ML^F : réconcilier ML et Système F.

Séminaire VERIF

16 Juin 2009

Didier Le Botlan

LAAS (équipe OLC)

INSA

Plan

- Contexte
- Position du problème (3 min de vulgarisation)
- ML^{F} de l'extérieur
- ML^{F}, sous le couvercle (récréation graphique)

Contexte

Le problème présenté aujourd'hui est étudié depuis une vingtaine d'années.

Boehm (1985),
Giannini & Ronchi Della Rocca (1988),
Pfenning (1988, 1993),
Rémy (1994),
Läufer & Odersky (1994, 1996),
Wells (1994),
Kfoury & Wells (1994, 1999),
Schubert (1998),
Pierce & Turner (1998),
Garrigue & Rémy (1999),
Odersky & Zenger & Zenger (2001),
Leijen & Löh (2005),
Vytiniotis & Weirich & Peyton Jones (2006).

Nous y reviendrons.

C'est devenu mon sujet de thèse...

ML^F : Une extension de ML avec polymorphisme de second ordre et instanciation implicite.

Mai 2004

Jury

Directeur de thèse	Didier Rémy
Rapporteurs	Benjamin Pierce
	Jacques Garrigue
Examinateurs	Roberto Di Cosmo
	Claude Kirchner
	Dale Miller

Et ça a été repris

- *Qualified Types for* ML^F (Daan Leijen, Andreas Löh, ICFP 2005)
- *A type directed translation of MLF to system F* (Daan Leijen, ICFP 2007)
- *Graphic type constraints and efficient type inference: from ML to MLF.* (Boris Yakobowski, mon successeur avec Didier Rémy, ICFP 2008)
- *Recasting* ML^F (Didier Le Botlan, Didier Rémy, Information and Computation 207 (2009) pp. 726-785)

Plan

- Contexte
- Position du problème
- ML^F de l'extérieur
- ML^F, sous le couvercle

Préambule

Le sujet de cette thèse est le typage

À quoi cela sert-il ?

Le typage vise à établir la sûreté des programmes, c'est-à-dire un fonctionnement sans erreur d'exécution.

Préambule

Si on compare un programme à une usine,
il s'agit de garantir, avant son inauguration,
que l'usine peut fonctionner.

Il s'agit également de repérer les erreurs.

Préambule

Pour détecter les erreurs, il est nécessaire de comprendre
— de manière schématique —
ce que fait l'usine / le programme.

C'est le rôle de l'inférence.

ML^{F} : Une extension de ML avec polymorphisme de second ordre
et instanciation implicite.

« *implicite* » signifie

qu'à partir d'un programme donné,

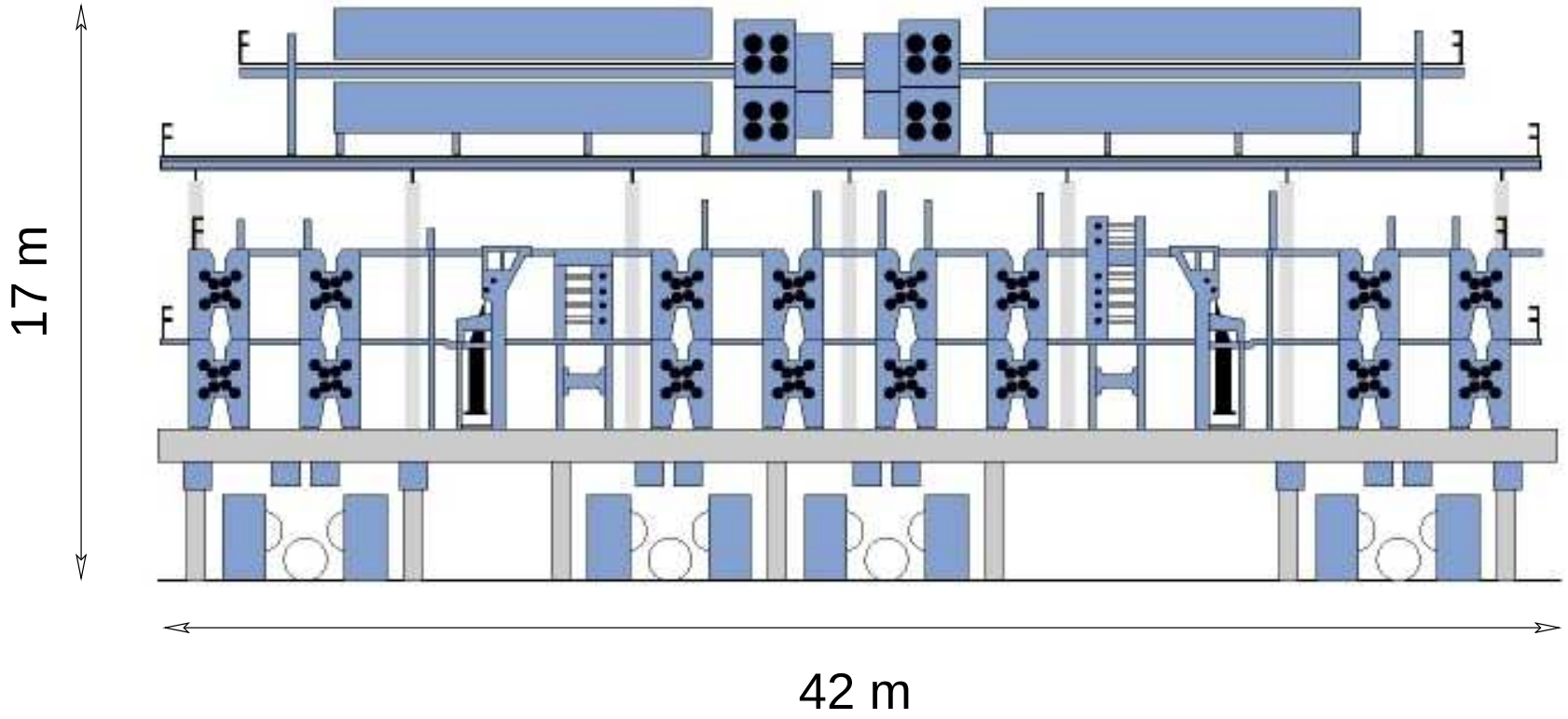
certaines informations sont devinées (inférées).

Devinette. . .

Que fait cette machine ?

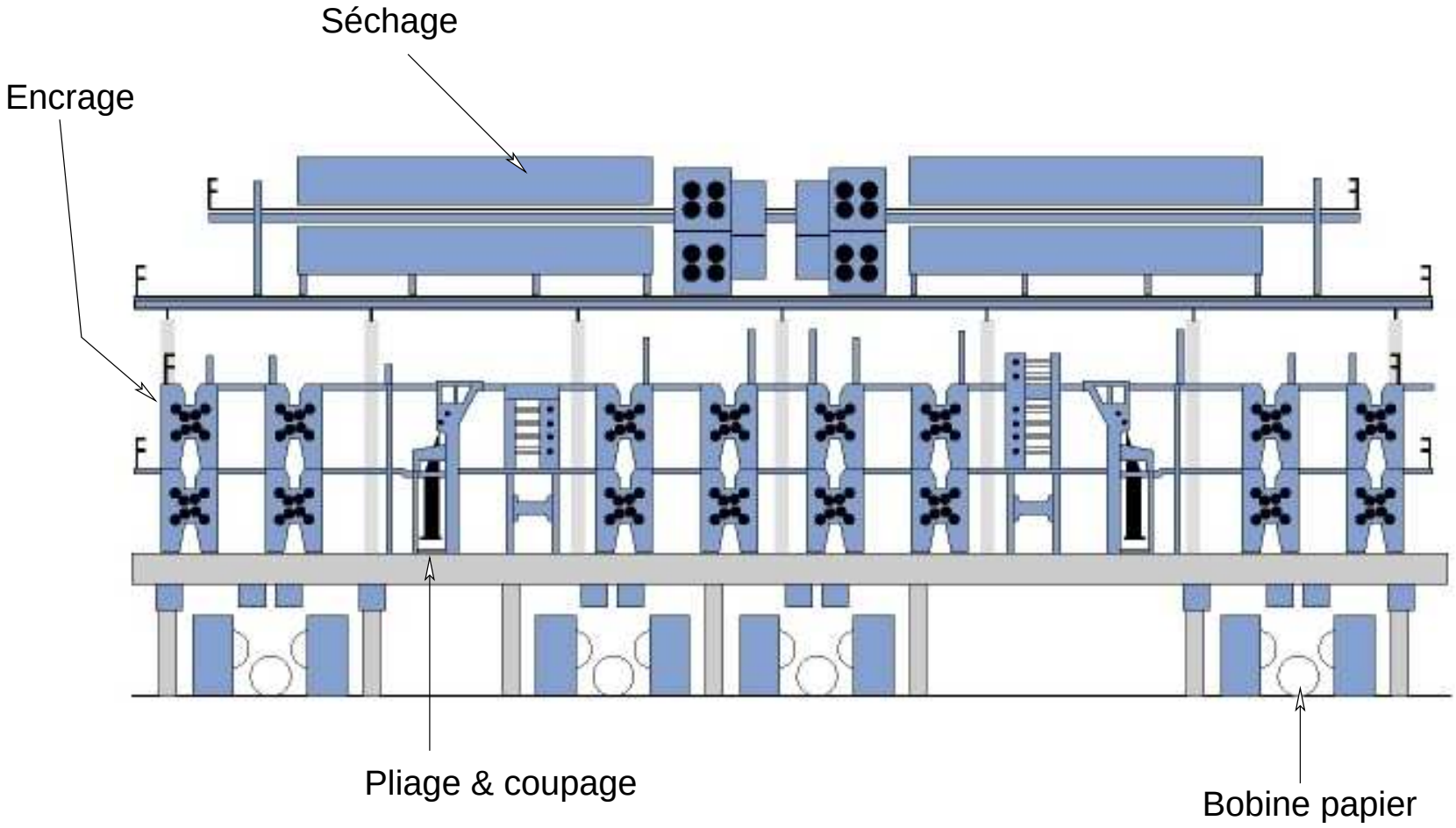
Devinette. . .

Son schéma de fonctionnement



Devinette. . .

Le schéma annoté



Conclusion

Quand on ne peut pas deviner,

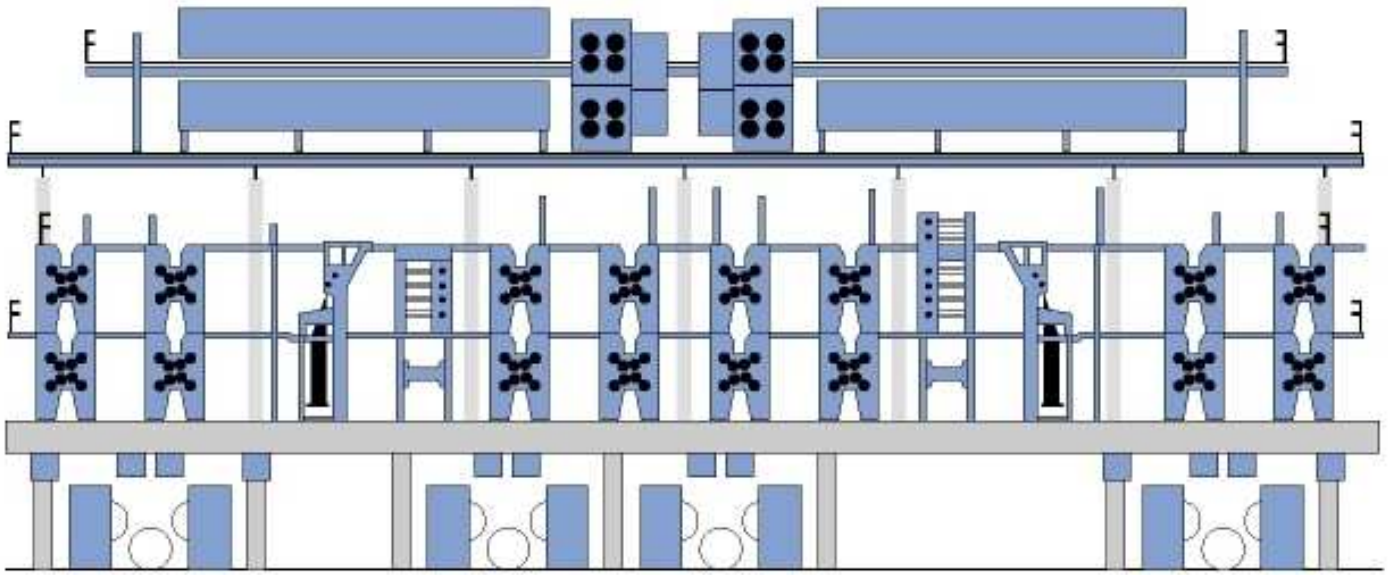
il faut des annotations.

Combien d'annotations ?

- Usine/programme Curry
- Usine/programme Church
- Usine/programme Hindley-Milner

Curry Church Hindley-Milner

Curry Church Hindley-Milner

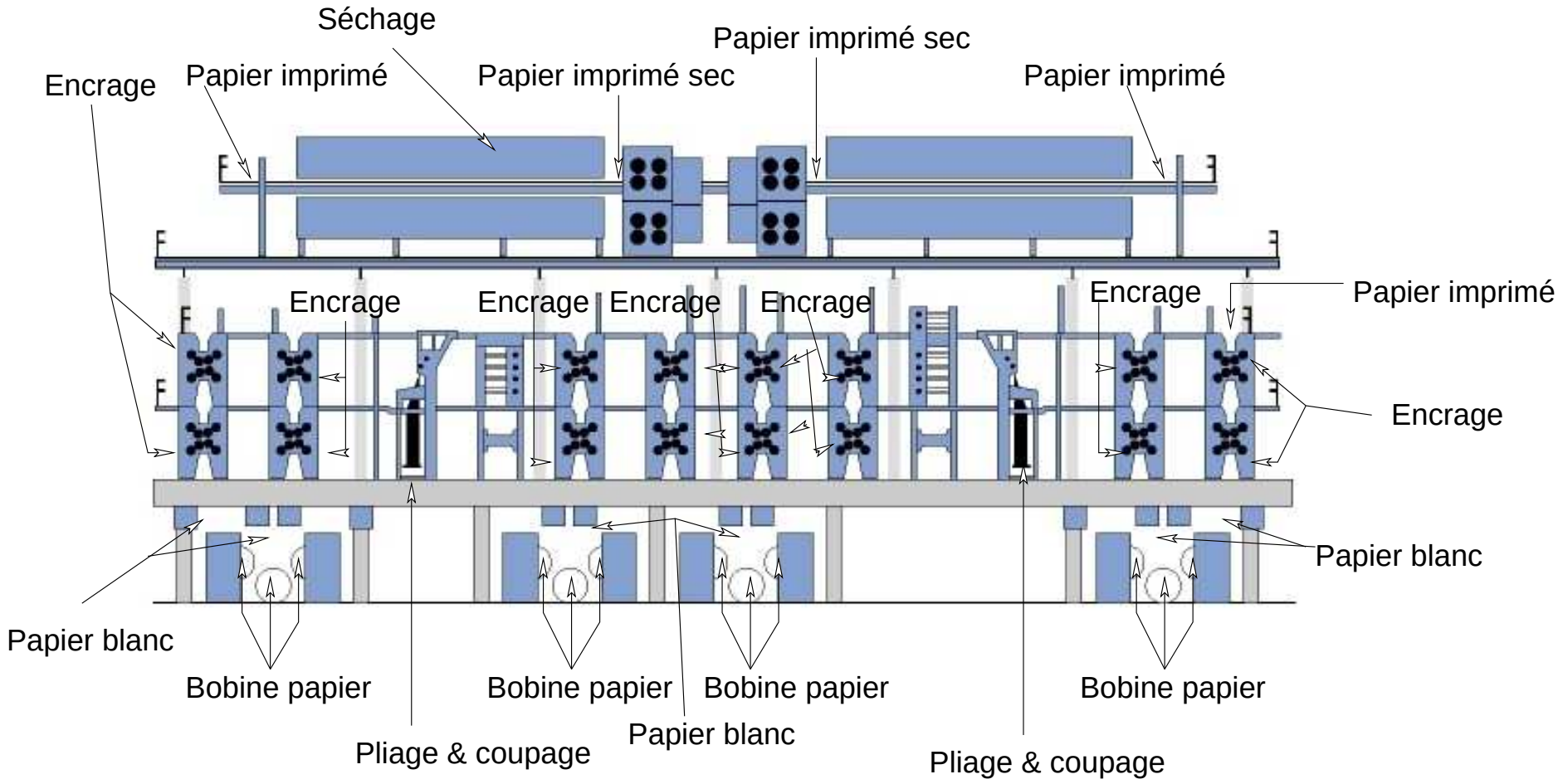


On ne peut pas deviner le type de l'usine dans le cas général.

J. B. Wells. (1994)

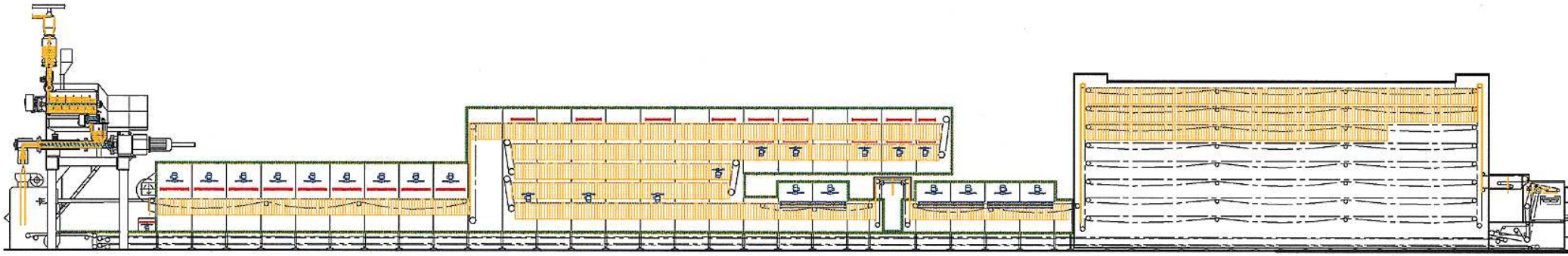
Typability and type checking in the second order λ-calculus are equivalent and undecidable.

Curry **Church** Hindley-Milner



Le programmeur passe plus de temps à poser des étiquettes qu'à écrire son programme

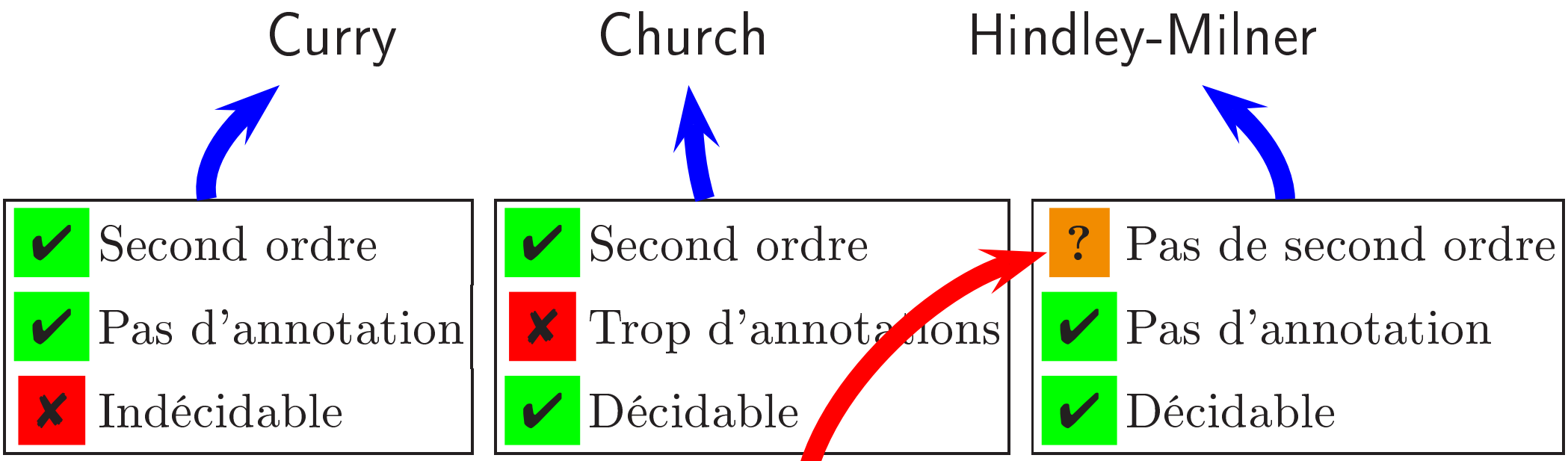
Curry Church Hindley-Milner



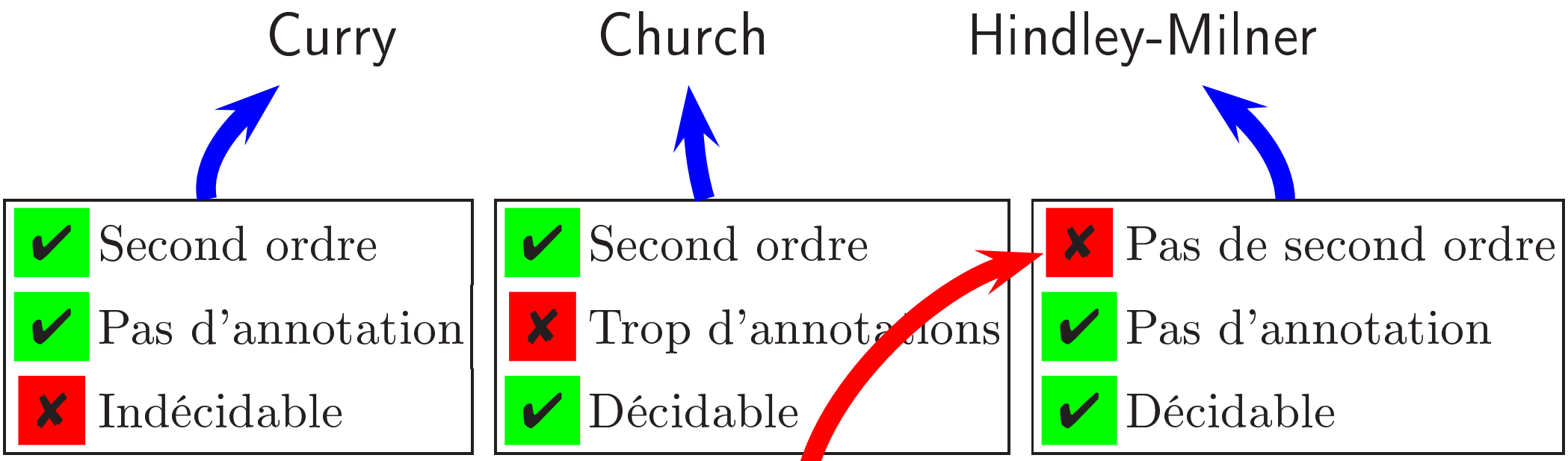
Pas de polymorphisme *de second ordre*.

Les programmes sont limités : moins compacts, d'ordre 1,
moins modulaires,
parfois moins sûrs.

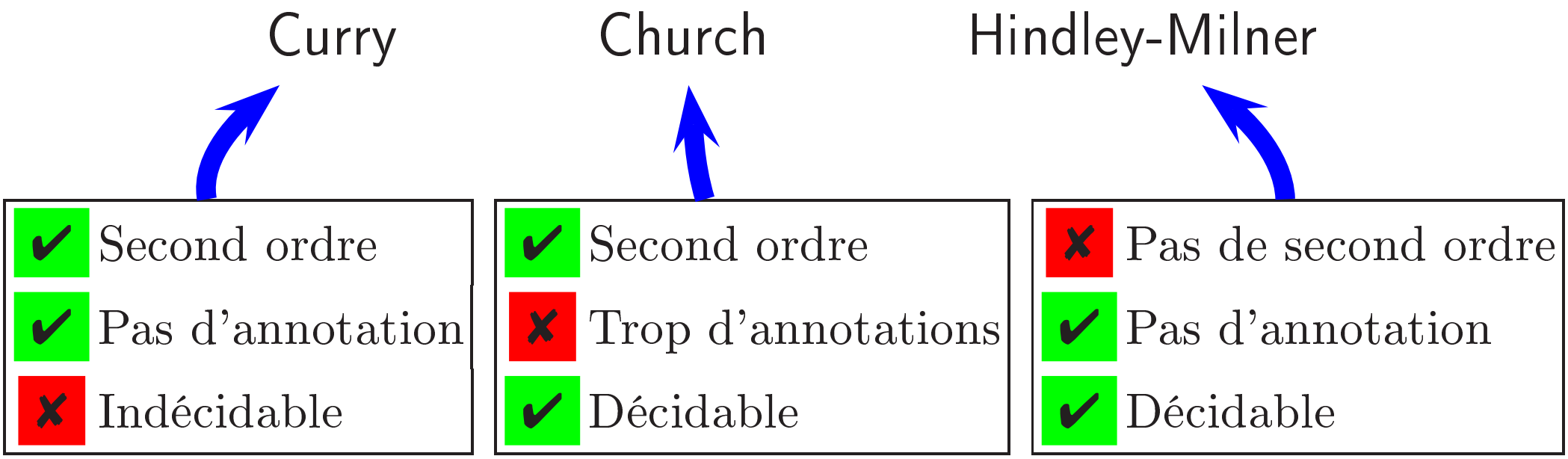
Le paradigme Hindley-Milner sert de base à des langages de programmation émérites ML, OCaml, SML, Haskell, Alice, ...



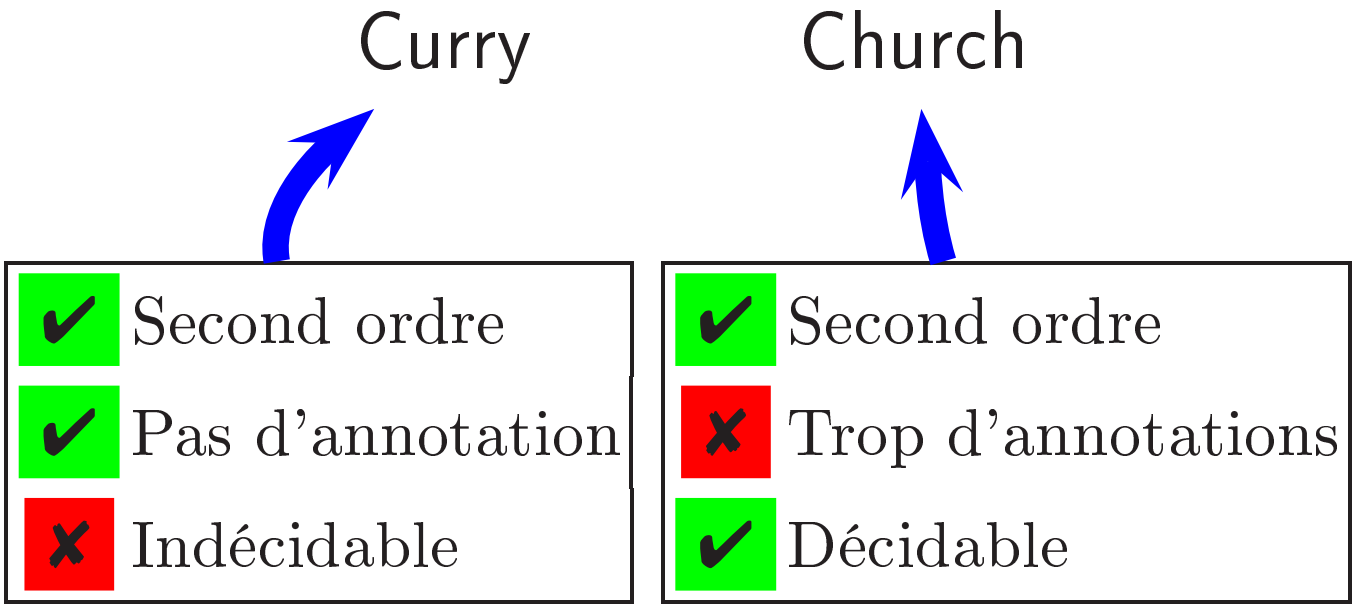
- ✘ Abstraction de valeurs polymorphes
- ✘ Monades, encapsulation
  (*e.g.* objets et méthodes polymorphes)
- ✘ Encodages divers
  (*e.g.* existentiels, syntaxe polytypique, schéma du visiteur)
- ✘ ...



- ✘ Abstraction de valeurs polymorphes
- ✘ Monades, encapsulation
  (*e.g.* objets et méthodes polymorphes)
- ✘ Encodages divers
  (*e.g.* existentiels, syntaxe polytypique, schéma du visiteur)
- ✘ ...



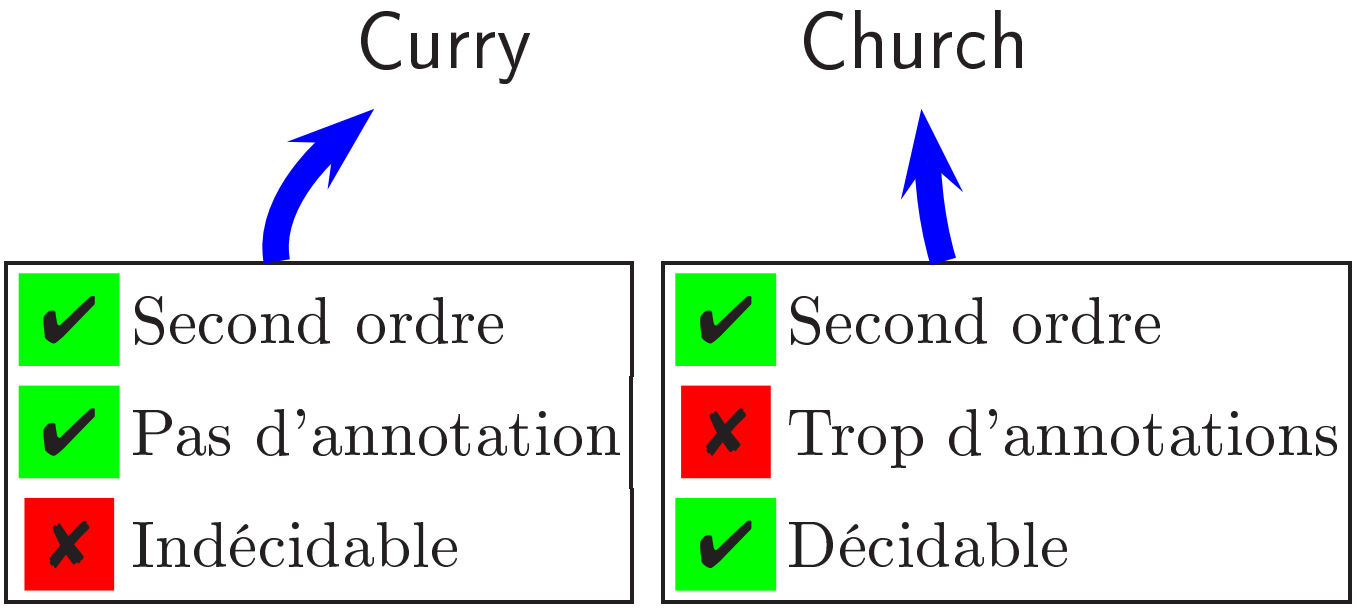
Une piste : combiner ces techniques.



F. Pfenning (1993)

On the undecidability of partial polymorphic type reconstruction.

✔ Second ordre
✔ Peu d'annotations
✘ Indécidable



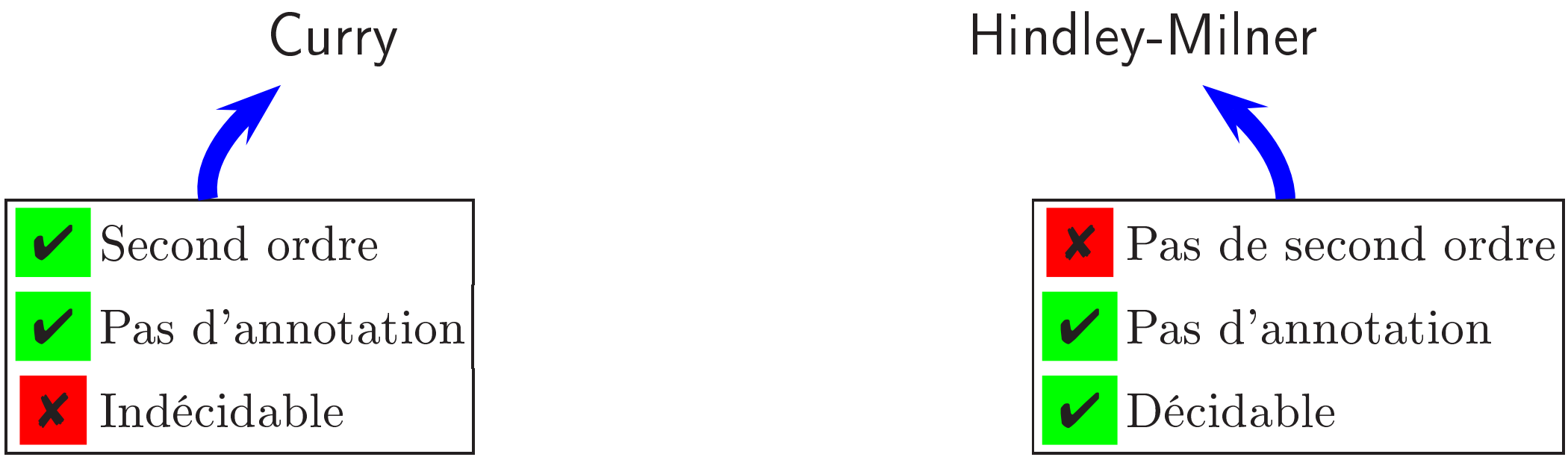
B.Pierce, D.Turner. (1998)
Local type inference.

M. Odersky, C. Zenger, M. Zenger (2001)
Colored local type inference.

✔ Sous-typage

✔ Une partie significative des annotations est devinée,

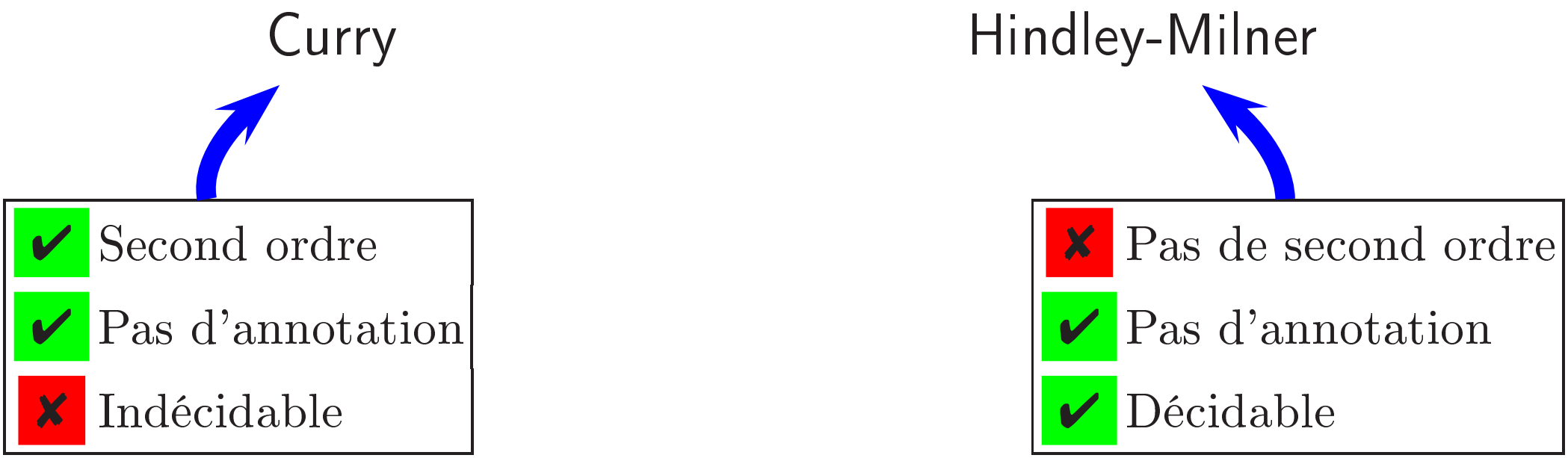
? mais pas suffisamment, notamment pour les programmes ML.



A. Kfoury, J.B. Wells. (1994)
A direct algorithm for type inference in the rank-2 fragment of the second-order lambda-calculus.

A. Kfoury, J.B. Wells. (1999)
Principality and decidable type inference for finite-rank intersection types.

T. Jim (2000)
A polar type system.

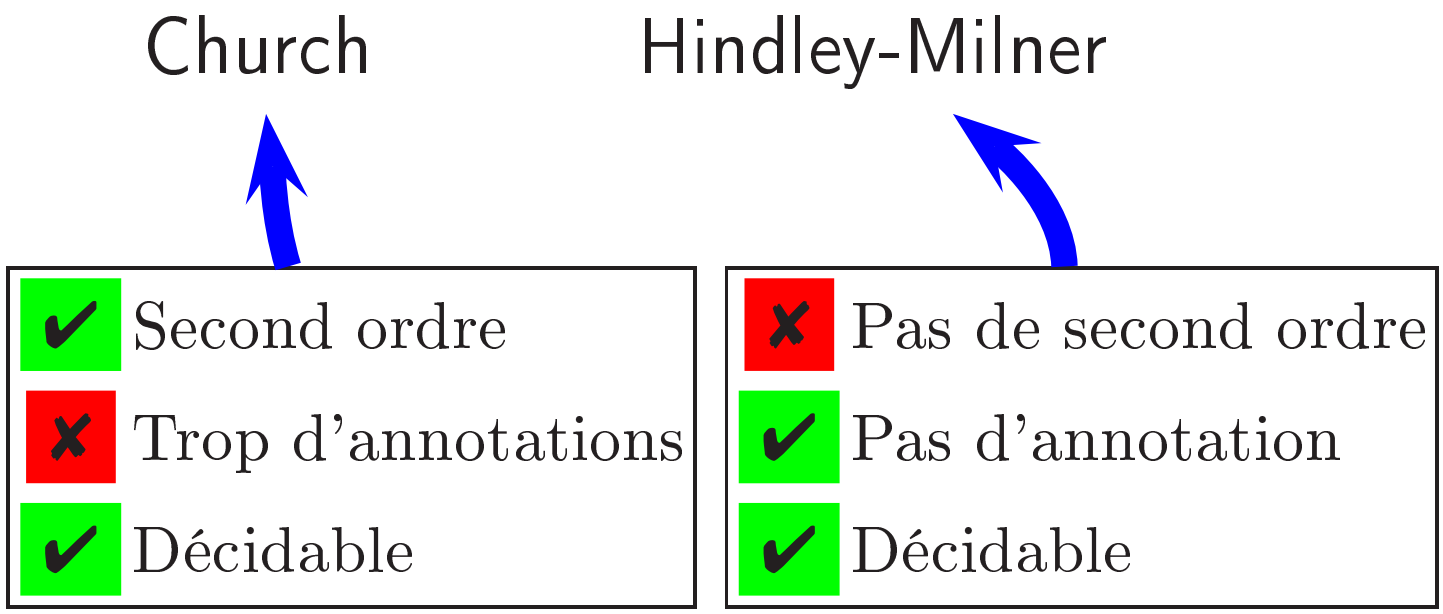


Systèmes de rangs finis

? Second ordre limité par construction

✔ Pas d'annotation

? Perte de modularité



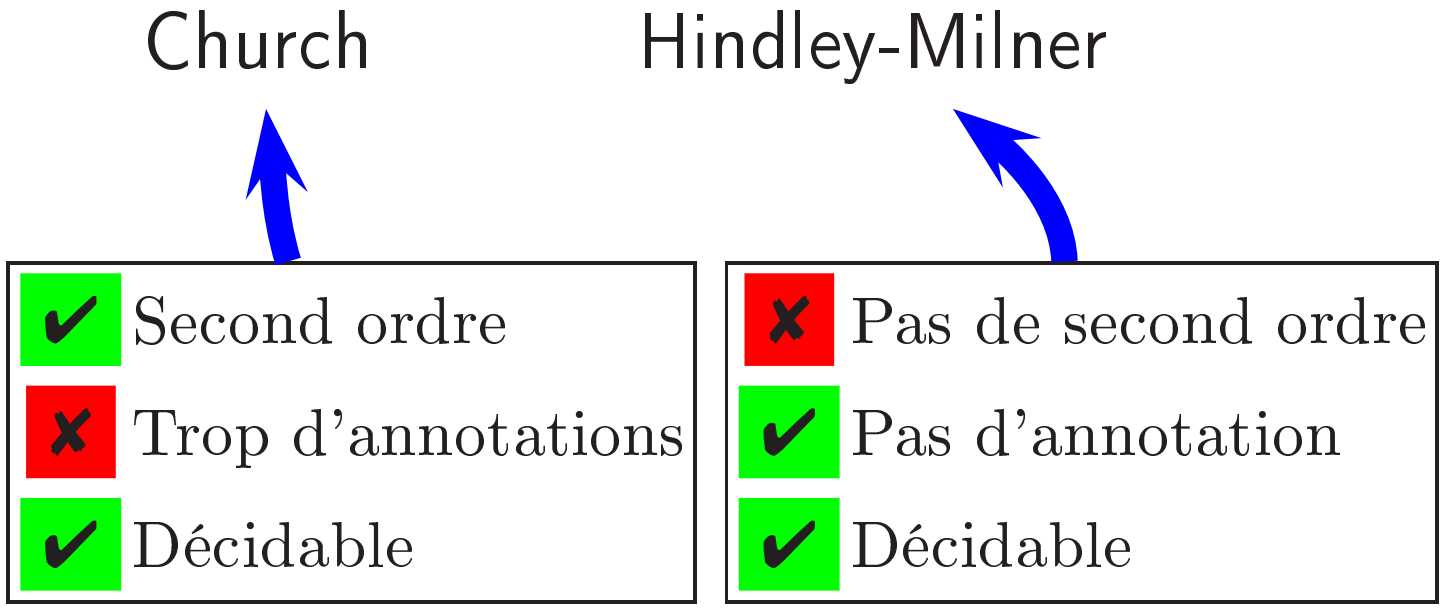
M. Odersky, K. Läufer (1996)

Putting type annotations to work.

