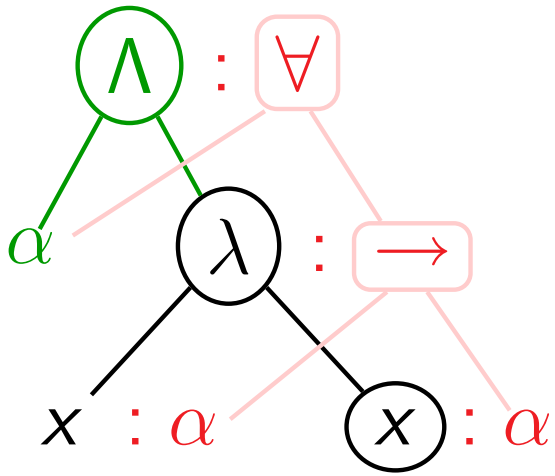


On retrouve les autres types par instantiation

Polymorphisme : implicite

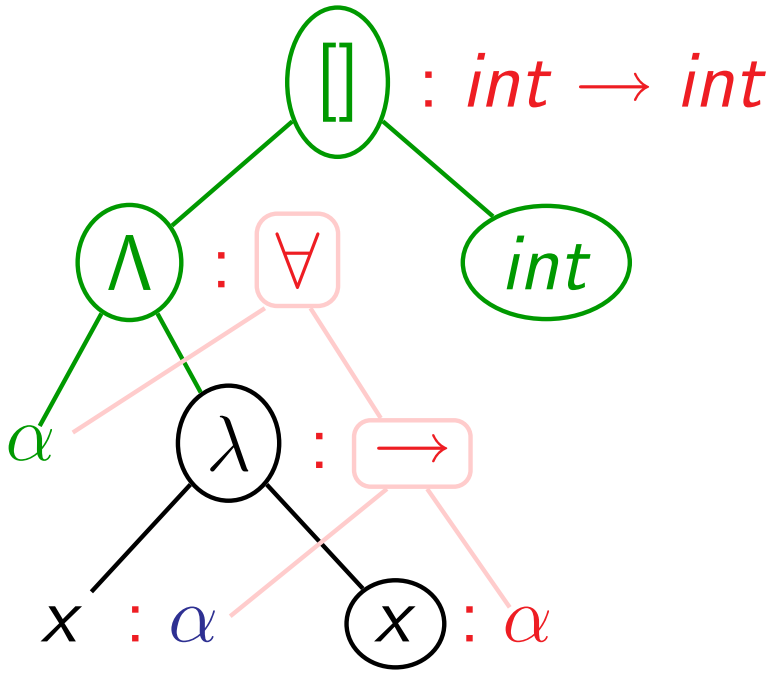


Polymorphisme : explicite



Abstraction de type

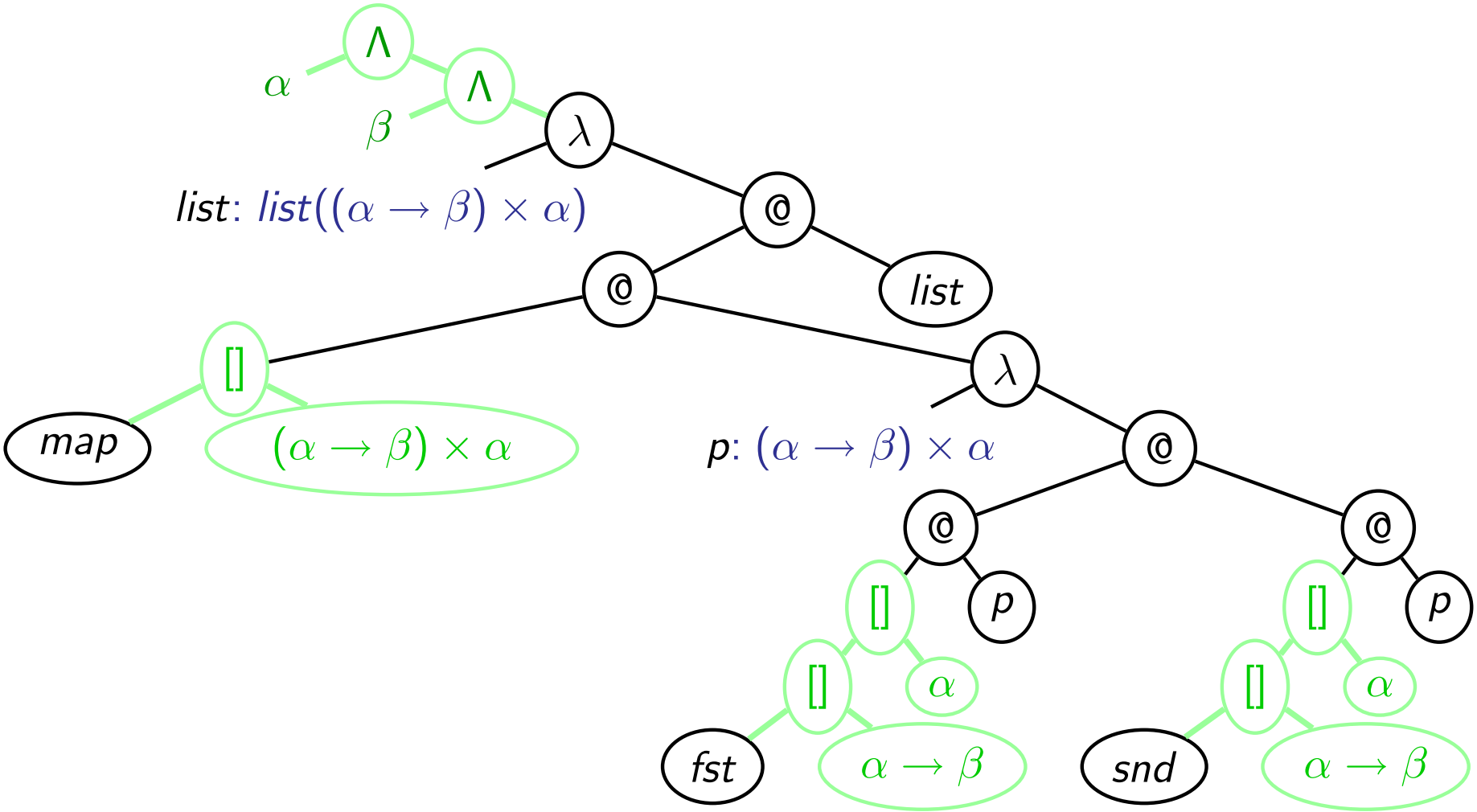
Polymorphisme : explicite



Application de type

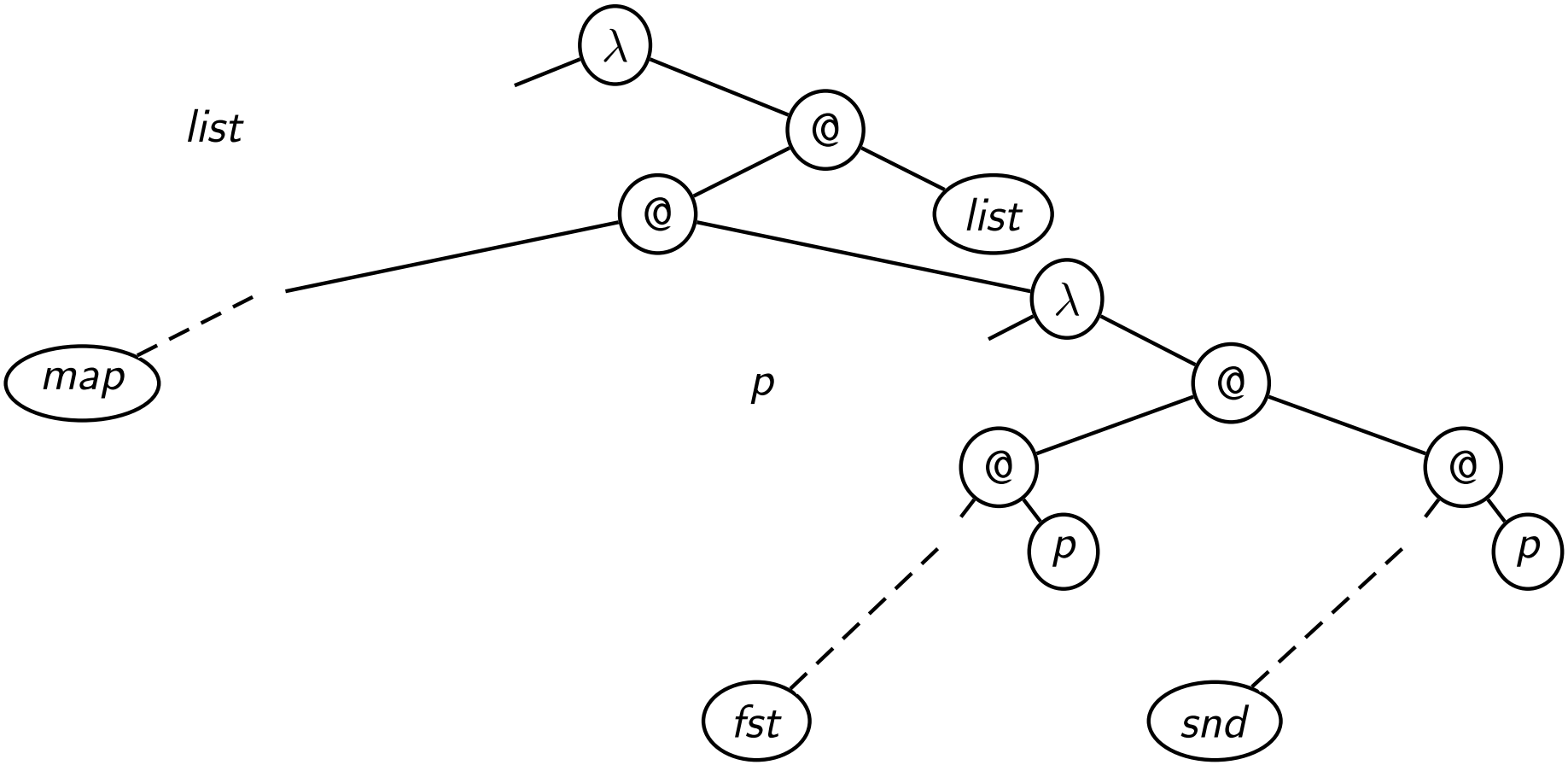
Typage : explicite

$\Lambda(\alpha)\Lambda(\beta)\lambda(\mathit{list}\colon \mathit{list}((\alpha \to \beta) \times \alpha))\ \mathit{map}\ [(\alpha \to \beta) \times \alpha]$
$(\lambda(p\colon (\alpha \to \beta) \times \alpha)\,(\mathit{fst}\ [\alpha \to \beta]\ [\alpha]\ (p))\ ((\mathit{snd}\ [\alpha \to \beta]\ [\alpha]\ (p))))\ (\mathit{list})$