J. Garrigue, D. Rémy. (1999)

Extending ML with semi-explicit higher-order polymorphism.

- ✔ Second ordre
- ✔ Pas d'annotation pour ML
- ✘ Trop d'annotations pour second ordre



ML^F : Une extension de ML avec polymorphisme de second ordre et instanciation implicite.

- ✔ Second ordre
- ✔ Pas d'annotation pour ML
- ✔ Pas d'annotation pour second ordre

Plan

- Contexte
- Position du problème
- ML^F de l'extérieur
- ML^F, sous le couvercle

Définitions

Système F

Le Système F à la Church (λ-calcul de second ordre),

$x \mid \lambda\,(x : t).\ a \mid \Lambda\alpha\,.a \mid a_1\ \ a_2 \mid a\ \ [t]$

ML

λ-calcul avec `let` et typage à la Hindley-Milner.

$x \mid c \mid \lambda\, x.\ a \mid a_1\ \ a_2 \mid \texttt{let}\ \ x = a_1\ \ \texttt{in}\ \ a_2$

ML^F

λ-calcul avec `let` et annotations

ML $+\ \lambda\,(x : t).\ a \mid (a : t)$ (langage externe)

Mode d'emploi de ML^F

☞ Les programmes ML sont typables tels quels.

Théorème

ML^F sans annotation est équivalent à ML.

Mode d'emploi de ML^F

- Les programmes ML sont typables tels quels.
- On ne devine pas le polymorphisme :
  $\lambda\, x.\ x\ \ x$ n'est pas typable dans ML^F sans annotation.
- Le polymorphisme de second ordre est explicite :
  `let` $\Delta = \lambda\,(x : \forall\alpha.\alpha \to \alpha\,).\ x\ \ x$ est typable
- Le polymorphisme est propagé implicitement :
  `List.fst` $[\Delta]\ \ (\lambda\, x.\ x)$ est typable sans annotation.

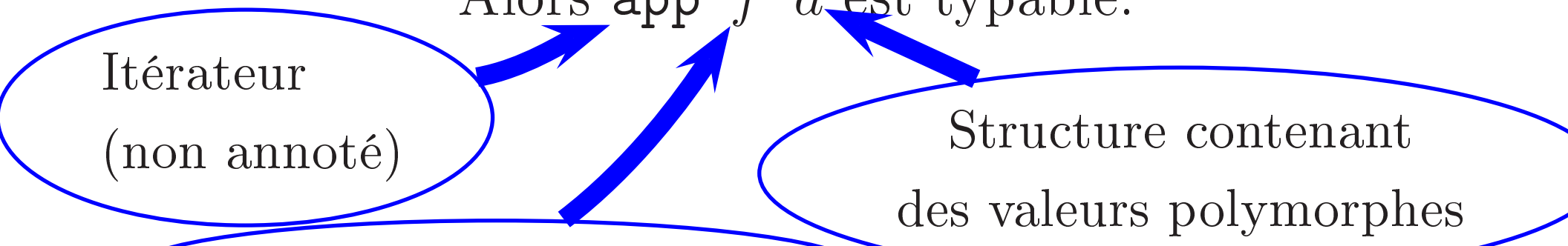
Seules les variables utilisées de manière polymorphe doivent être annotées

ML^F est compositionnel

$\texttt{let}\ \Delta = \lambda\,(x : \forall\alpha.\alpha \to \alpha\,).\ x\ \ x$	une annotation sur x
$\texttt{let}\ \texttt{app} = \lambda\,x.\,\lambda\,y.\,x\ \ y$	aucune annotation
$\texttt{let}\ \texttt{id} = \lambda\,x.\,x$	aucune annotation
$\texttt{app}\ \Delta\ \texttt{id}$	aucune annotation

Plus généralement, si l'expression $f\ \ a$ est typable,

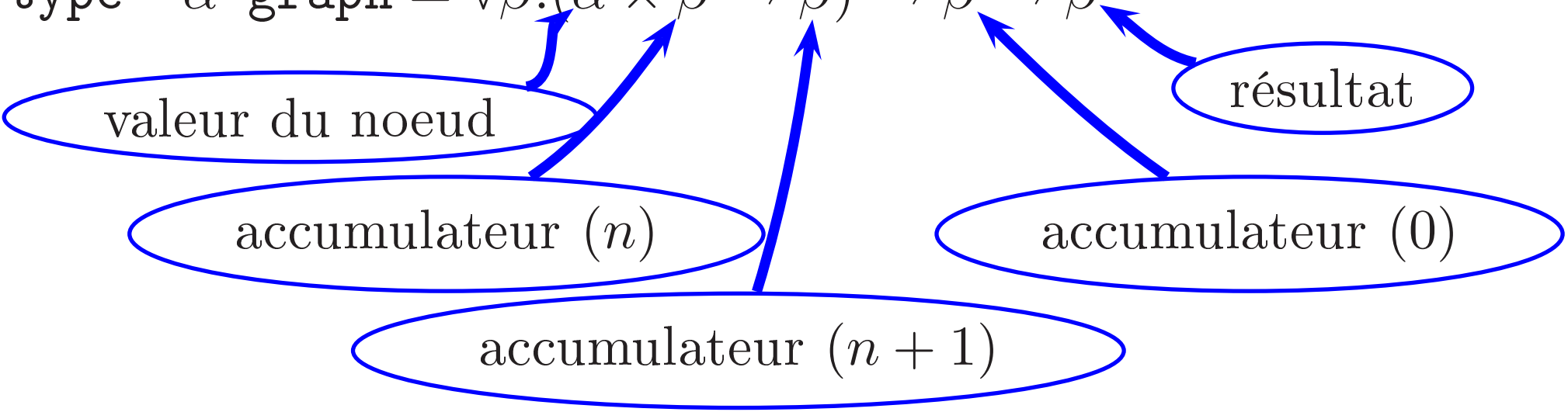
Alors $\texttt{app}\ \ f\ \ a$ est typable.



Exemple

Un graphe représenté par sa fonction de parcours.

$$\texttt{type}\quad \alpha\ \texttt{graph} = \forall\beta.(\alpha \times \beta \to \beta) \to \beta \to \beta$$



Exemple

Un graphe représenté par sa fonction de parcours.

type α graph $= \forall\beta.(\alpha \times \beta \to \beta) \to \beta \to \beta$

```
let size  gr = gr (λ (_,n) n + 1)  0
let nodes gr = gr (λ (x,l) x :: l) []

let flatten (gr: α graph ) =
      if size gr ⩽ 1000 then nodes gr
      else ...
```

Encodage du Système F à la Church

F $\quad x \mid \lambda\,(x : t).\ a \mid \Lambda\alpha\,.a \mid a_1\ a_2 \mid a\ [t]$

$\Downarrow$

ML$^{\mathsf{F}}$ $\quad x \mid \lambda\,(x : t).\ a \mid \Lambda\alpha\,.a \mid a_1\ a_2 \mid a\ [t]$

Les variables libres des annotations seront remplacées par un type adéquat.

Exemple :

$\lambda\,(x : \beta \to \forall\alpha.\alpha \to \alpha).\ x\ \ 1\ \ x$ $\quad$ β sera remplacé par `int`

Encodage du Système F à la Church

F $\quad x \mid \lambda\,(x : t).\ a \mid \Lambda\alpha\,.a \mid a_1\ a_2 \mid a\ [t]$

$\Downarrow$

ML$^{\mathsf{F}}$ $\quad x \mid \lambda\,(x : t).\ a \mid \cancel{\Lambda\alpha\,.a} \mid a_1\ a_2 \mid a\ \cancel{[t]}$

Théorème

L'ensemble du Système F est typable dans ML$^{\mathsf{F}}$.

Remarquer que l'encodage est une simple projection des termes.

Les annotations

En ML^F, les annotations sont des primitives (constantes)

L'application $(\_ : \sigma)\ a$ se note $(a : \sigma)$ (sucre syntaxique).

Le lambda annoté $\lambda\ (x : \sigma)\ .\ a$ est également du sucre pour

$$\lambda\, x.\, \texttt{let}\ \ x\ = (x :\ \sigma\ )\ \ \texttt{in}\ \ a$$

L'expressivité des annotations ne provient pas
d'une extension ad-hoc du langage,
mais uniquement de la puissance du système de types.

Comparaison avec d'autres travaux

ML^F ne devine pas le polymorphisme :

$\lambda\, x.\, x\ \ x$ n'est pas typable,

mais $\lambda\, x.\, (x : \sigma_{\mathtt{id}})\ \ x$ l'est.

Certains systèmes sont capables d'inférer le polymorphisme...

Les types intersection, par exemple, permettent de typer $\lambda\, x.\ x\ \ x$

malheureusement, ils permettent aussi de typer

```
let k = λx.λy.x                 and k' = λx.λy.y
and two = λf.λx.f (f x)         and three = λf.λx.f (f (f x))
and app = λf.λx.f x
let t y =
   (λh.h (h (h (h (λx.y)))))
   (λf.λn.n (λv.k') k app (λg.λx.n (f (n (λp.λs.s (p k')
   (λf.λx.f (p k' f x))) (λs.s k' k') k) g) x)) three
```

. . . en 1 heure sur un PIII 2GHz, en prenant 4Go de mémoire.

Source : `http://types.bu.edu/modular/compositional/system-i/`

A. Kfoury, J. B. Wells (1999)

Principality and decidable type inference for finite-rank intersection types

Avec ML^F, le résultat est instantané...

...le typage échoue ! (et nous en sommes fiers)

Remarquer qu'une seule annotation de type suffit à rendre le programme typable.

```
type Int = ∀α (α → α) → (α → α)
let t y =
  (λh.h (h (h (h (λx.y)))))
  (λf.λ(n: Int ).n (λv.k') k app (λg.λx.n (f (n (λp.λs.s (p k')
                                                  (λf.λx.f (p k' f x)))
                                              (λs.s k' k') k) g) x))
  three
```

Cet exemple et plusieurs autres (entiers de Church) ont été testés dans une implémentation prototype de ML^F.

→ Les types intersection typent exactement les programmes qui terminent...le typage est donc indécidable.

→ L'inférence dans le Système F sans annotation est indécidable.

☞ Pour échapper à l'indécidabilité, les systèmes suivants imposent des contraintes sur les rangs.

A. Kfoury, J. B. Wells (1999)
Principality and decidable type inference for finite-rank intersection types

A. Kfoury, J. B. Wells (1994)
A direct algorithm for type inference in the rank-2 fragment of the second-order λ–calculus.

T. Jim (2000)
A polar type system.

Contraintes sur les rangs

Les systèmes basés sur des limitations de rang souffrent d'un manque de modularité :

Étant donné un terme a typable, `id` a n'est pas toujours typable.

En effet, si le type de a est t, le type de `id` doit être $t \to ...$

Le rang de chaque quantificateur apparaissant dans t est donc incrémenté dans le type de `id`.

Ces systèmes ne sont pas compositionnels par construction.

L'inférence locale

Pour éviter l'unification globale, les techniques d'inférence locale propagent l'information localement dans le programme.

Ces systèmes incluent le sous-typage.

B. Pierce, D. Turner (1998)
Local type inference.

H. Hosoya, B. Pierce (1999)
How good is local type inference?

M. Odersky, C. Zenger, M. Zenger (2001)
Colored local type inference.

L'inférence locale

Par construction, l'inférence locale propage l'information de voisin en voisin.

- ☞ L'inférence fonctionne très bien avec le sous-typage
- ☞ Des annotations "évidentes" restent parfois nécessaires, notamment dans le cas de ML.

ML^F n'inclut pas le sous-typage, mais type tous les programmes ML. (D'ailleurs, veut-t-on vraiment le sous-typage ?)

L'encapsulation

Les polytypes sont encapsulés dans des monotypes.

Les coercions entre polytypes et monotypes encapsulés sont explicites.

D. Rémy. (1994)
ML-ART: An extension to ML with abstract and record types.

M. Odersky, K. Läufer. (1996)
Putting type annotations to work.

J. Garrigue, D. Rémy (1999)
Extending ML with semi-explicit higher-order polymorphism.

L'encapsulation

ML^F s'inscrit dans la suite directe de ces travaux.

Les progrès accomplis :

- ✔ Strictement moins d'annotations nécessaires
- ✔ L'encodage du Système F est une transformation locale et ne nécessite pas d'analyse de la dérivation.
- ✔ Un programme ML^F est toujours moins annoté que le programme Système F correspondant.
- ✔ Plus d'encapsulation : la frontière entre monotypes et polytypes est plus perméable (plus de passeport).

Putting Type Annotations to Work

Outre l'encapsulation, ce système permet d'annoter un λ avec un schéma de type.

- ✘ Cette construction ne suffit pas à encoder le Système F,
- ✘ et n'est pas compositionnelle (propriété "`app`") : l'expression `app` Δ `id` n'est pas typable.

Propagation des annotations

Une idée de l'inférence locale est de propager en profondeur les annotations du niveau supérieur.

Dans ML^F, un mécanisme "purement syntaxique", précédant le typage, permet de propager les annotations.

Exemple :

`file.mli` `let` $\Delta : (\forall\alpha.\alpha \to \alpha) \to \_$

`file.ml` `let` $\Delta = \lambda\, x.\, x\ \ x$

pas d'annotation

Contributions

Le polymorphisme devient accessible au programmeur

- ✔ Compatibilité avec ML
- ✔ Types principaux, Algorithme d'unification
- ✔ Un minimum d'annotations

Réduire les annotations, c'est passer du côté indécidable.

Plan

- Contexte
- Position du problème
- ML^F de l'extérieur
- ML^F, sous le couvercle

Ingrédients

- Un langage de types avec une relation d'instance
- Des règles de typage
- Un algorithme d'inférence
- Des annotations

Ingrédients

- Un langage de types avec une relation d'instance
- Des règles de typage
- Un algorithme d'inférence
- Des annotations

Exemple

Soit `id` l'identité $\lambda\, x.\, x$.

Soit `choix` une fonction de type $\forall\,(\gamma)\ \gamma \to \gamma \to \gamma$.

Quel est le type de `choix id` ?

Dans le Système F, deux typages pour `choix id` :

`choix` $[\forall\alpha.\alpha\to\alpha]$ `id` : $(\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$

`choix` $[\alpha\to\alpha]$ (`id` $[\alpha]$) : $(\alpha \to \alpha) \to (\alpha \to \alpha)$

Exemple

Soit `id` l'identité $\lambda\, x.\, x$.

Soit `choix` une fonction de type $\forall\,(\gamma)\ \gamma \to \gamma \to \gamma$.

Quel est le type de `choix id` ?

Dans ML^F, deux typages pour `choix id` :

`choix` $[\forall\alpha.\alpha\to\alpha]$ `id` : $(\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$

`choix` $[\alpha\to\alpha]$ (`id` $[\alpha]$) : $(\alpha \to \alpha) \to (\alpha \to \alpha)$

Exemple

Soit `id` l'identité $\lambda\, x.\, x$.

Soit `choix` une fonction de type $\forall\,(\gamma)\ \gamma \to \gamma \to \gamma$.

Quel est le type de `choix id` ?

Dans ML^F, deux typages pour `choix id` :

`choix id` : $(\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$

`choix id` : $(\alpha \to \alpha) \to (\alpha \to \alpha)$

Exemple

Dans ML^F, deux typages pour `choix id` :

`choix id` : $(\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$

`choix id` : $(\alpha \to \alpha) \to (\alpha \to \alpha)$

Exemple

Dans ML^F, deux typages pour `choix id` :

`choix id` : $(\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$

`choix id` : $(\alpha \to \alpha) \to (\alpha \to \alpha)$

Exemple

Dans ML^F, deux typages pour `choix id` :

`choix id` : $\forall\,(\beta = \forall\alpha.\alpha \to \alpha)\ \beta \to \beta$

`choix id` : $\forall\,(\beta = \alpha \to \alpha)\ \beta \to \beta$

Exemple

Le type σ donné à `choix id` peut s'instancier en :

$$\forall\,(\beta = \forall\alpha.\alpha \to \alpha)\ \beta \to \beta$$

et

$$\forall\,(\beta = \alpha \to \alpha)\ \beta \to \beta$$

Nous le notons

$$\forall\,(\beta \geq \forall\alpha.\alpha \to \alpha)\ \beta \to \beta$$

De manière similaire, $\forall\,(\alpha)\ \alpha \to \alpha$ est en réalité $\forall\,(\alpha \geq \bot)\ \alpha \to \alpha$.

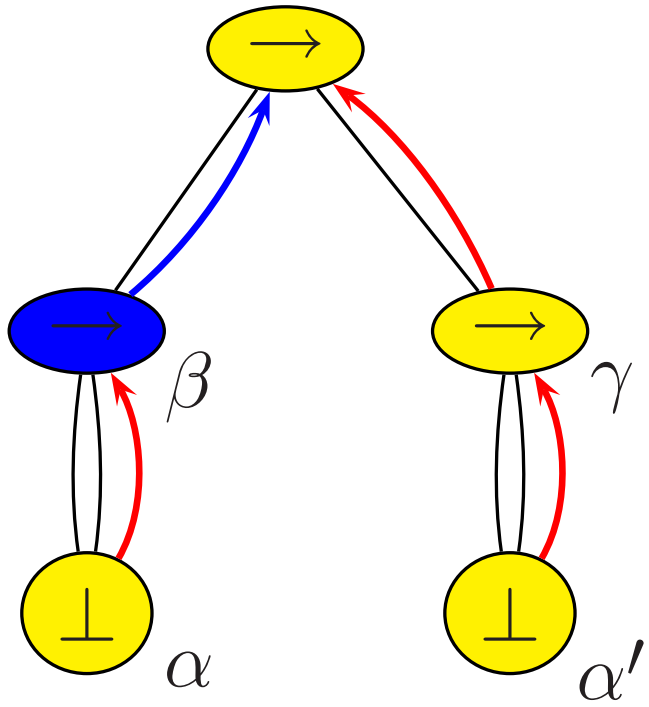
Syntaxe des types

monotypes $\quad \tau ::= \alpha \mid \tau \to \tau'$

polytypes $\quad \sigma ::= \tau \mid \bot \mid \forall\,(\alpha \geq \sigma_1)\ \sigma_2 \mid \forall\,(\alpha = \sigma_1)\ \sigma_2$

Remarquer que $\alpha \geq \tau$ et $\alpha = \tau$ sont équivalents.

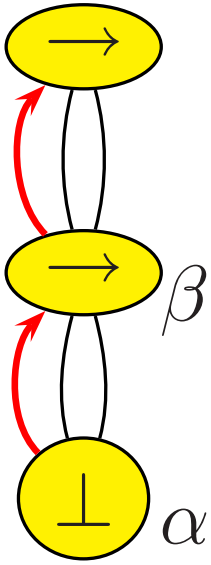
Représentation graphique des types



$$\forall\,(\beta = \forall\,(\alpha)\ \alpha \to \alpha\ )\ \forall\,(\gamma \geq \forall\,(\alpha')\ \alpha' \to \alpha'\ )\ \beta \to \gamma$$

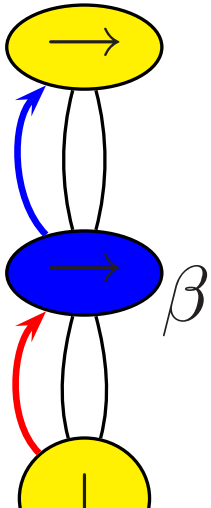
Déplacement d'un quantificateur flexible

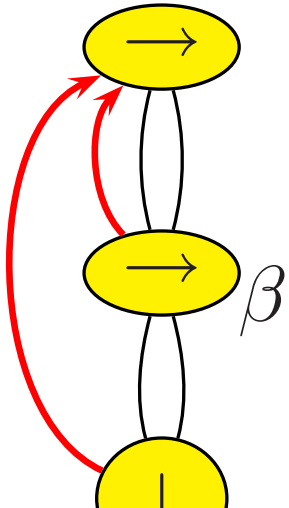
Le type $\forall\,(\beta \geq \forall\alpha.\alpha \to \alpha)\ \beta \to \beta$



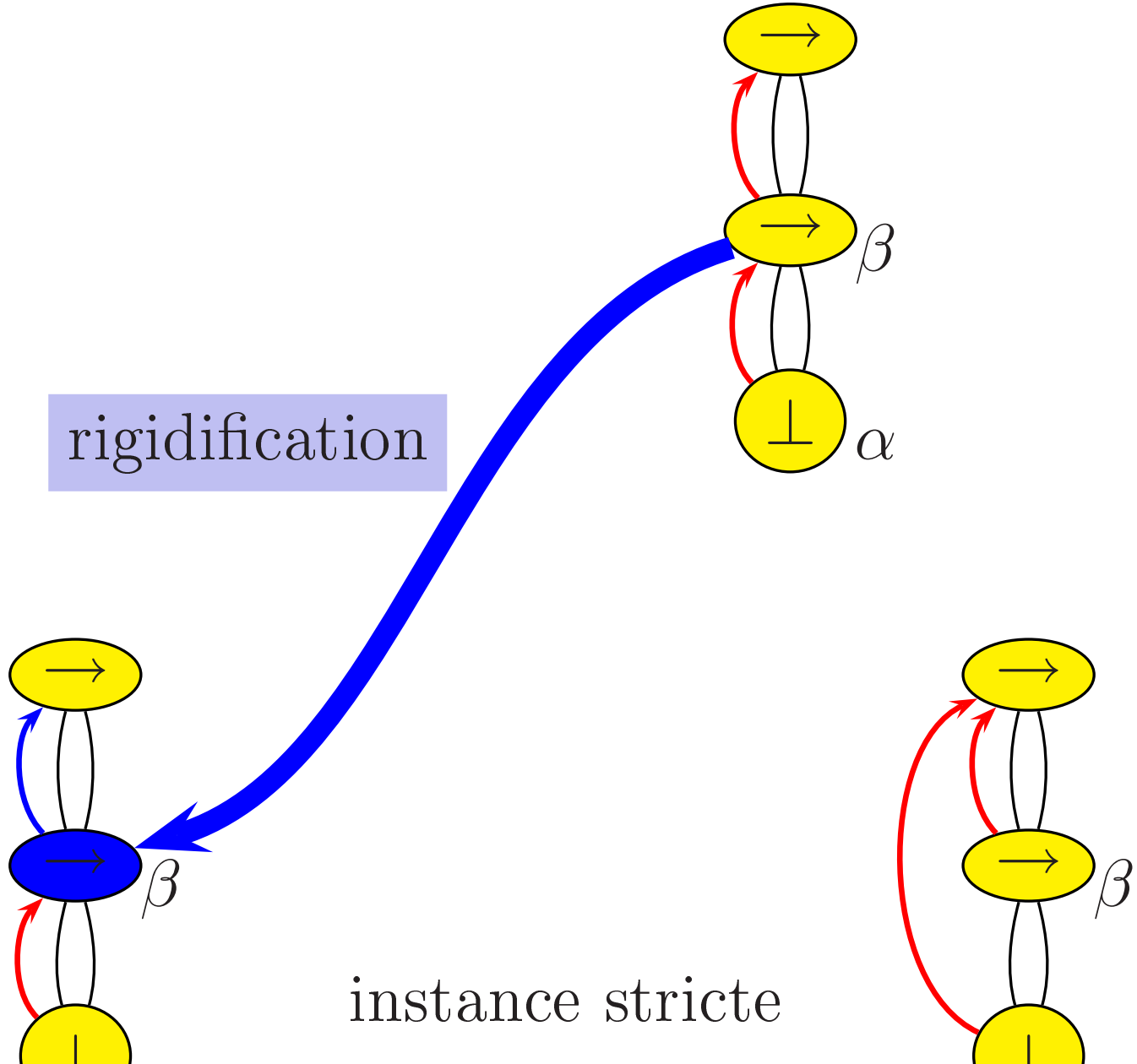
peut s'instancier en

$\forall\,(\beta = \forall\alpha.\alpha \to \alpha)\ \beta \to \beta$ et $\forall\,(\alpha)\ \forall\,(\beta = \alpha \to \alpha)\ \beta \to \beta$

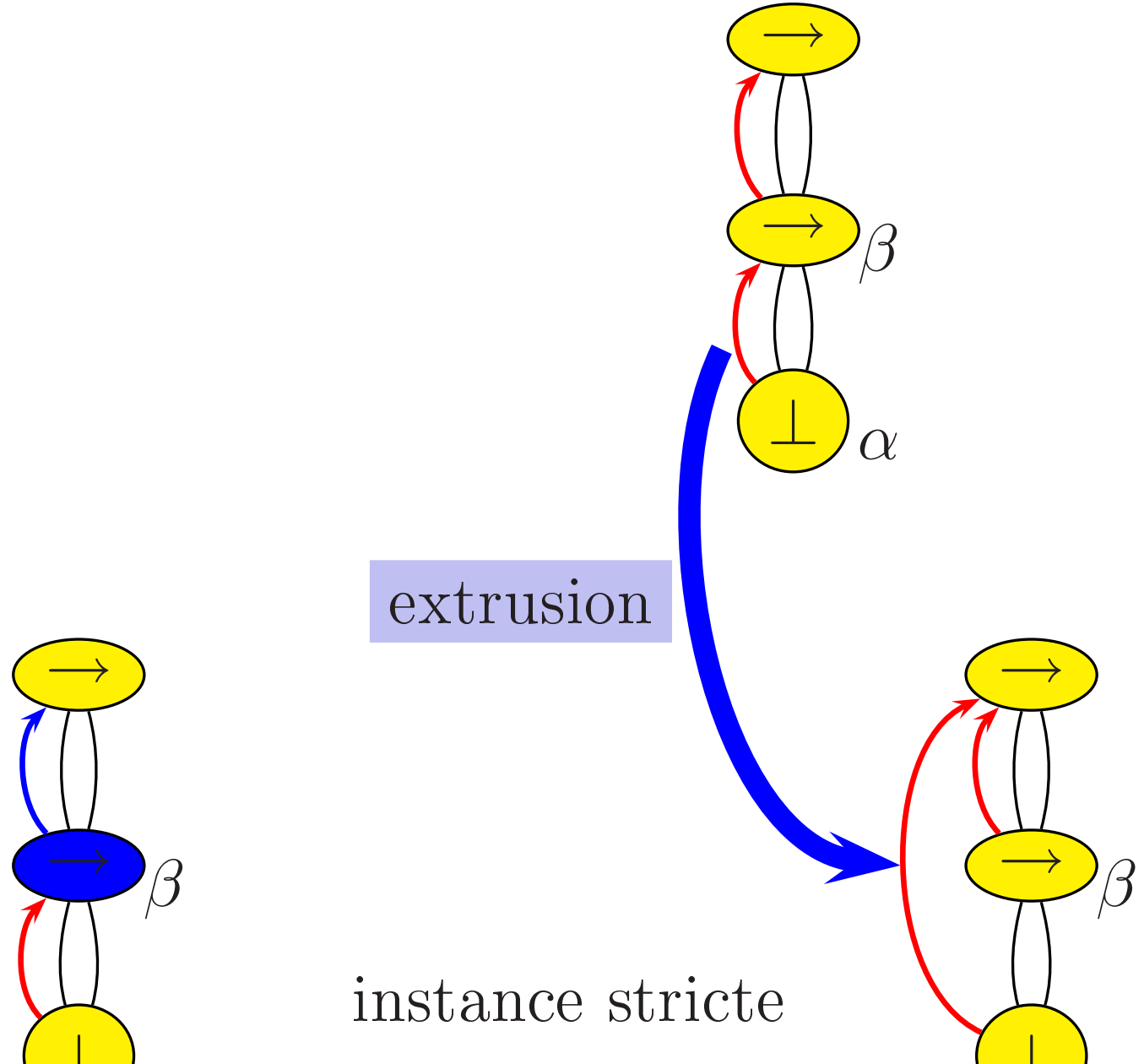




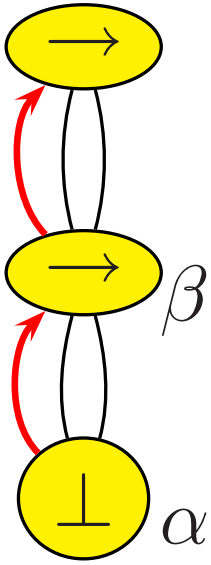
Déplacement d'un quantificateur flexible

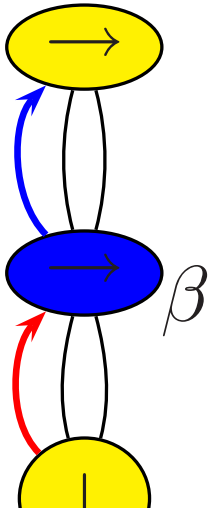


Déplacement d'un quantificateur flexible

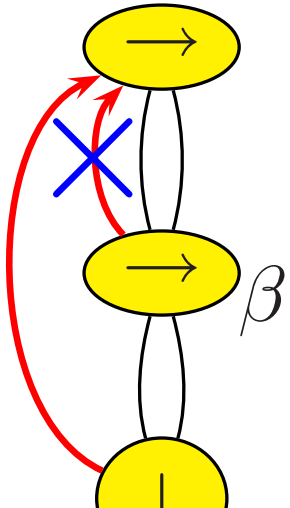


Déplacement d'un quantificateur flexible

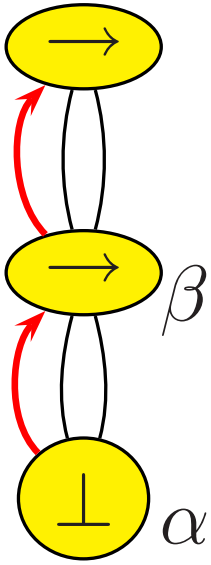


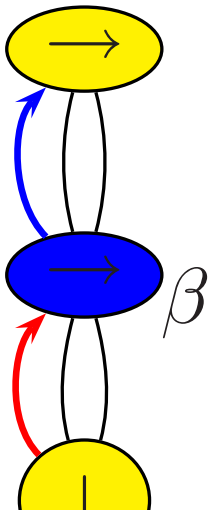


Monotype



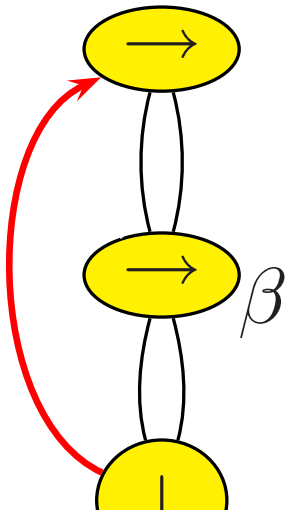
Déplacement d'un quantificateur flexible





Monotype

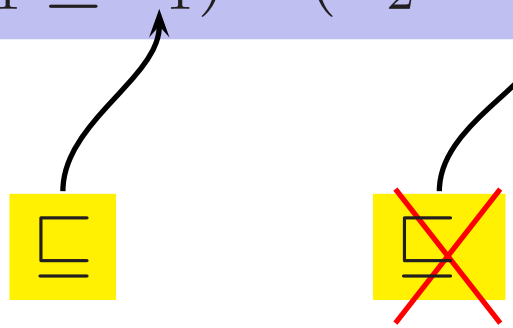
Équivalence



L'instanciation

Cas de base $\bot \sqsubseteq \sigma$

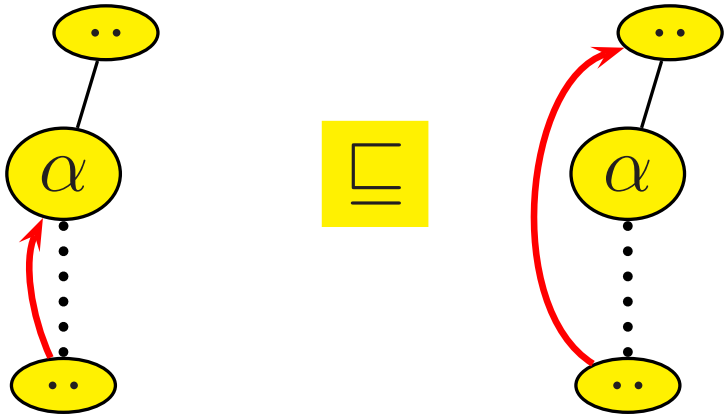
Contextes $\forall\,(\alpha_1 \geq \sigma_1)\;\forall\,(\alpha_2 = \sigma_2)\;\sigma_3$



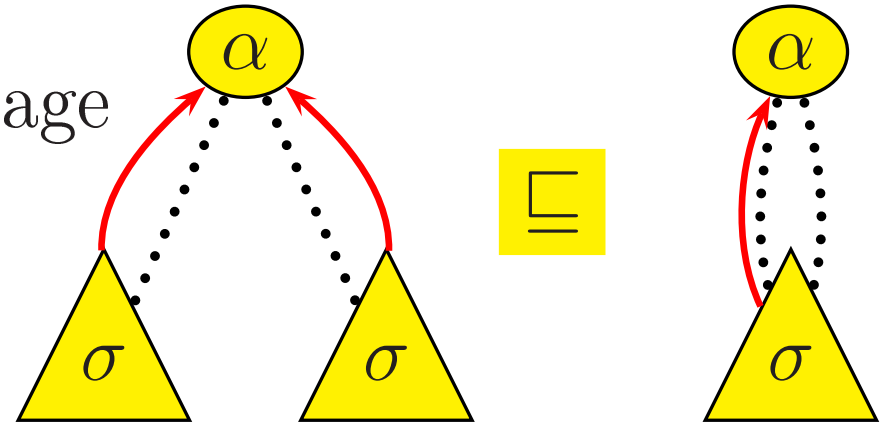
L'occurrence de α_1 est flexible

Instanciation et quantificateurs

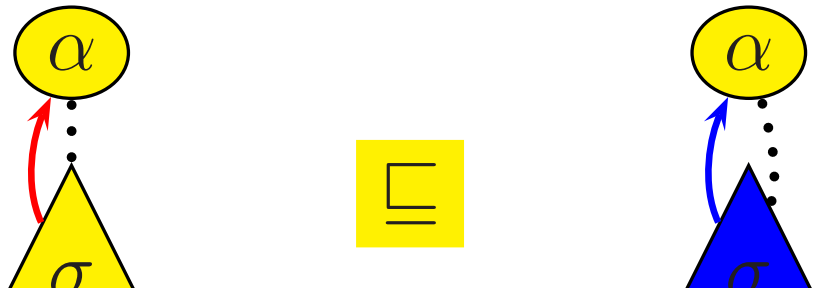
Extrusion d'un quantificateur



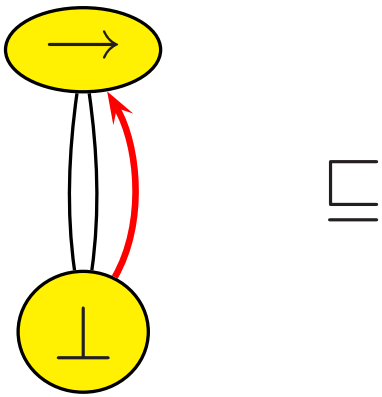
Partage



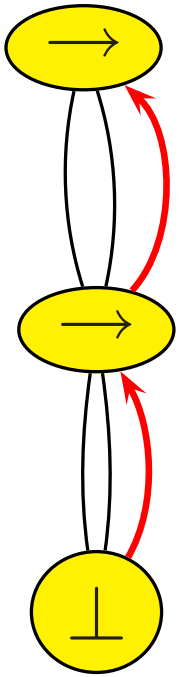
Rigidification



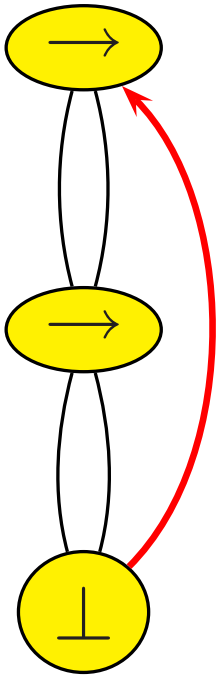
Exemple



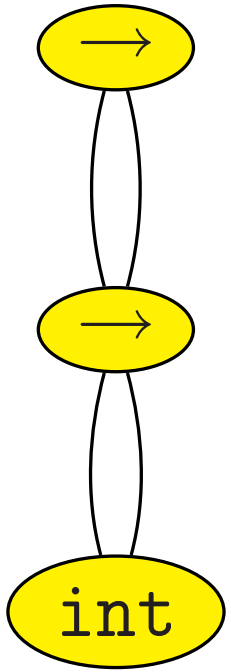
$\sqsubseteq$



$\sqsubseteq$



$\sqsubseteq$



$\forall\,(\alpha \geq \boxed{\bot}\,)\;\alpha \to \alpha$

$\forall\,(\alpha \geq \boxed{\forall\beta.\beta \to \beta}\,)\;\alpha \to \alpha$

$\forall\,(\beta)\;\forall\,(\alpha = \boxed{\beta \to \beta}\,)\;\alpha \to \alpha$

Le préfixe

Un type peut avoir des variables libres.

- Dans le Système F, elles doivent être introduites dans l'environnement de typage.
- En ML, elles ne sont pas introduites.
- En ML^F, elles doivent apparaître dans un préfixe.

Ce préfixe est de la forme $\alpha_1 \geq \sigma_1, \alpha_2 = \sigma_2, \ldots$

En ML, le préfixe est laissé implicite : $\alpha_1 \geq \bot, ..\alpha_n \geq \bot$

Le préfixe

En conséquence, la relation d'instance et les jugements de typage sont définis sous préfixe.

$Q \quad \sigma_1 \sqsubseteq \sigma_2$ $\qquad$ $Q \quad \Gamma \vdash a : \sigma$

Ingrédients

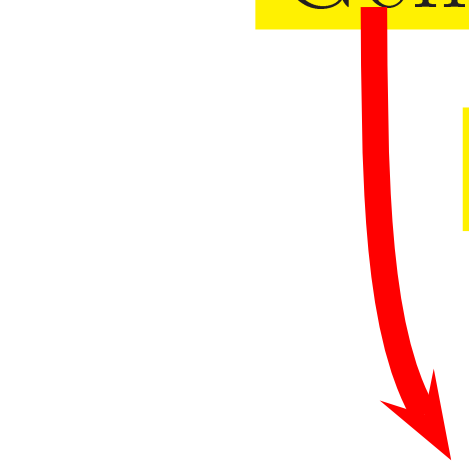
- Un langage de types avec une relation d'instance
- Des règles de typage
- Un algorithme d'inférence
- Des annotations

Règles de typage de ML

Var Fun App

Gen Inst

Let



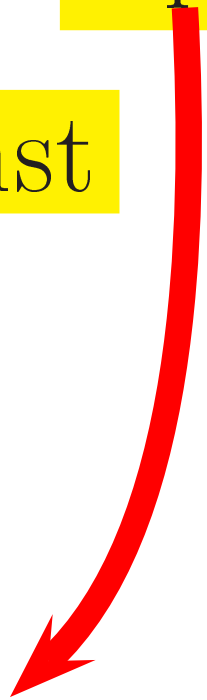
$$\frac{\Gamma \vdash a : \sigma \qquad \alpha \notin \mathbf{ftv}(\Gamma)}{\Gamma \vdash a : \forall\,(\alpha)\ \sigma}$$

Règles de typage de ML

Var Fun App

Gen Inst

Let



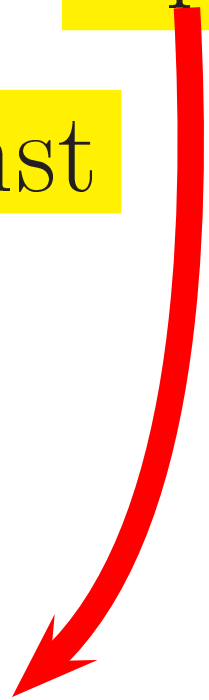
$$\frac{\Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1 \ a_2 : \tau_1}$$

Règles de typage de ML sous préfixe

Var Fun App

Gen Inst

Let



$$\frac{Q \quad \Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad Q \quad \Gamma \vdash a_2 : \tau_2}{Q \quad \Gamma \vdash a_1 \;\; a_2 : \tau_1}$$

Règles de typage de ML sous préfixe

Var Fun App

Gen Inst

Let

En ML, les préfixes Q sont de la forme $\alpha_1 \geq \bot, \ldots \alpha_n \geq \bot$

En ML^F, les préfixes sont de la forme $\alpha_1 \diamond \sigma_1, \alpha_2 \diamond \sigma_2, \ldots$

$\geq$ ou $=$

Règles de typage de ML^F

Var Fun App

Gen Inst

Let

$$\frac{Q \quad \Gamma \vdash a : \sigma \qquad Q \quad \sigma \sqsubseteq \sigma'}{Q \quad \Gamma \vdash a : \sigma'}$$

Règles de typage de ML^F

Var Fun App

Gen Inst

Let

$$\frac{Q, \alpha \diamond \sigma' \quad \Gamma \vdash a : \sigma \qquad \alpha \notin \mathbf{ftv}(\Gamma)}{Q \quad \Gamma \vdash a : \forall\,(\alpha \diamond \sigma')\ \sigma}$$

Typage de ML^F

Les règles de typage de ML^F sont celles de ML exceptés

- ☞ Le préfixe, qui est explicite.
- ☞ La relation d'instance, qui est plus générale.

Théorème

Le typage est sûr (*subject reduction** et *progress*).

* montrée dans une variante de ML^F : $\mathsf{ML}^\mathsf{F}_\star$

Ingrédients

- Un langage de types avec une relation d'instance
- Des règles de typage
- Un algorithme d'inférence
- Des annotations

Inférence

L'algorithme d'inférence de ML^F est similaire à celui de ML.

- Si l'expression est typable, il retourne son type principal.
- Il repose essentiellement sur l'unification.

Théorème

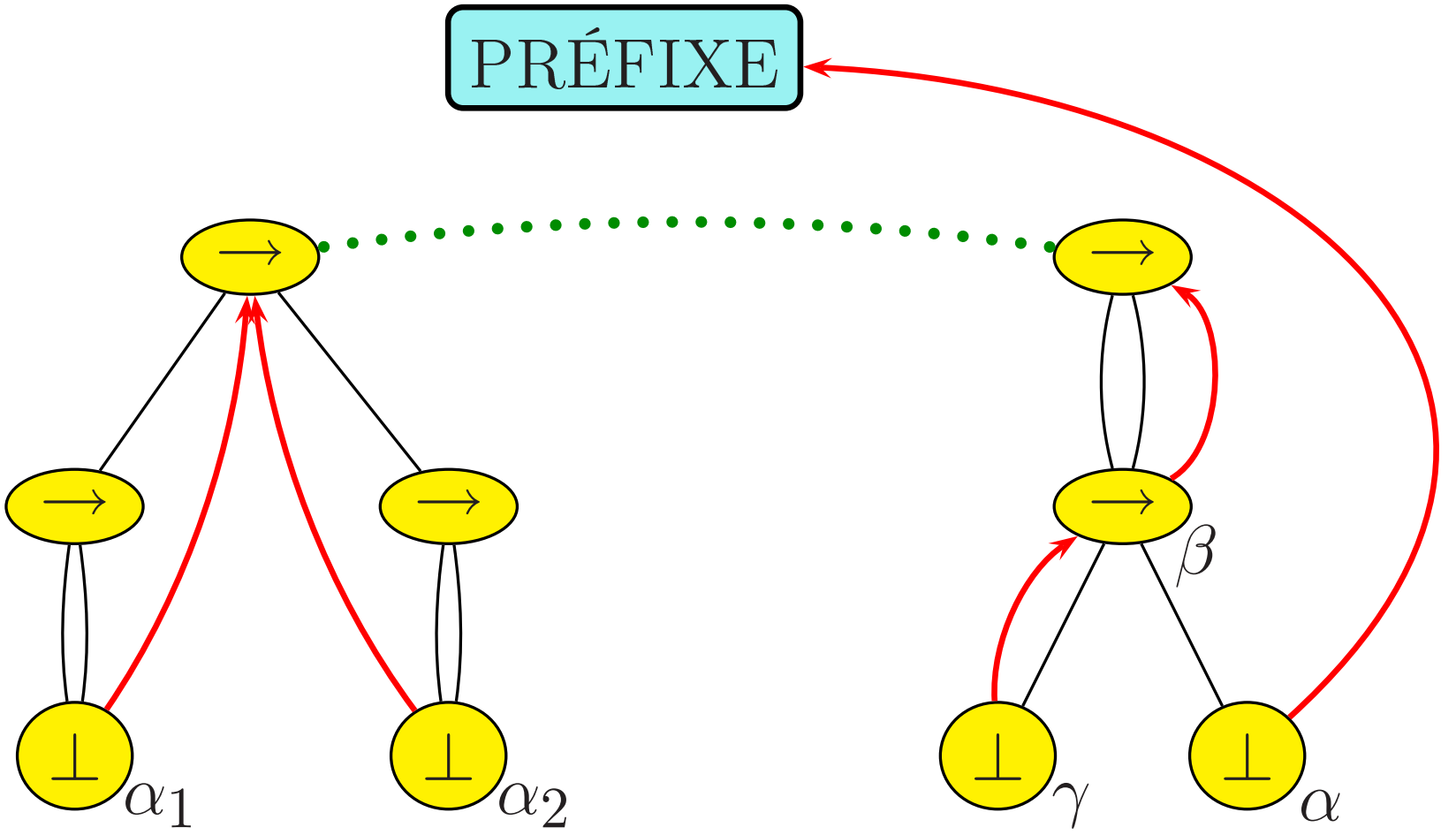
L'algorithme d'inférence est correct et complet

L'algorithme d'unification de ML^F est en partie similaire à celui de ML mais doit en plus unifier les quantificateurs.

Théorème

L'algorithme d'unification est correct et complet

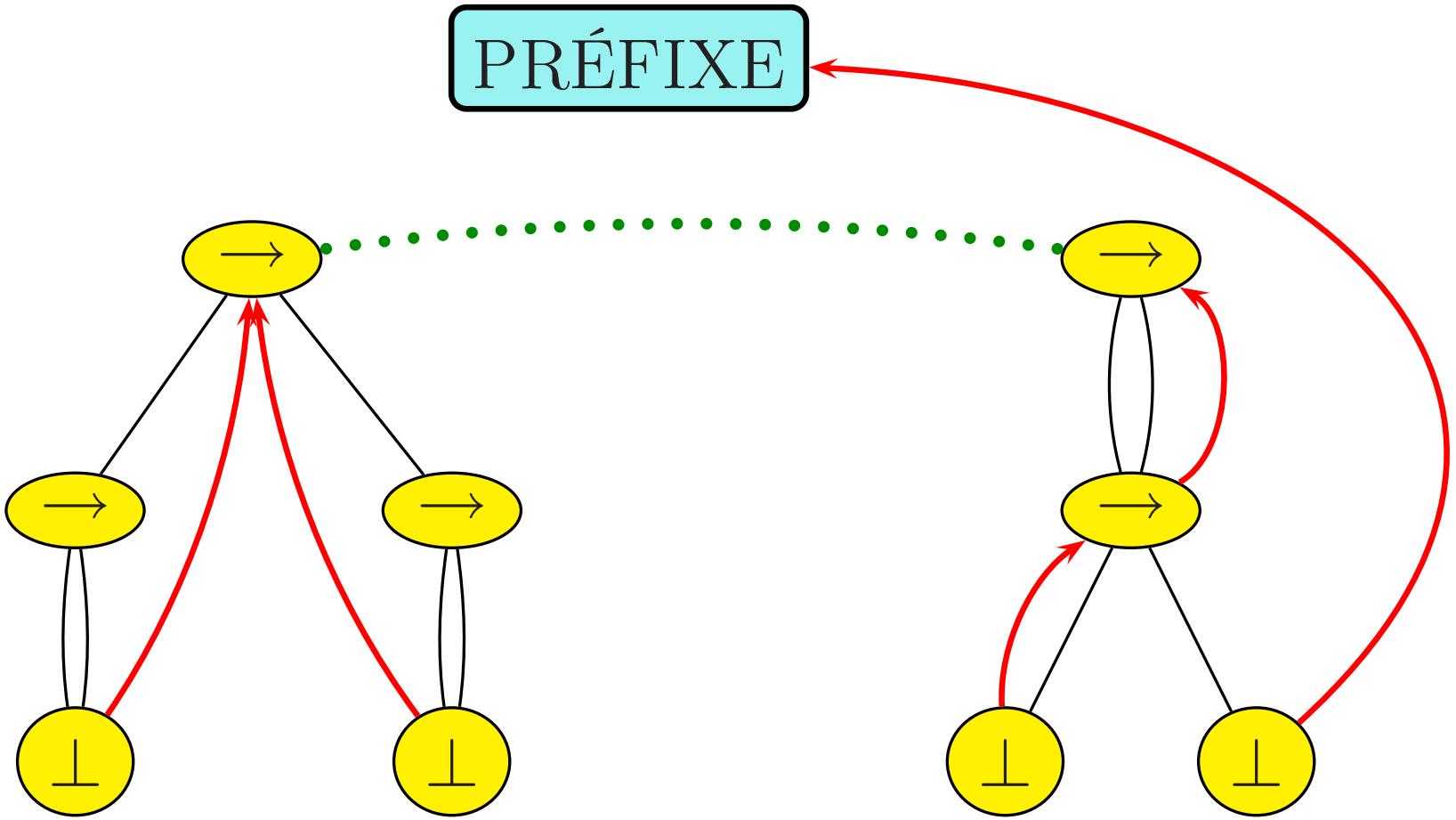
Exemple d'unification

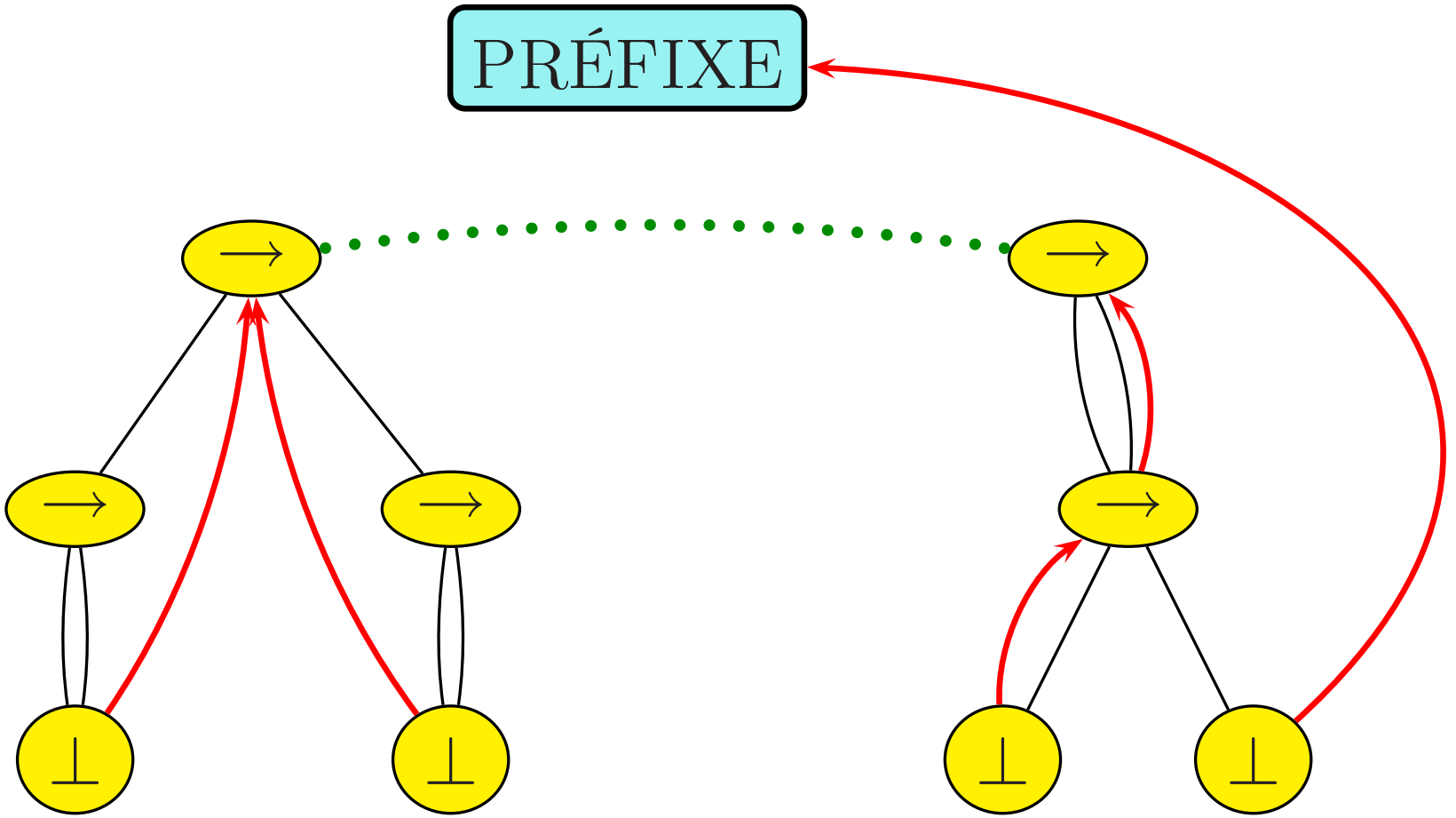


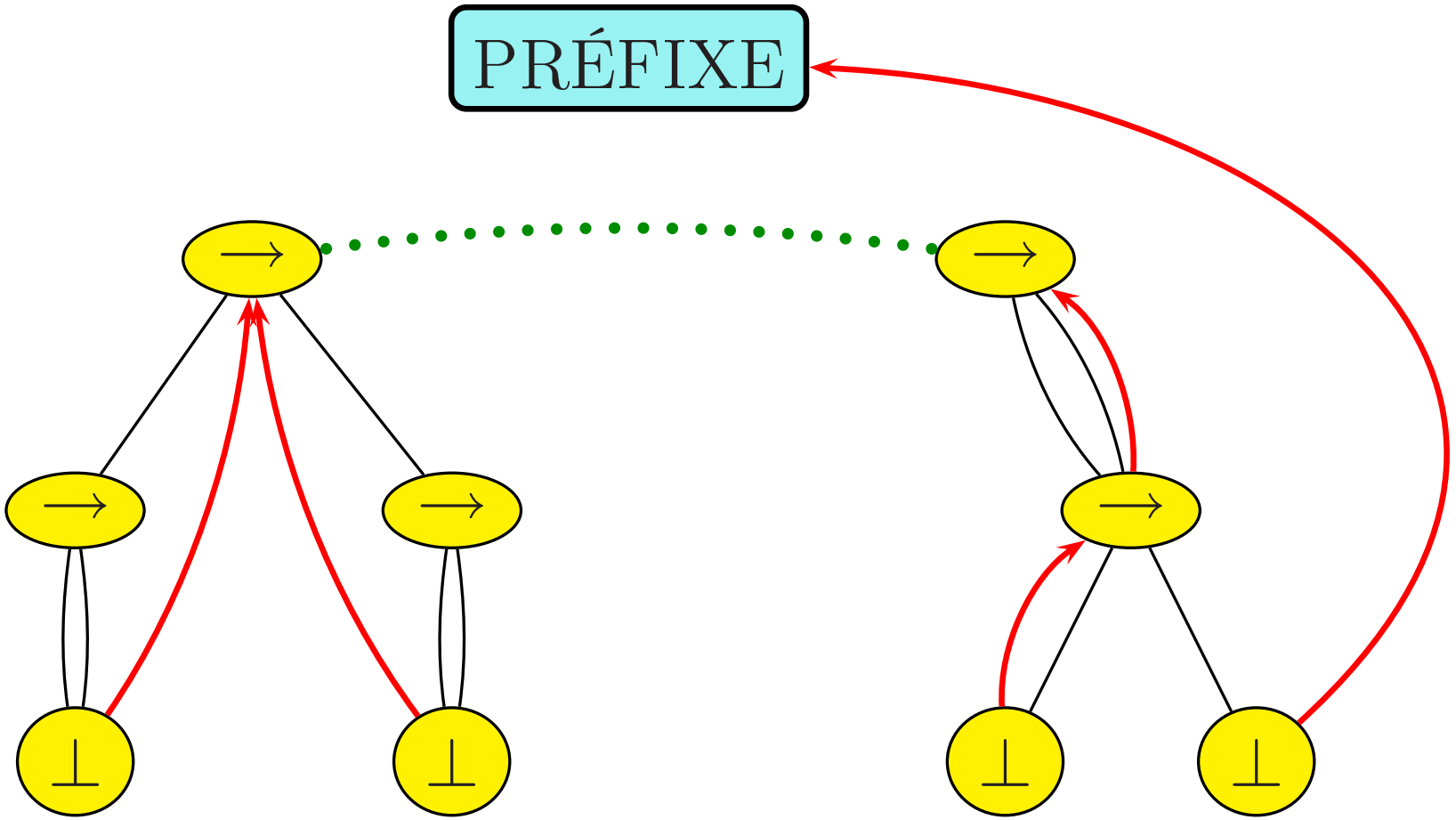
$\forall \alpha_1 \alpha_2.\ (\alpha_1 \to \alpha_1) \to (\alpha_2 \to \alpha_2)$

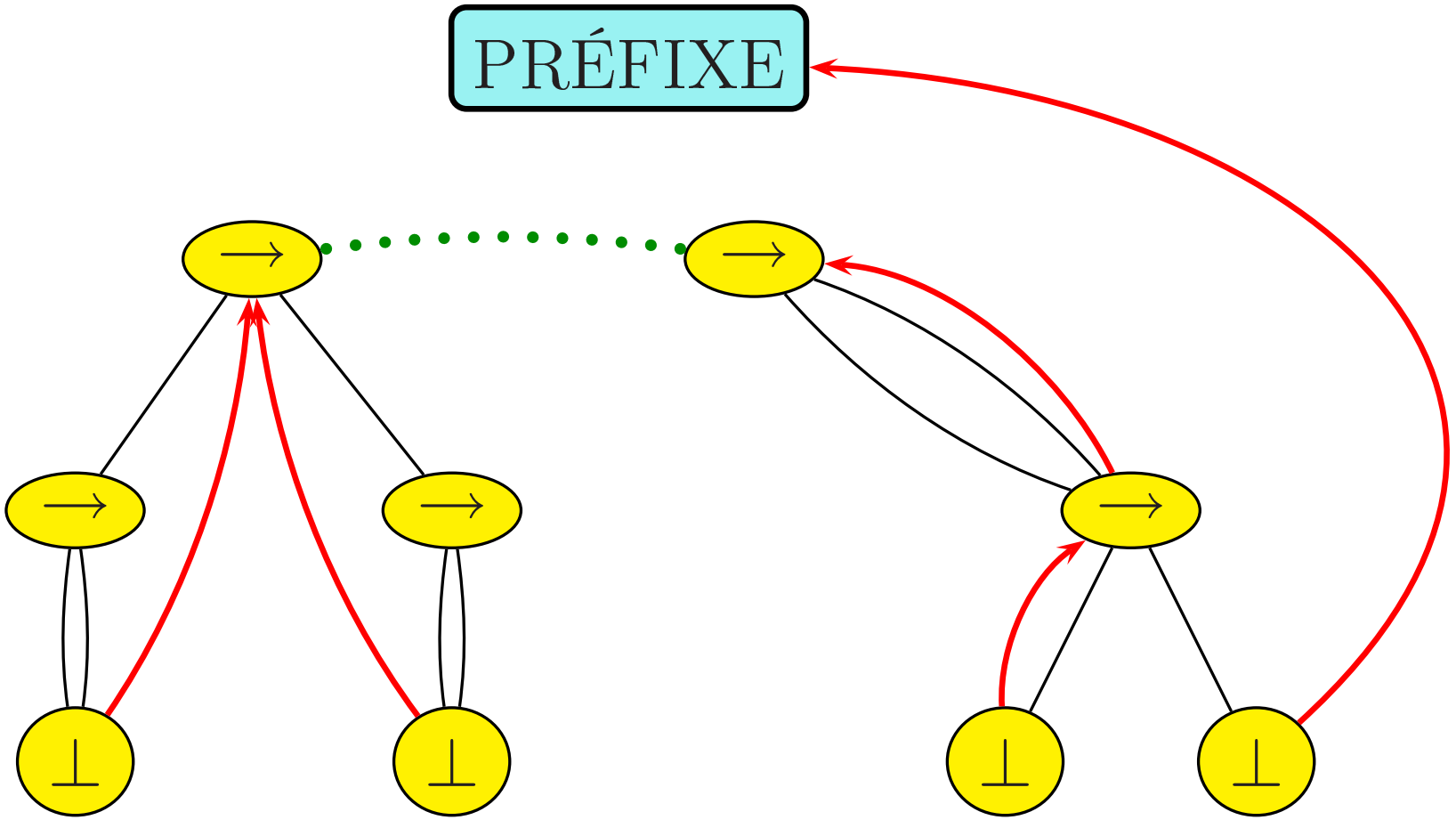
$\forall\,(\beta \geq \forall \gamma.\ \gamma \to \alpha\ )\quad \beta \to \beta$

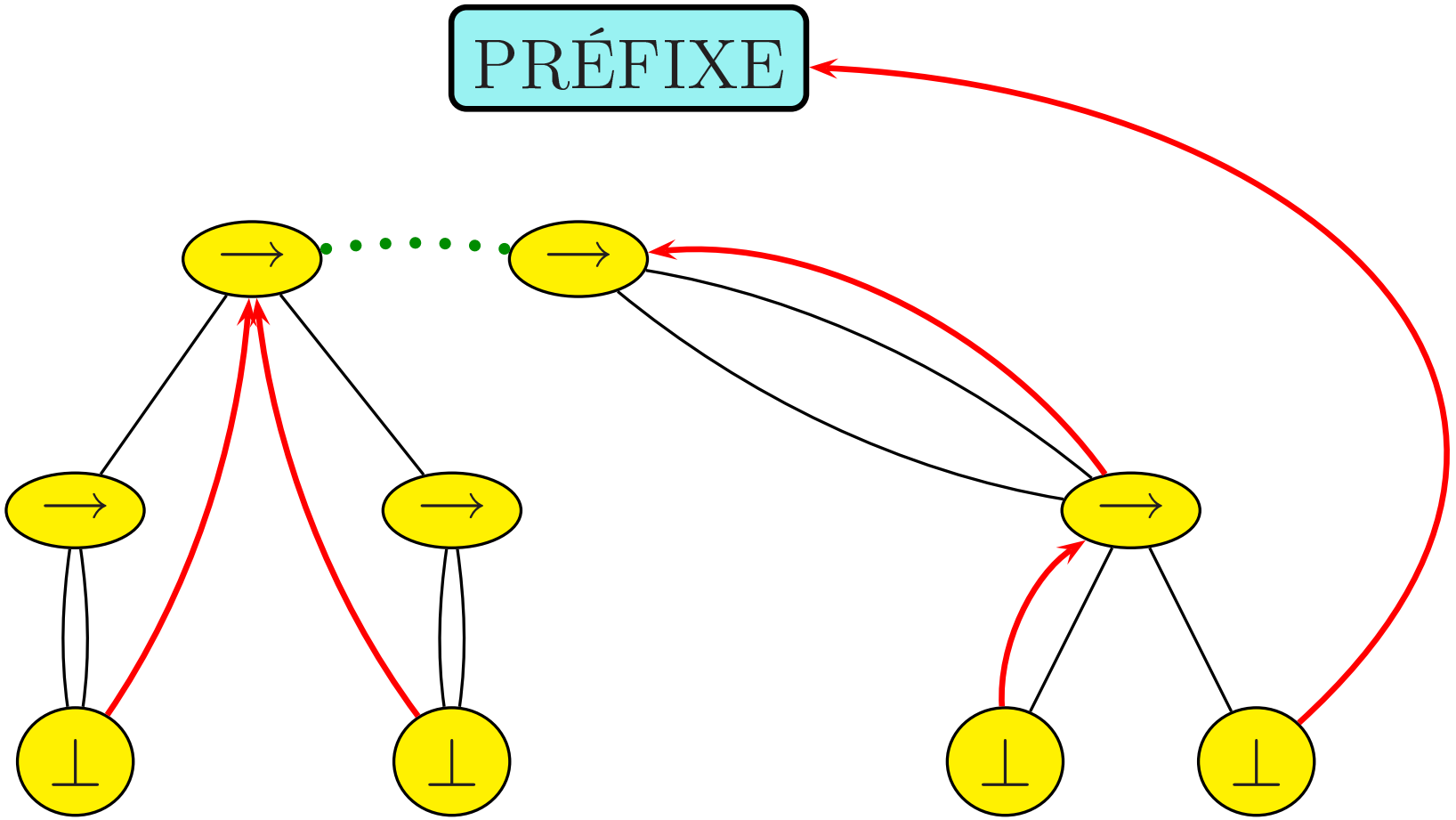
Résultat : $(\alpha \to \alpha) \to (\alpha \to \alpha)$