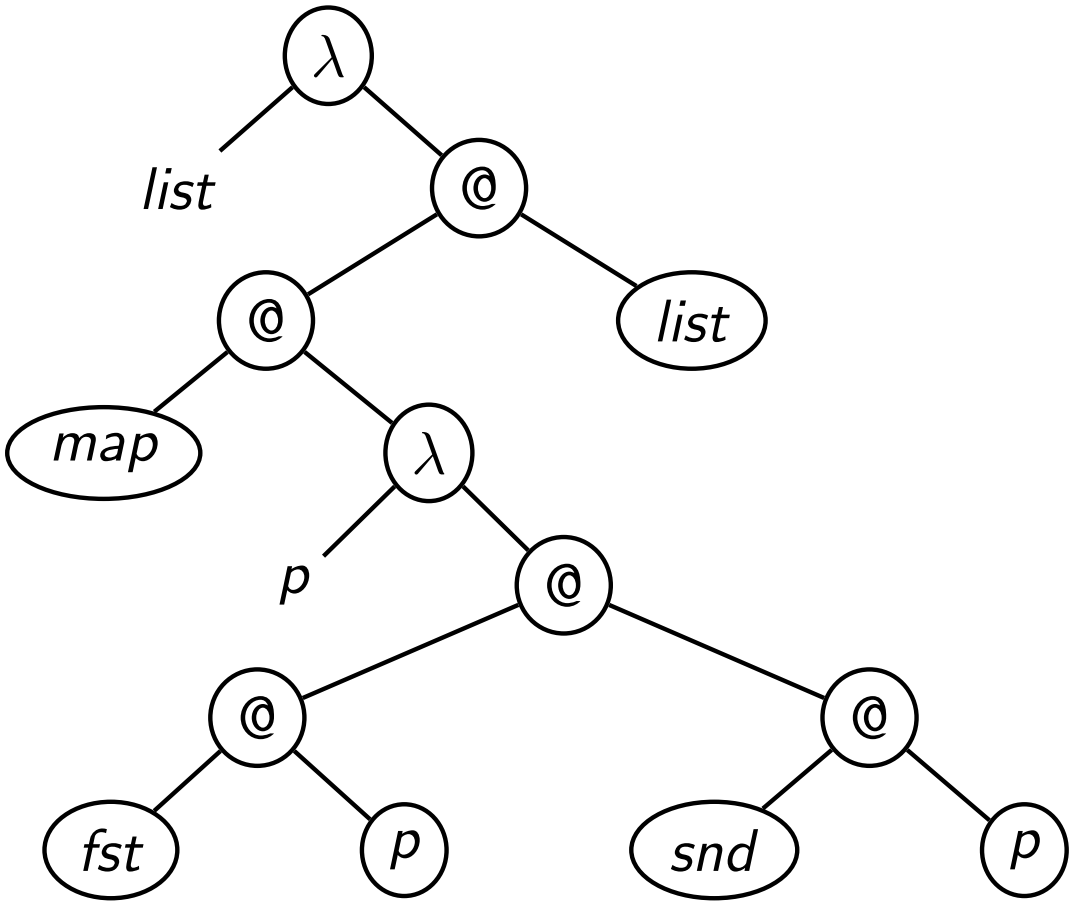
Typage : implicite

$\lambda(list) (\lambda(p) (fst) (p)) ((snd) (p)))) (list)) map$



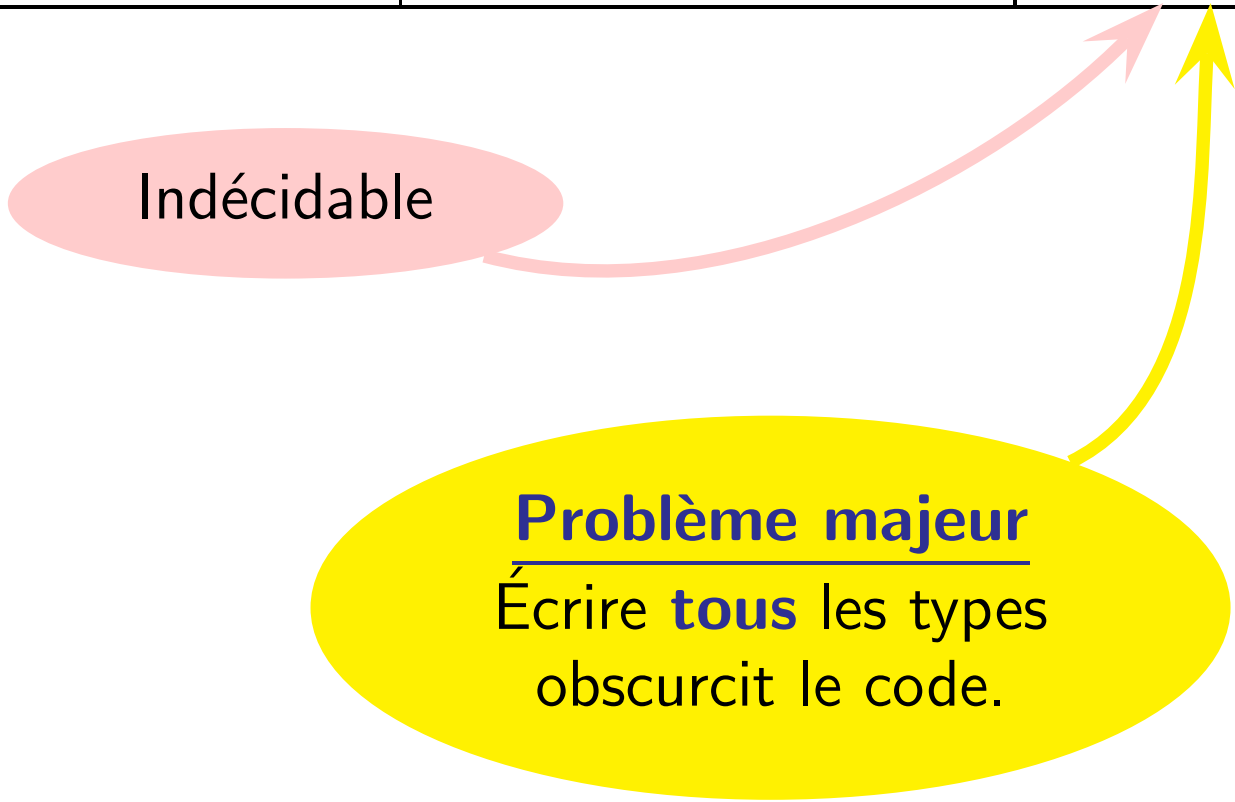
Typage : implicite

$$\lambda(list)map\,(\lambda(p)(fst\,(p))\,((snd\,(p))))\,(list)$$



Zoologie

Variante	**Polymorphisme**	**Inférence**
Types simples	**Non**	Oui
Système F	Oui	**Non**



Zoologie

Variante	Polymorphisme	Inférence
Types simples	**Non**	Oui
Système F	Oui	**Locale**

Inférence partielle

On propage les types entre noeuds voisins. quand on peut.

Ad hoc

- Difficile de prévoir les annotations nécessaires
- Peu robuste aux changements, même mineurs

Zoologie

Variante	Polymorphisme	Inférence
Types simples		Oui
Système F		**Locale**

Pas d'inférence «propre» dans le Système F

Inf

On
entr
qu

Au fil

- Difficile de prévoir les annotations nécessaires
- Peu robuste aux changements, même mineurs

Zoologie

Variante	**Polymorphisme**	**Inférence**
Types simples	**Non**	Oui
Système F	Oui	**Non**
ML	Restreint	Oui
ML + F.	Oui	Oui

ML

C'est plus de 30 ans de succès...

- d'abord académique,
- puis (un peu) industriel

ML

C'est plus de 30 ans de succès...

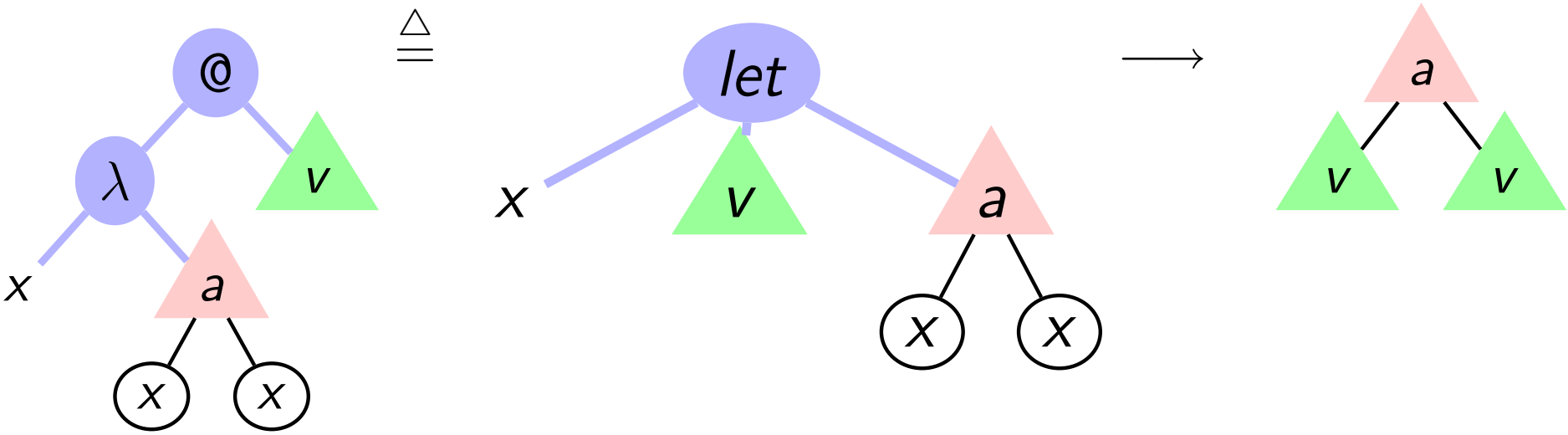
- d'abord académique,
- puis (un peu) industriel

Il est veillissant...

- plus beaucoup de grandes nouveautés.
- besoin d'un nouveau souffle, d'une nouvelle génération avec :
  - de l'ordre supérieur,
  - un meilleur traitement des effets,
  - une meilleure intégration avec les systèmes de preuve.

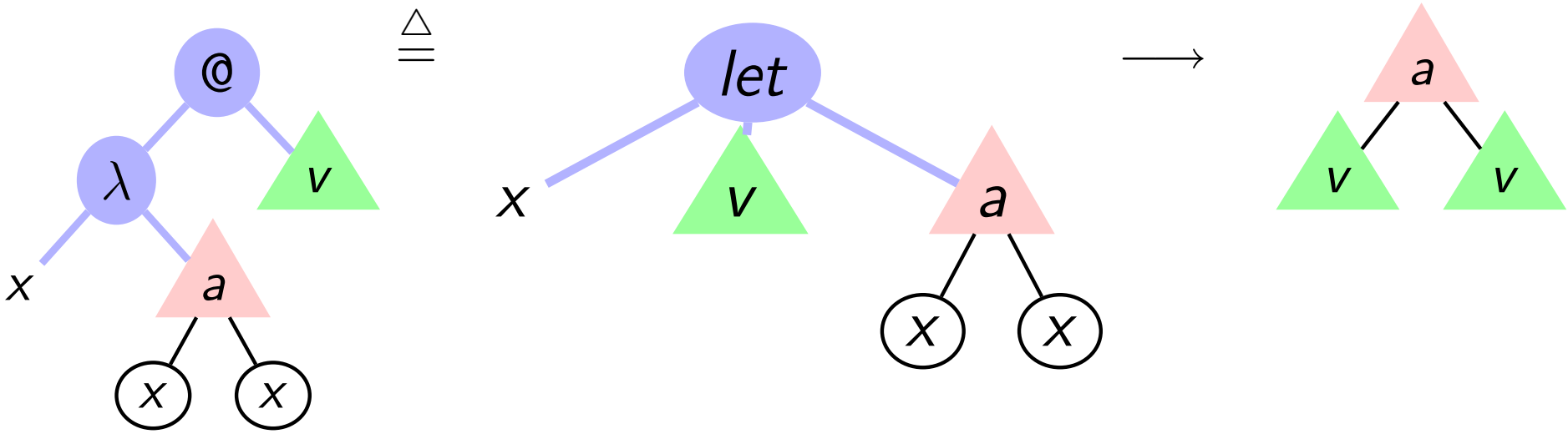
ML

Distingue les fonctions immédiatement appliquées



ML

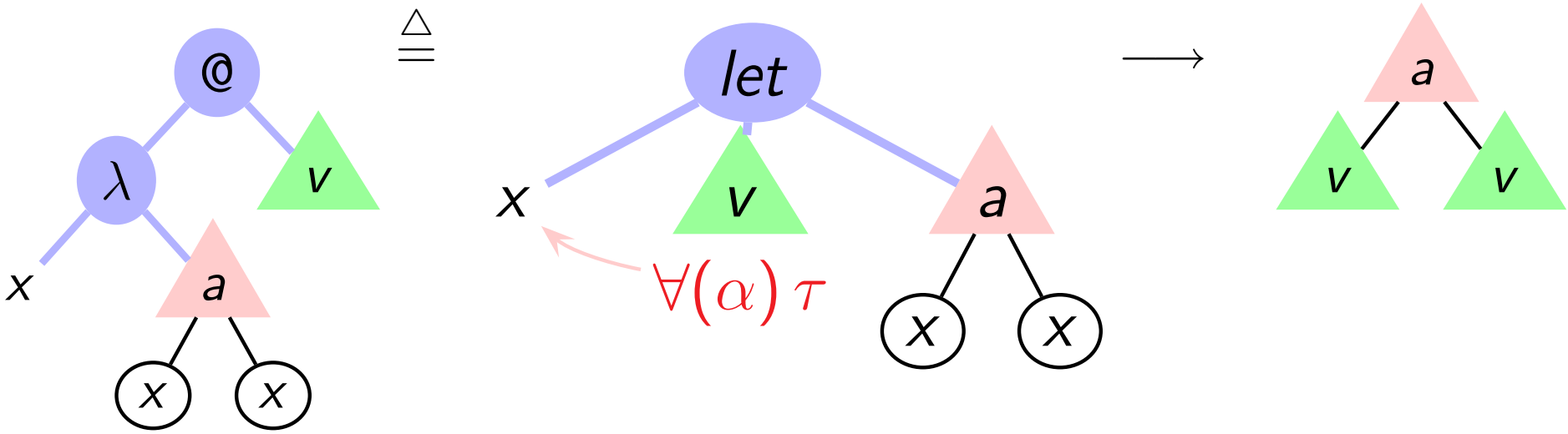
Distingue les fonctions immédiatement appliquées



– Ce sont les seules à avoir un argument de type polymorphe $\forall(\bar{\alpha})\,\tau$.

ML

Distingue les fonctions immédiatement appliquées

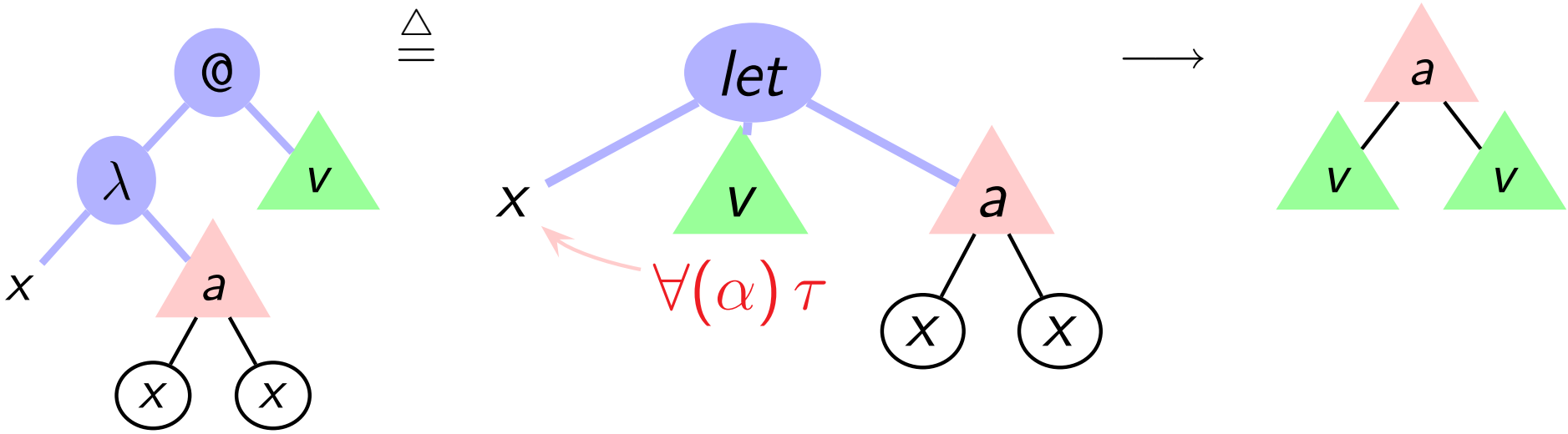


– Ce sont les seules à avoir un argument de type polymorphe $\forall(\bar{\alpha})\,\tau$.

– *Astuce* : on infère un type polymorphe pour v que l'utilise pour x dans a, mais on n'a pas à deviner le type polymorphe de x dans a.

ML

Distingue les fonctions immédiatement appliquées



– Ce sont les seules à avoir un argument de type polymorphe $\forall(\bar{\alpha})\,\tau$.

– *Astuce* : on infère un type polymorphe pour v que l'utilise pour x dans a, mais on n'a pas à deviner le type polymorphe de x dans a.

En particulier, pas de polymorphisme en profondeur : $\left(\forall(\bar{\alpha})\,\tau\right) \to \tau'$

ML

Intérêt de ML

Guère plus compliqué que les types simples, mais beaucoup plus expressif et plus modulaire.

- On peut écrire des fonctions polymorphes pour les utiliser à différents types.
- Le code peut être mieux partagé, il plus concis, plus facile à modifier.
- Le code est plus général, ce qui parfois le simplifie.

ML

Intérêt de ML

Guère plus compliqué que les types simples, mais beaucoup plus expressif et plus modulaire.

- On peut écrire des fonctions polymorphes pour les utiliser à différents types.
- Le code peut être mieux partagé, il plus concis, plus facile à modifier.
- Le code est plus général, ce qui parfois le simplifie.

Limitations de ML

Les types vraiment polymorphes sont (de plus en plus) incontournables :
– pour exprimer des invariants plus fins.
– pour factoriser les constructions du langage.

Inférence de types

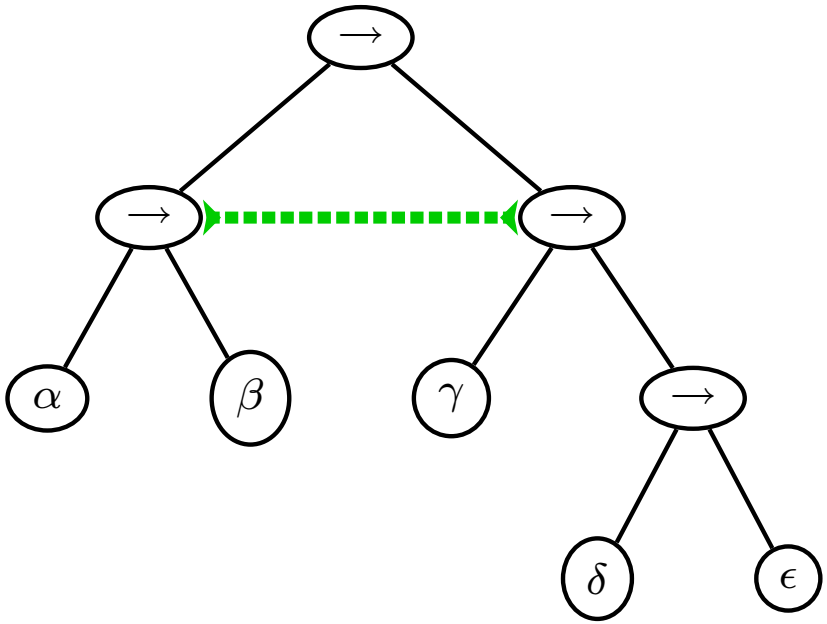
Repose sur l'unification

L'inférence de type est basée sur la résolution de problèmes d'unification de premier ordre, c'est-à-dire la résolution d'équations sur des structures de terme avec variables.

Les problèmes d'unification peuvent être représentés directement sur la structure de type, en notant les équations par des arcs et leurs solutions par le partage des noeuds.

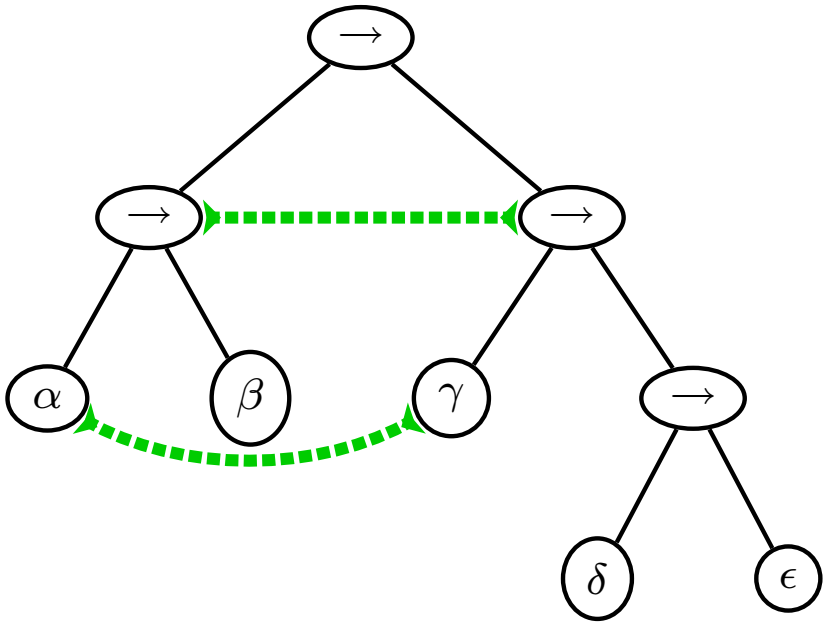
Inférence de types

Exemple d'unification



Inférence de types

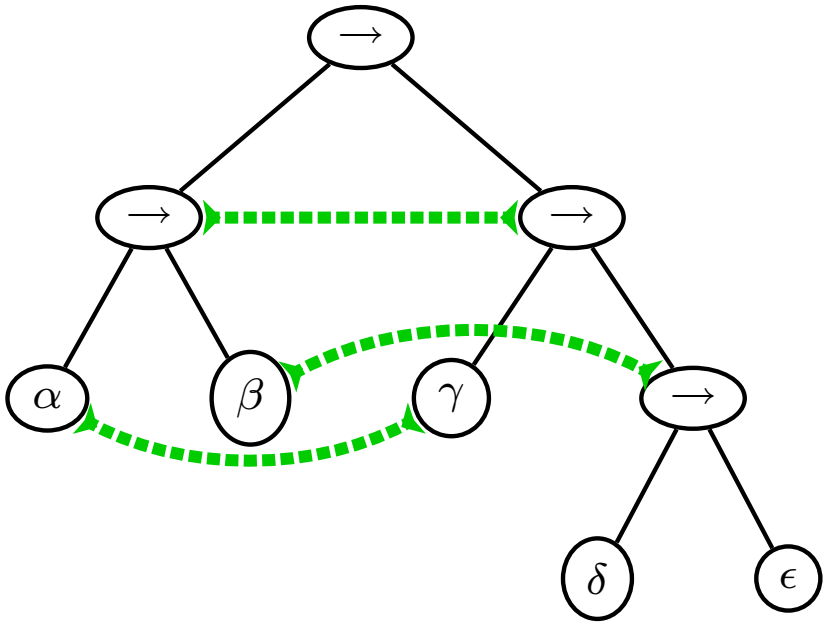
Exemple d'unification



Congruence closure

Inférence de types

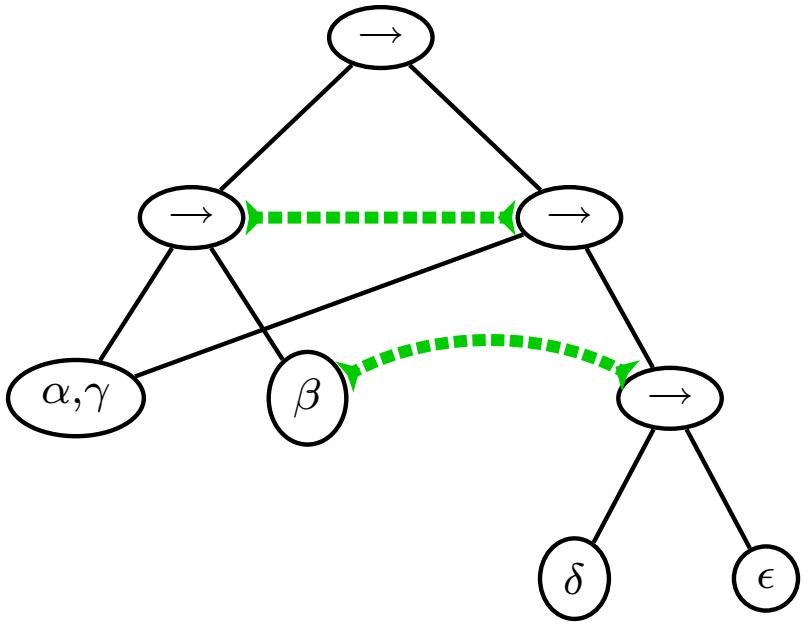
Exemple d'unification



Congruence closure

Inférence de types

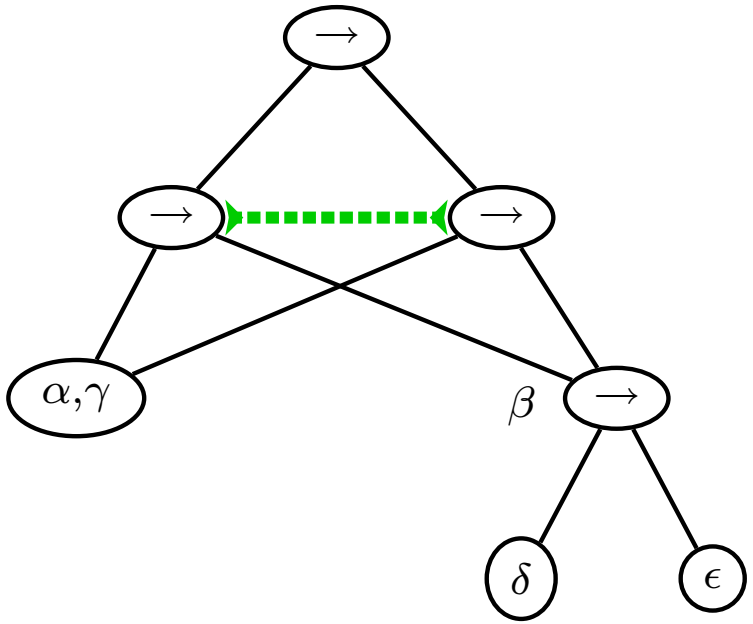
Exemple d'unification



Merging

Inférence de types

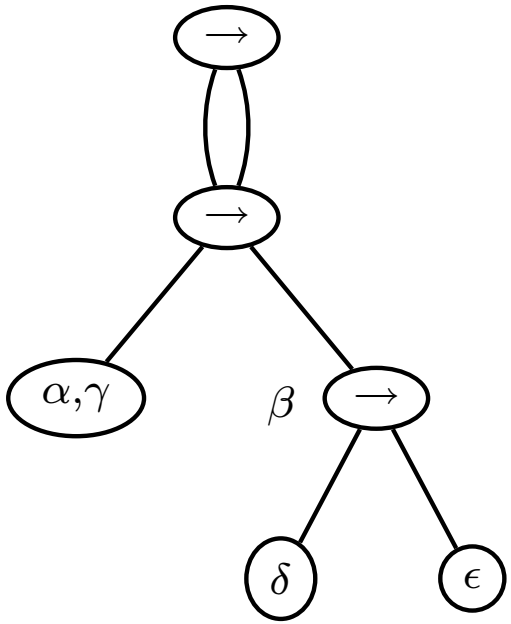
Exemple d'unification



Grafting

Inférence de types

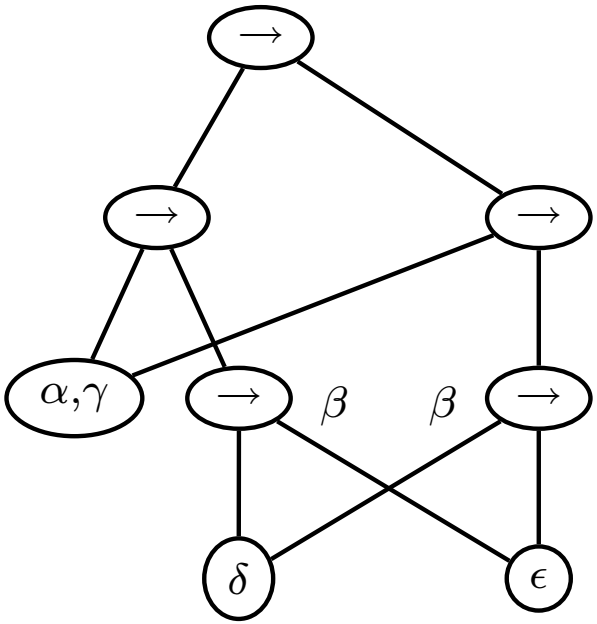
Exemple d'unification



- L'unification construit un dag, en identifiant des variables ou des nœuds internes.

Inférence de types

Exemple d'unification



- L'unification construit un dag, en identifiant des variables ou des nœuds internes.
- Ce dag peut être relu comme un terme en départageant les noeuds internes.

Inférence pour le λ-calculus simplement typé

Exemple :

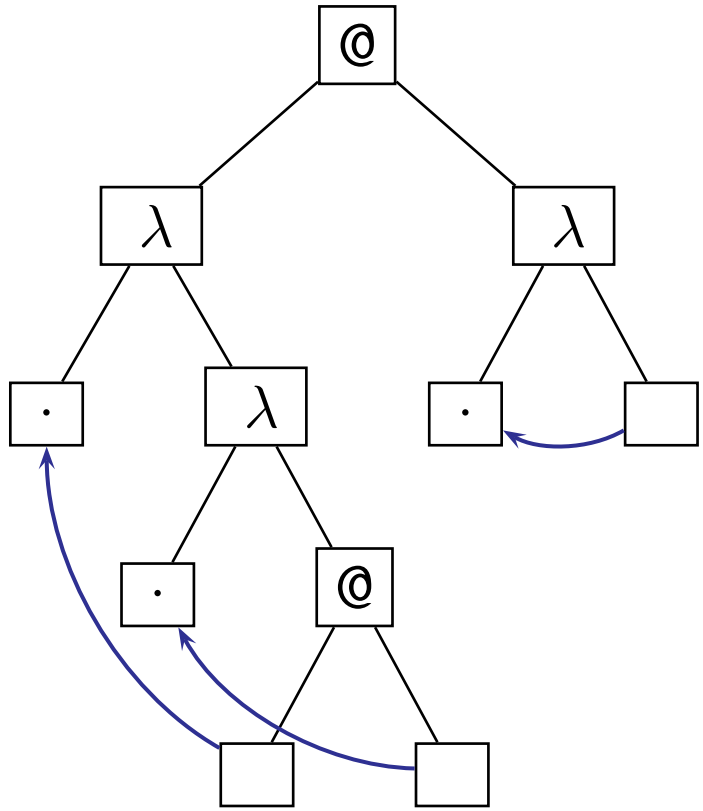
$$\Big(\lambda(f)\,\lambda(x)\,f\ x\Big)\ \big(\lambda(y)\,y\big)$$

Inférence pour le λ-calculus simplement typé

Graphiquement : son λ-terme

Exemple :

$$\Big(\lambda(f)\,\lambda(x)\,f\ x\Big)\ \big(\lambda(y)\,y\big)$$

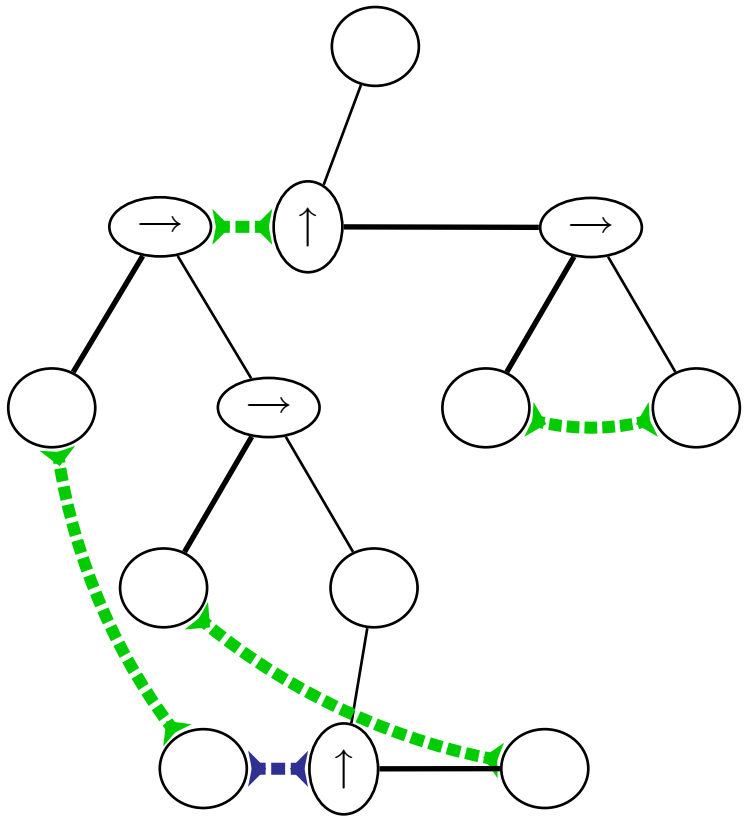


Inférence pour le λ-calculus simplement typé

Graphiquement : sa constrainte

Exemple :