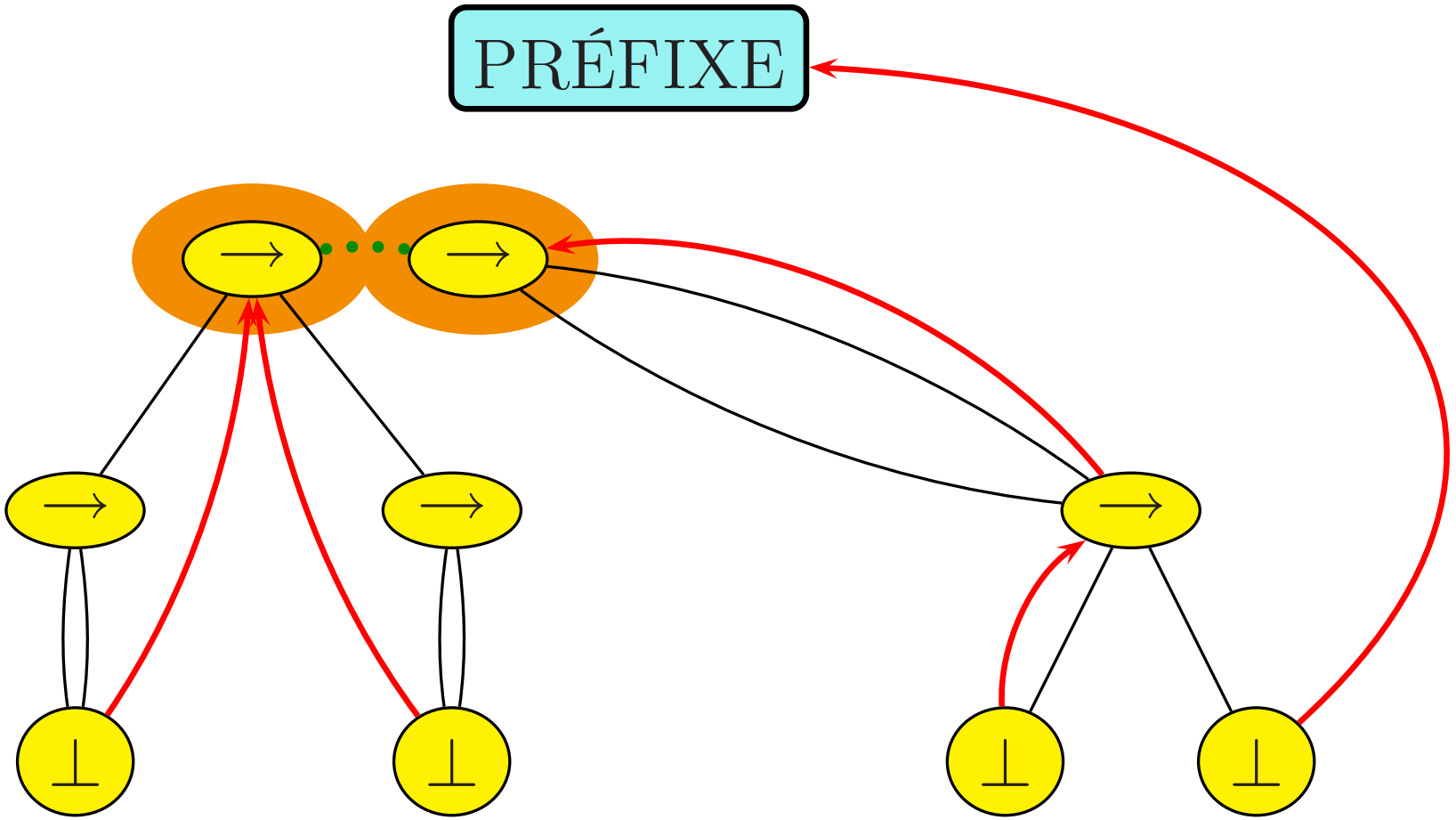

PRÉFIXE

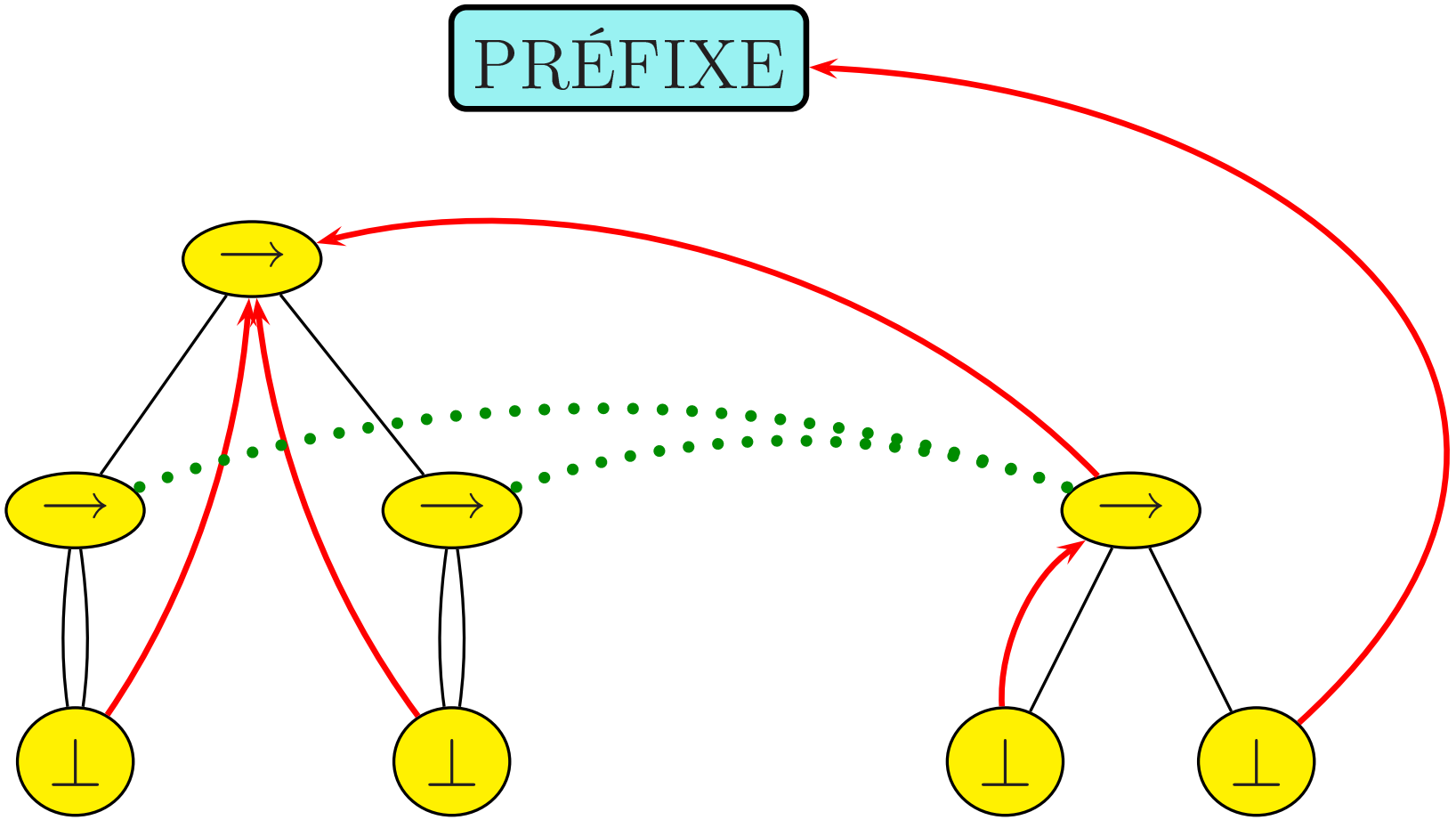

PRÉFIXE

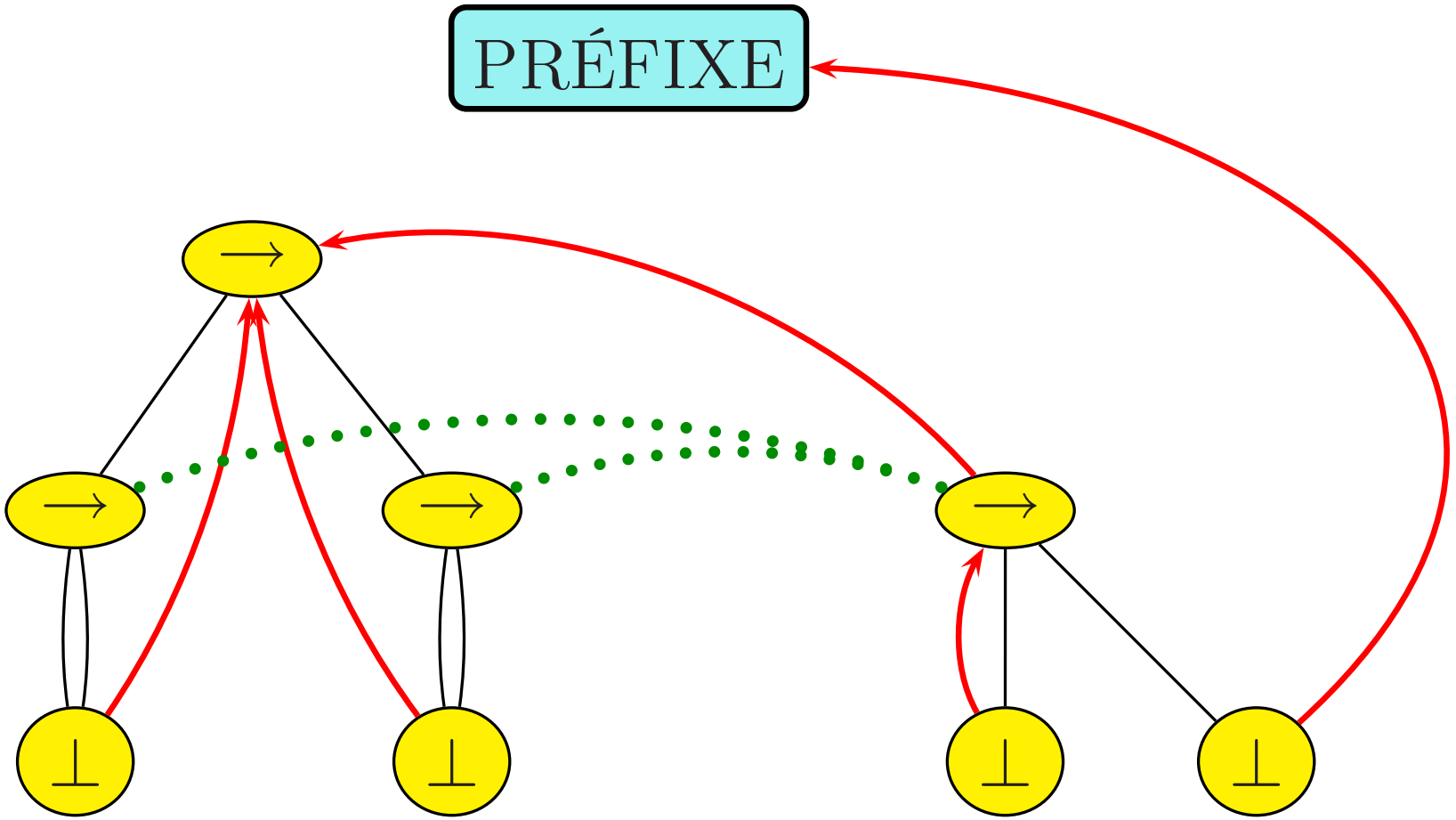

PRÉFIXE

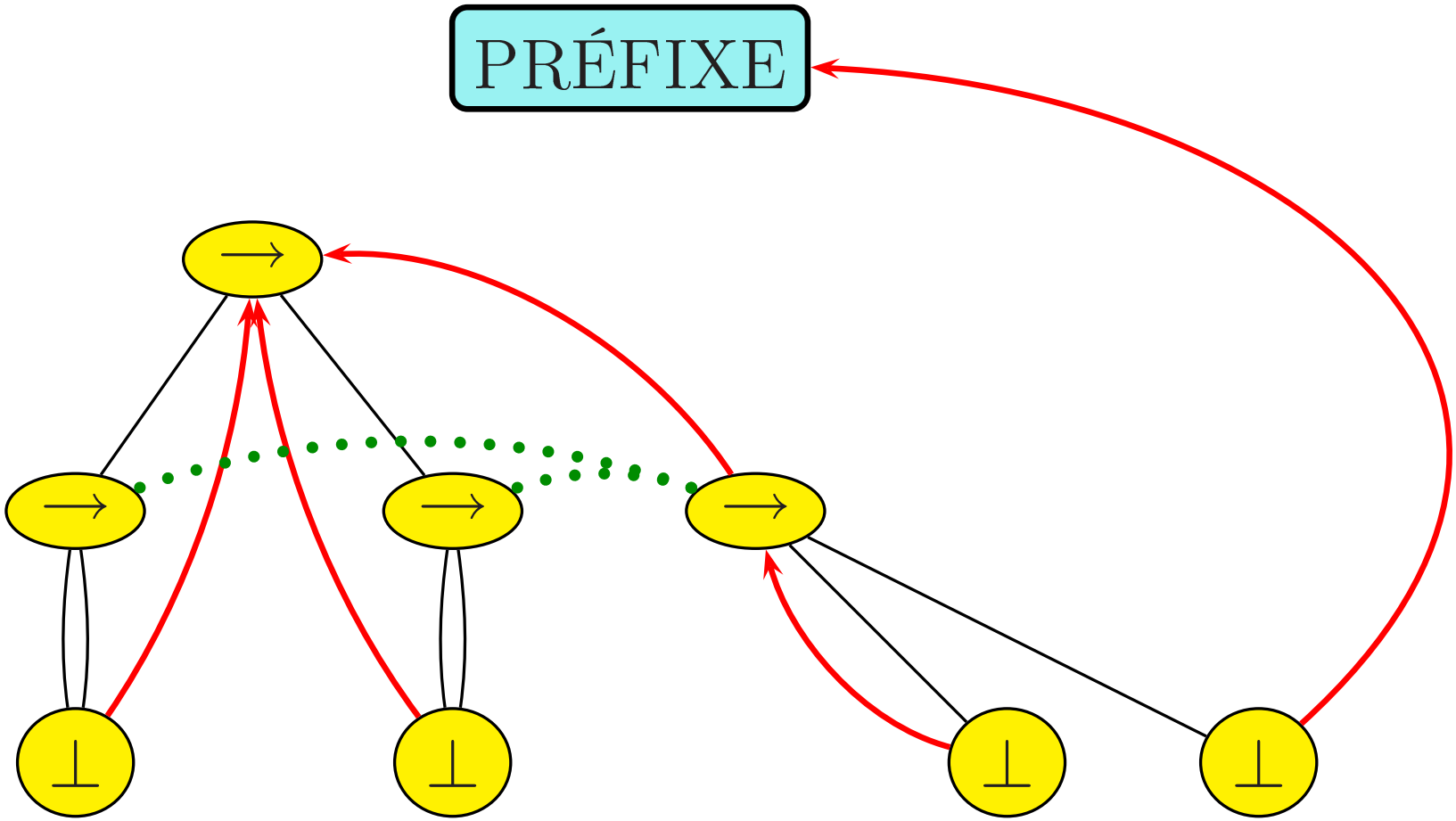

PRÉFIXE

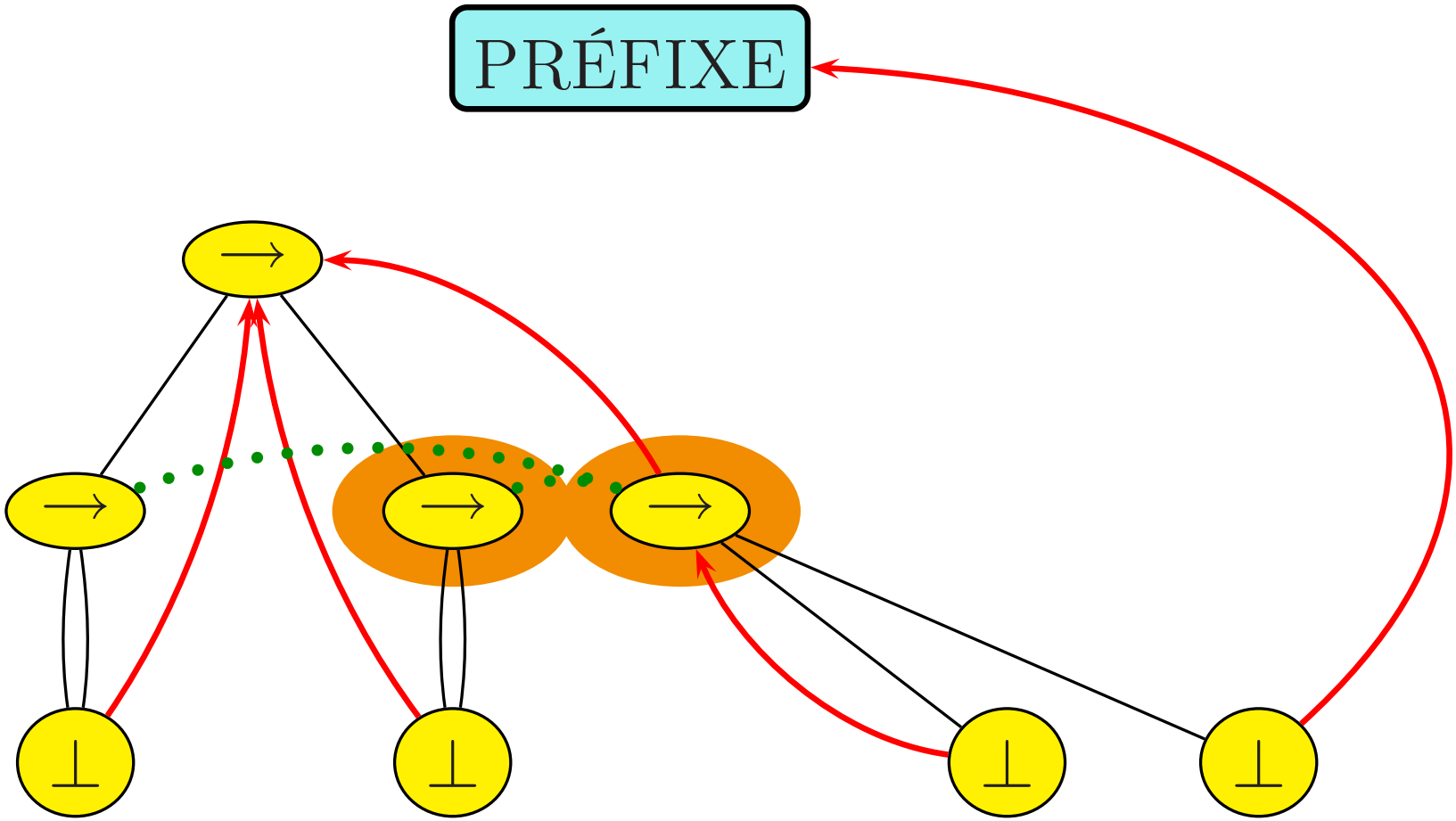

PRÉFIXE

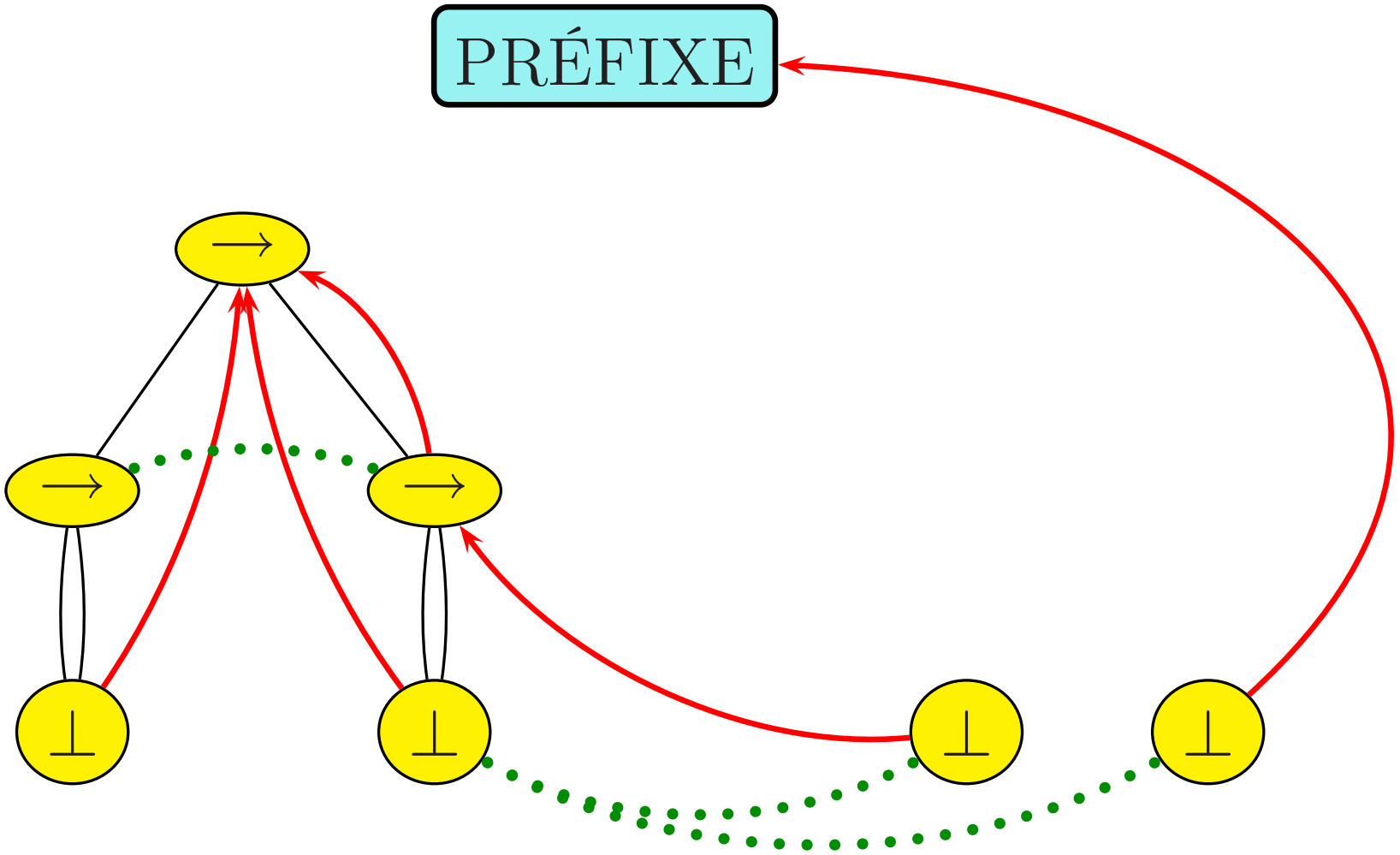

PRÉFIXE

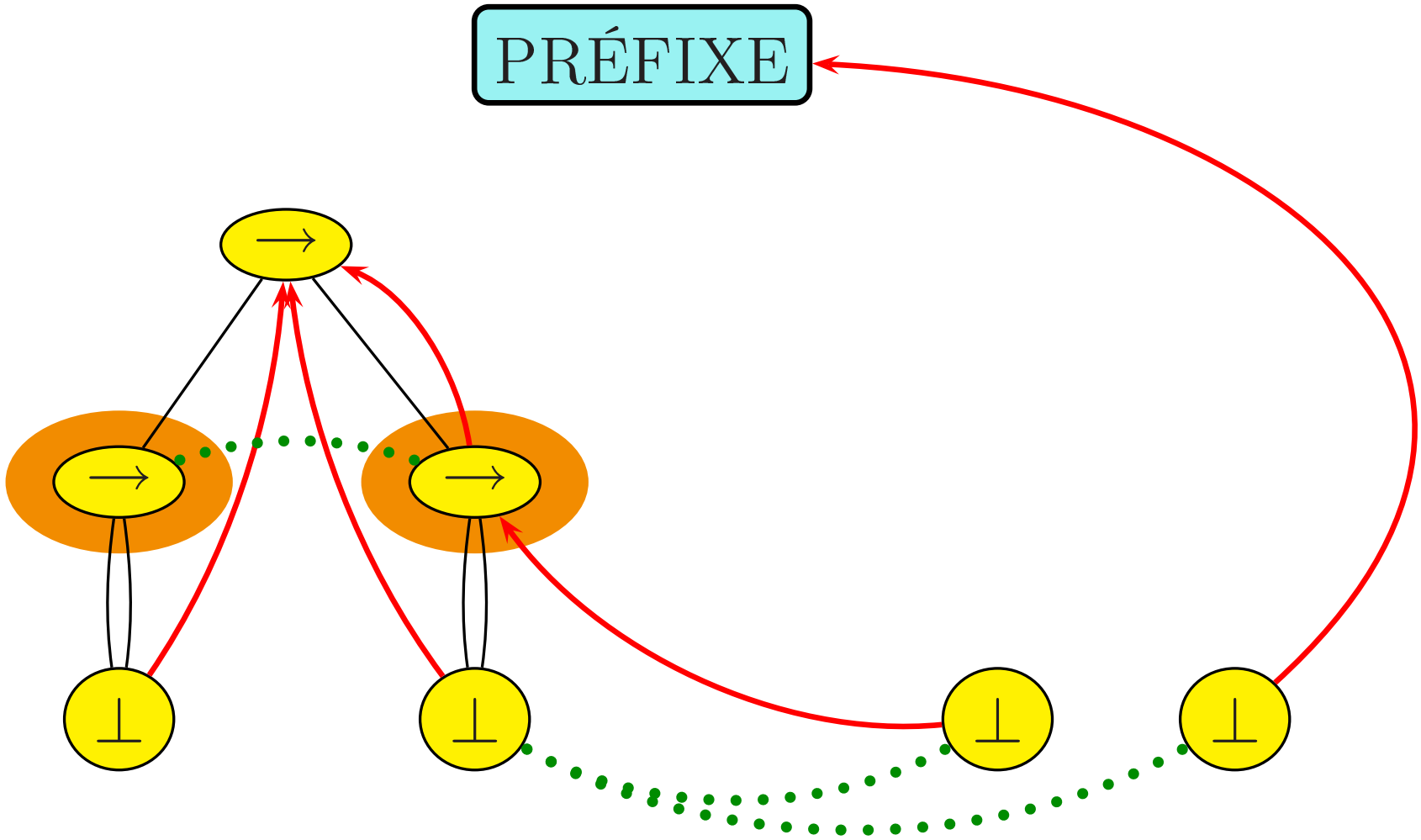

PRÉFIXE

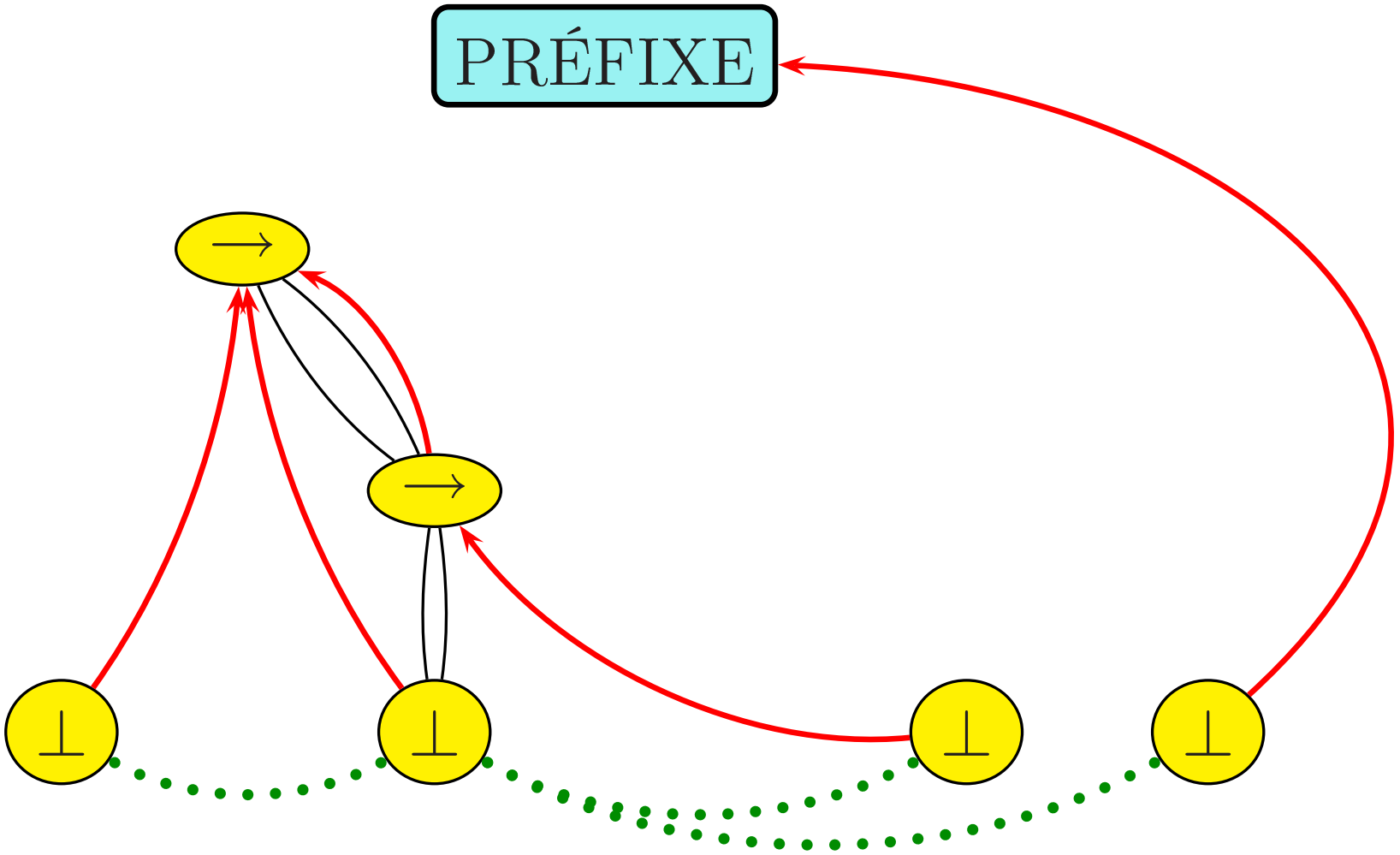

PRÉFIXE

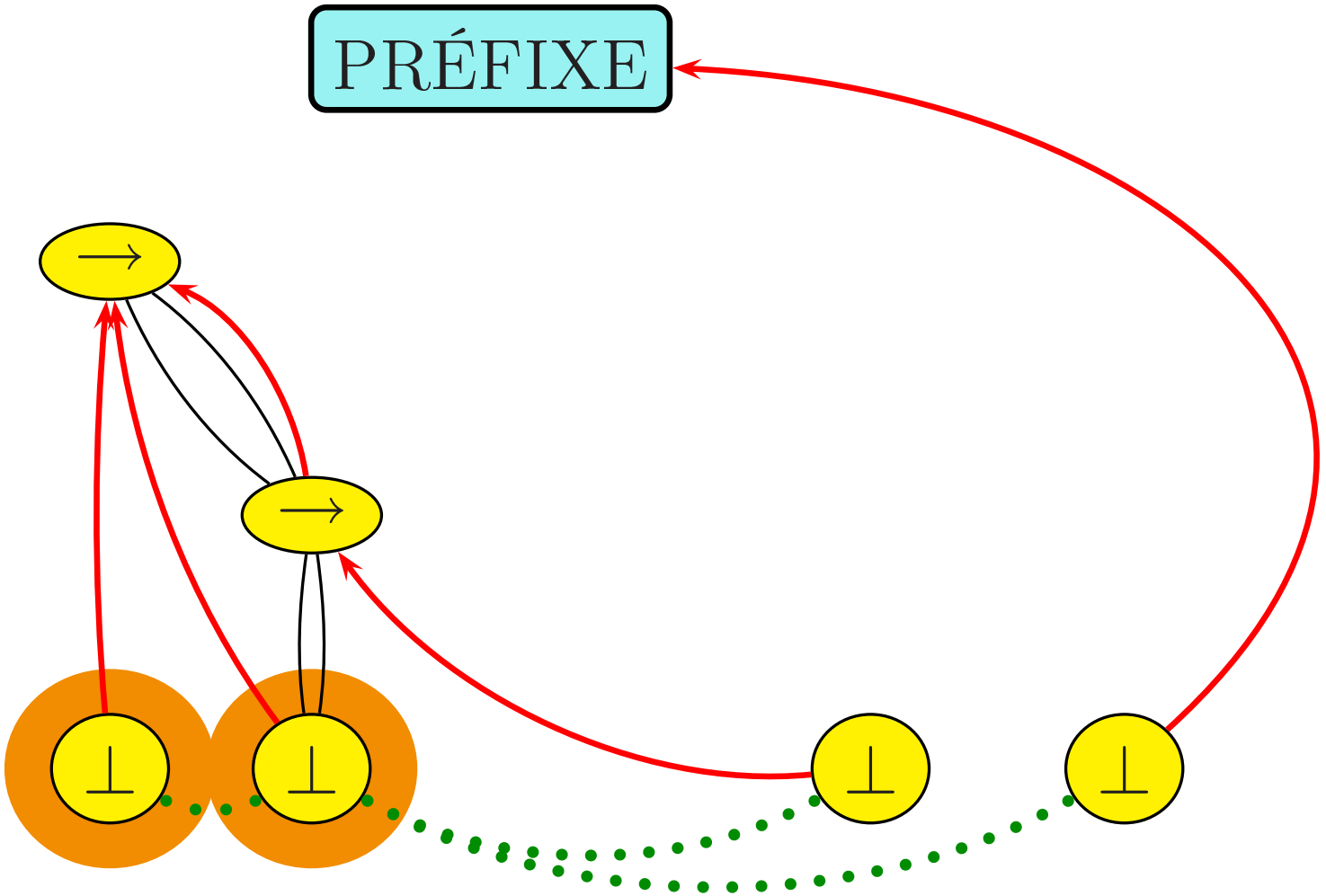

PRÉFIXE

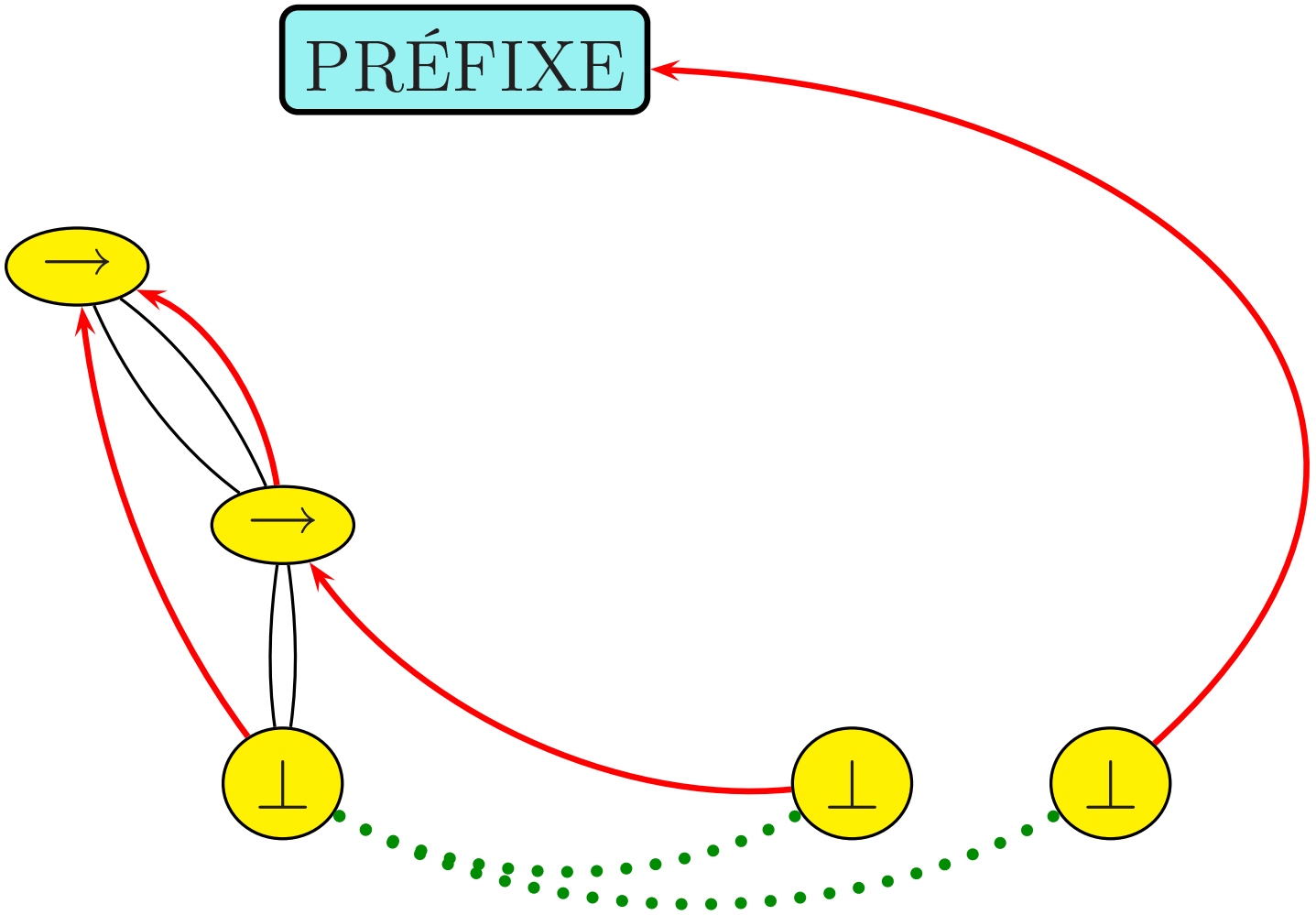

PRÉFIXE

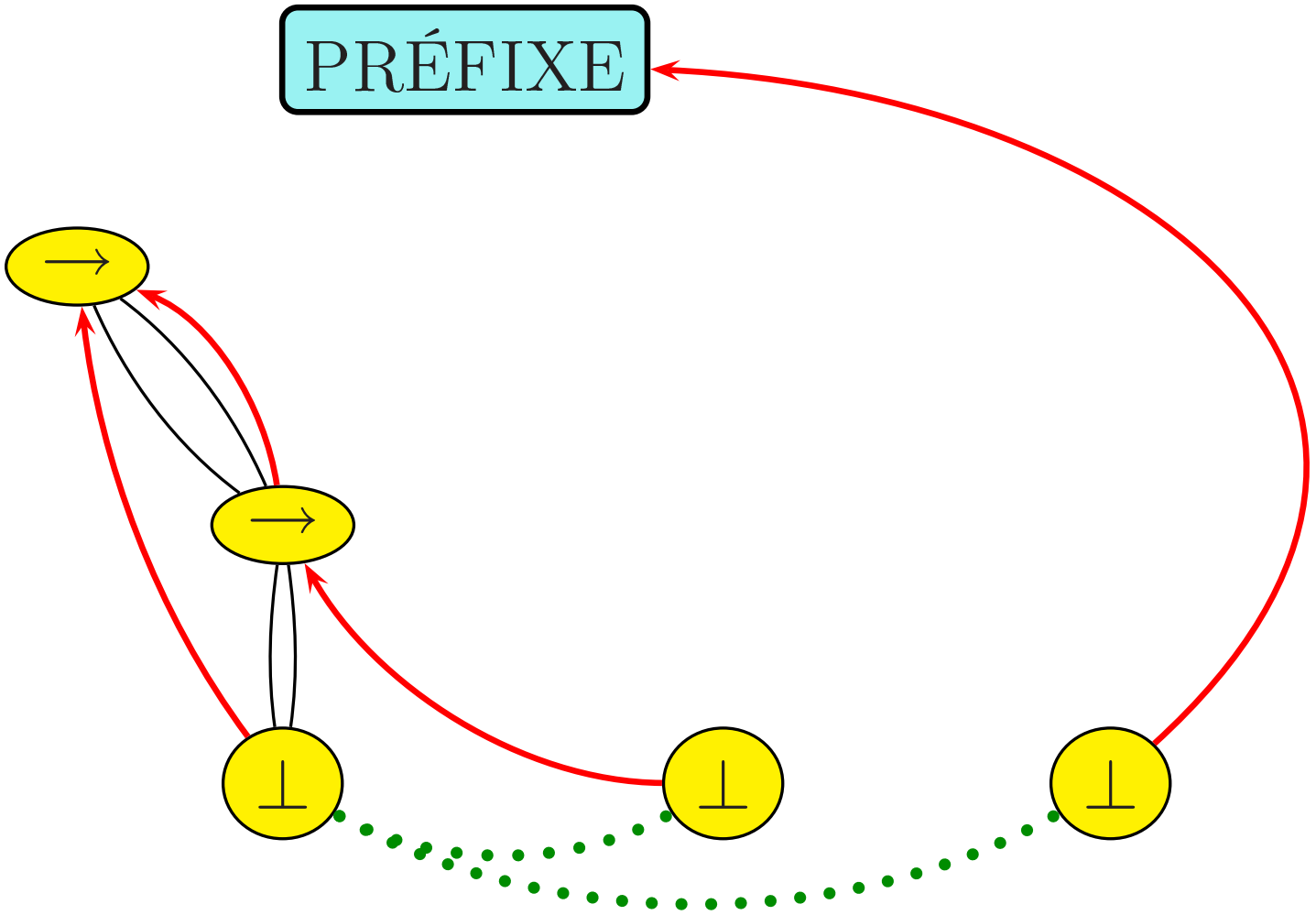

PRÉFIXE

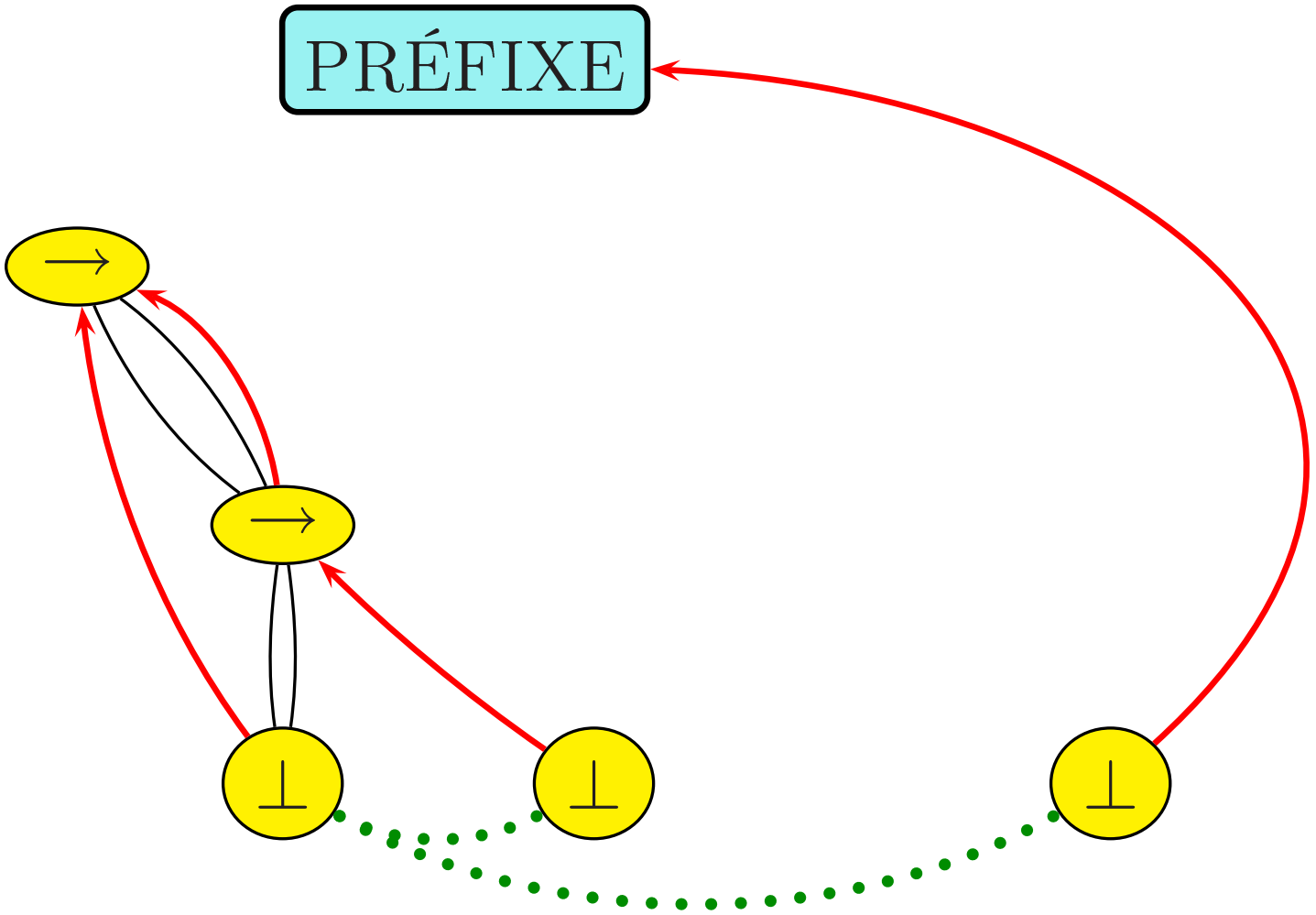

PRÉFIXE

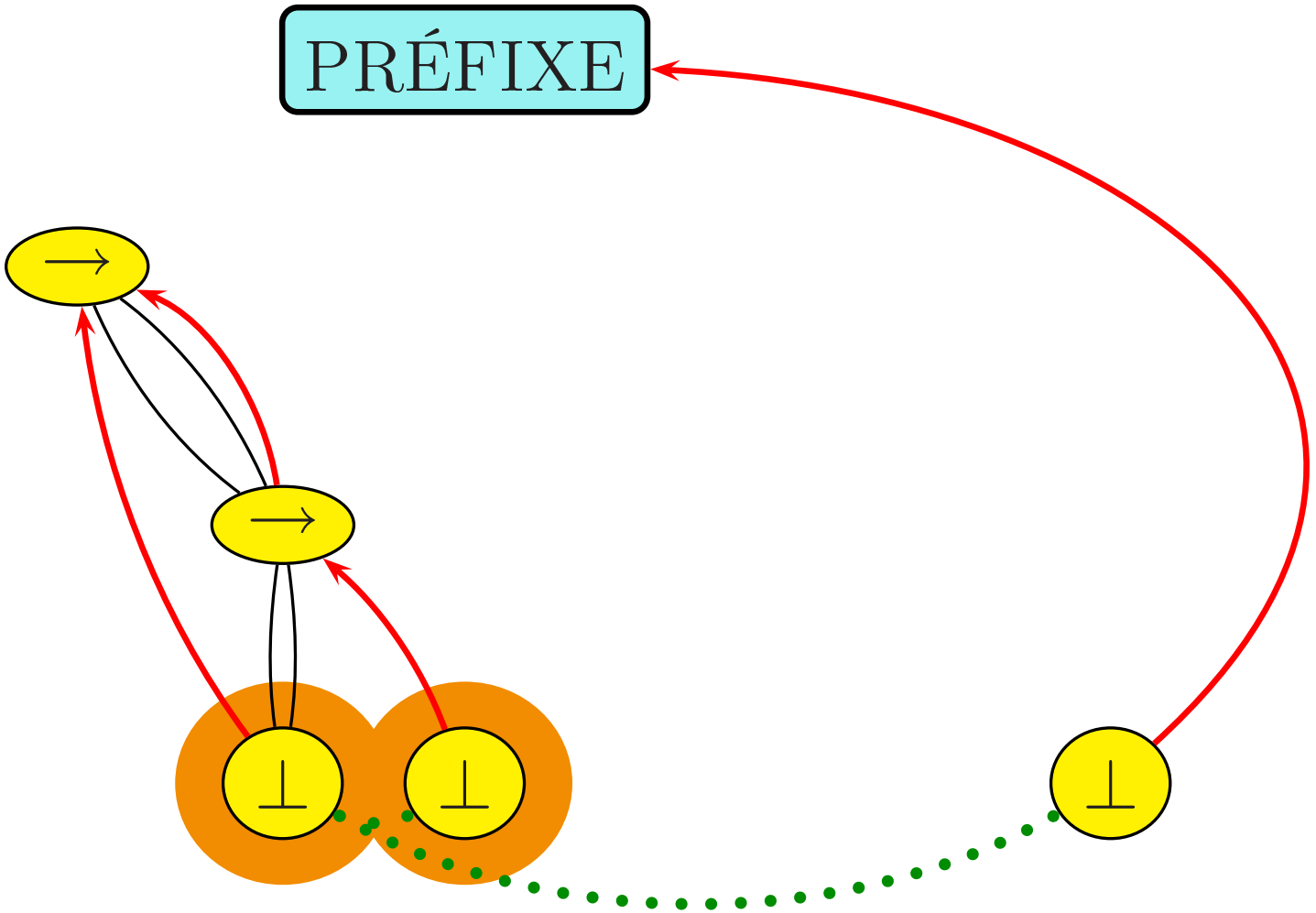

PRÉFIXE
→
→
⊥
⊥
⊥
⊥

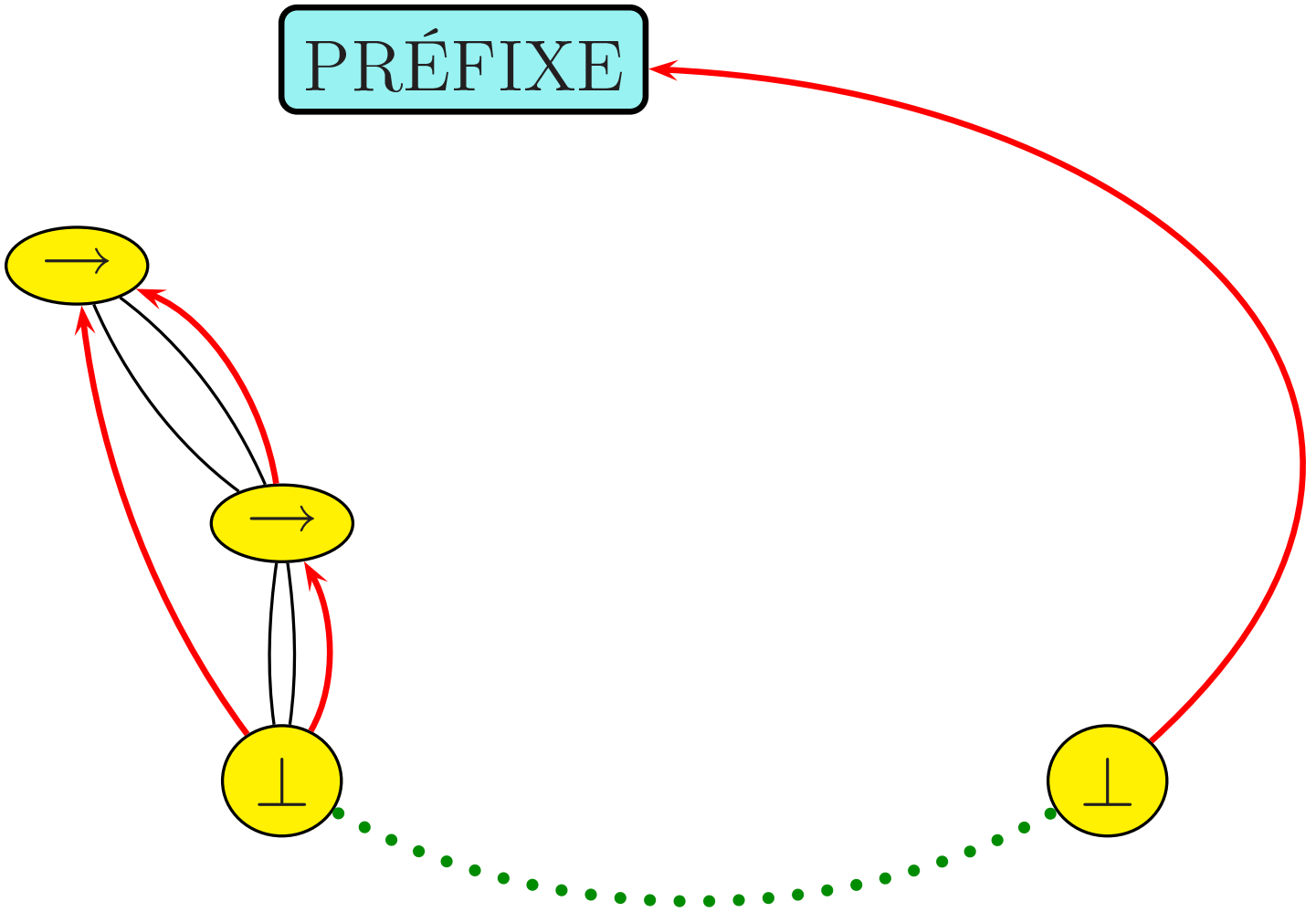

PRÉFIXE