$$\Big(\lambda(f)\,\lambda(x)\,f\ x\Big)\ \big(\lambda(y)\,y\big)$$

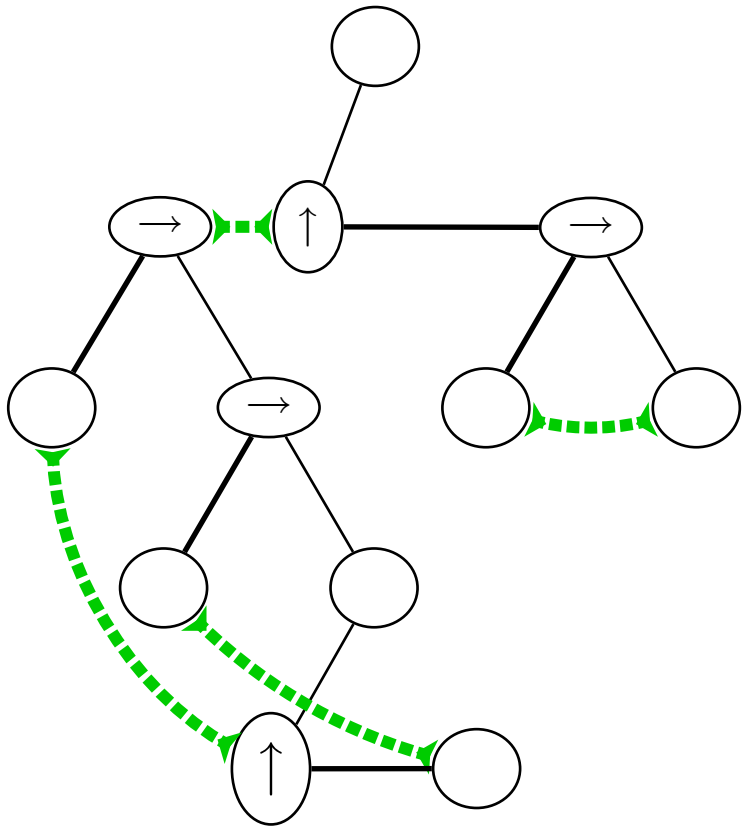


Inférence pour le λ-calculus simplement typé

Graphiquement : sa constrainte

Exemple :

$$\Big(\lambda(f)\,\lambda(x)\,f\ x\Big)\ \big(\lambda(y)\,y\big)$$

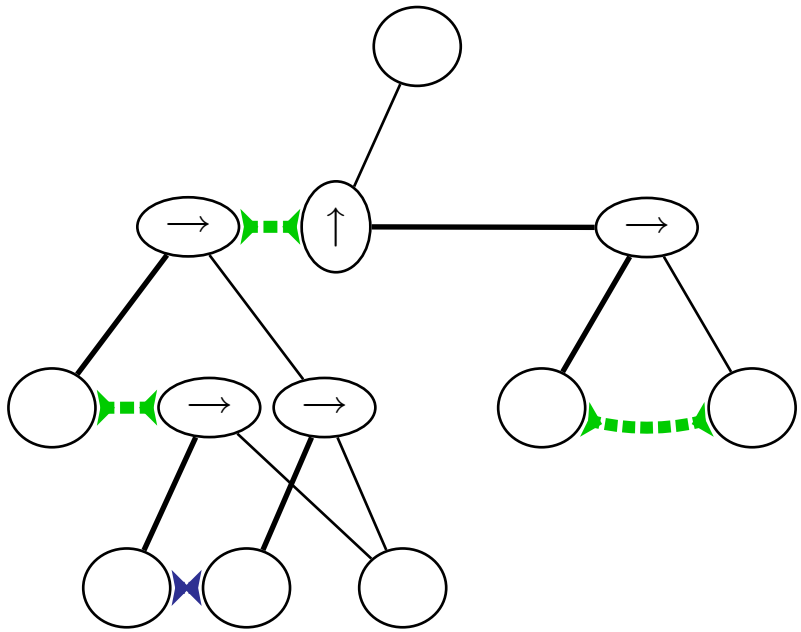


Inférence pour le λ-calculus simplement typé

Graphiquement : sa constrainte

Exemple :

$$\Big(\lambda(f)\,\lambda(x)\,f\ x\Big)\ \big(\lambda(y)\,y\big)$$

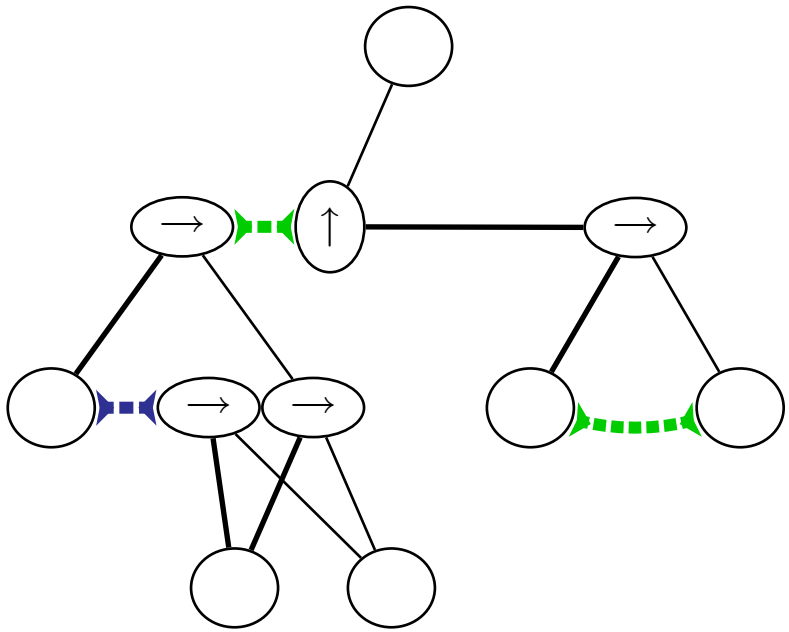


Inférence pour le λ-calculus simplement typé

Graphiquement : sa constrainte

Exemple :

$$\Big(\lambda(f)\,\lambda(x)\,f\ x\Big)\ \big(\lambda(y)\,y\big)$$

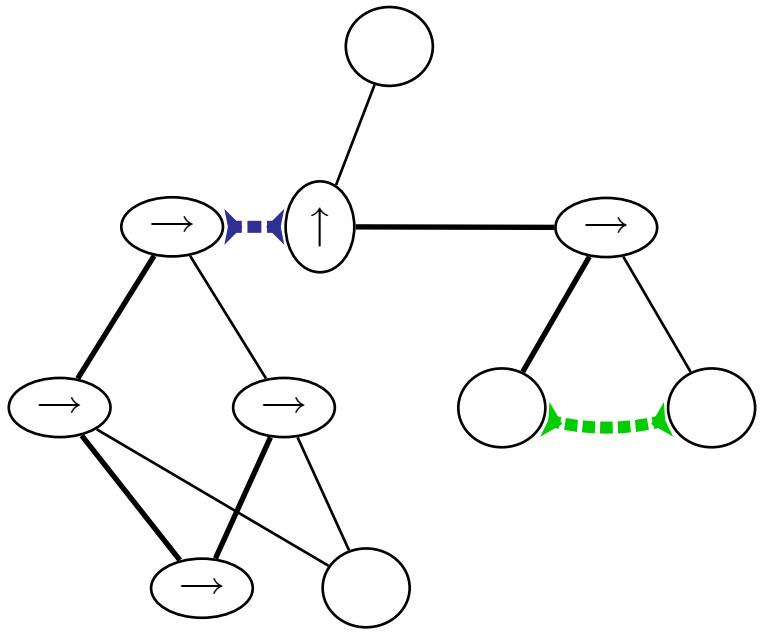


Inférence pour le λ-calculus simplement typé

Graphiquement : sa constrainte

Exemple :

$$\Big(\lambda(f)\,\lambda(x)\,f\ x\Big)\ \big(\lambda(y)\,y\big)$$

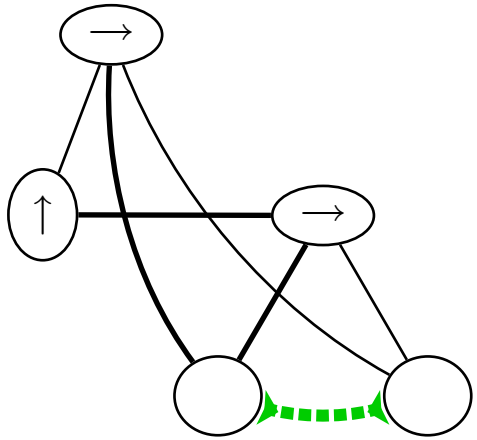


Inférence pour le λ-calculus simplement typé

Graphiquement : sa constrainte

Exemple :

$$\Big(\lambda(f)\,\lambda(x)\,f\ x\Big)\ \big(\lambda(y)\,y\big)$$

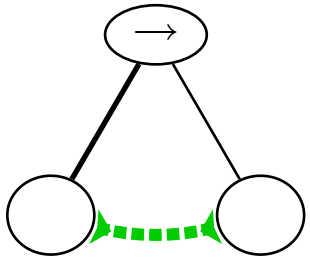


Inférence pour le λ-calculus simplement typé

Graphiquement : sa forme résolue

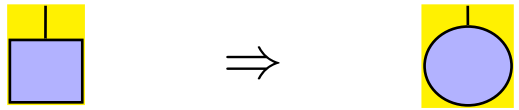
Exemple :

$$\Big(\lambda(f)\,\lambda(x)\,f\ x\Big)\ \big(\lambda(y)\,y\big)$$

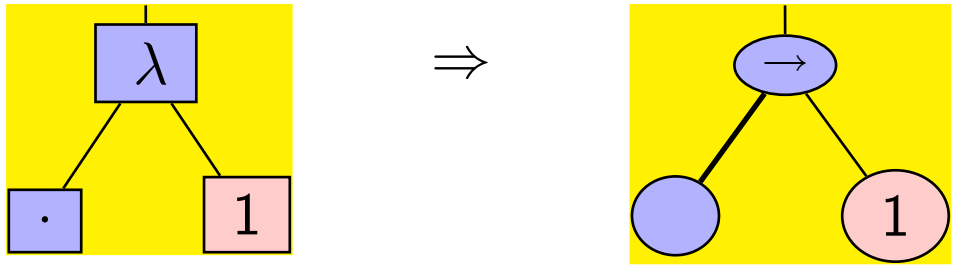


Génération de contraintes (détails)

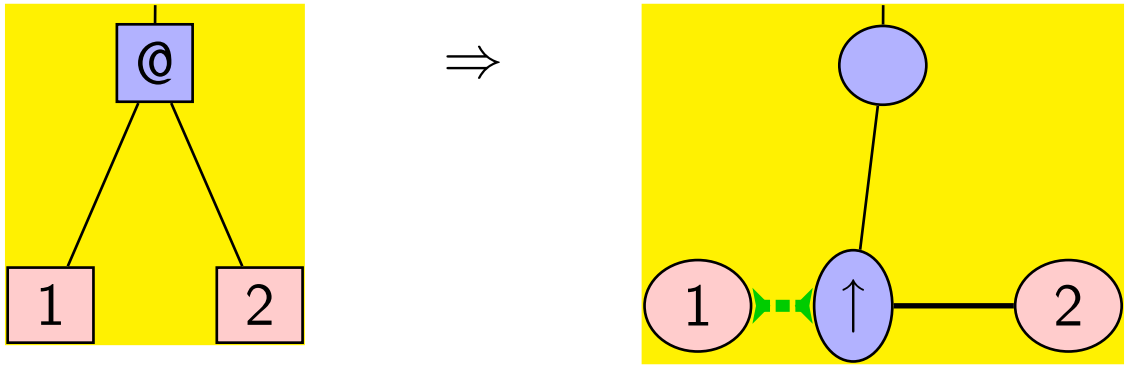
Variables



Functions

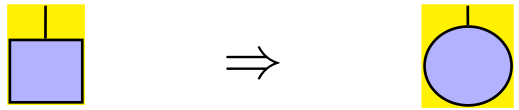


Applications

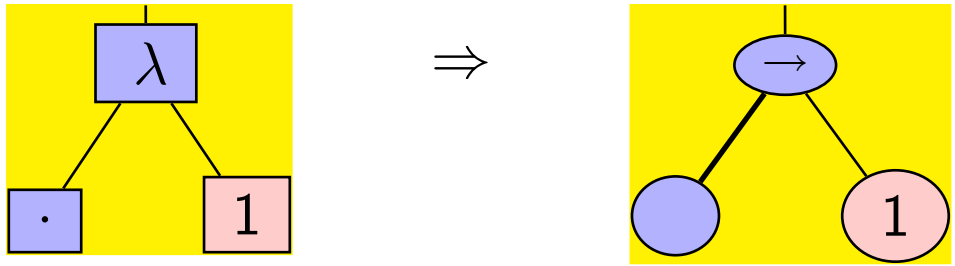


Génération de contraintes (détails)

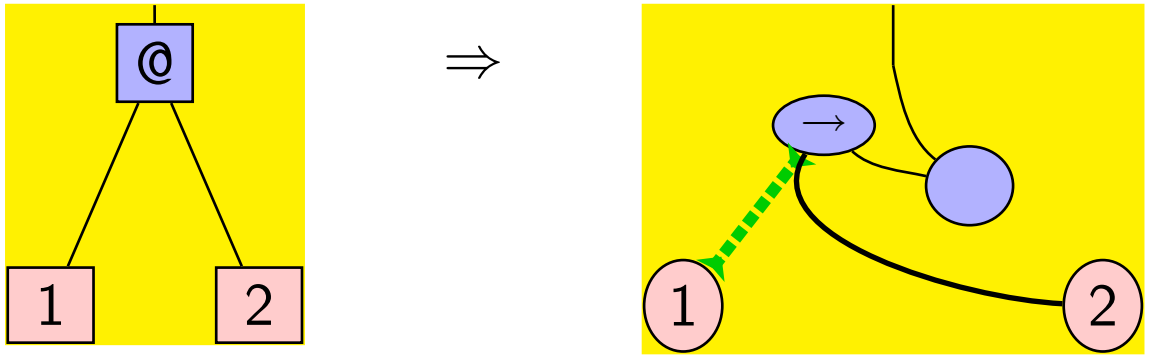
Variables



Functions

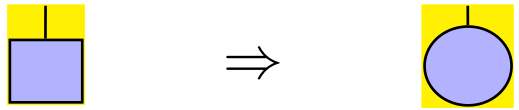


Applications

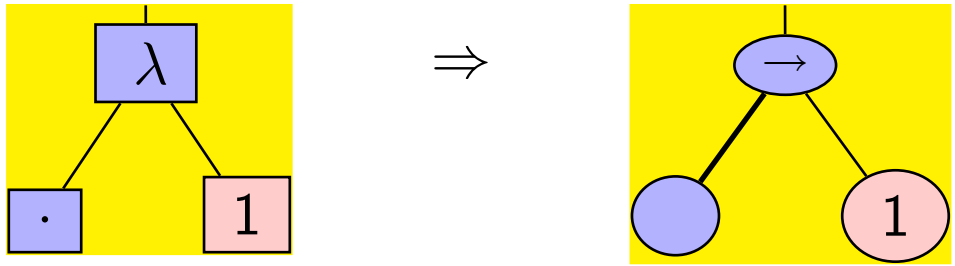


Génération de contraintes (détails)

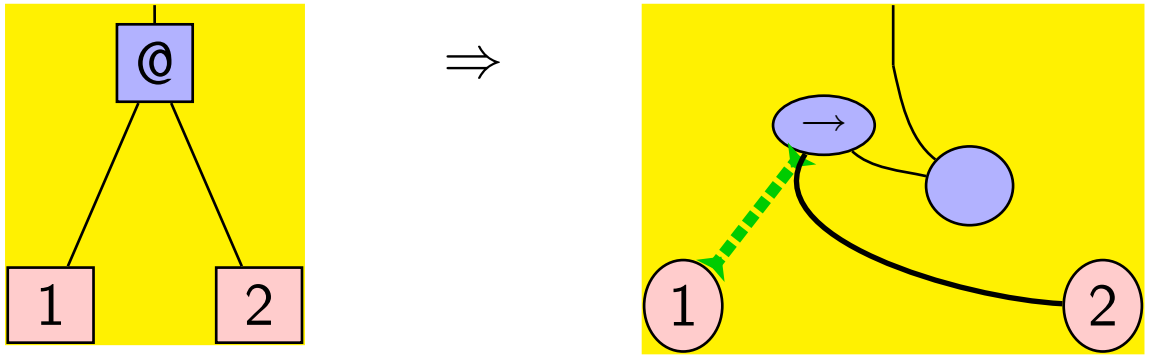
Variables



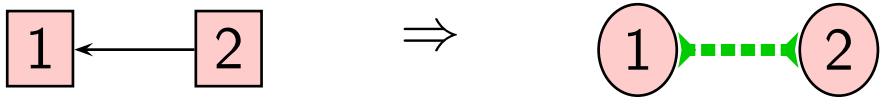
Functions



Applications



Bindings



Inférence avec contraintes

Intérêt

Séparer la génération des contraintes de leur résolution permet

- De ne pas préciser la stratégie de résolution : les algorithmes W et M pour ML ne sont que deux instances du même algorithme pour des stratégies différentes.
- D'étendre facilement le langage : on génére des contraintes pour les nouveaux nœuds, sans modifier le langage de contraintes.