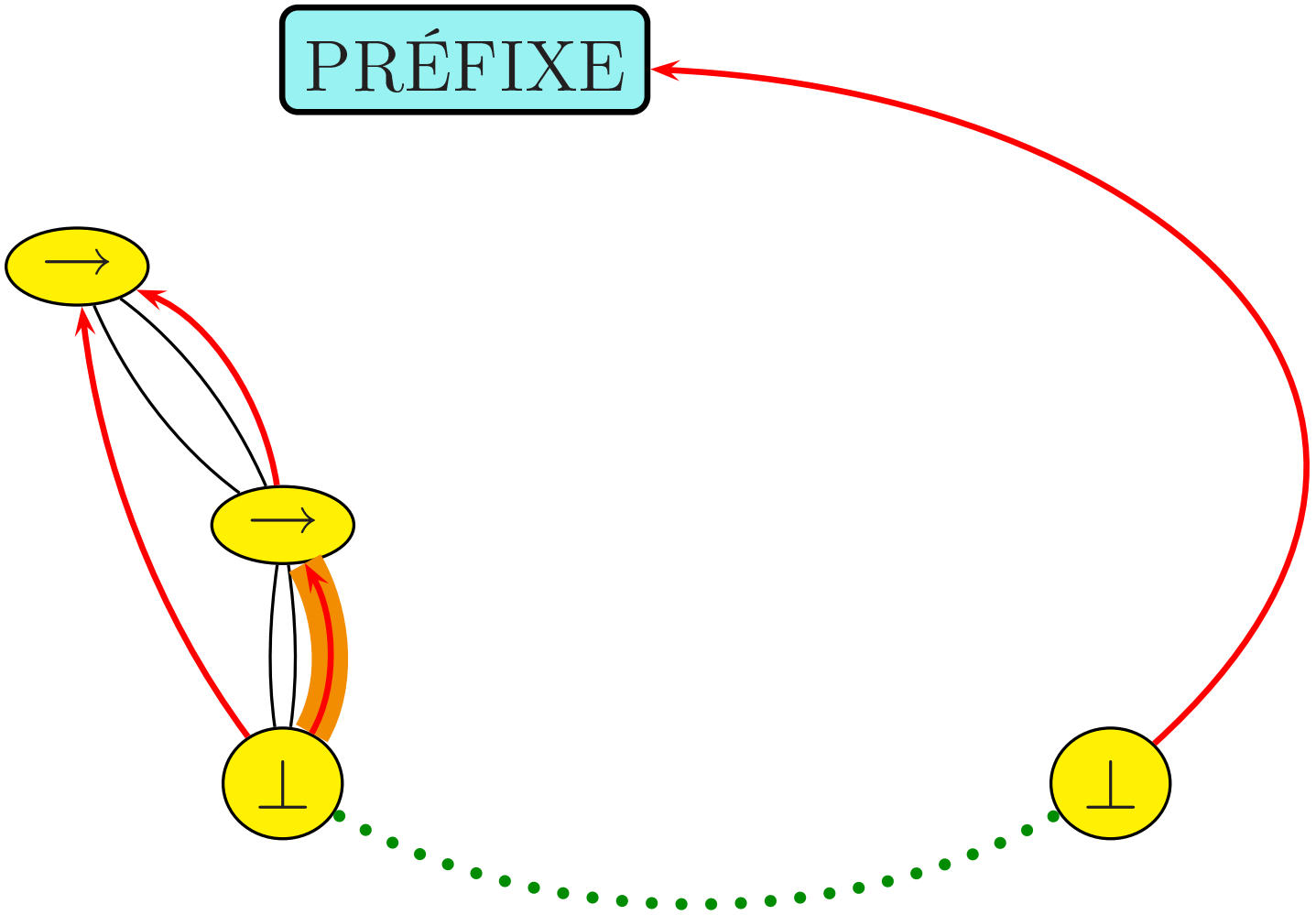

PRÉFIXE
→
→
⊥
⊥
⊥

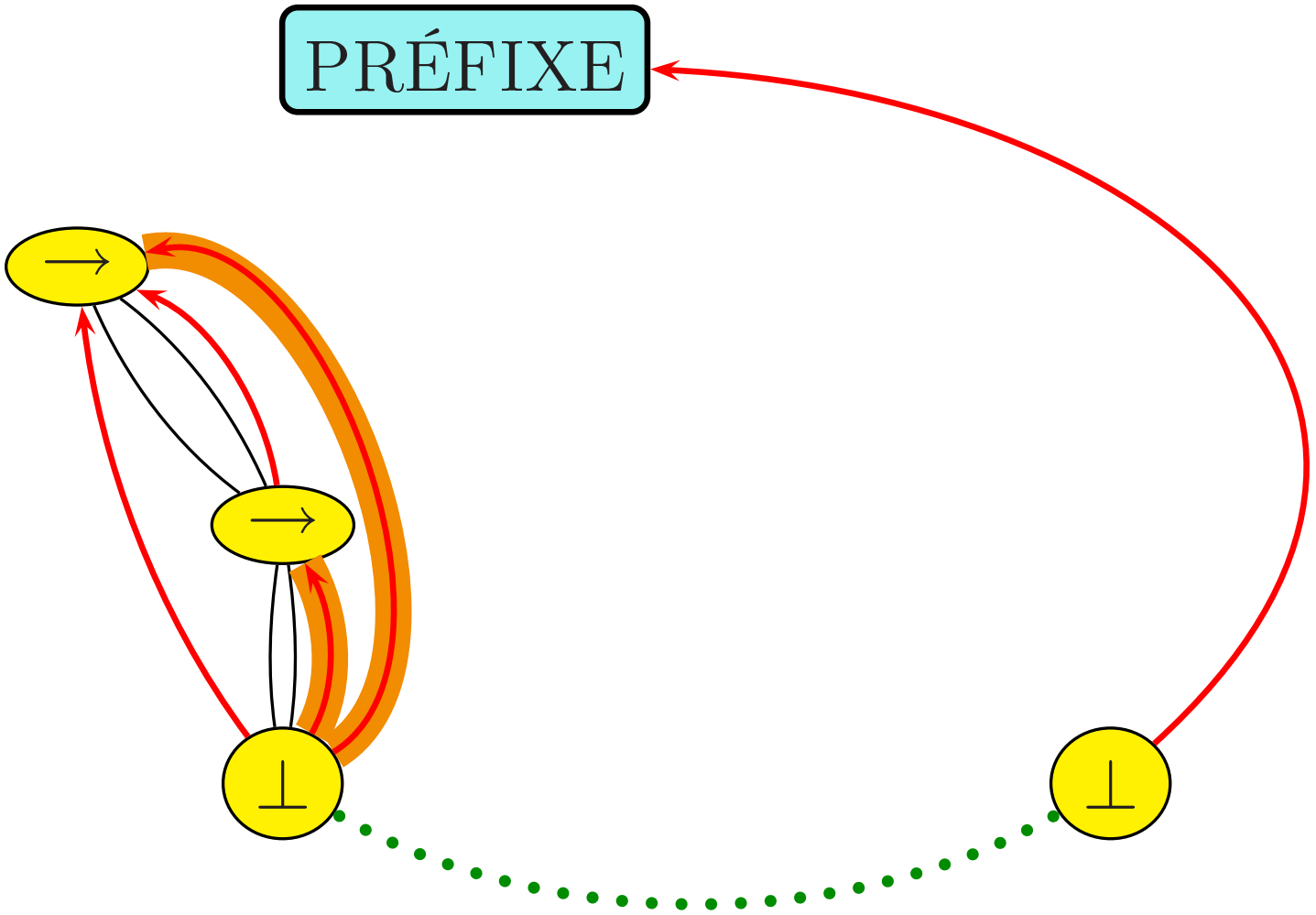

PRÉFIXE
→
→
⊥
⊥
⊥

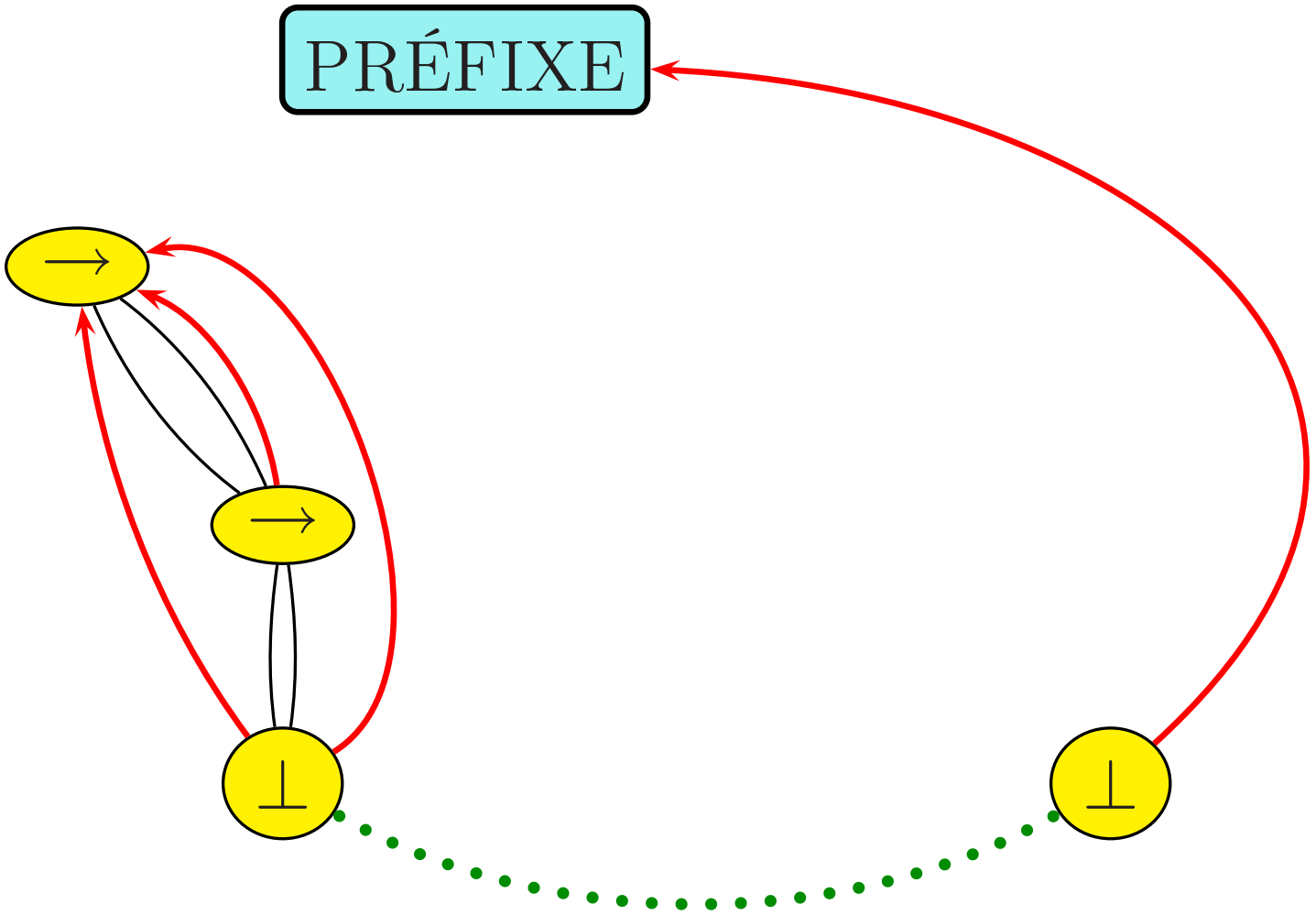

PRÉFIXE
→
→
⊥
⊥
⊥

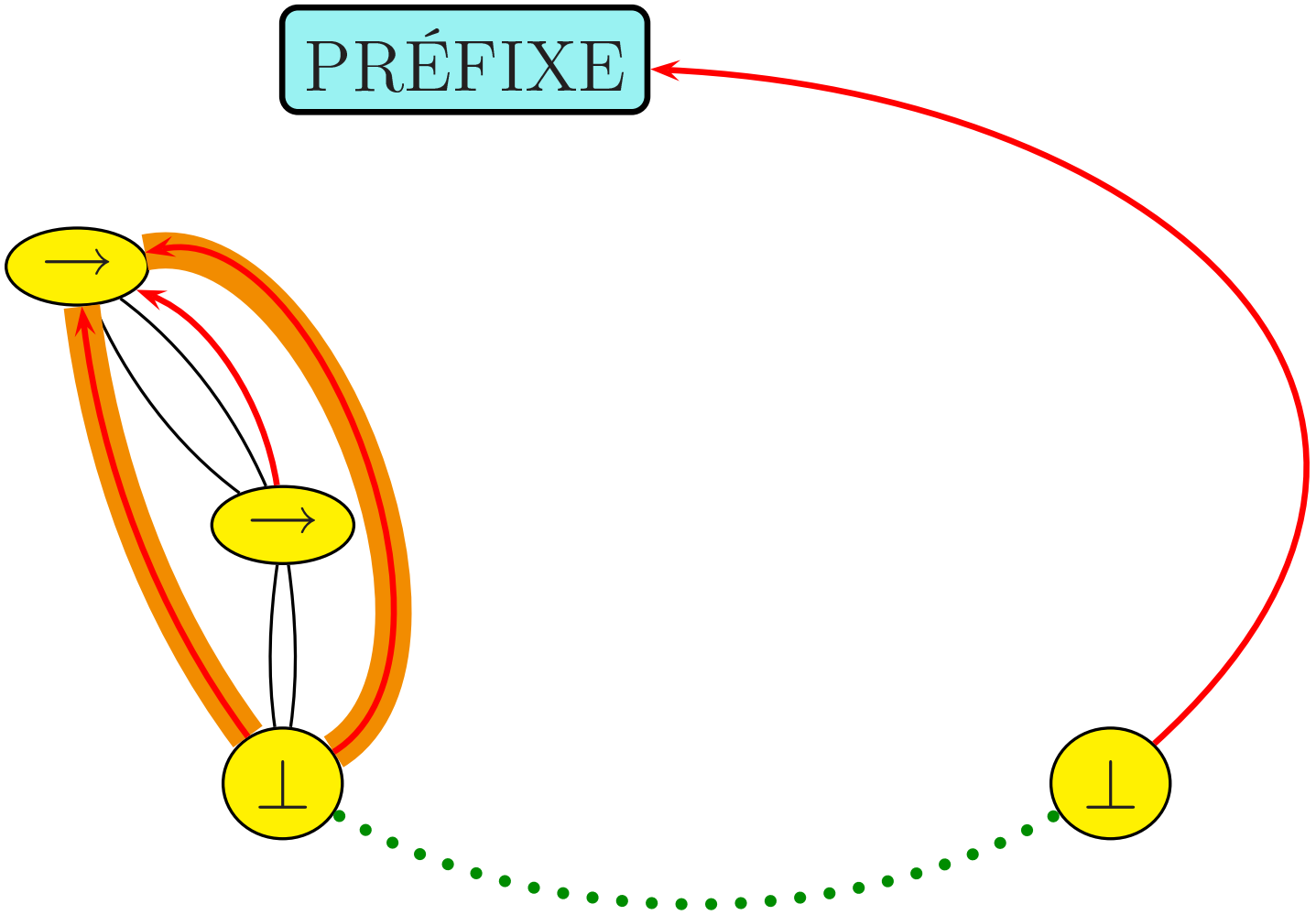

PRÉFIXE

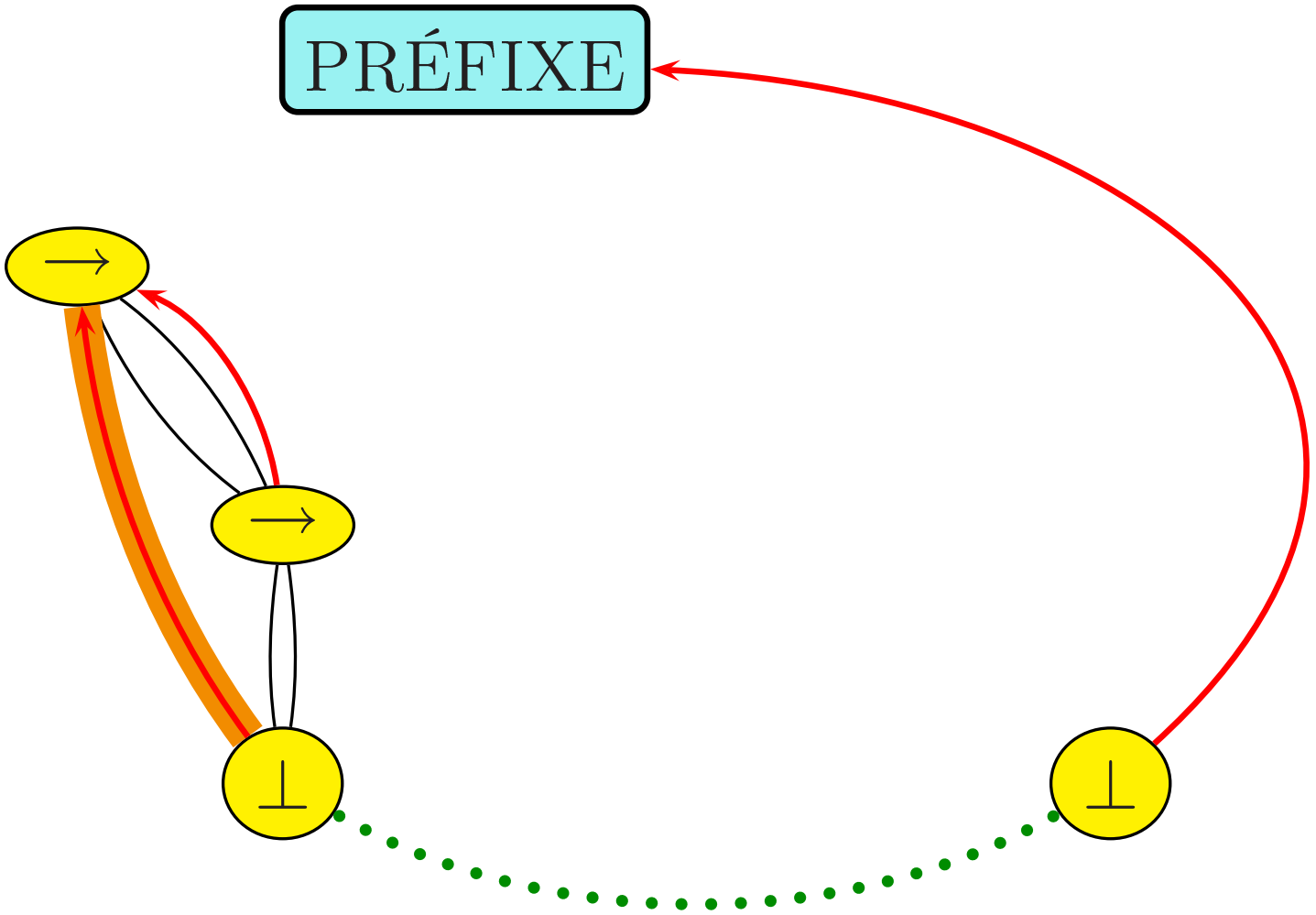

PRÉFIXE
→
→
⊥
⊥

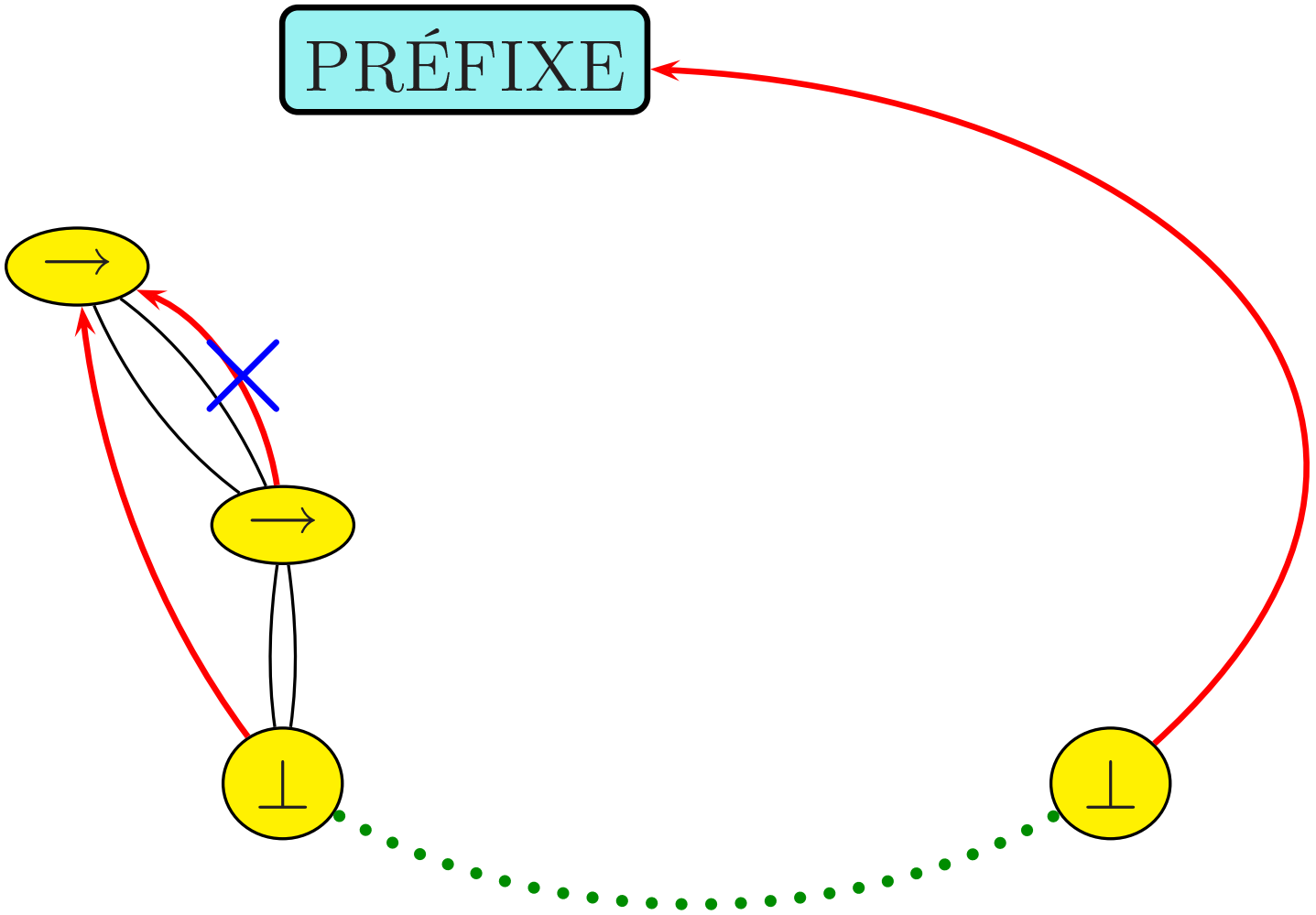

PRÉFIXE
→
→
⊥
⊥

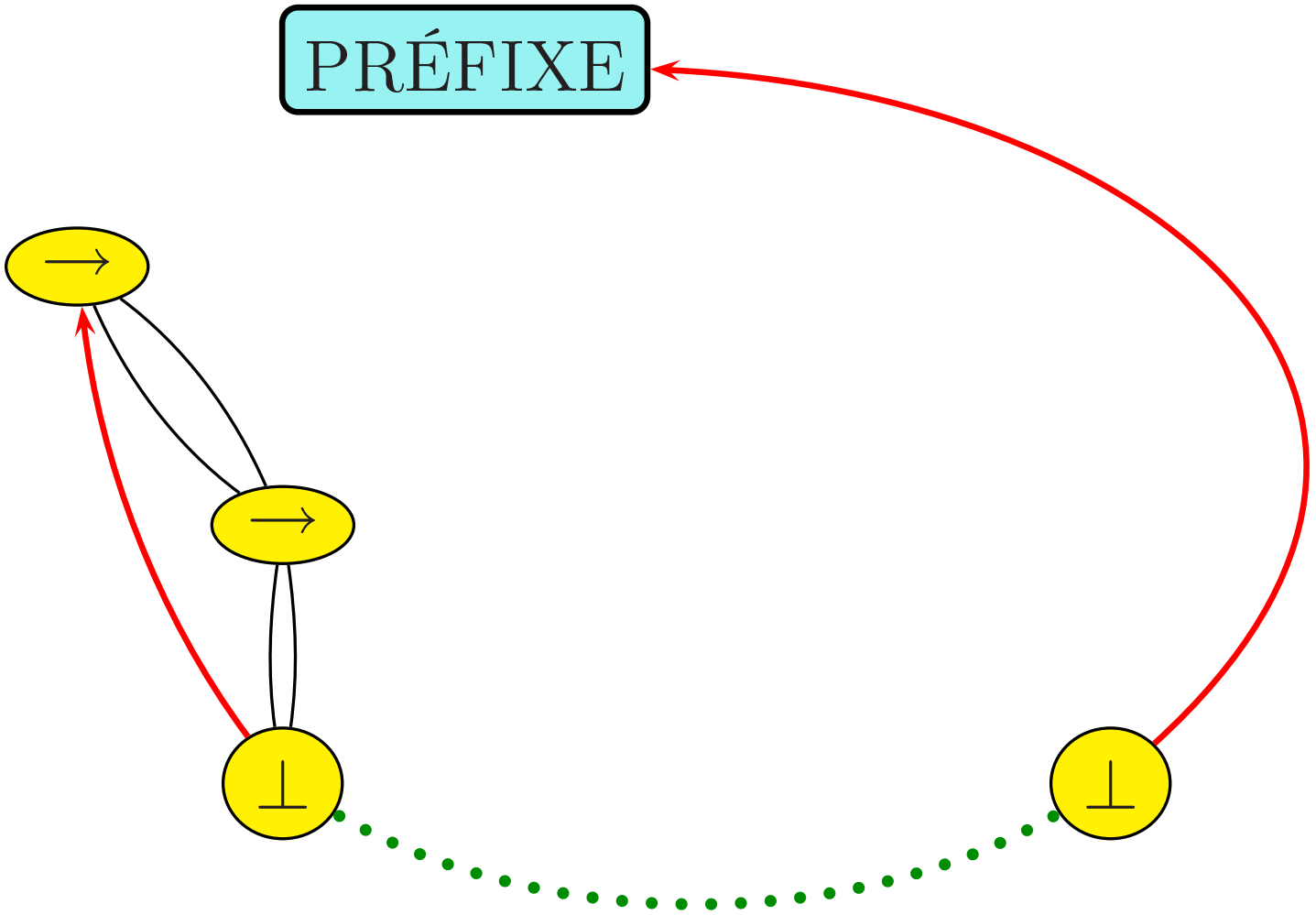

PRÉFIXE
→
→
⊥
⊥

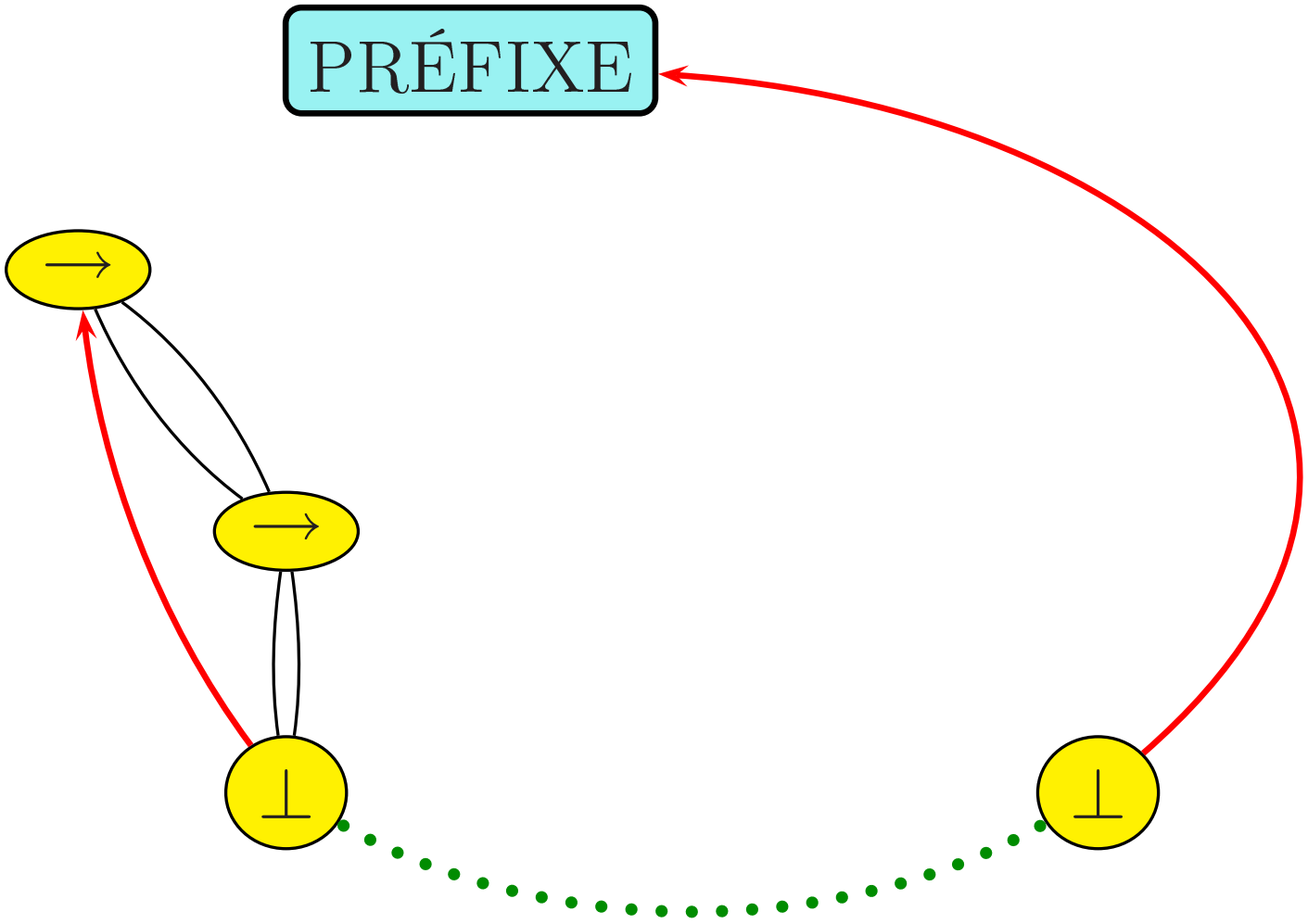

PRÉFIXE
→
→
⊥
⊥

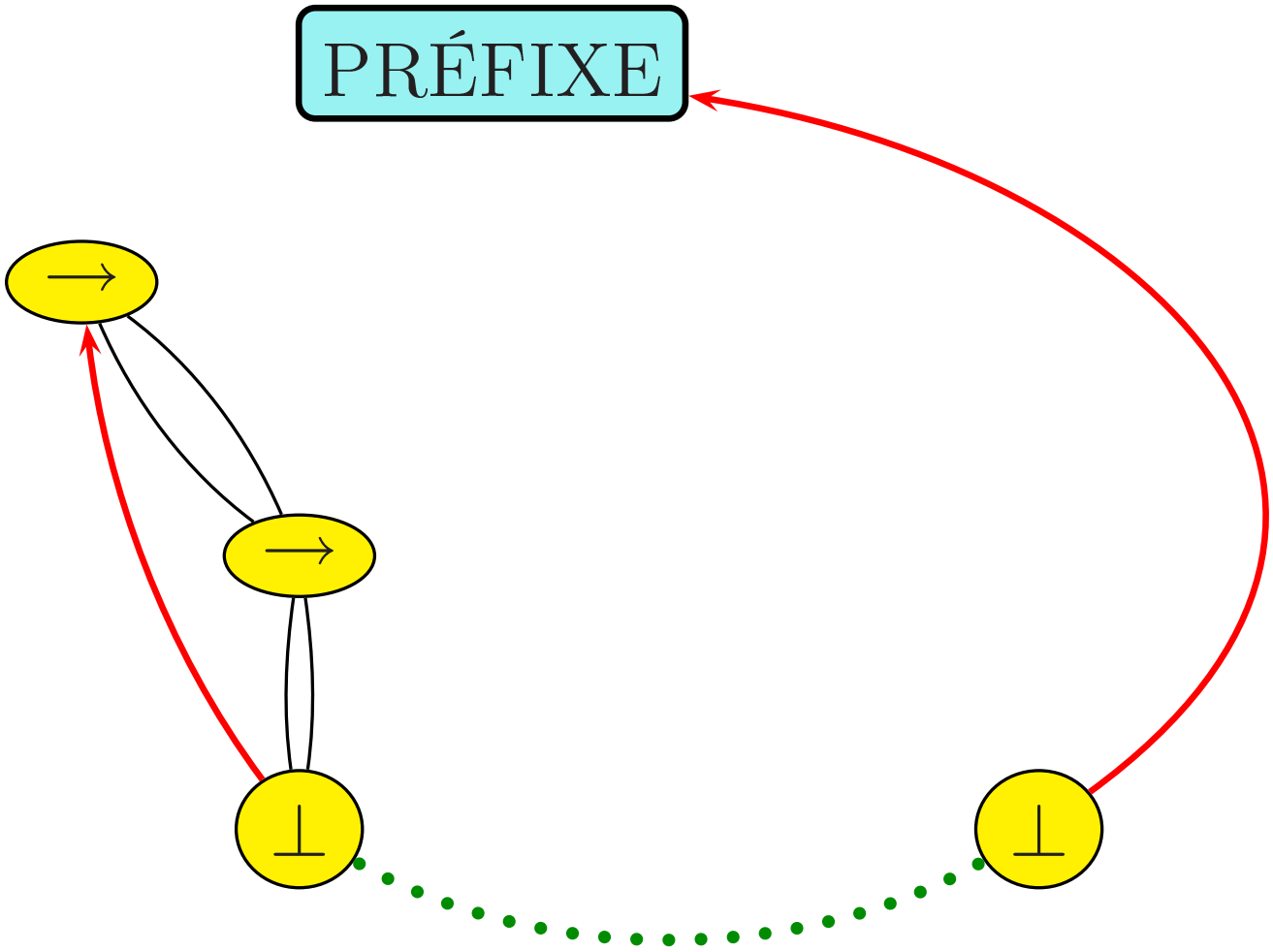

PRÉFIXE
→
→
⊥
⊥

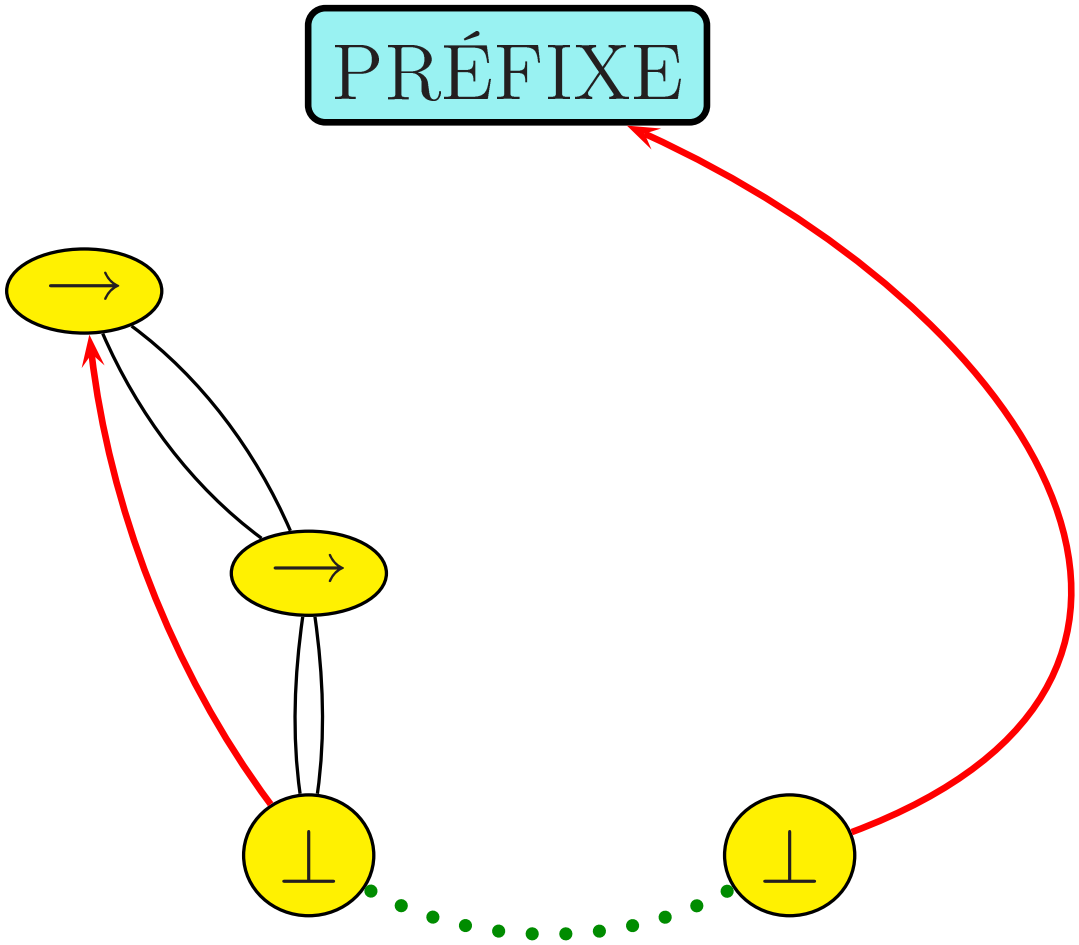

PRÉFIXE

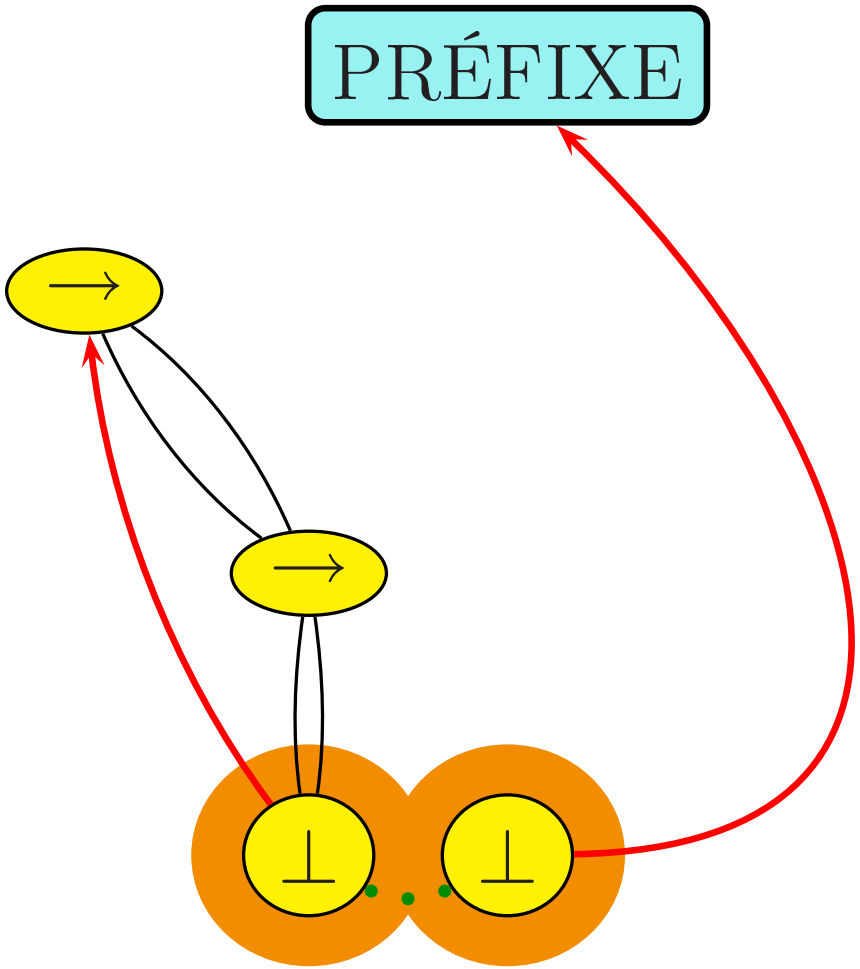

PRÉFIXE

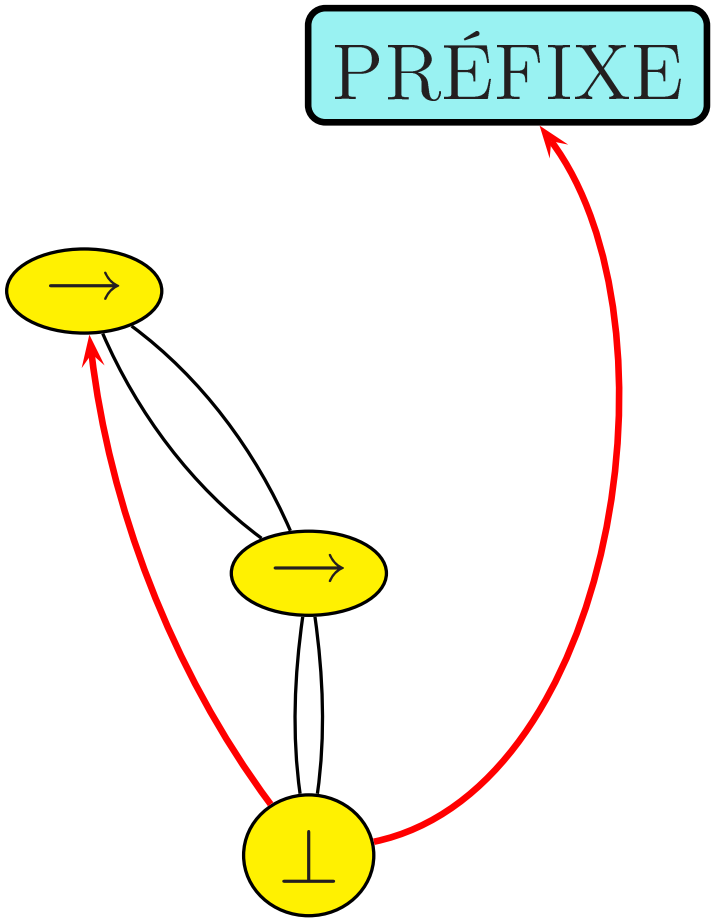

PRÉFIXE
→
→
⊥
⊥

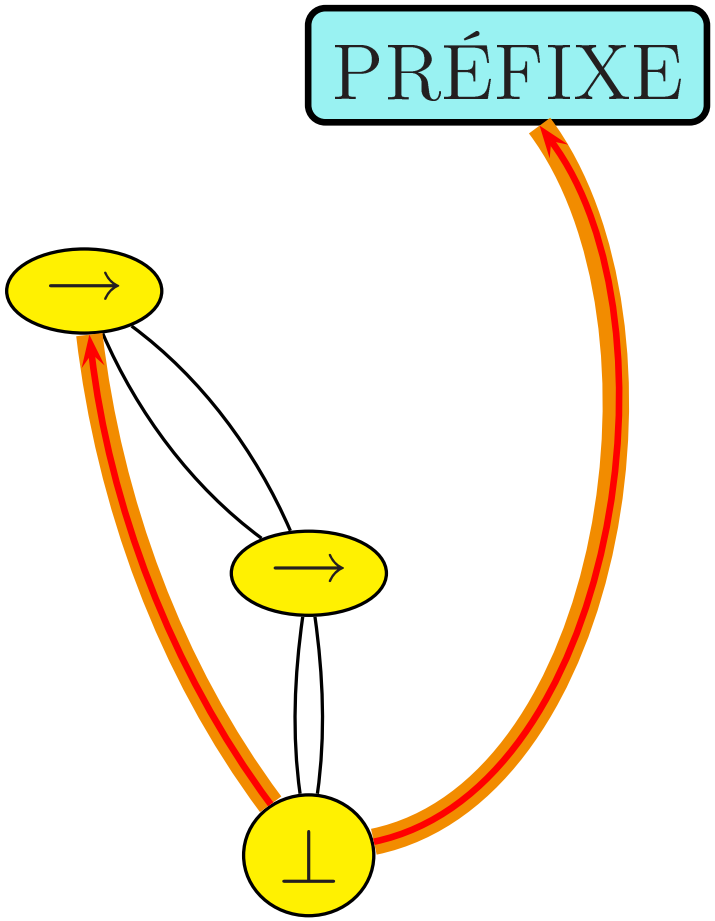

PRÉFIXE
→
→
⊥