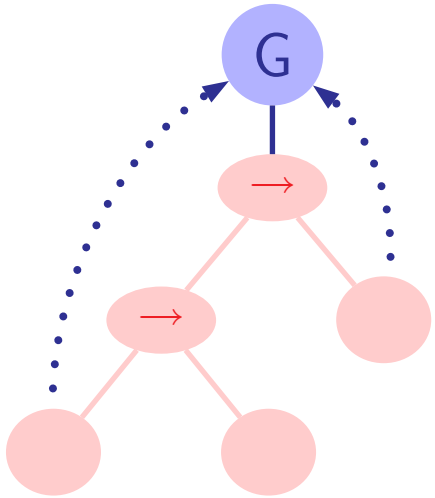
Peut-on étendre le schéma précédent à ML ?

- La question est souvent éludée dans les libres, qui présente un algorithme particulier (W) où la résolution des contraintes est entrelacée avec leur génération.
- Pourtant la solution précédente s'étend bien à ML, si on la regarde graphiquement...

Type inference for let-bindings

On introduit des nœuds G (points de généralisation)

pour représenter les schémas de type et les distinguer des types



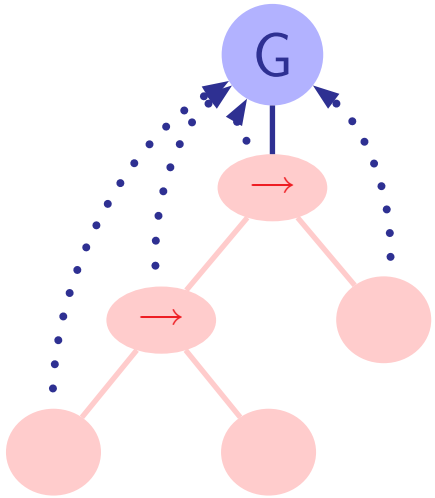
$$\forall(\alpha\beta)\ (\alpha \to \beta) \to \gamma$$

Les variables generalisées sont dessinées par des arcs allant sur un nœud G.

Type inference for let-bindings

On introduit des nœuds G (points de généralisation)

pour représenter les schémas de type et les distinguer des types



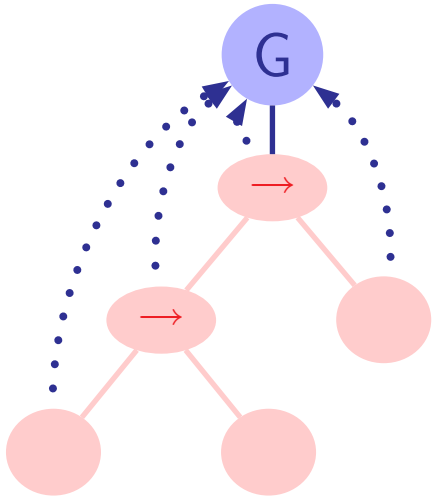
$$\forall(\alpha\beta)\ (\alpha \to \beta) \to \gamma$$

Les variables generalisées sont dessinées par des arcs allant sur un nœud G.
Pour simplifier, les noeuds internes sont aussi rattachés à un noeud G.

Type inference for let-bindings

On introduit des nœuds G (points de généralisation)

pour représenter les schémas de type et les distinguer des types



$$\forall(\alpha\beta)\ (\alpha \to \beta) \to \gamma$$

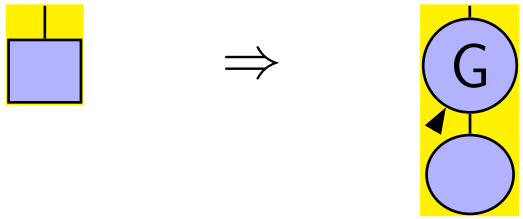
Les variables generalisées sont dessinées par des arcs allant sur un nœud G.

Génération de contraintes

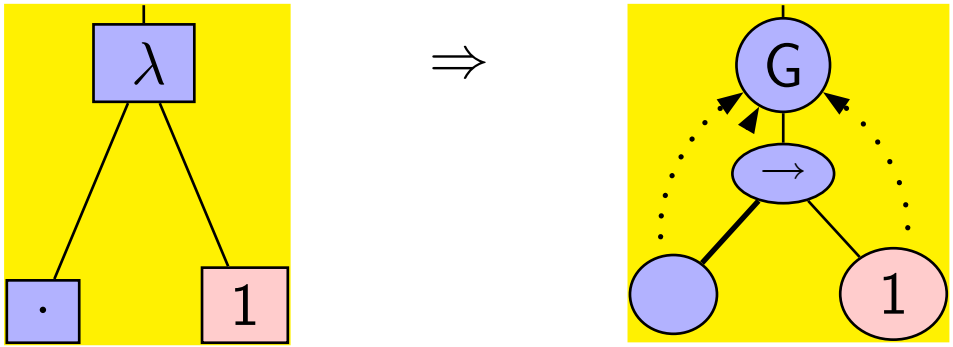
Les expressions représentes des nœuds G, *i.e.* des schémas.

Génération de constraintes pour ML (revue)

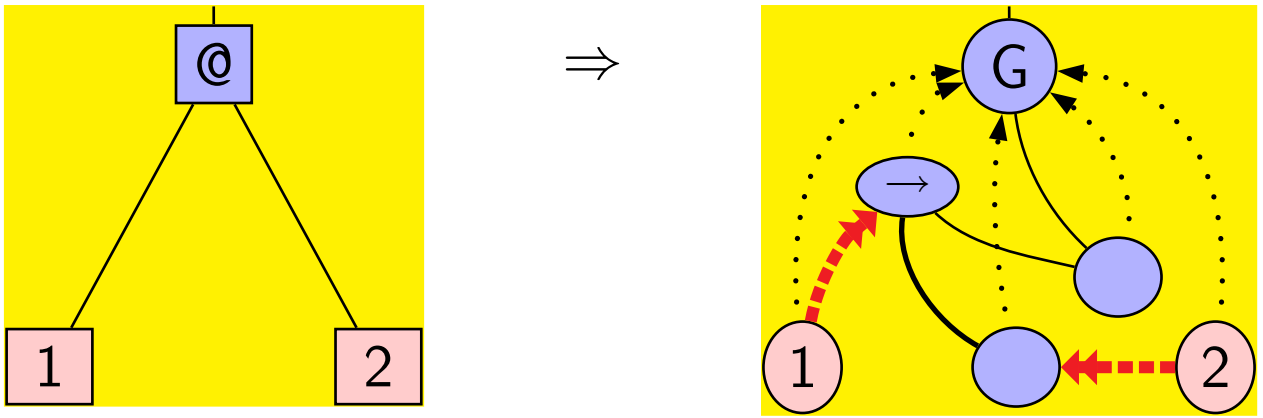
Variables



Functions

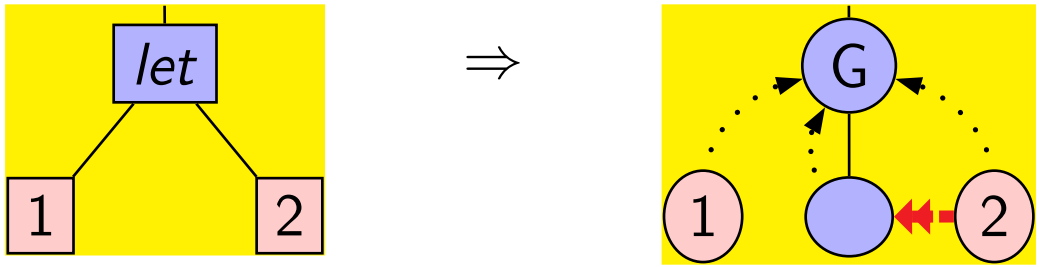


Applications

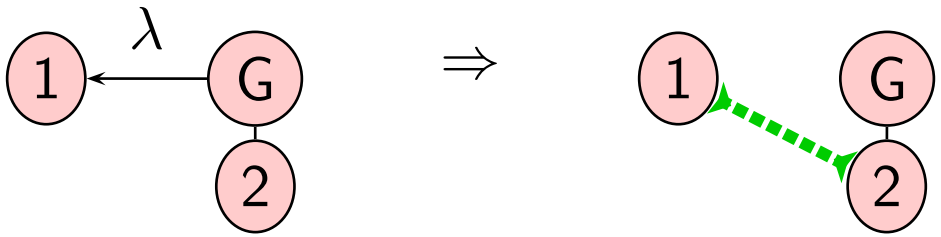


Génération de constraintes pour ML (revue)

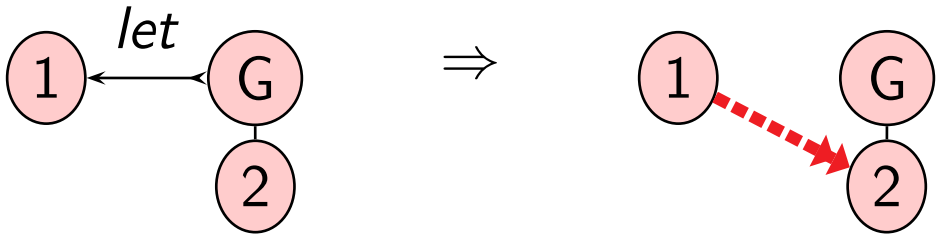
Let-bindings



λ-bindings



let-bindings



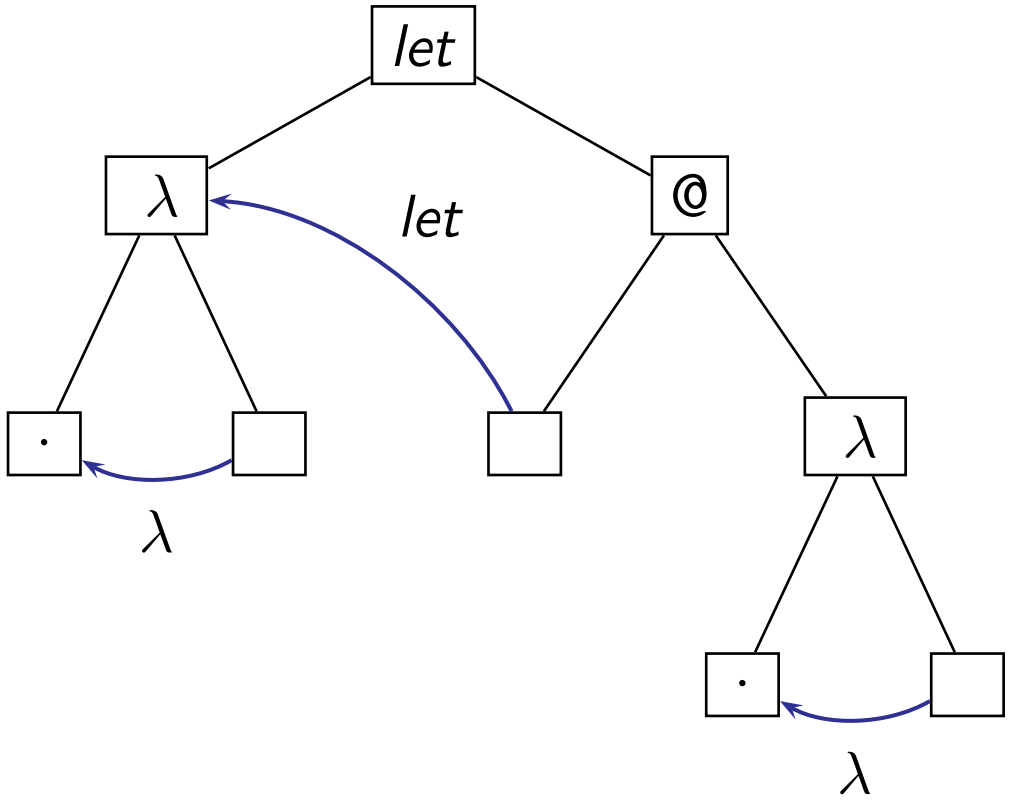
Génération de contraintes pour ML

Exemple :

$$\mathit{let}\ g = \lambda(x)\, x\ \mathit{in}$$
$$g\ (\lambda(y)\, y)$$

Graphiquement
(à droite) :