⊥

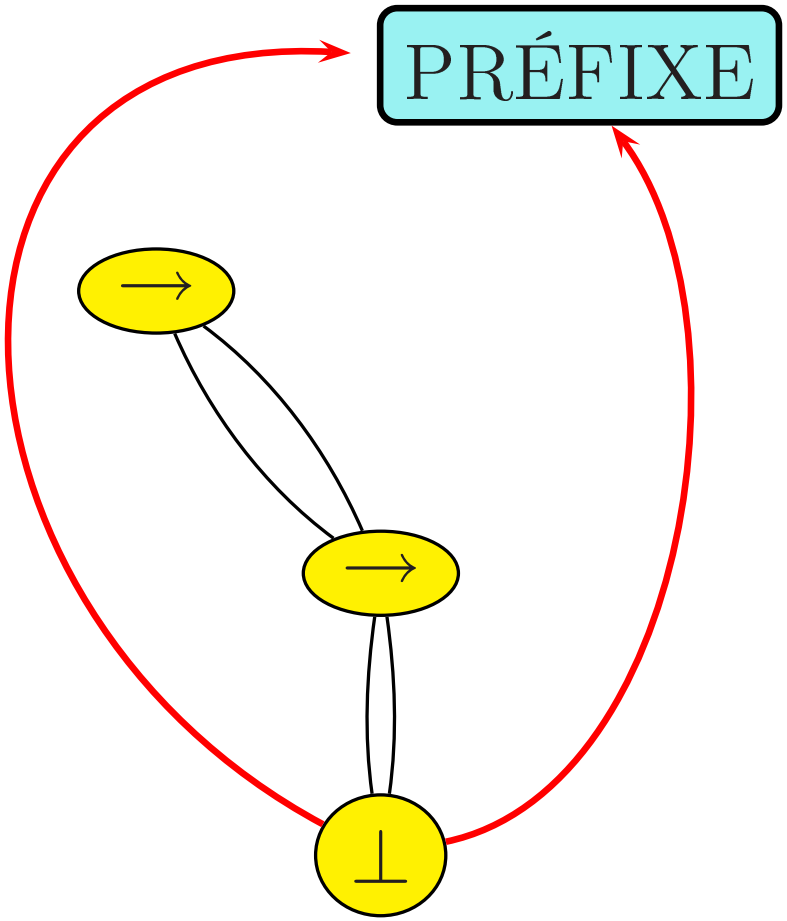

PRÉFIXE
→
→
⊥
⊥

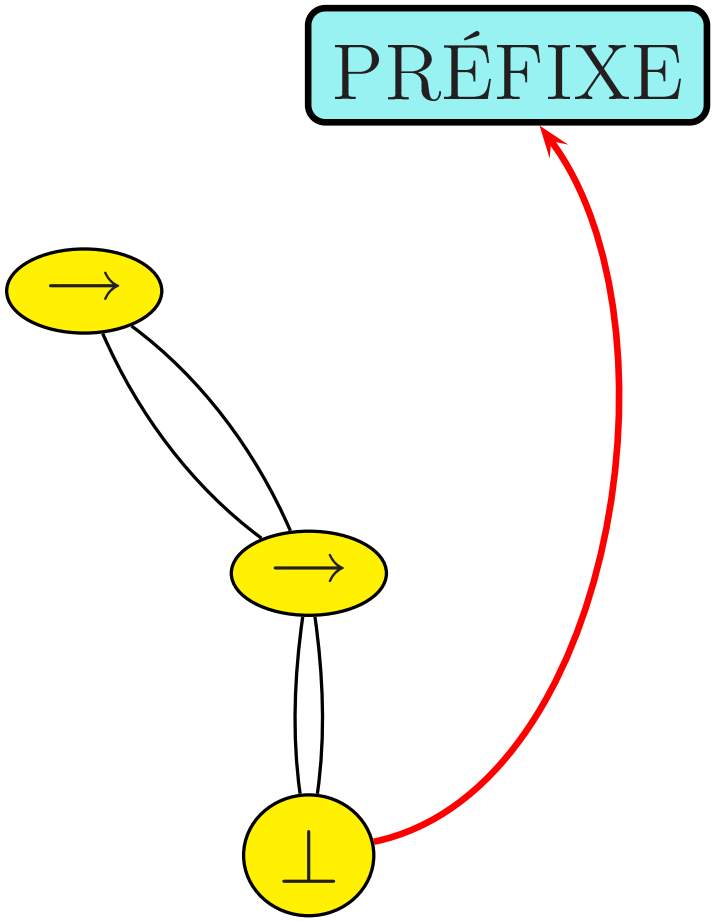

PRÉFIXE
→
→
⊥
⊥

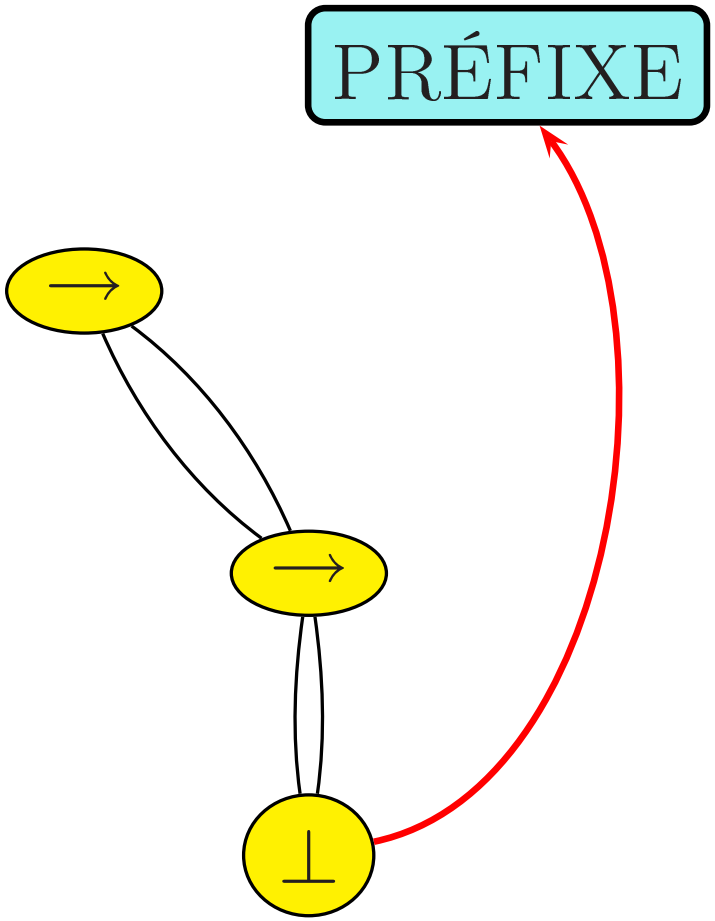

PRÉFIXE
→
→
⊥
⊥


PRÉFIXE
→
→
⊥


PRÉFIXE
→
→
⊥


PRÉFIXE
→
→
⊥

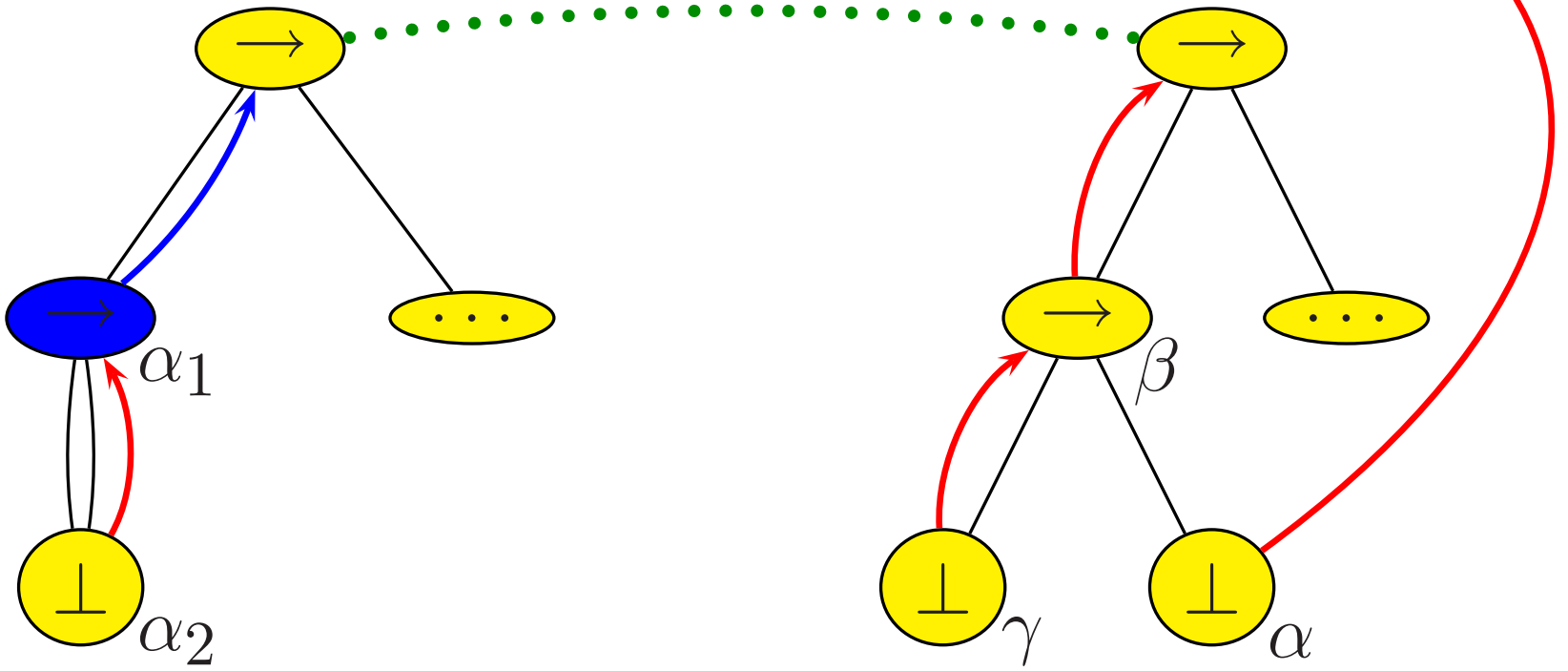

PRÉFIXE
→
→
⊥



$$(\alpha \to \alpha) \to (\alpha \to \alpha)$$

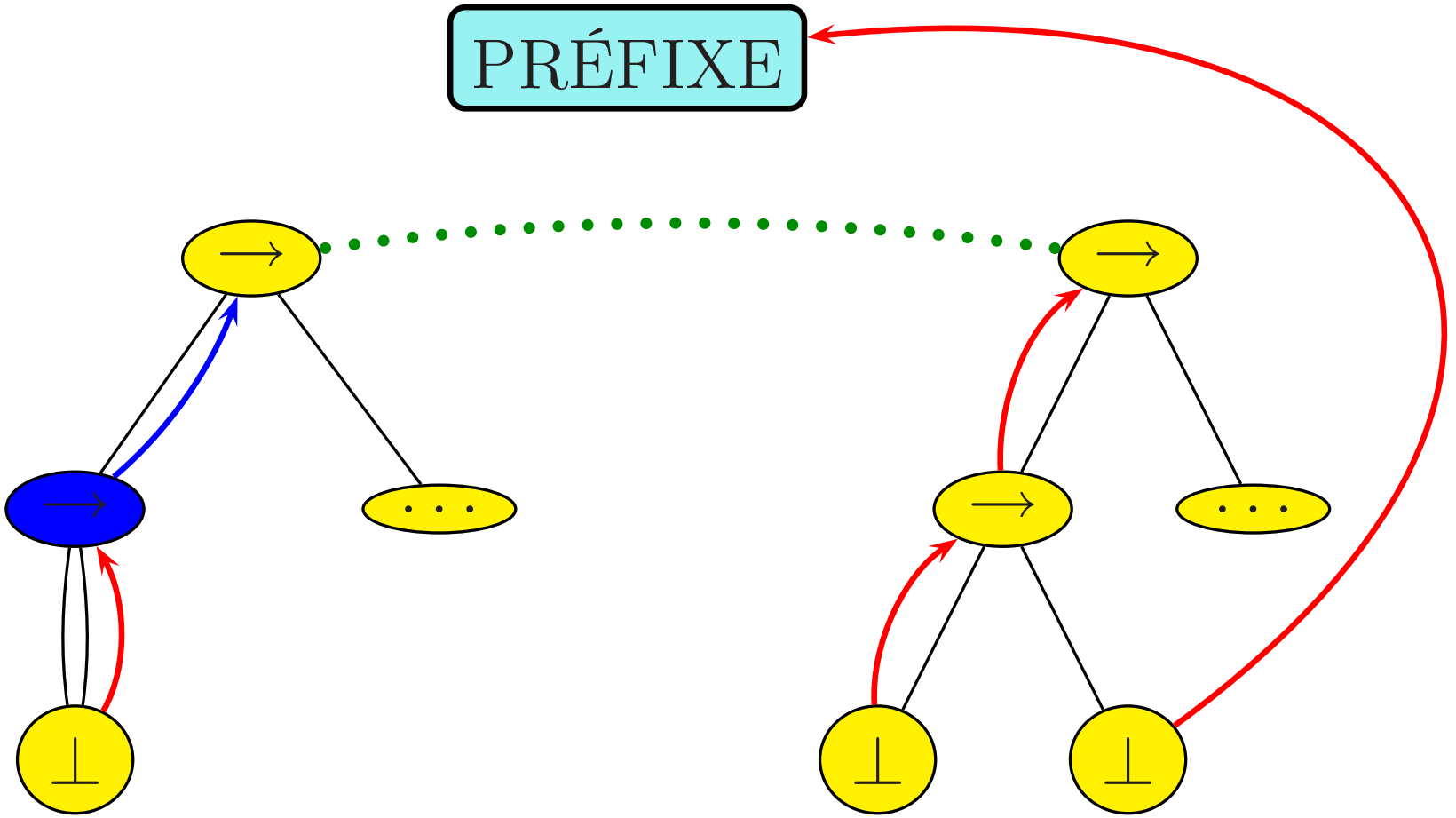
Exemple «rigide»

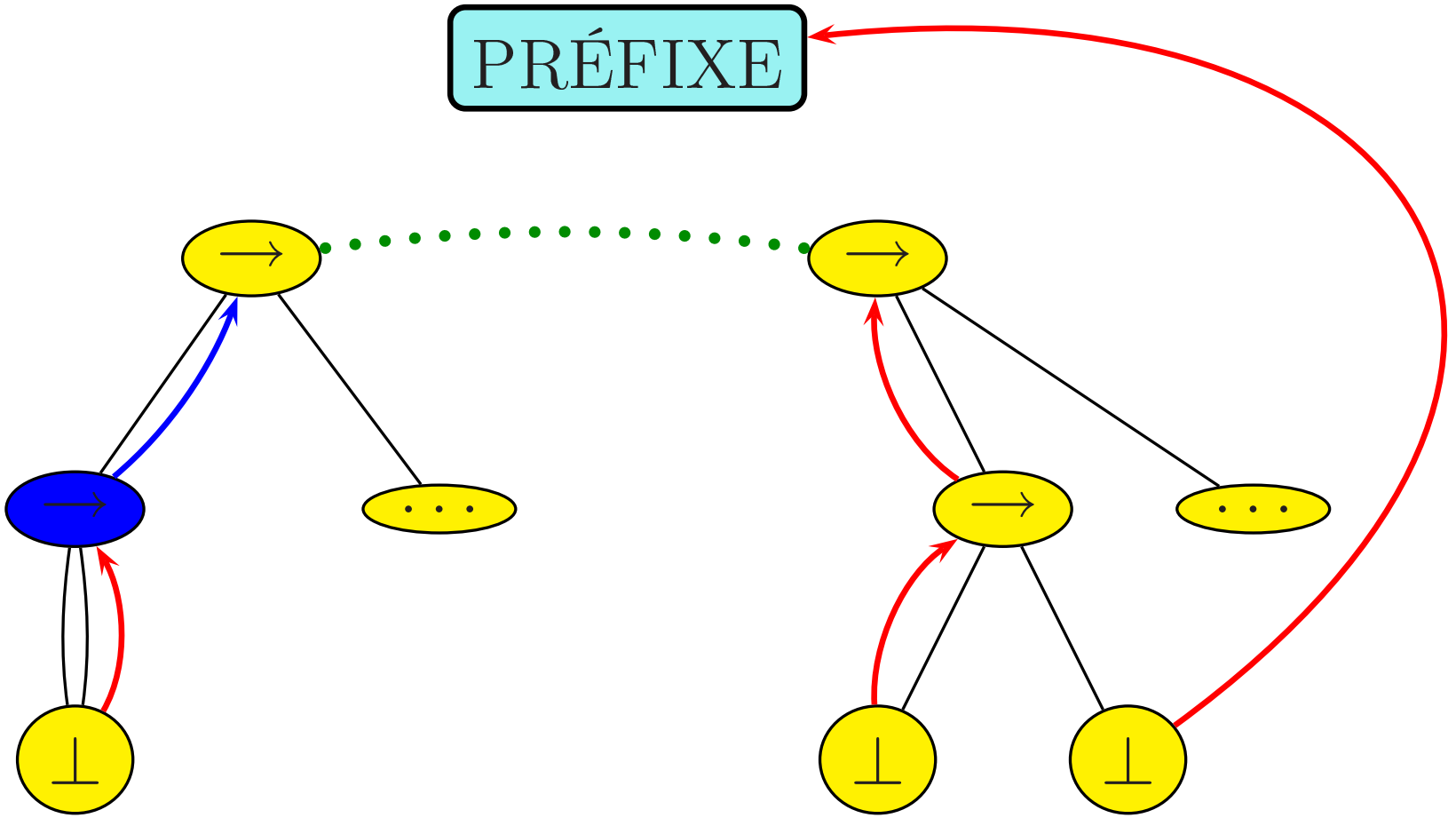
PRÉFIXE

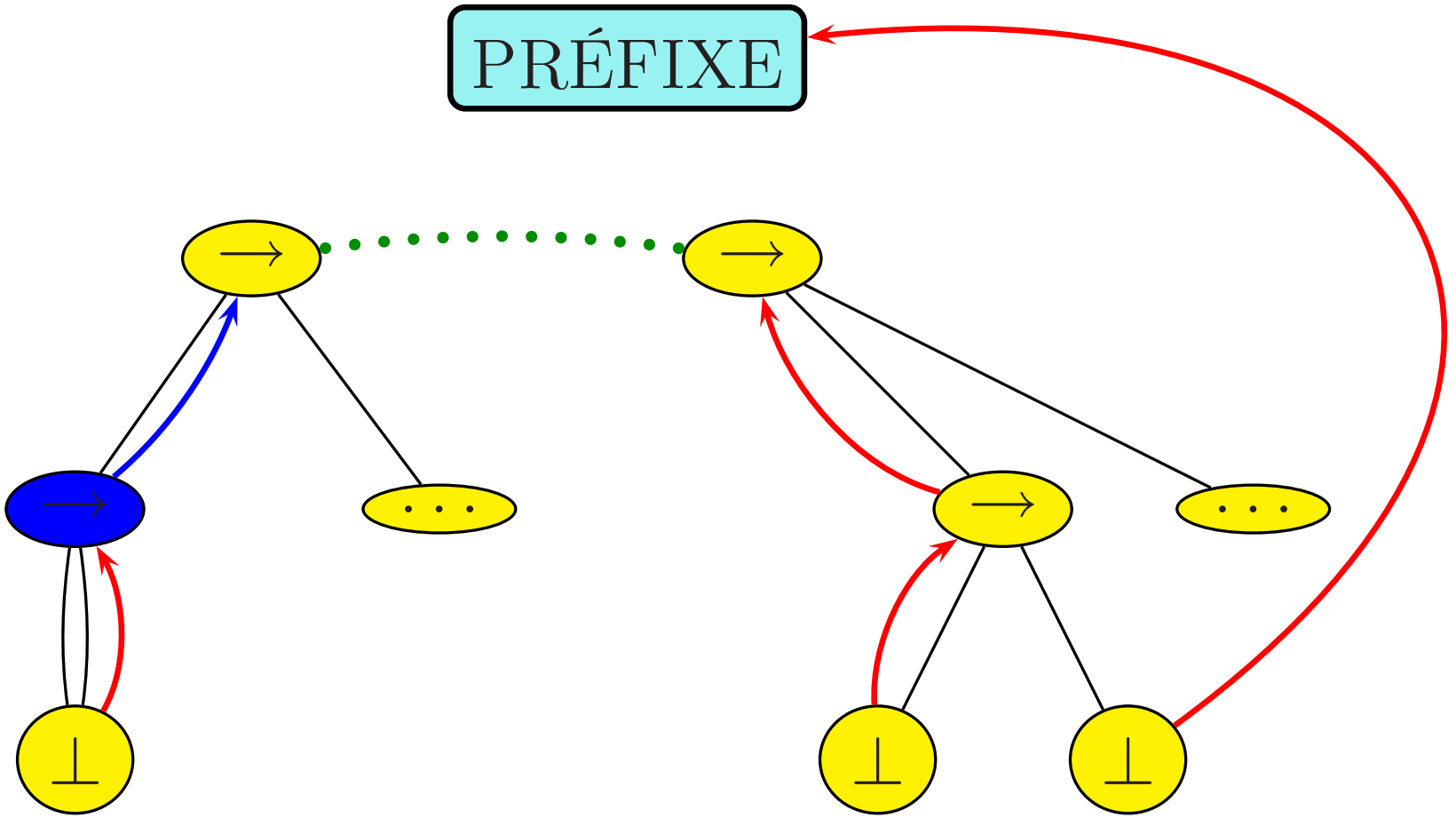


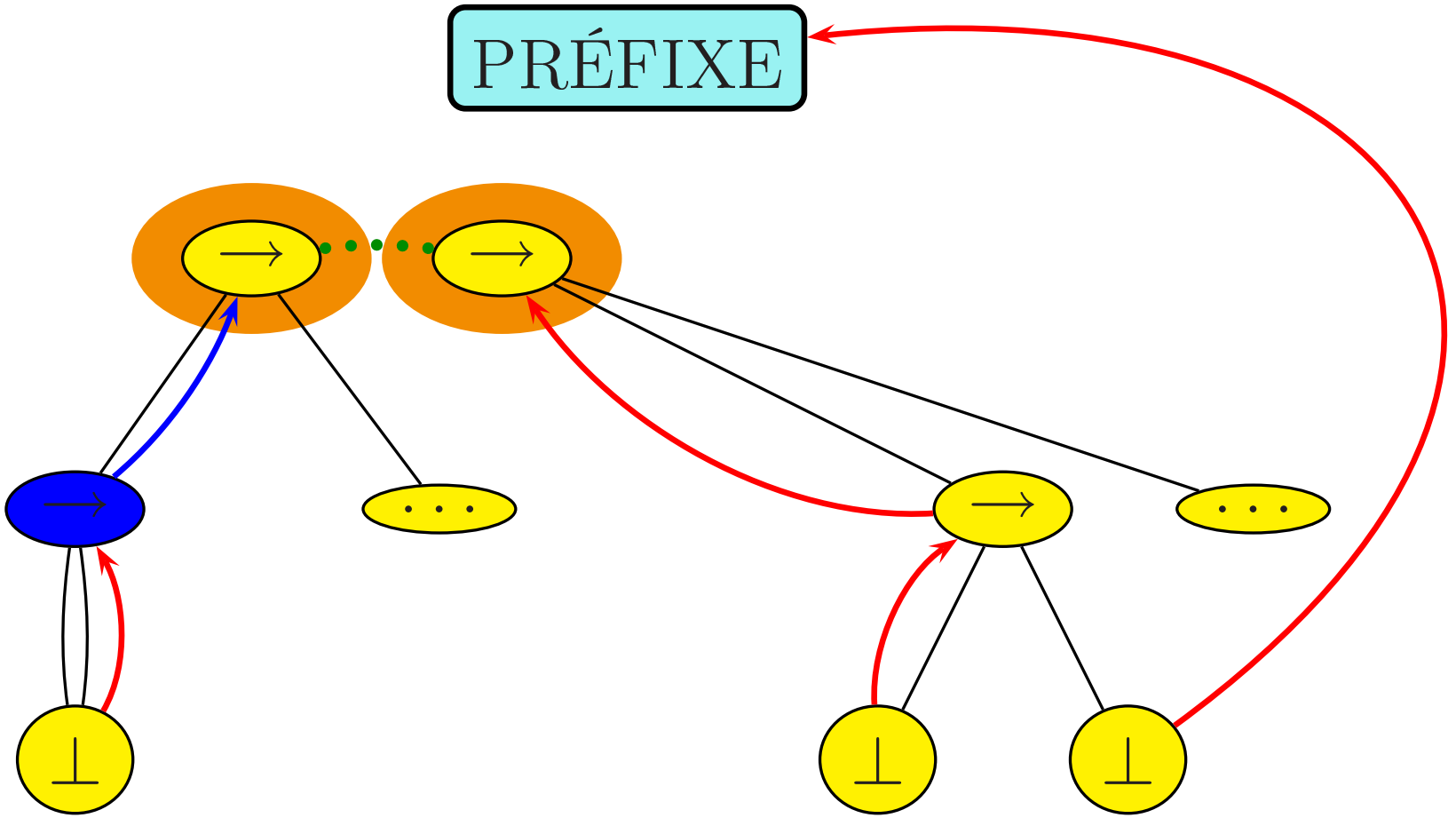
$\forall\,(\alpha_1 = \forall\alpha_2.\alpha_2 \to \alpha_2\,)$ $\alpha_1 \to \ldots$

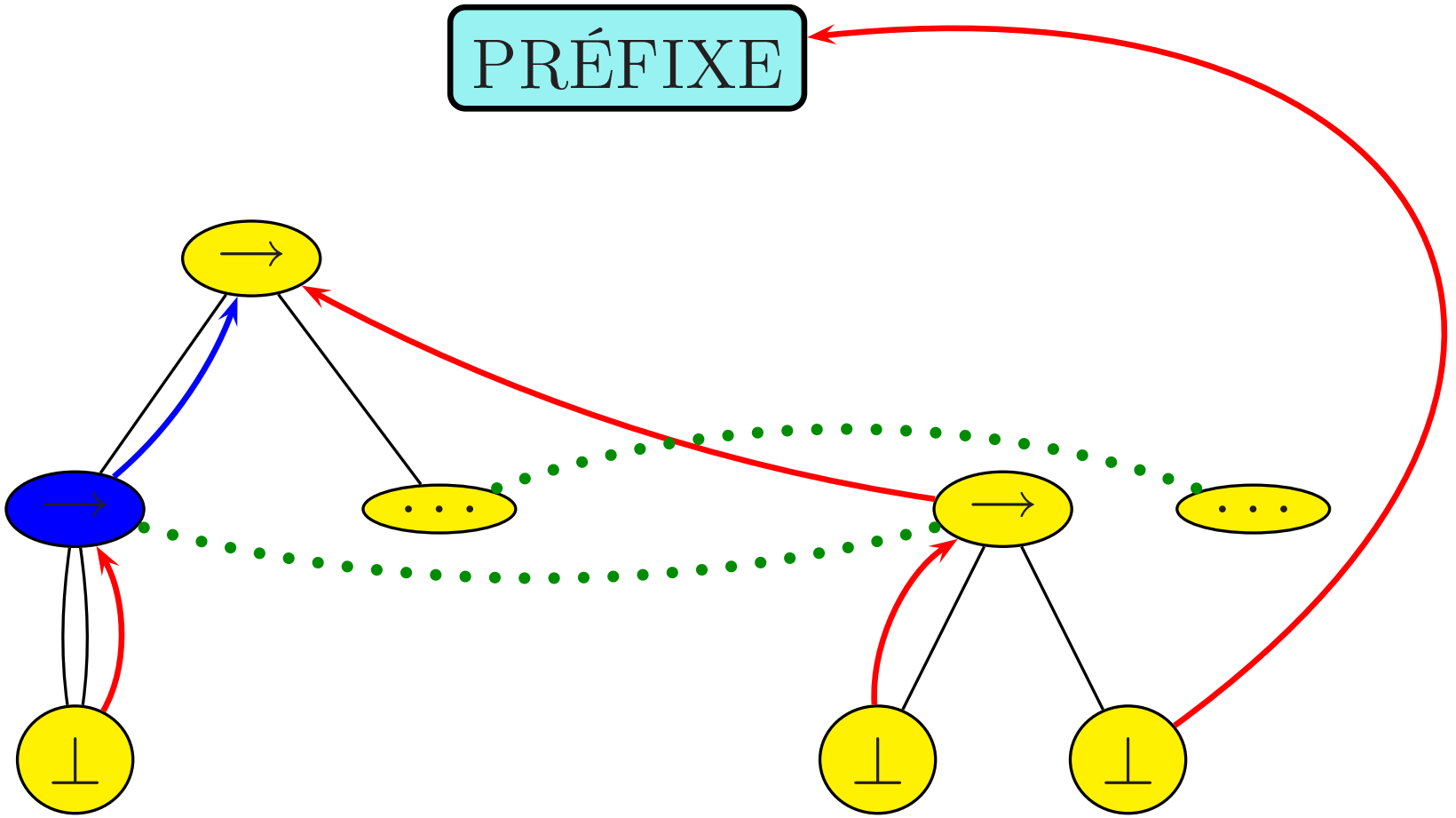
$\forall\,(\beta \geq \forall\gamma.\,\gamma \to \alpha\,)$ $\beta \to \ldots$