le λ-terme



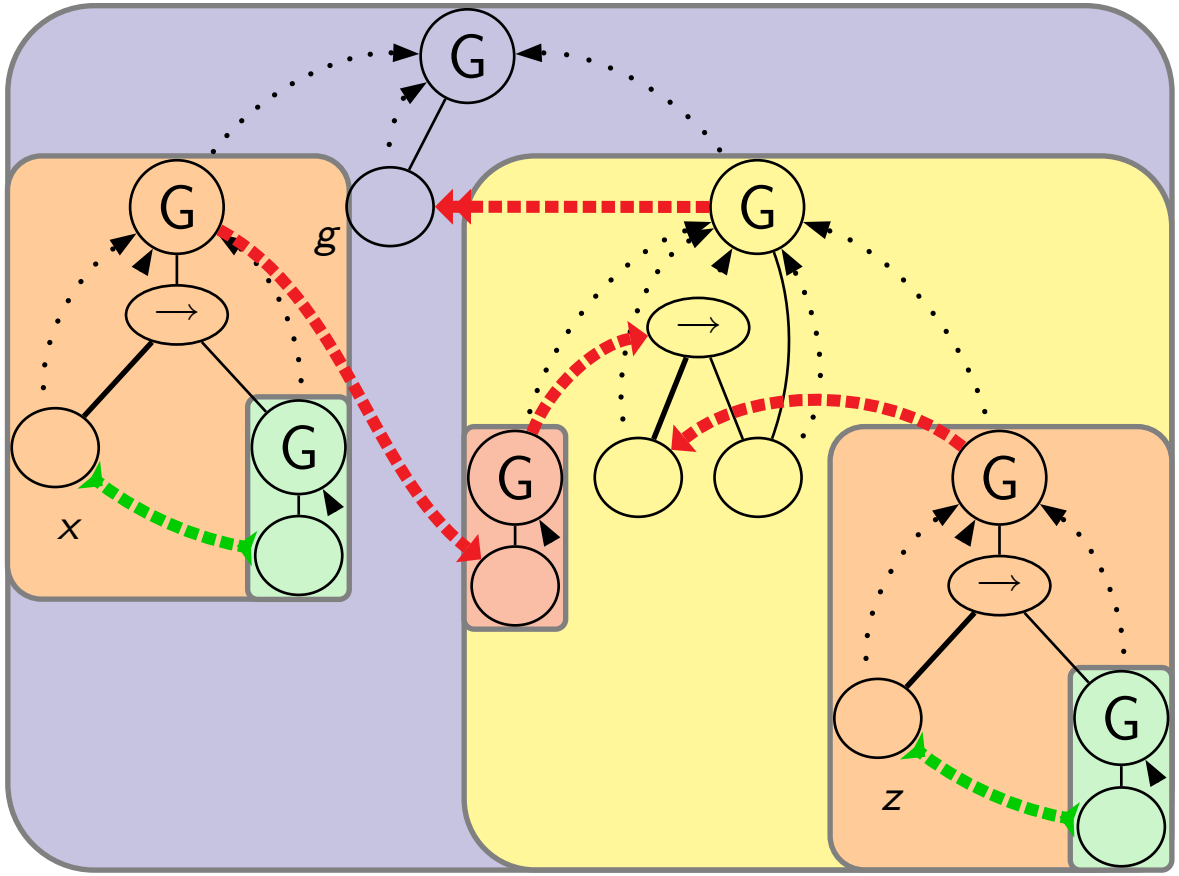
Génération de contraintes pour ML

Exemple :

$$\textit{let } g = \lambda(x)\, x \textit{ in } g\ (\lambda(y)\, y)$$

Graphiquement (à droite) :

sa contrainte de type



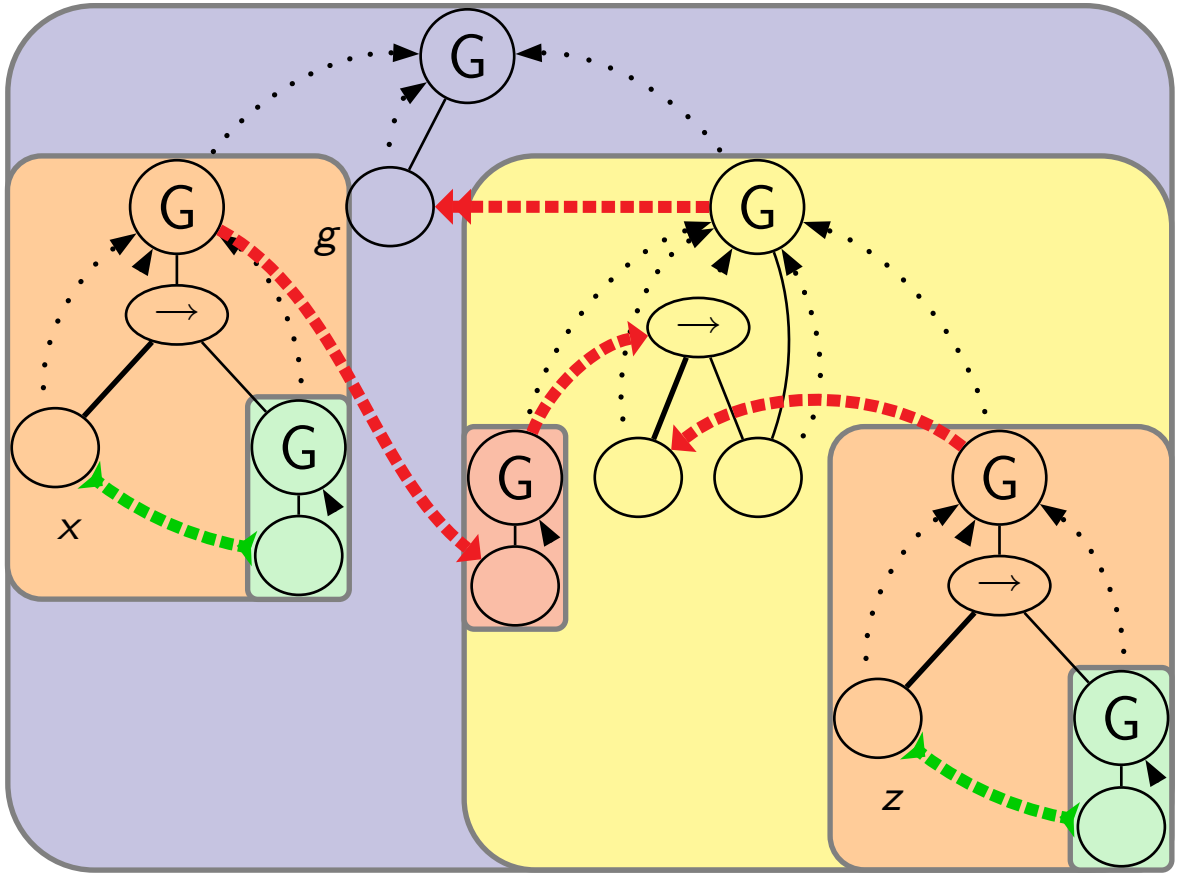
Génération de contraintes pour ML

Exemple :

$$let\ g = \lambda(x)\, x\ in$$
$$g\ (\lambda(y)\, y)$$

Graphiquement (à droite) :

sa contrainte de type



Stratégie

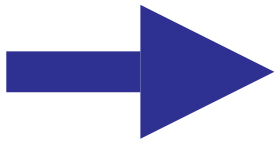
- Les fléches rouges ordonnent les boîtes (pas de cycle)
- Simplifier les boîtes dans l'ordre (par unification)
- Prendre des instances des formes résolues le long des flèches rouges.

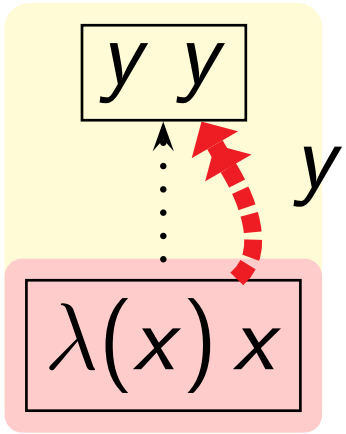
Inférence de type pour ML (exemple)

$$\text{let } y = \lambda(x)\, x \text{ in } y\ y$$

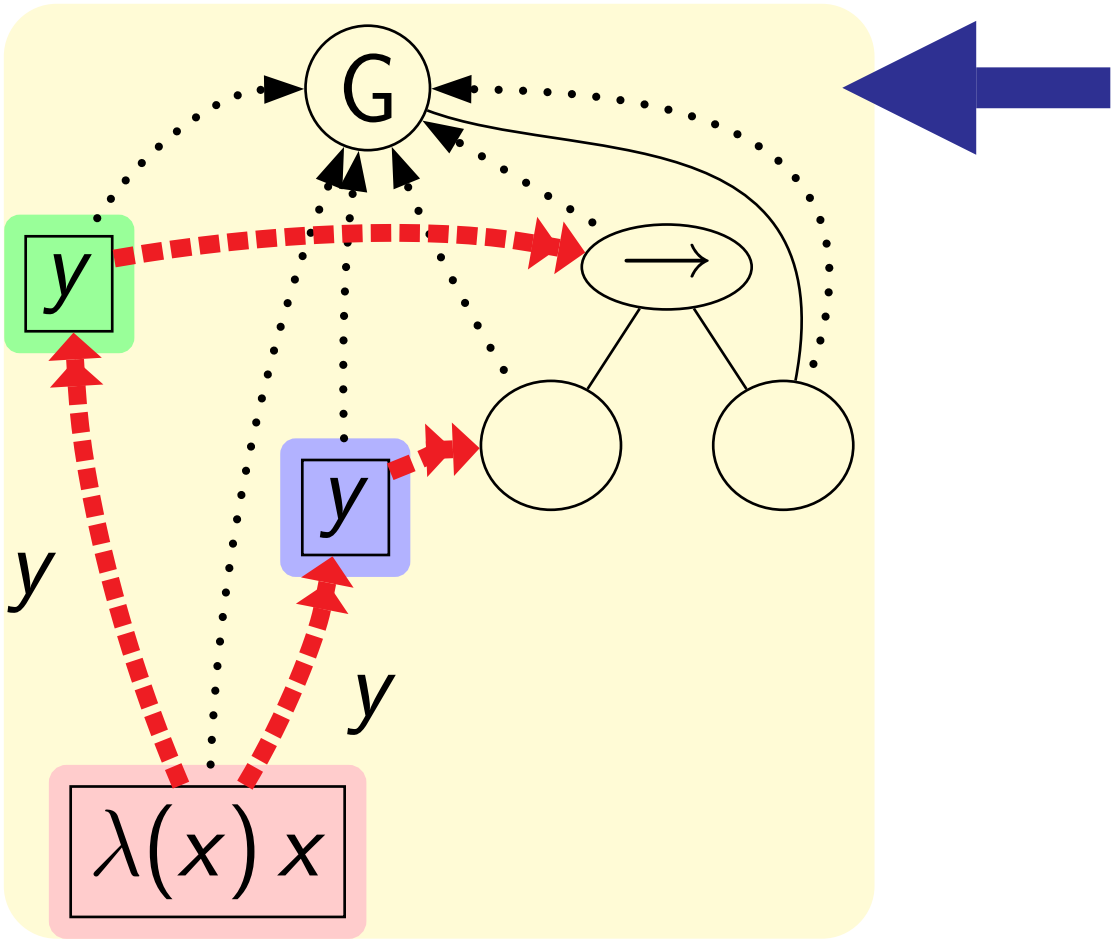
Inférence de type pour ML (exemple)

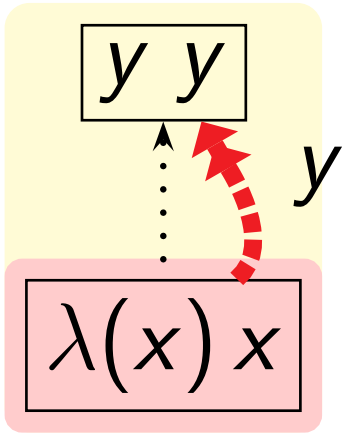
$$\text{let } y = \lambda(x)\, x \text{ in } y\ y$$



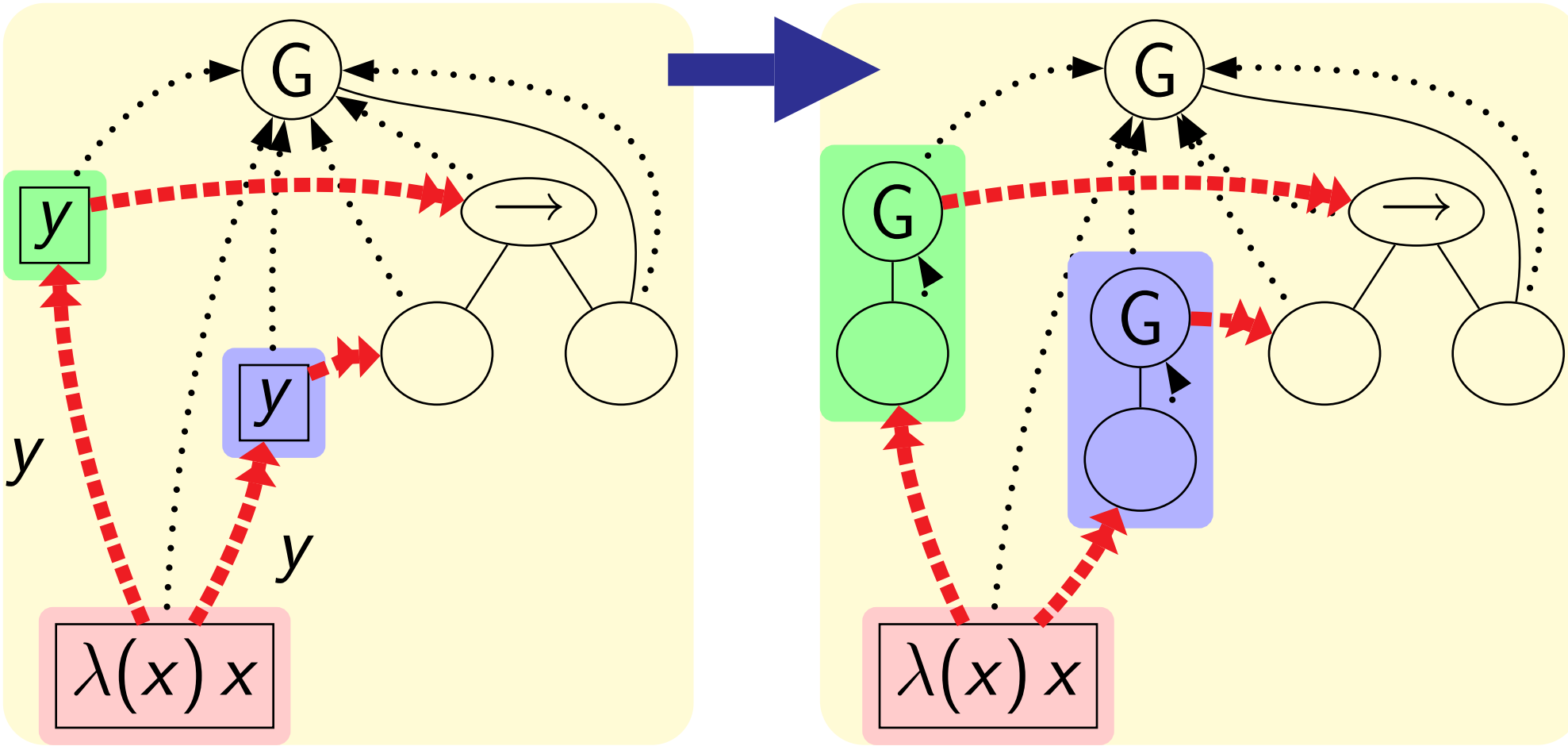


Inférence de type pour ML (exemple)

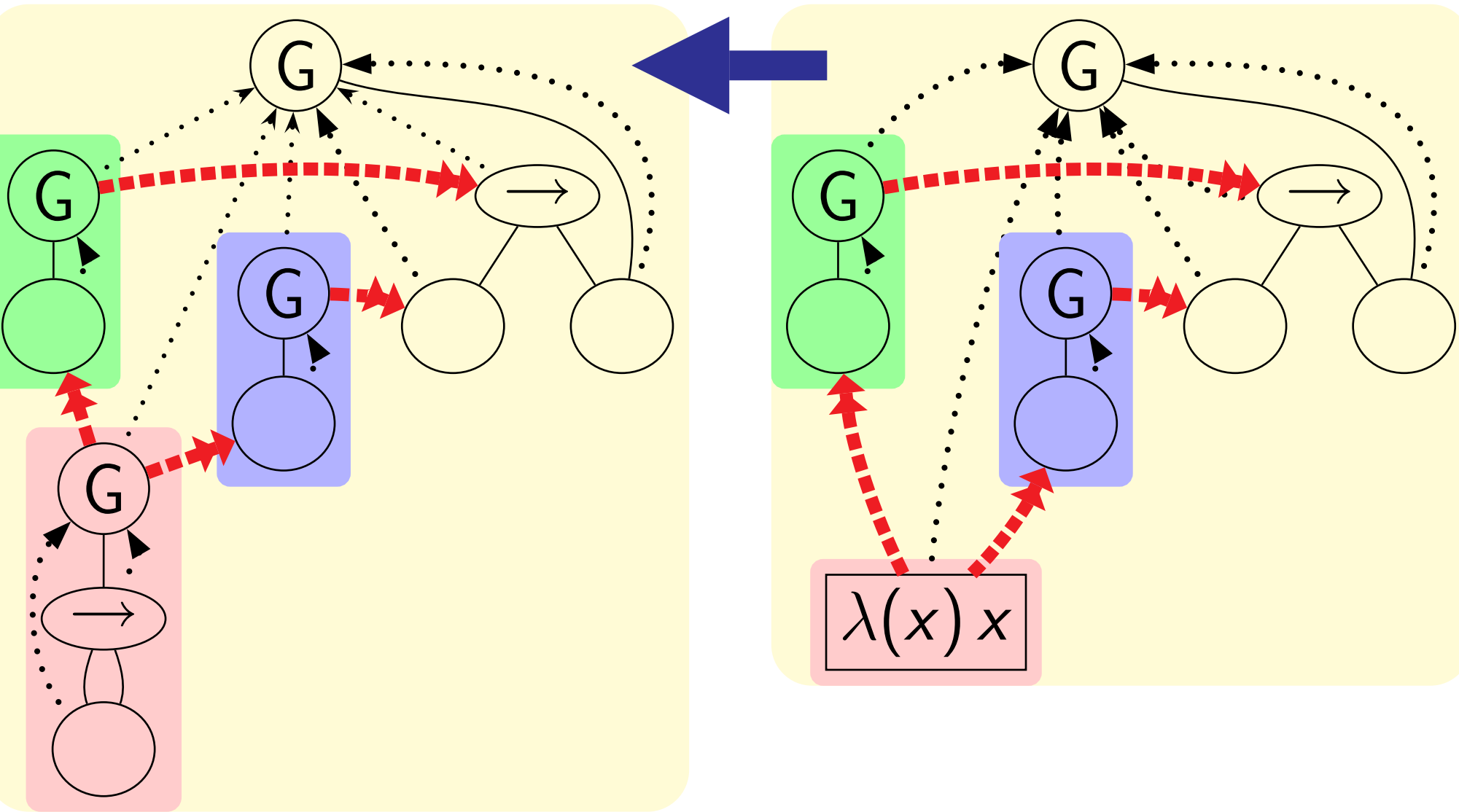




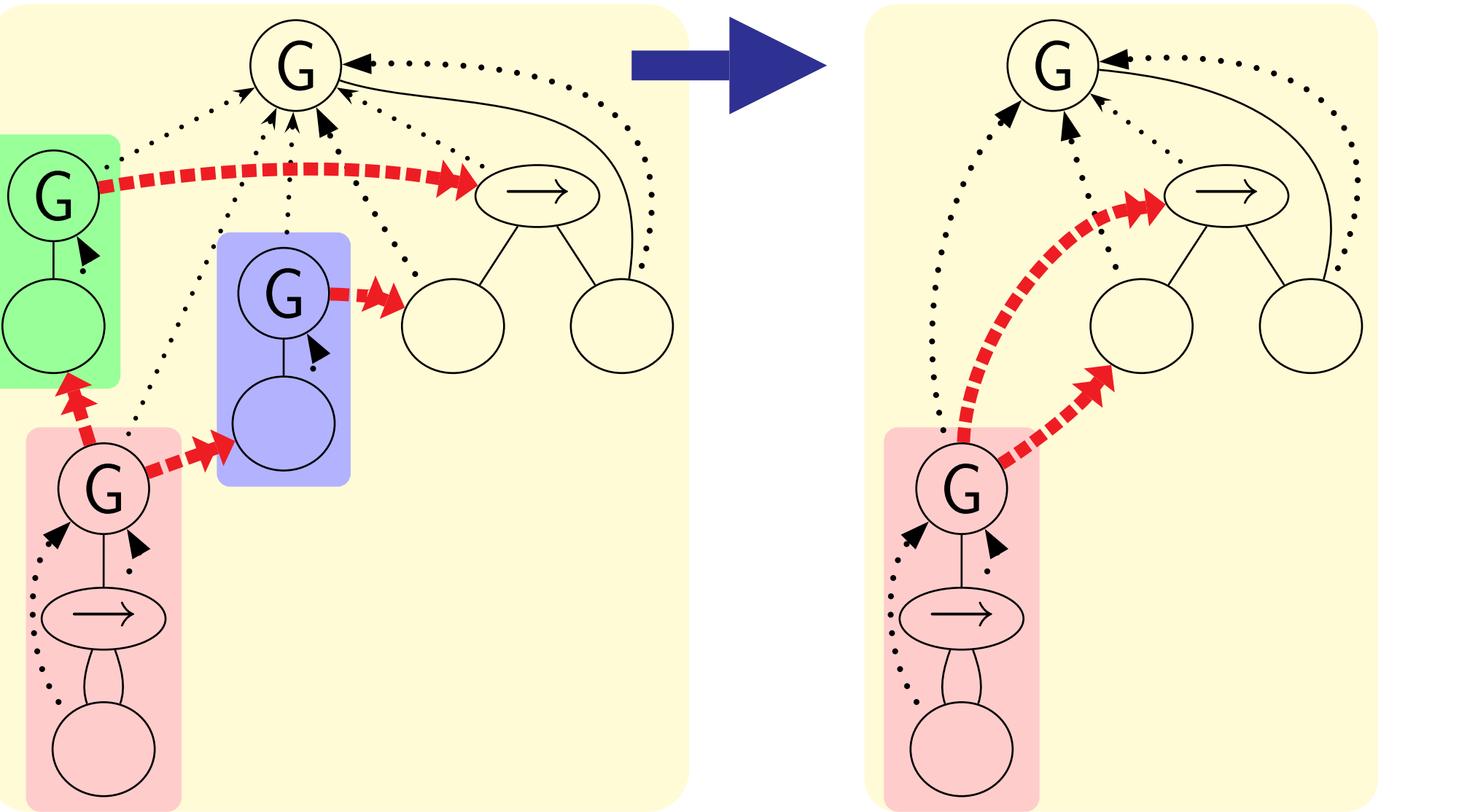
Inférence de type pour ML (exemple)



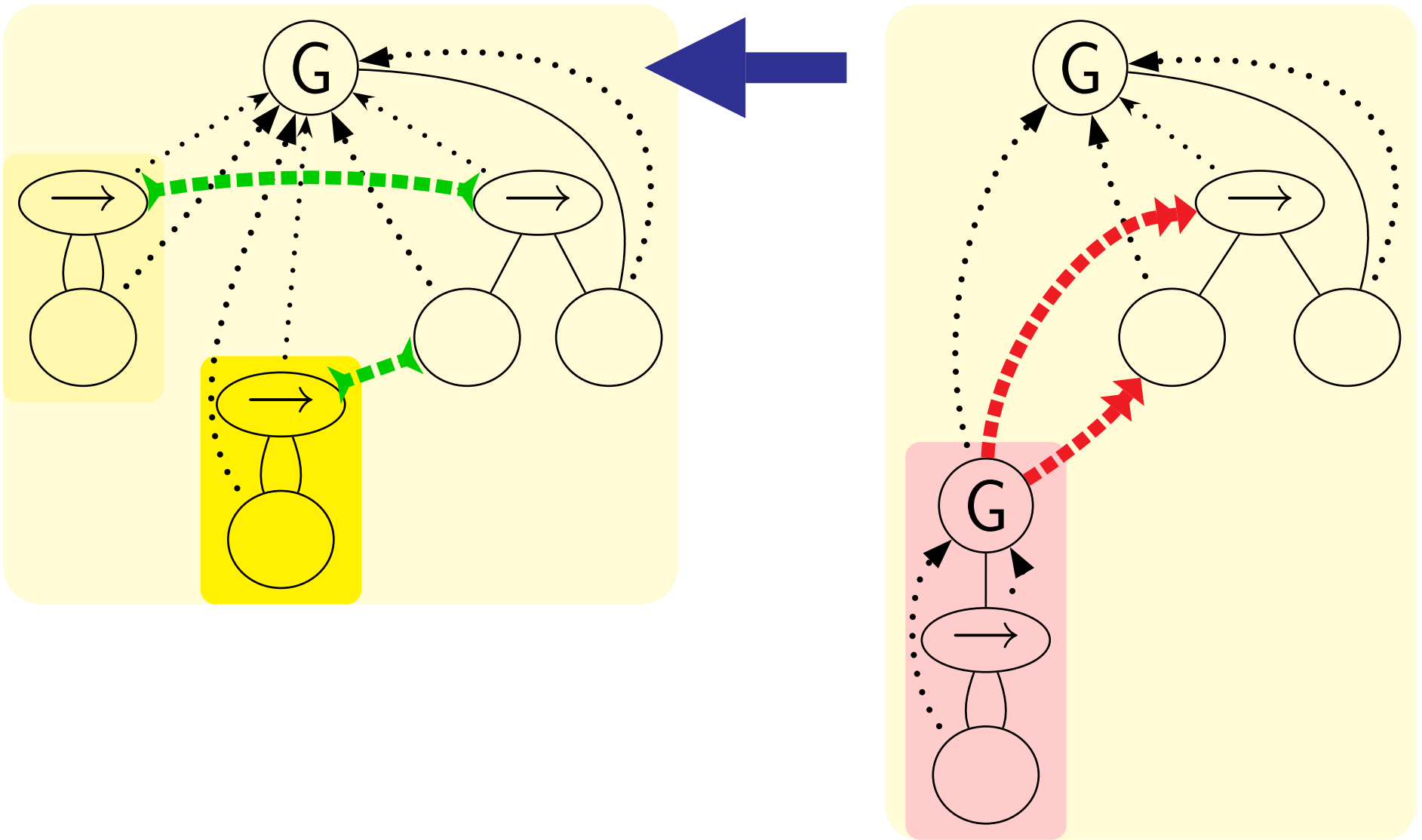
Inférence de type pour ML (exemple)



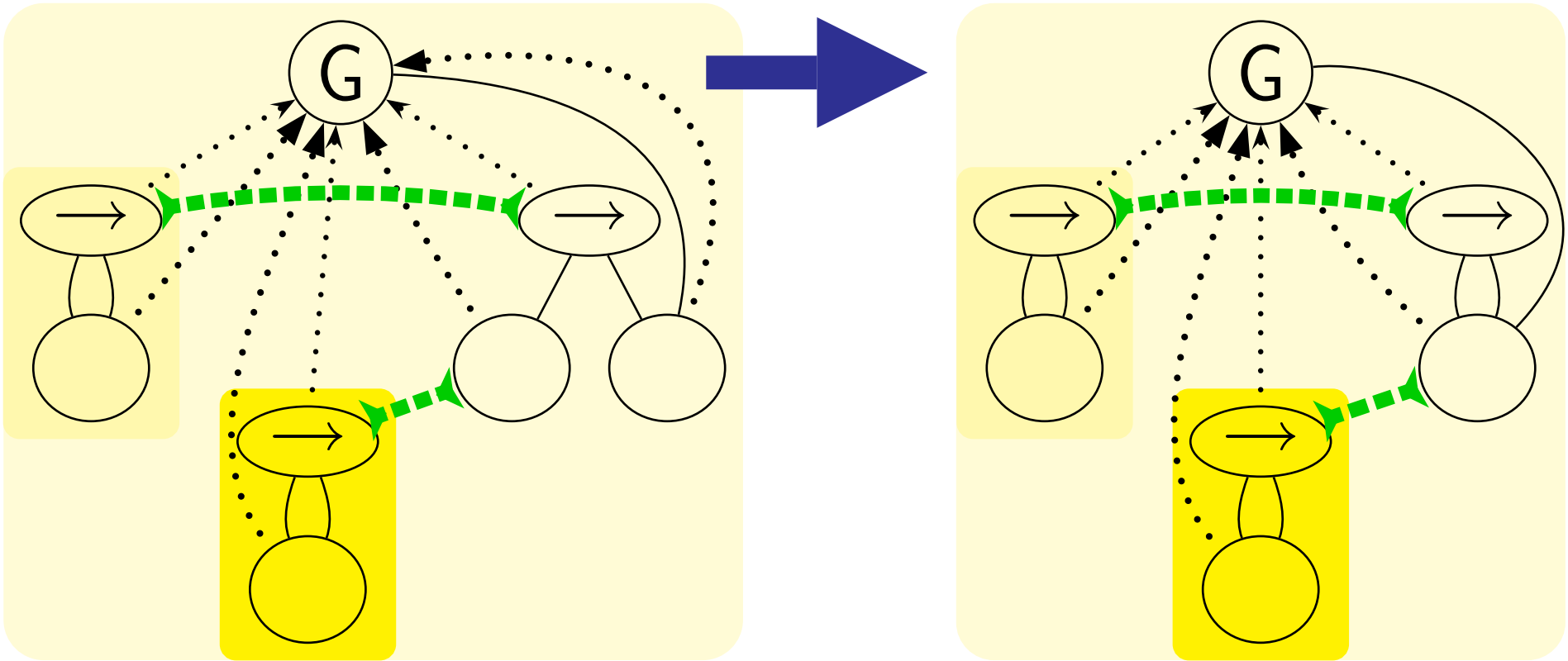
Inférence de type pour ML (exemple)