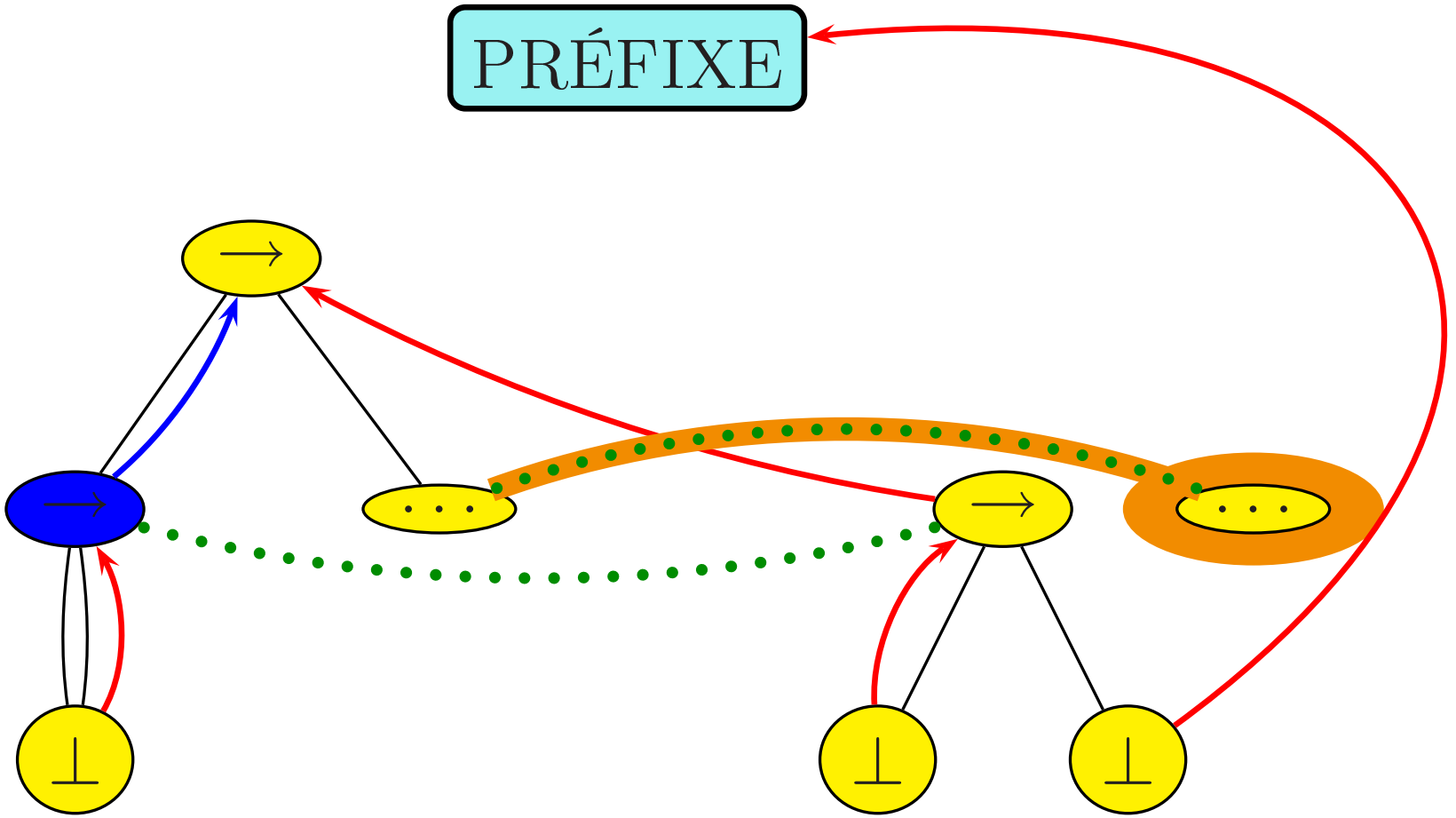

PRÉFIXE

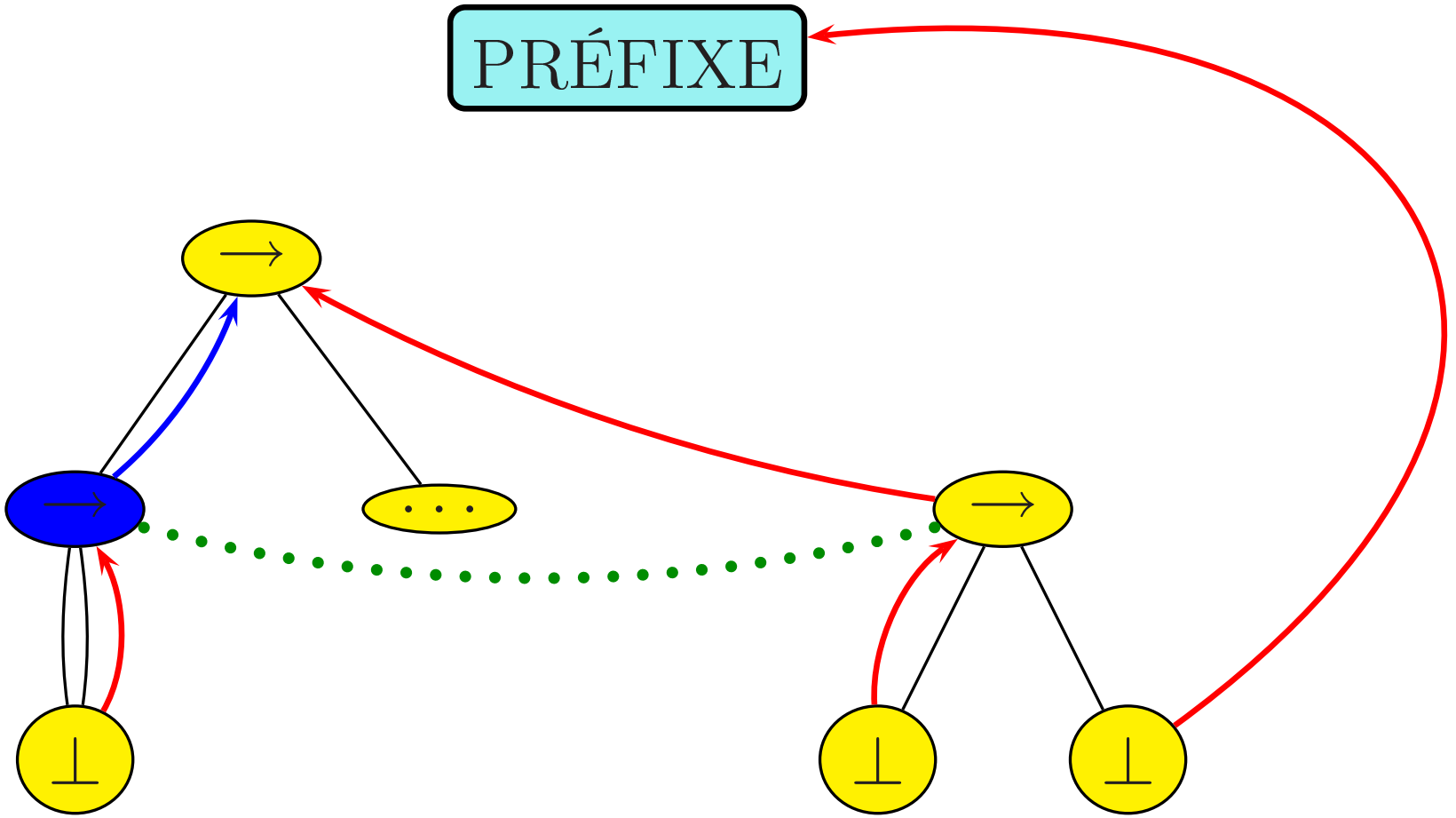

PRÉFIXE

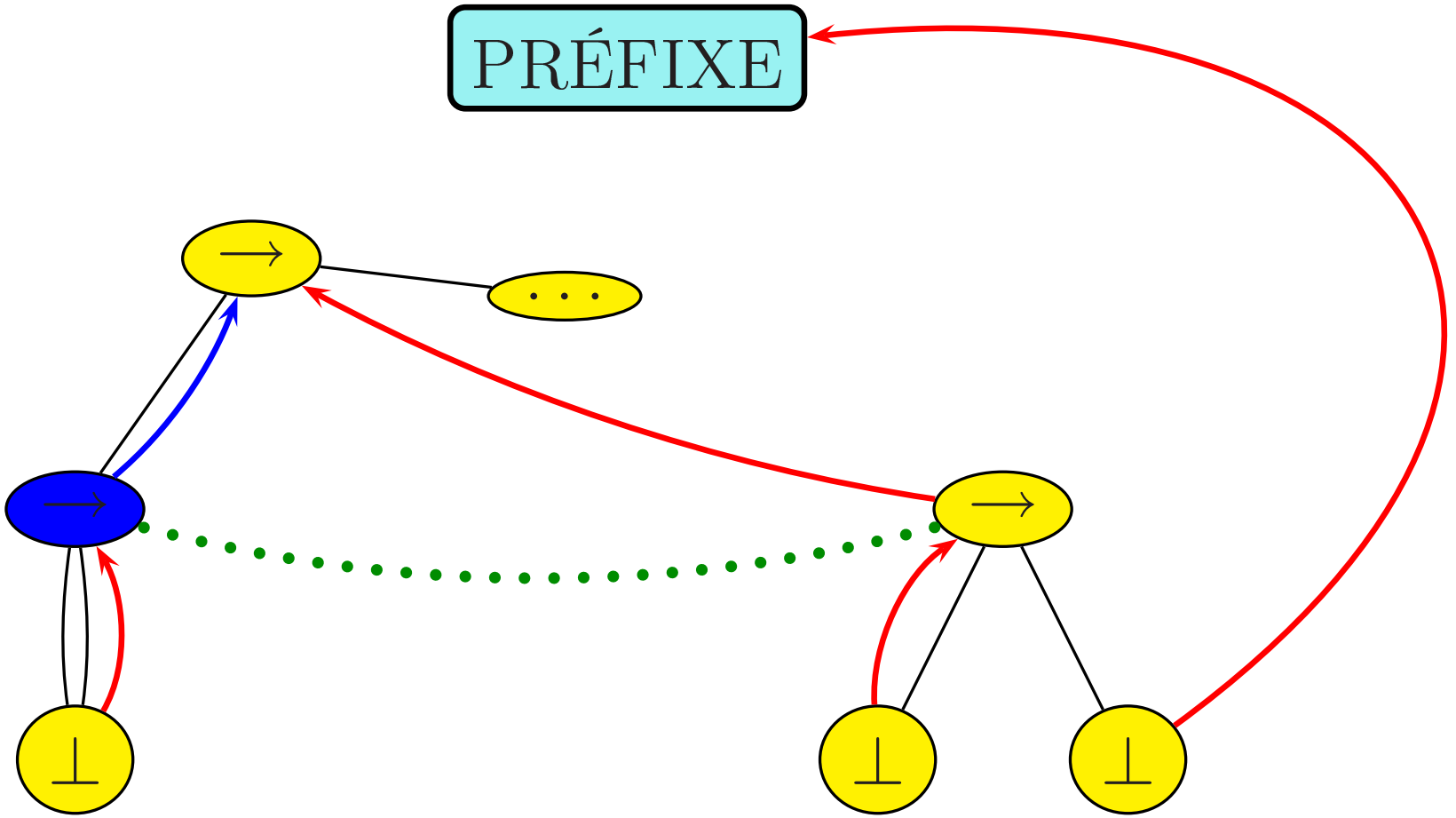

PRÉFIXE

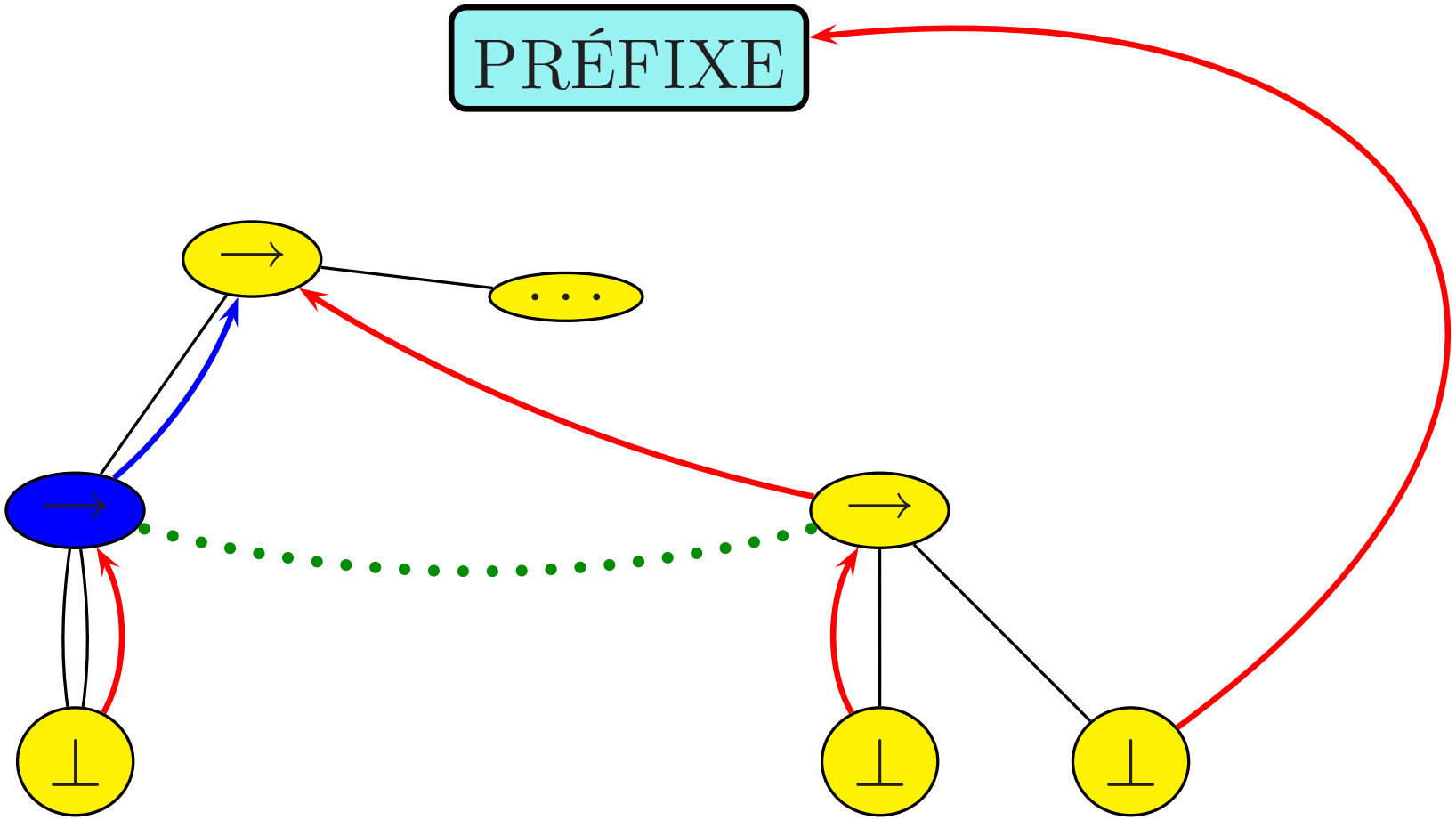

PRÉFIXE

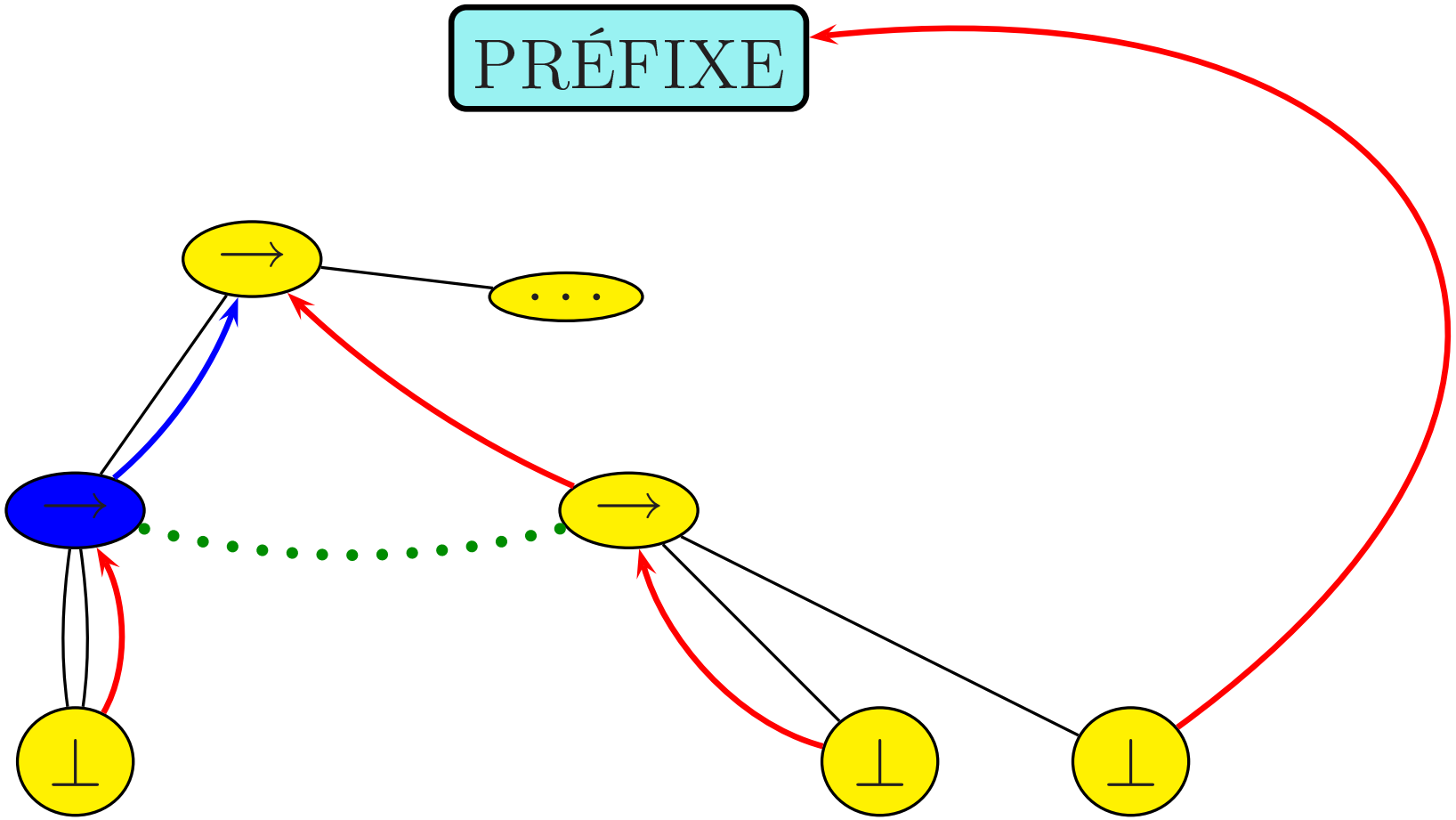

PRÉFIXE

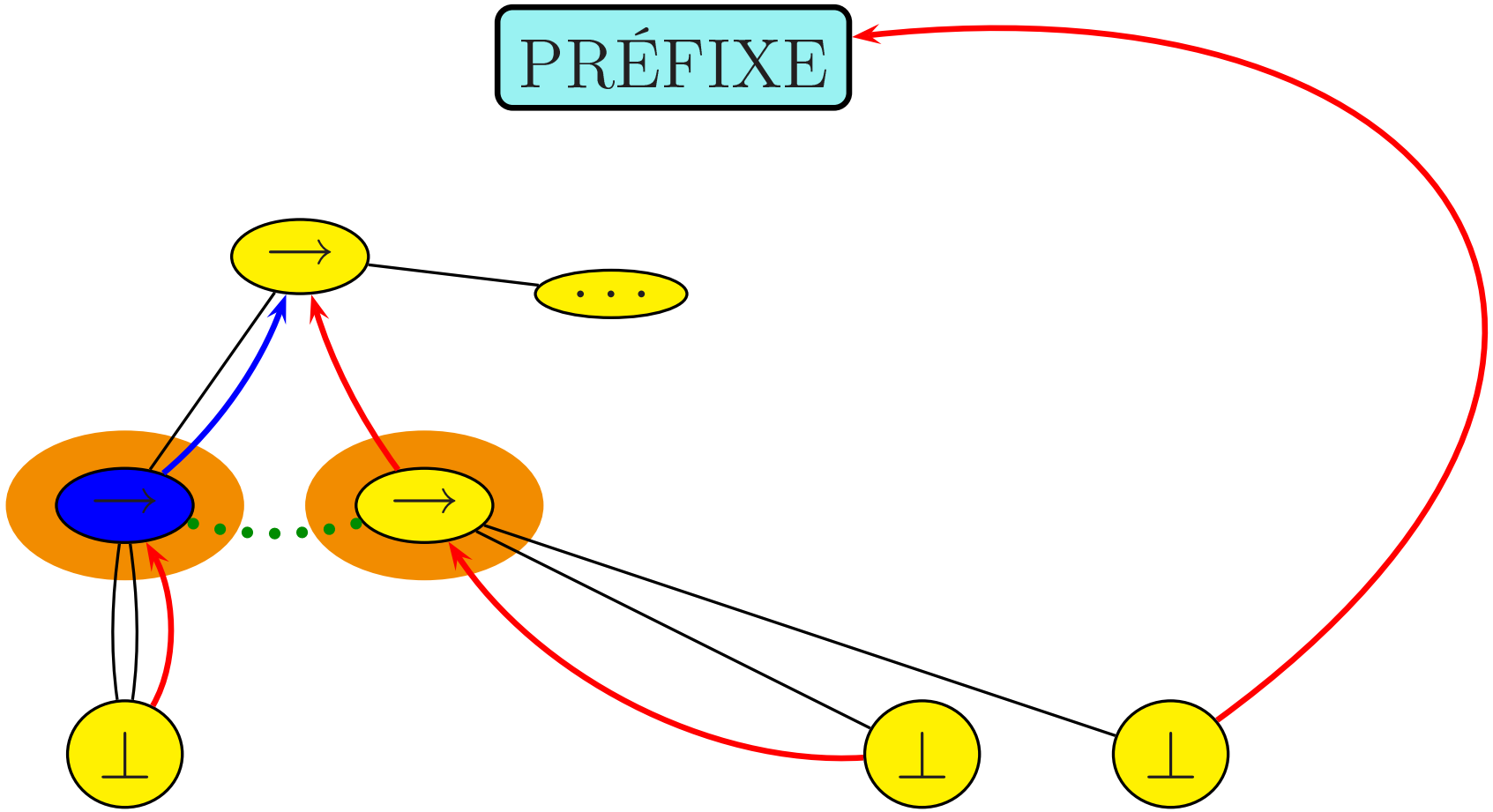

PRÉFIXE

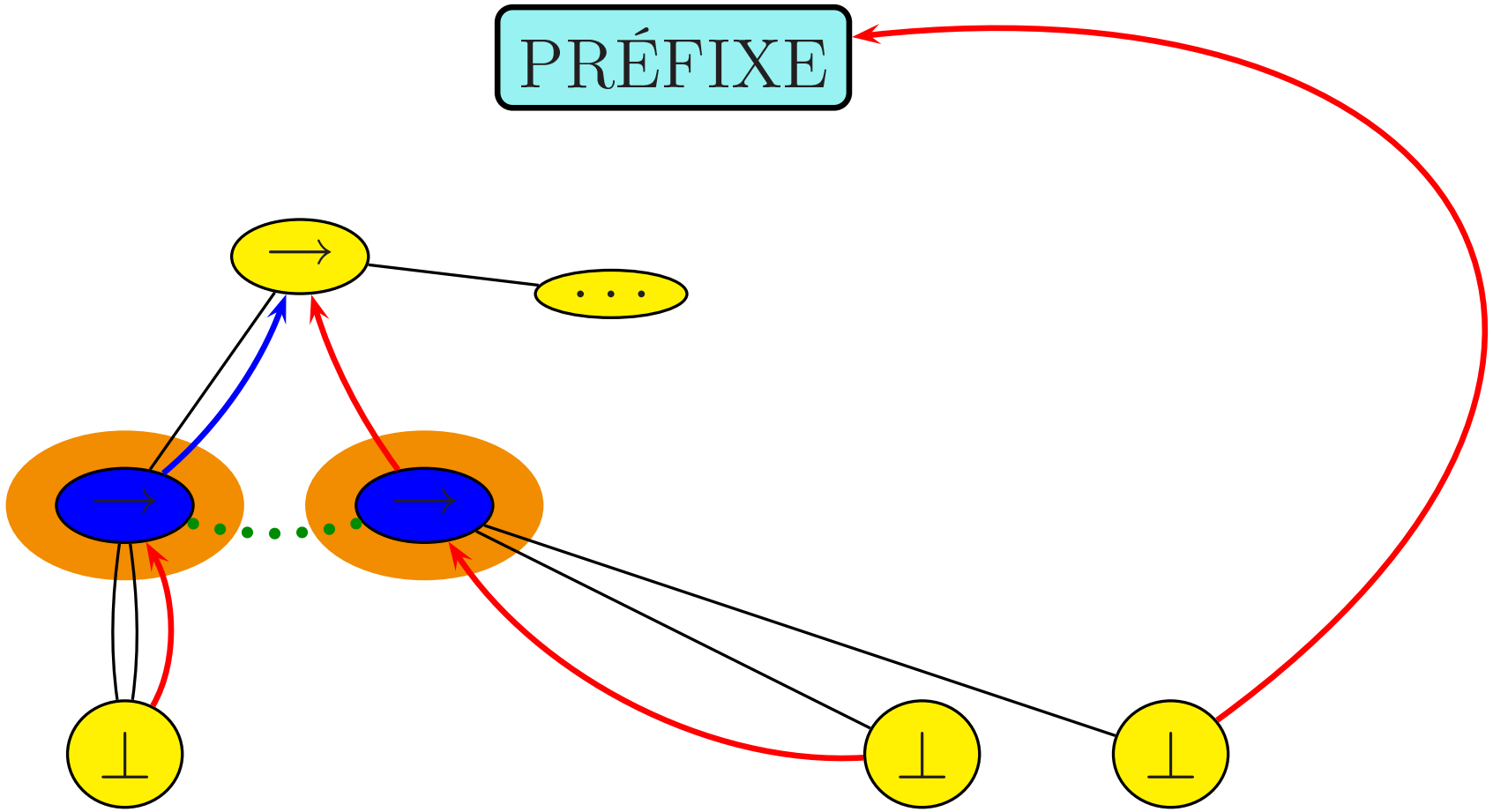

PRÉFIXE

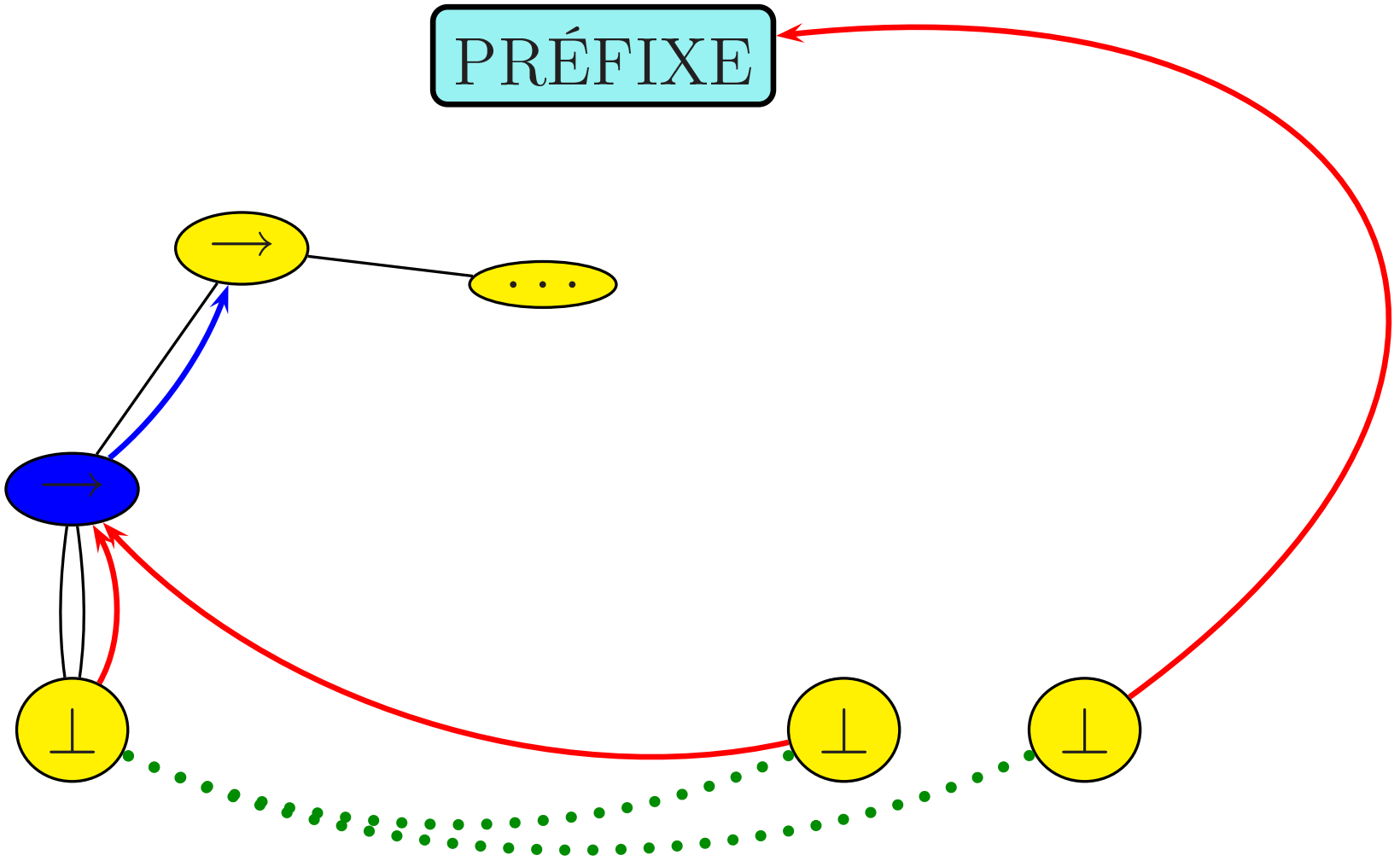

PRÉFIXE

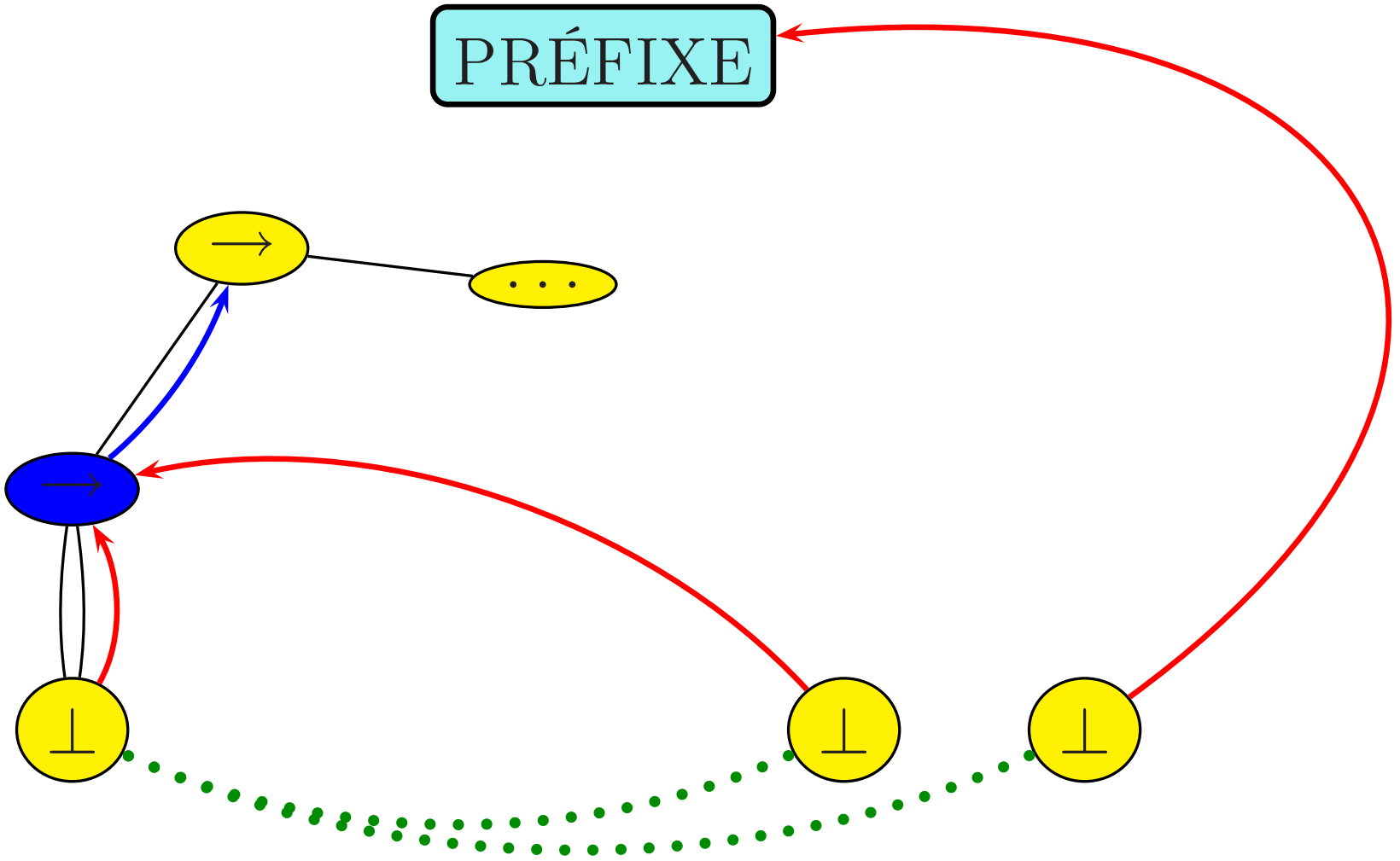

PRÉFIXE

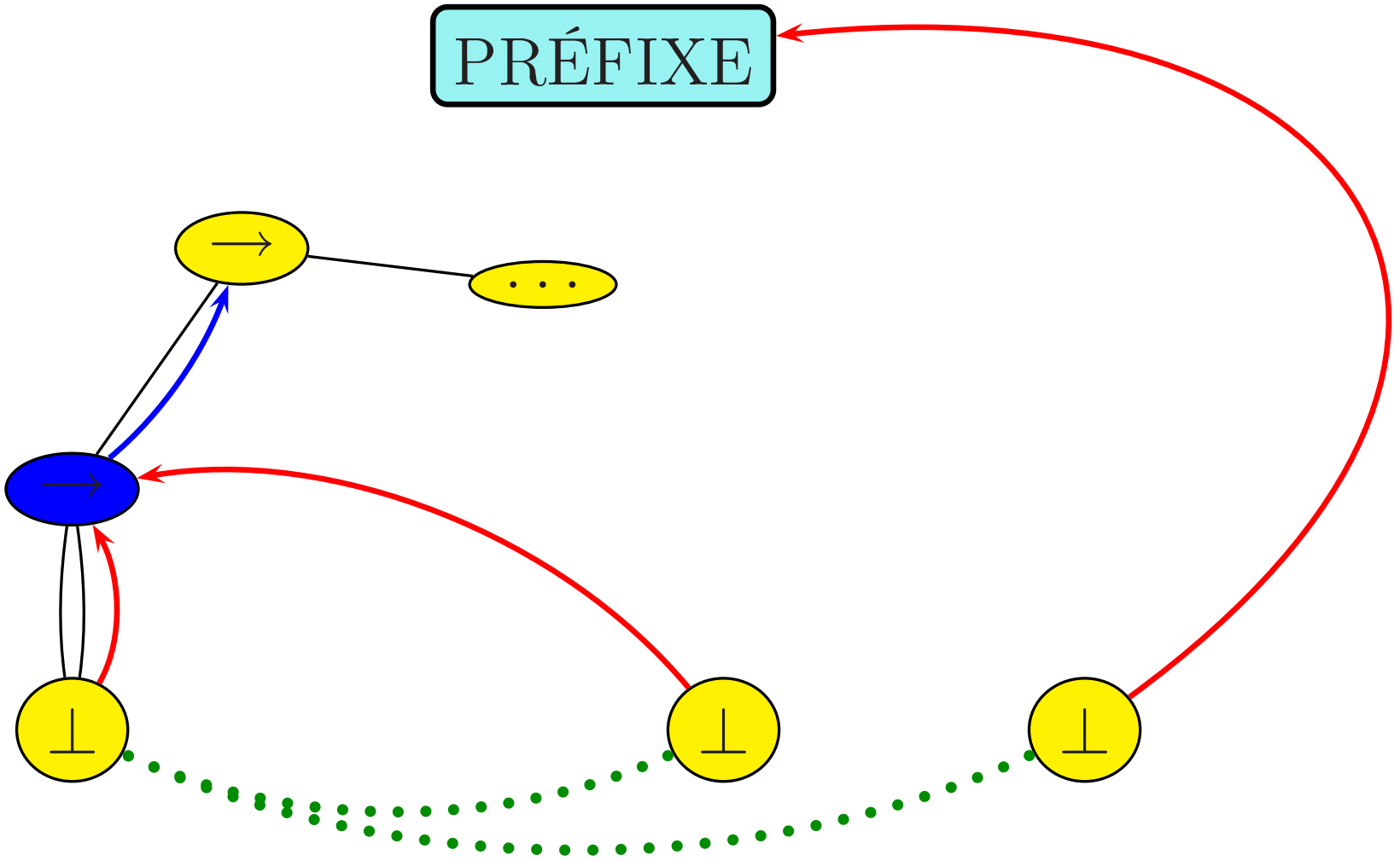

PRÉFIXE

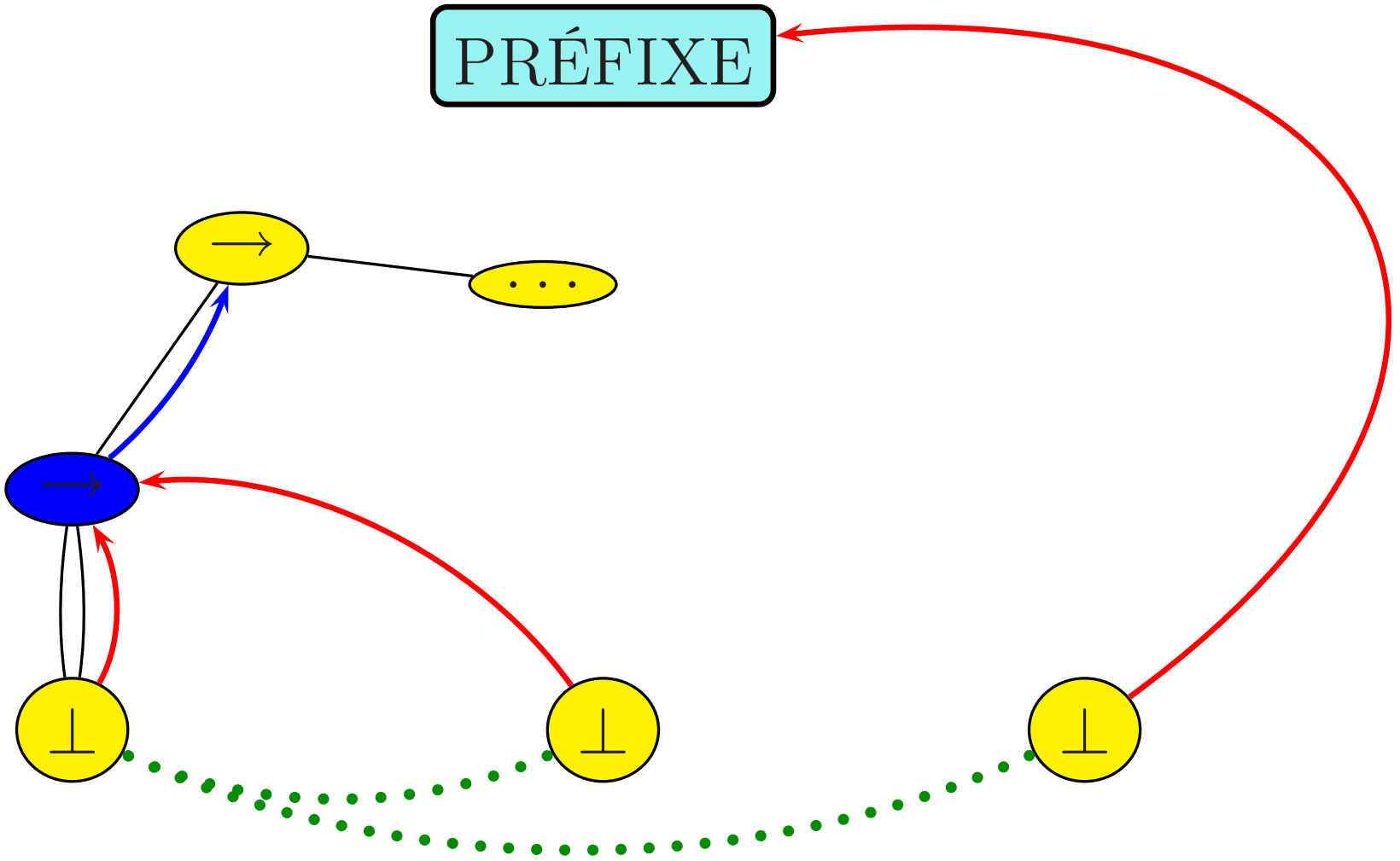

PRÉFIXE

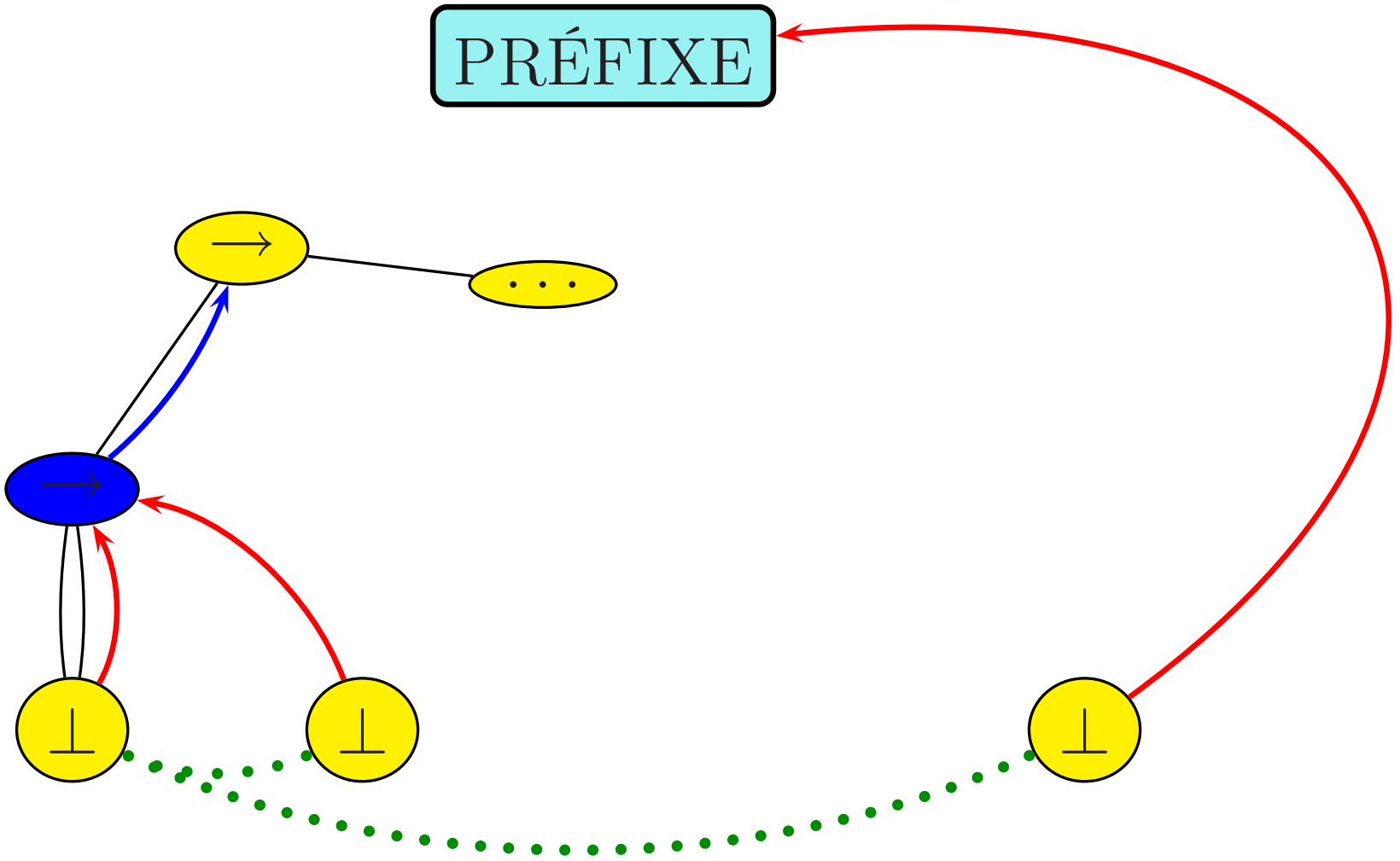

PRÉFIXE

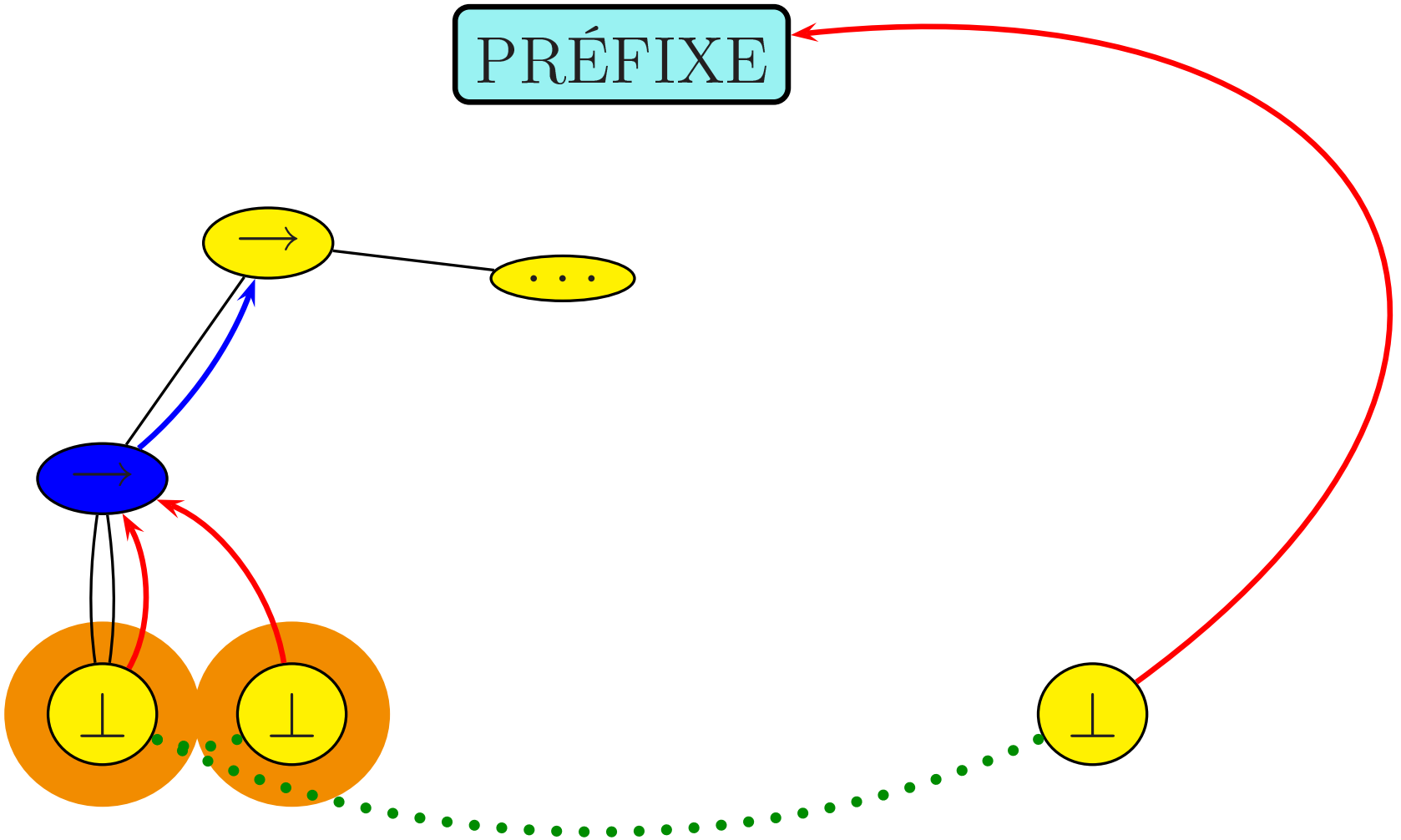

PRÉFIXE

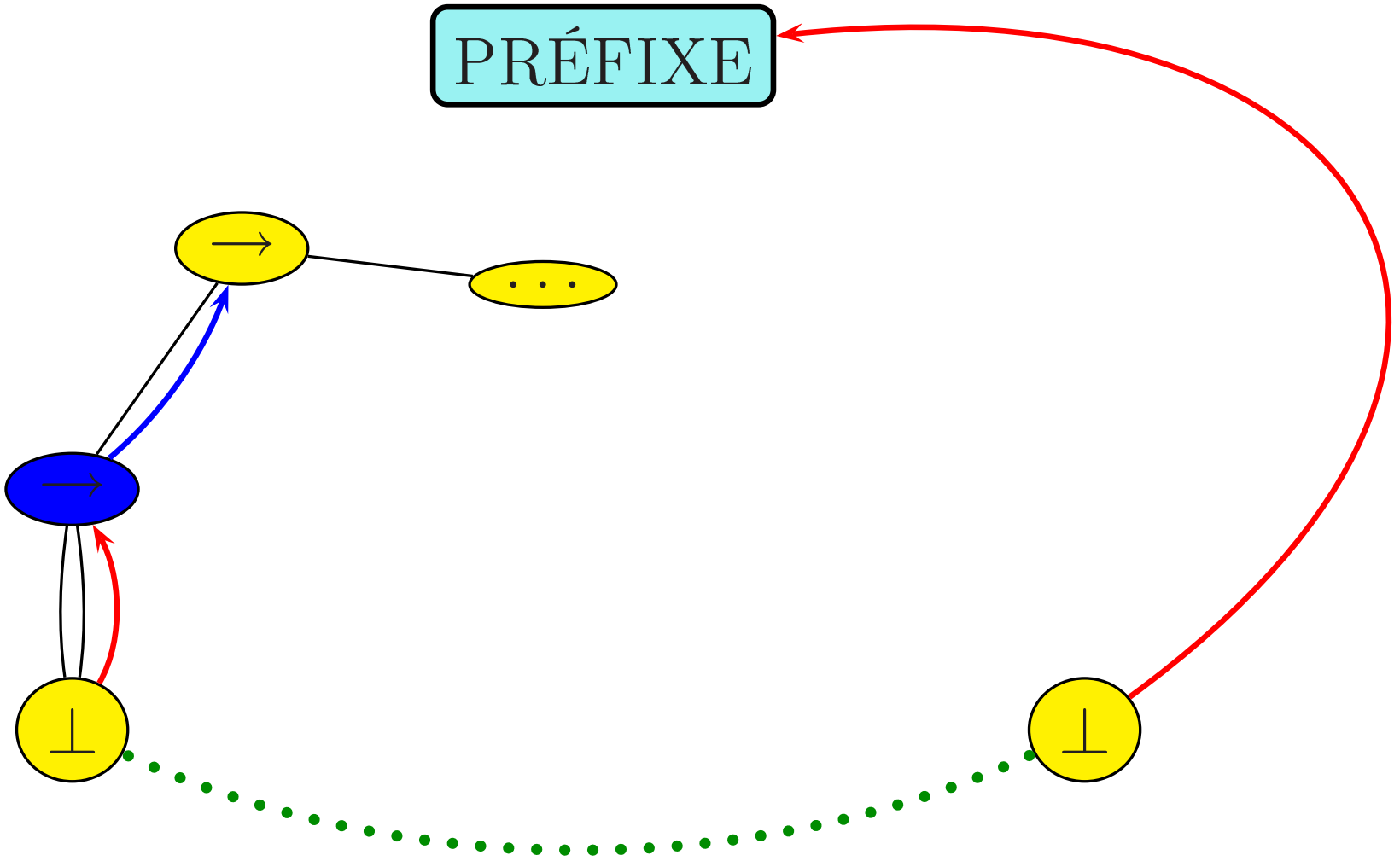

PRÉFIXE

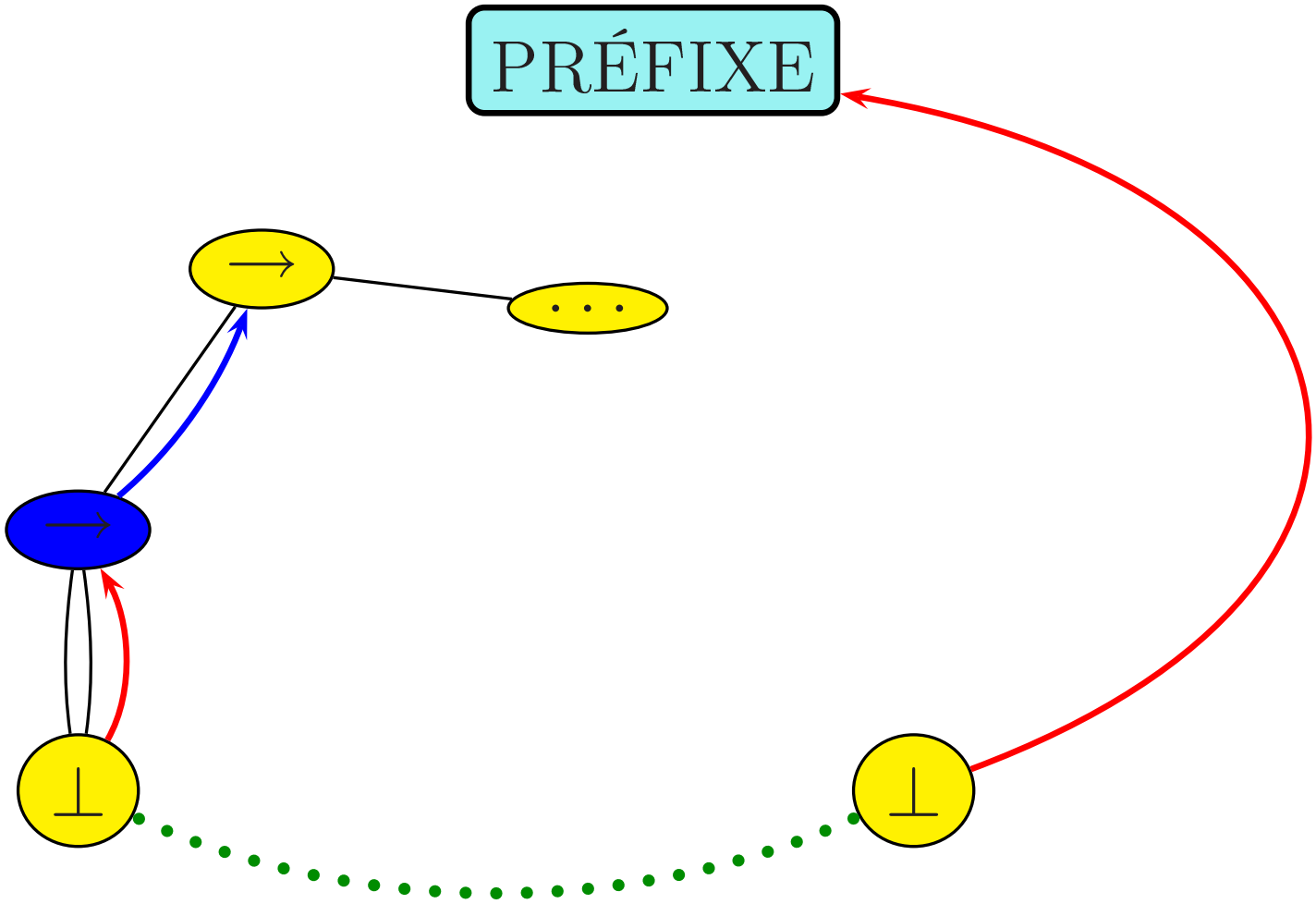

PRÉFIXE

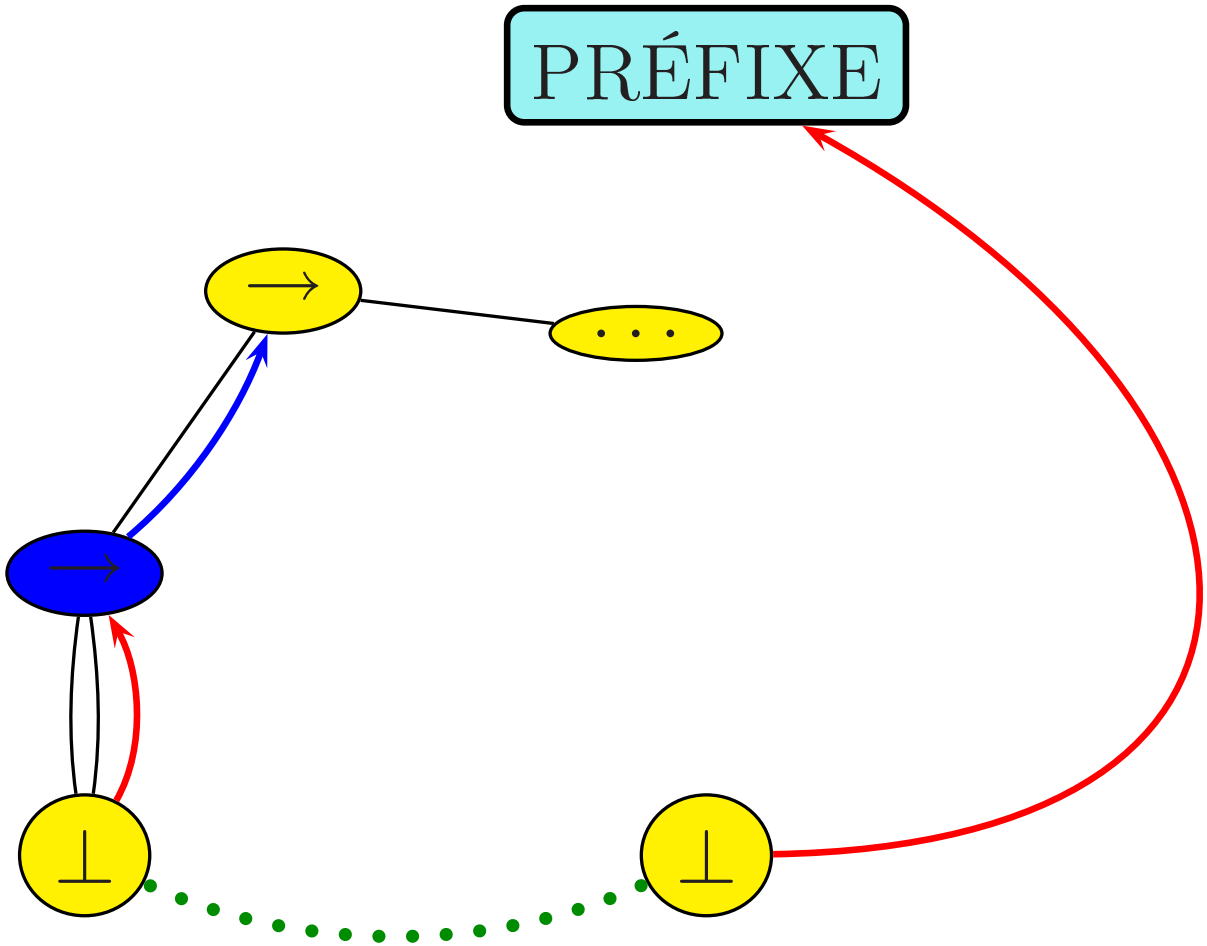

PRÉFIXE

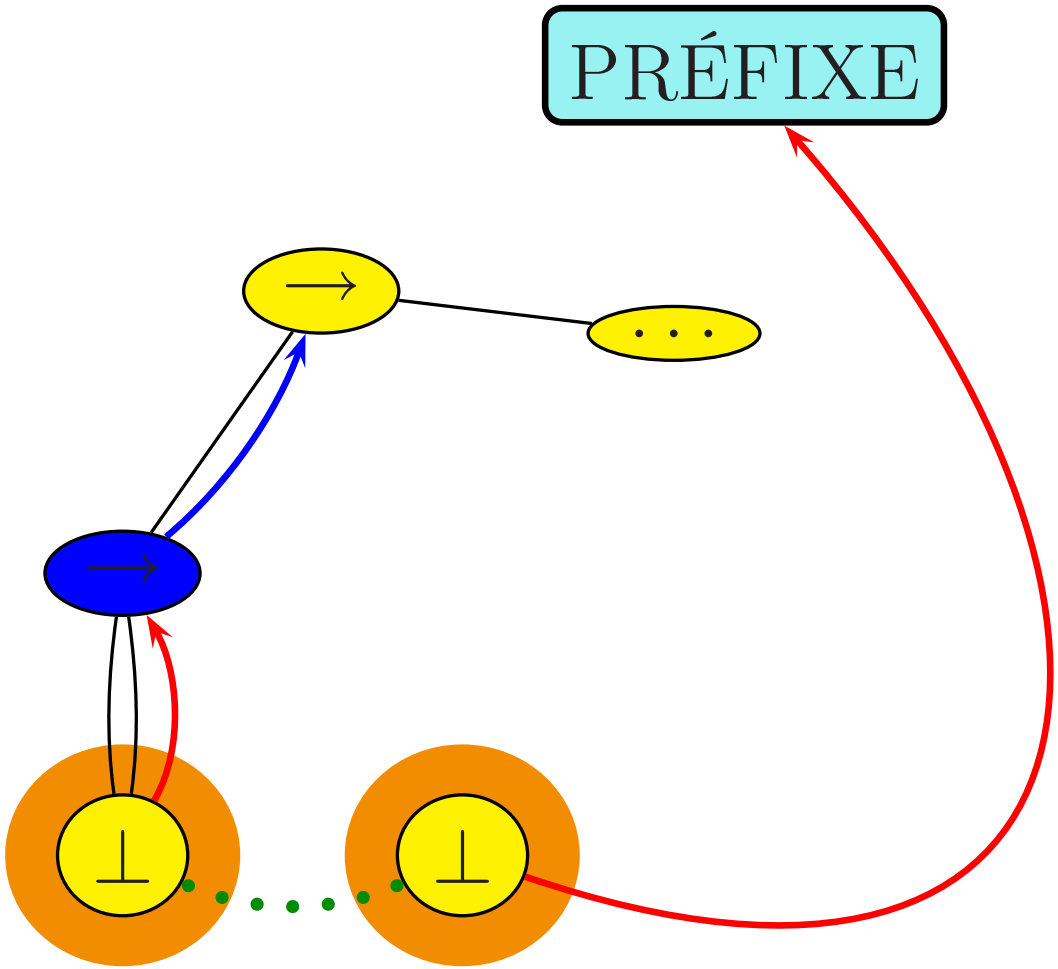

PRÉFIXE

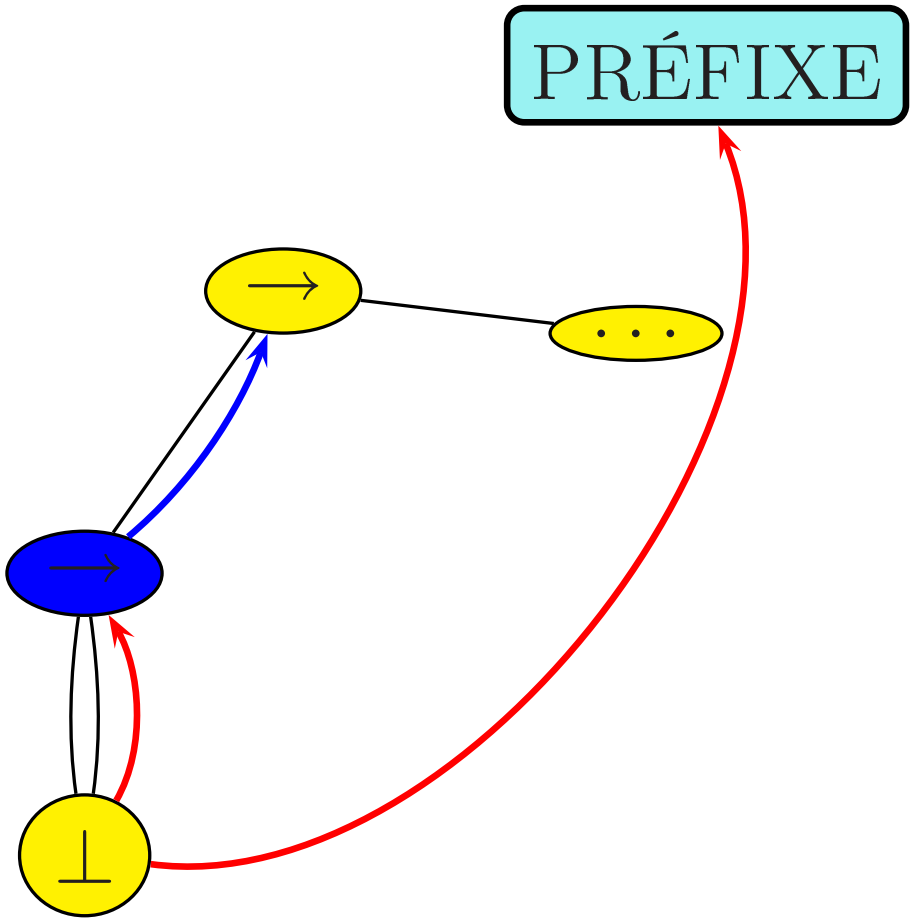

PRÉFIXE

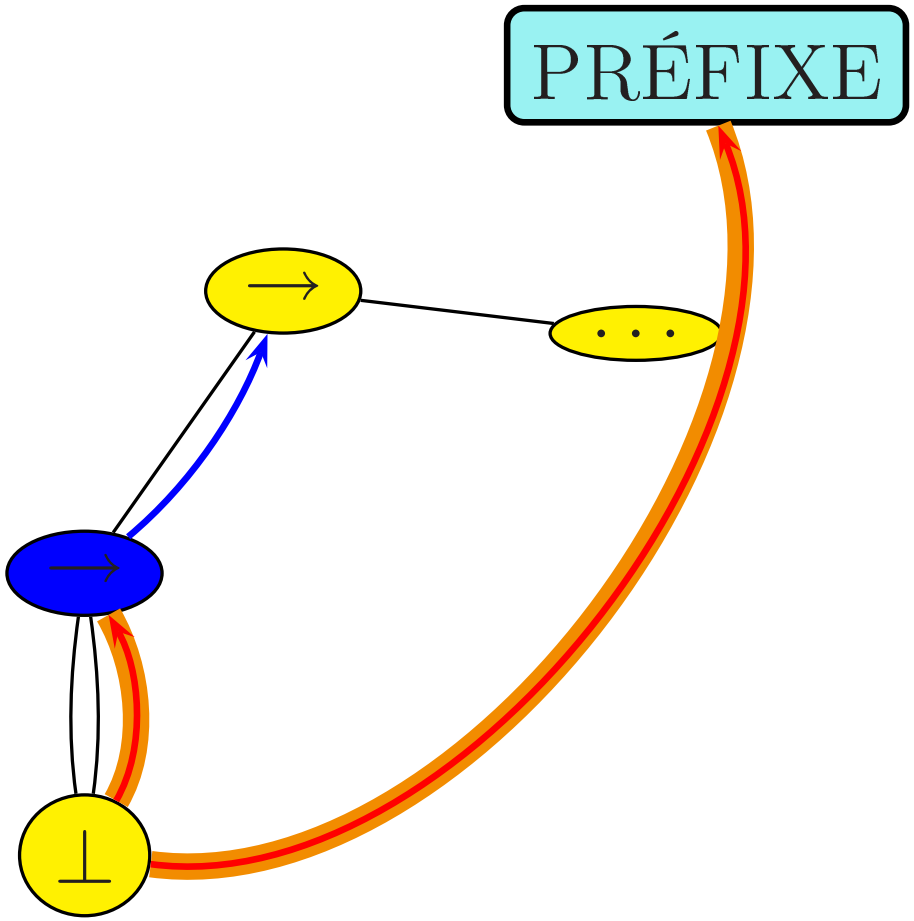

PRÉFIXE

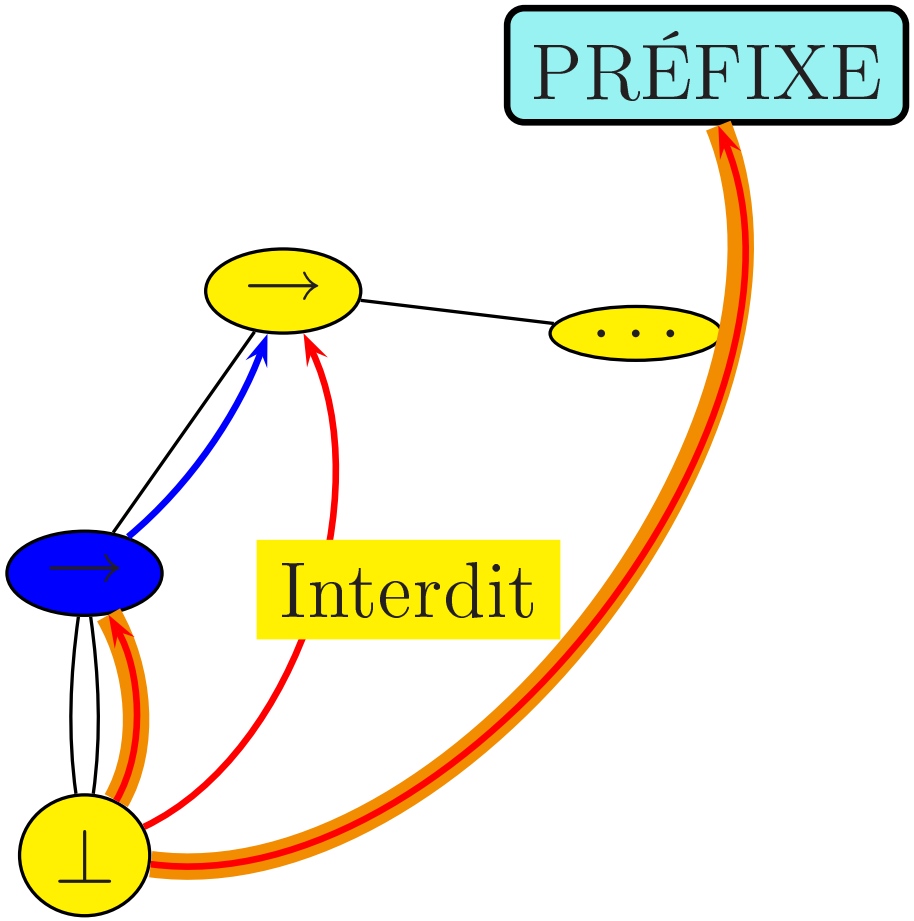

PRÉFIXE
→
• • •
→
⊥
⊥

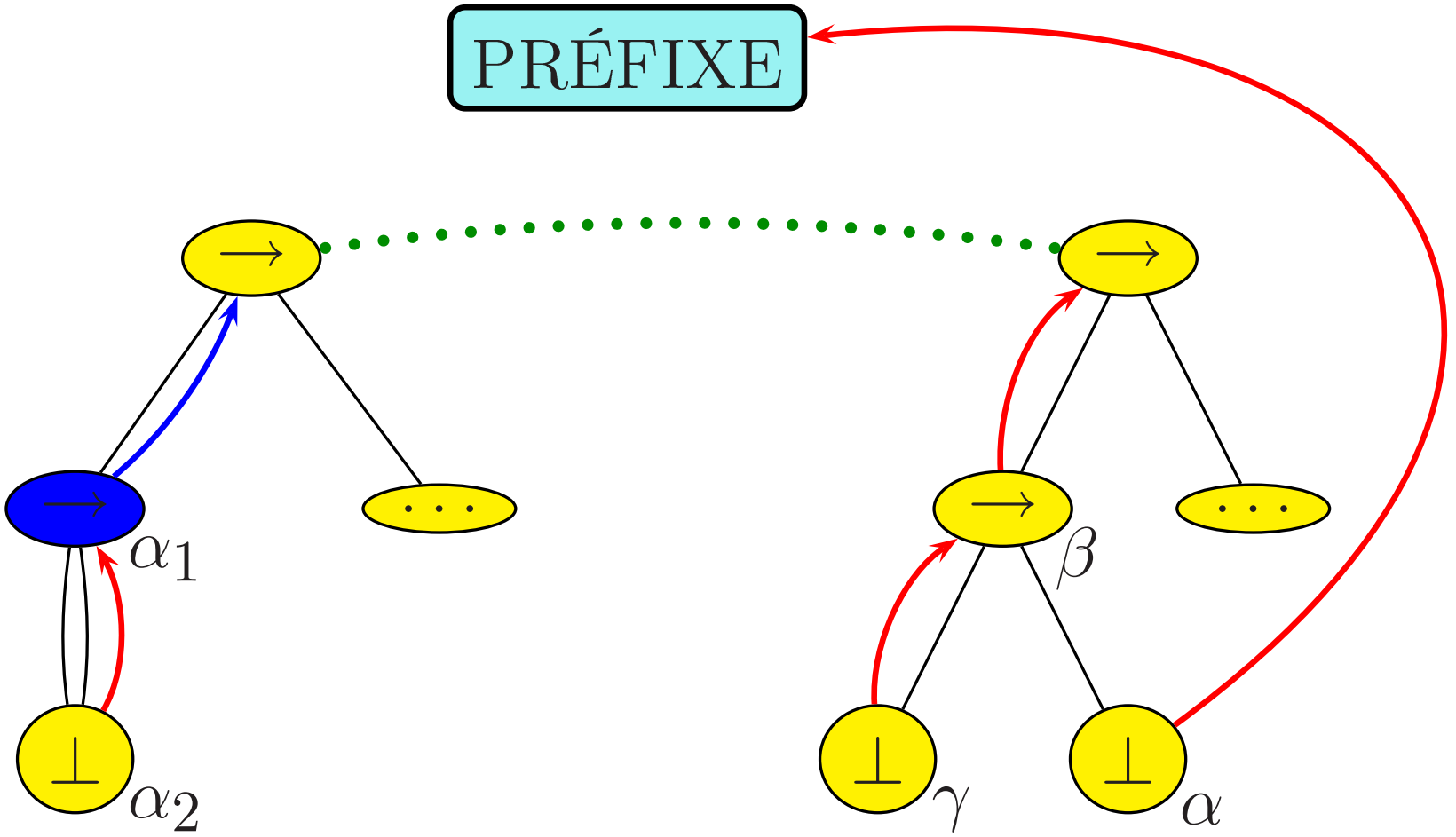

PRÉFIXE
→
• • •
→
⊥
⊥


PRÉFIXE
⊥
⊥


PRÉFIXE
→
• • •
→
⊥


PRÉFIXE


PRÉFIXE
Interdit



$\forall\,(\alpha_1 = \forall\alpha_2.\alpha_2 \to \alpha_2\,)$ $\alpha_1 \to \ldots$ n'est pas unifiable avec

$\forall\,(\beta \geq \forall\gamma.\,\gamma \to \alpha\,)$ $\beta \to \ldots$

Unification

L'unification reste de "premier ordre"

mais avec des types de second ordre

Elle reste notamment décidable.

Ingrédients

- Un langage de types avec une relation d'instance
- Des règles de typage
- Un algorithme d'inférence
- Des annotations

Introduction aux annotations

Comme pour ML, le typage du λ est monomorphe :

$$\textsc{Fun}\ \frac{(Q)\ \ \Gamma, x : \tau_0 \vdash a : \tau}{(Q)\ \ \Gamma \vdash \lambda\, x.\ a : \tau_0 \to \tau}$$

Mais alors quel est le type de x dans $\lambda\, x.\ (x : \sigma_{\mathtt{id}})\ \ x$?

Le type de x est α, et $\alpha = \sigma_{\mathtt{id}}$ est mis dans le préfixe.

L'abstraction

Sans annotation, le type de x demeure abstrait

$$\alpha = \sigma_{\texttt{id}} \quad x : \alpha \vdash x : \alpha$$

Avec annotation, le polymorphisme de x est révélé

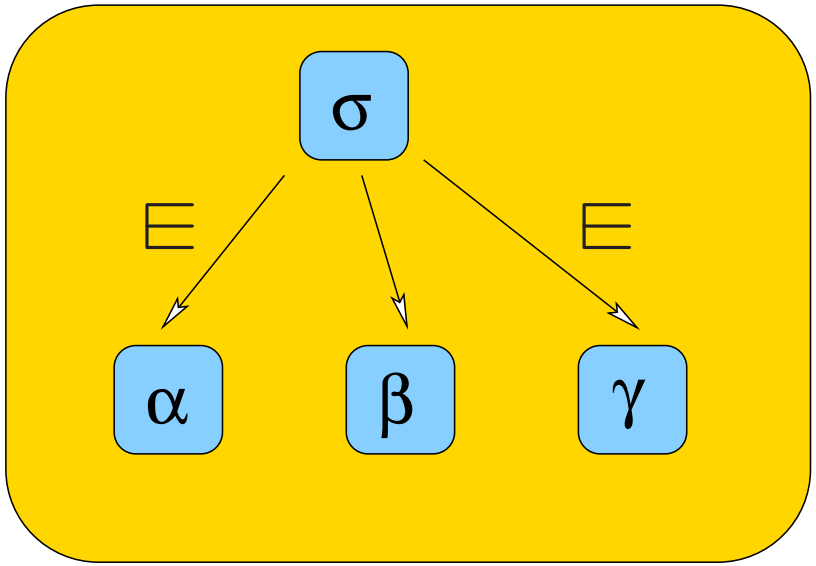
$$\alpha = \sigma_{\texttt{id}} \quad x : \alpha \vdash (x : \sigma_{\texttt{id}}) : \sigma_{\texttt{id}}$$