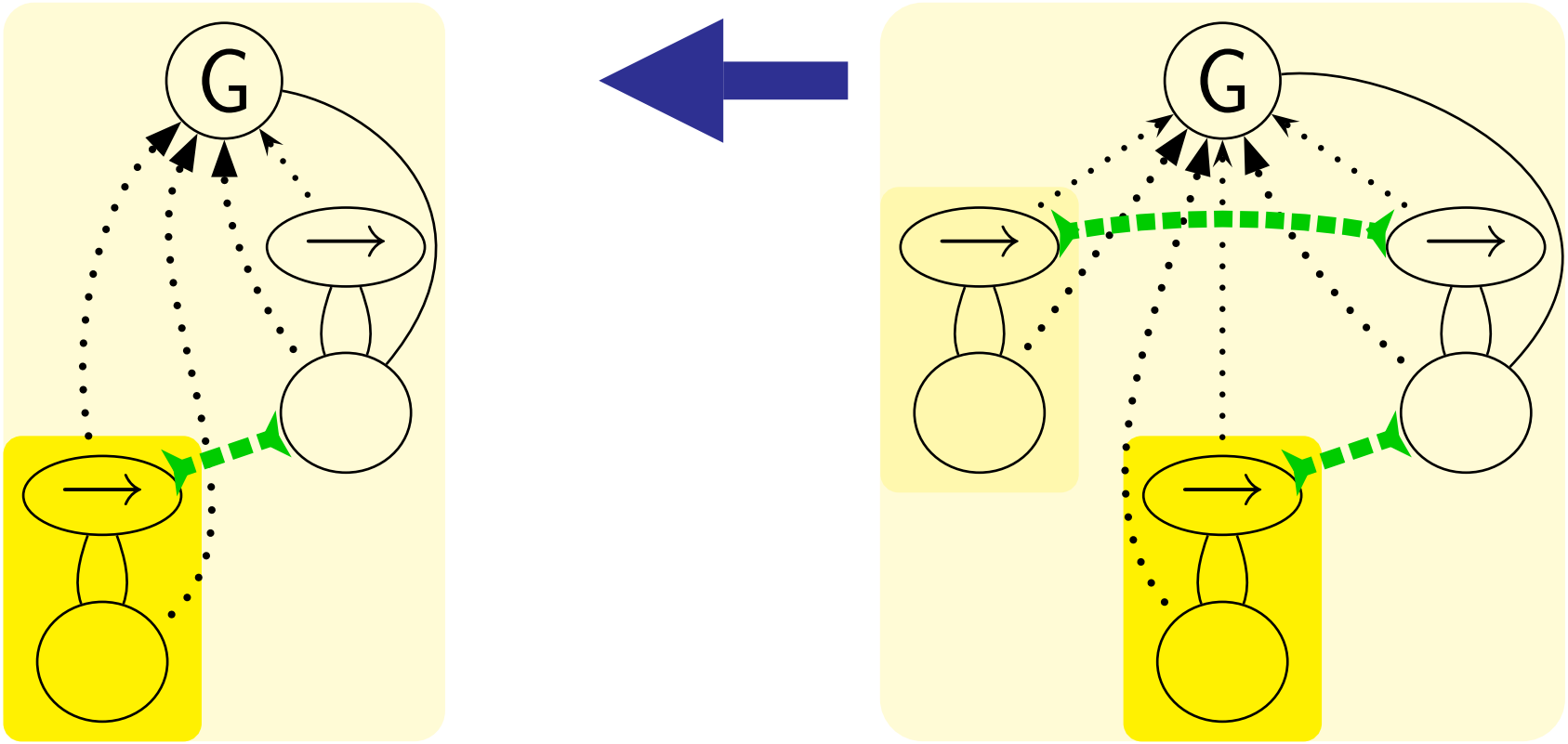
Inférence de type pour ML (exemple)



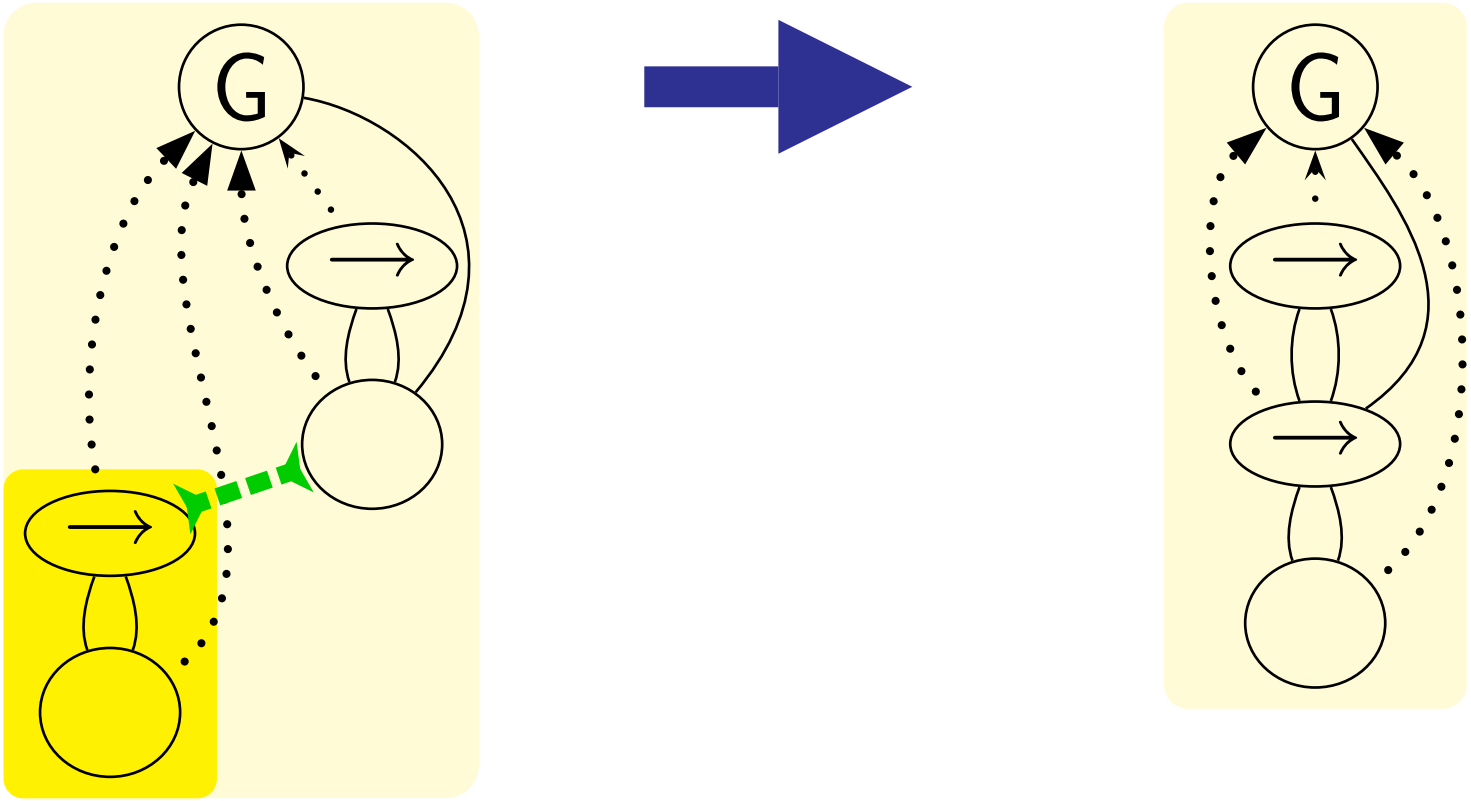
Inférence de type pour ML (exemple)



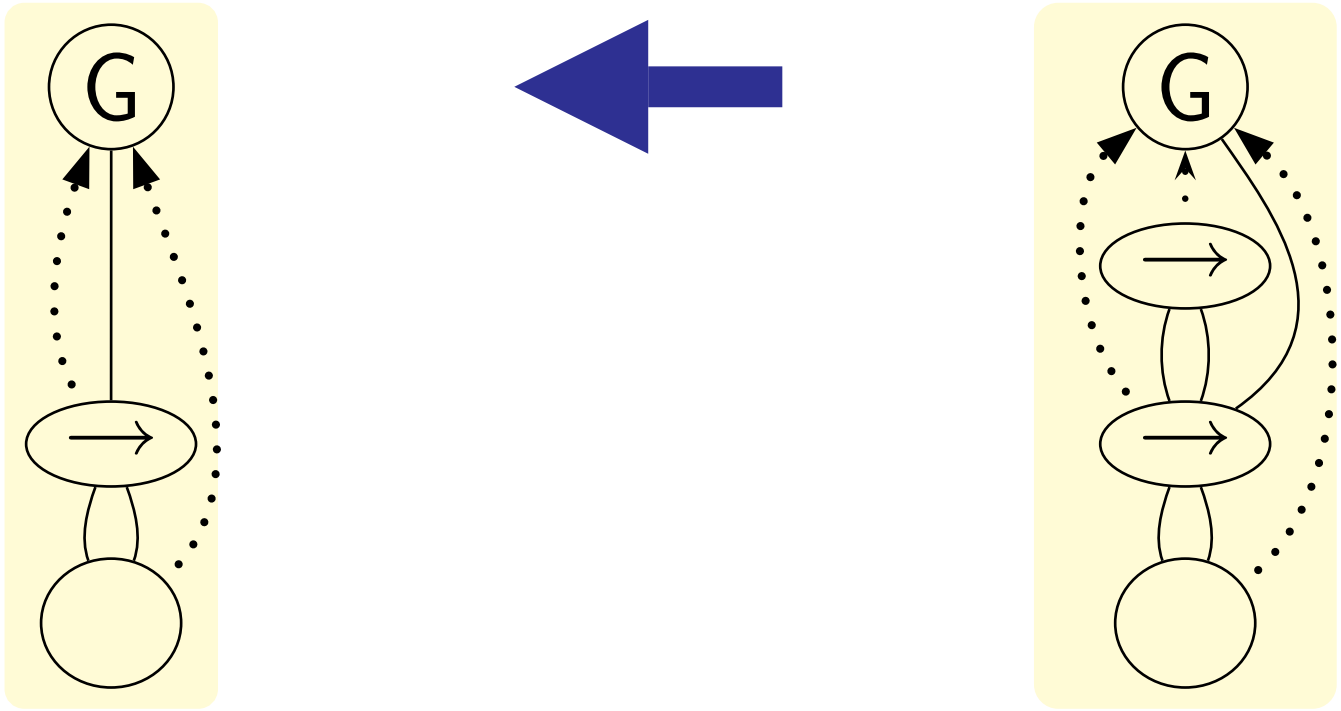
Inférence de type pour ML (exemple)



Inférence de type pour ML (exemple)



Inférence de type pour ML (exemple)



Type inference

Complexité

- exponentielle, en théorie
- quasi-linéaire, en pratique (avec une bonne implémentation)

Explication

- Complexité en $O(kn(\alpha(kn) + d)) \approx O(kdn)$ où
  - k est la taille maximale des types.
  - d est la profondeur d'imbrication à gauche des let.
- k et d sont tous deux bornées en pratique...
  lorsque les programmes sont écrits manuellement.
- peut poser des problèmes pour des programmes générés.

Au delà de ML : problèmes et enjeux

Typage fin des produits et des sommes

- Plus besoin de définir les enregistrements ou les sommes.
- Le système infére leurs types.
- Problème résolu, mais les types (et parfois les messages d'erreurs) sont compliqués.
- Les sommes fines (variantes) sont implémentées dans OCaml.

Au delà de ML : problèmes et enjeux

Typage encore plus fin des sommes (GADT)

- Les sommes permettent de d'étiquetter les valeurs.
- En testant un constructeur, on apprend des informations sur les valeurs, mais aussi dans certains cas, sur les types.

```
type 'a tag with
| Int : int -> int tag
| String : string -> string tag
type 'a dyamic = Dynamic of 'a tag * 'a

let f (Dynamic (tag, x)) = match tag with
| Int -> x + 1                    (* on sait que x est un entier *)
| String -> String.length x
```

- Permet de typer plus finement les invariants dans les structures de données. C'est une technique de plus en plus exploitée ou demandée.

Du polymorphisme de première classe

Pourquoi ML n'est-il pas suffisant ?

Une structure de donnée est un enregistrement de fonctions permettant de construire et d'explorer cette structure.

Typiquement, les fonctions sont polymorphes, comme map

Pour paramétrer un algorithme par une structure de donnée (par exemple, un conteneur, dont la représentation n'est pas fixée), il faut lui passer une structure de donnée en paramètre, donc un enregistrement dont les champs sont des fonctions polymorphes. Ce n'est pas possible en ML.

Autre applications du le polymorphisme de première classe :

- Interaction avec des types existentiels
- Codage des modules
- Codage des objets

Besoin de types plus expressifs

Types abstraits

On ne peut pas définir de types abstraits an ML (sans les modules)
Par exemple, on ne peut pas enfermer ensemble une fonction et ses arguments en cachant le type des arguments :

```
let x = (1, string_of_int ) : ∃t. t * (t -> string)
let x' = (1.0, string_of_float ) : ∃t. t * (t -> string)
let bag = [x ; x' ]
```

et les mettre dans une même liste...
pour exécuter plus tard, le calcul suivant :

```
let calcul x =
  let (v, operation) : ∃t. t * (t -> string)) = x in
  operation v in
map calcul bag
```

Typage en grand : typage des modules

Une approche stratifiée

Le calcul des modules est un système de typage explicite au dessus du système de typage du langage de base.

- Plus de problème d'inférence pour le langage des modules.
- Mais beaucoup de dupplication entre les deux langages :
  - dédoublement des notions de fonction, de produit, etc.
  - complique le langage : sa théorie et sa présentation aux programmeurs.
  - il y a parfois plusieurs façons de faire la même chose.

Vers une approche unifiée

- Un seul calcul, économe (pas de redondance) permetant de programmer à petite et à grande échelle.
- Donner à chaque monde le meilleur des deux.
- Des difficultés, sujet de recherche.

Autres techniques d'inférence

L'inférence dans ML repose sur l'unification de premier ordre qui trouve des types principaux et rend le typage robuste (invariant par de petites transformations des programmes).

Le polymorphisme de première classe indui des problèmes d'unification d'ordre supérieur qui ne sont pas décidables et n'admettent plus des solutions principales.

Une solution alternative, l'inférence locale n'a pas de bonne propriétés (pas de types principaux, fragile par rapport aux transformations de programme)

Trouver de bonnes techniques d'inférence pour des systèmes de types plus expressifs reste un problème de recherche difficile.

Mélanger types et preuves

Les systèmes de typages sont de plus en plus compliqués

- exprimer des invariants fins dans les structures de données, oun exprimer des liens forts entre plusieurs arguments.
- typer plus finement la mémoire
  - typage des effets de bords, permet de garantir qu'une expression est pure, ce qui permet de mieux la compiler.
  - tracer la consommation des ressources (complexité en espace)

Des systèmes qui deviennent très puissants mais aussi trop complexes.

C'est une course sans fin : le typage ne pourra jamais tout dire sur les programmes (indécidable).

Mélanger types et preuves

Les systèmes de typages sont de plus en plus compliqués

La preuve de programmmes devient une réalité

- De plus en plus de preuves de gros programmes
- Contrairement au typage, la preuve est manuelle ou assistée, elle peut dire plus, mais seulement lorsqu'on le souhaite (pas besoin de prouver tous les programmes ou toutes les parties des programmes)
- Est-ce une alternative au typage ? Ou un complément ?

Mélanger types et preuves

Les systèmes de typages sont de plus en plus compliqués

La preuve de programmmes devient une réalité

Combiner typage et preuve

- Laisser au typeur les choses simples, liées à la structure du programme et des données qui peuvent être complétement automatisées.
- Exprimer les invariants complexes comme des propriétés logiques qui se rajoutent au typage, et que l'on prouve semi-manuellement.

Mélanger types et preuves

Les systèmes de typages sont de plus en plus compliqués

La preuve de programmmes devient une réalité

Un gros enjeu

> Trouver de bonnes façons de combiner système de type et système de preuve dans un même langage de programmation

Conclusions

Le langage ML est un point d'équilibre remarquable entre expressivité et simplicité.

Mais il craque sous les demandes d'enrichissement.
Le besoin d'un successeur se faire de plus en plus pressant.

Mais il reste des questions difficiles à résoudre pour gagner en expressivité sans (trop) perdre en simplicité.