Le mécanisme clef est l'abstraction / révélation :

$$(Q_1, \alpha = \sigma, Q_2) \;\; \sigma \Subset \alpha$$

L'abstraction

Sous un préfixe donné contenant $\alpha = \sigma, \beta = \sigma, \gamma = \sigma$, un schéma σ a plusieurs représentants :



Concret Information explicite

Abstrait Information inférée

Pseudo classe d'équivalence

Les annotations et l'abstraction

L'annotation $(\_ : \sigma)$ est une primitive permettant de remonter dans la pseudo classe d'équivalence de σ.

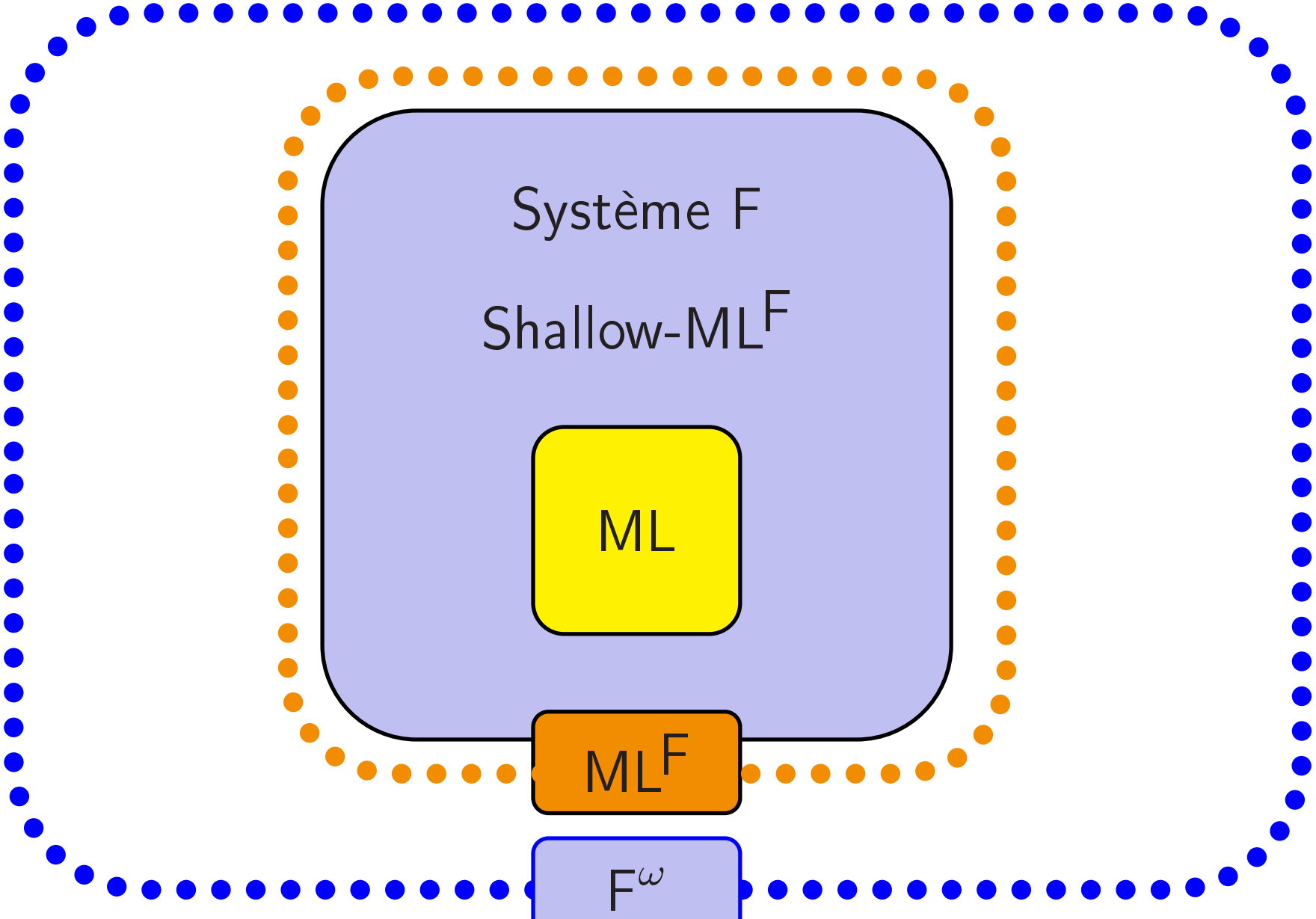
Elle attend en argument un élément quelconque de la classe, et retourne l'élément le plus haut.

Ses types possibles sont donc $\alpha \to \sigma$ $\beta \to \sigma$ $\gamma \to \sigma$ et $\sigma \to \sigma$

Ce dernier type subsume tous les autres par instanciation (abstraction) implicite.

Le type de l'annotation $(\_ : \sigma)$ est donc $\sigma \to \sigma$,
c'est-à-dire $\forall\,(\alpha = \sigma, \alpha' \geq \sigma)\ \alpha \to \alpha'$

Topologie



Conclusion

- Le mécanisme d'abstraction–révélation permet de propager le polymorphisme via le préfixe.
- Le noyau ML est conservé tel quel, seul change le langage des types.
- L'expressivité des types permet de définir des annotations polymorphes, qui suffisent à encoder le Système F de manière concise.
- La spécification de ML^F est intuitive :

  Les variables utilisées de manière polymorphe doivent être annotées.

La suite...

- Typage des références, en s'inspirant de *value-restriction* et *relaxed value-restriction*.
- Nouvel algorithme d'unification (prouvé linéaire)
- Le passage à F^{ω}
- Nouvelle théorie pour MLF, basée sur le Système F.

Merci . . .

Logiciels Libres

L'ensemble de la thèse a été réalisé uniquement avec des logiciels libres.

Merci à toute la communauté du libre.

En particulier,

Linux, Ocaml, WhizzyTeX, Advi, Emacs, Unison, LaTeX

Et bien d'autres.

Zoggy, xfig, DemoLinux, MLDonkey, mozilla, mplayer, gimp, gv, blender, OpenOffice, Mandrake, Aurox, wget, Frozen Bobble, FreeCiv, gzip, Sylpheed, BladeEnc ...

ML^F et le Système F^ω

- ✔ Le noyau ML^F est le même.
- ✔ Les types de *kind* supérieures sont manipulés explicitement.

Exemple

$$\texttt{let}\ \ fak = \Lambda(F : \star \to \star).\lambda\,(f : \forall\alpha.\alpha \to F\ \ \alpha).\ \lambda\, x.\ f(f\ \ x)$$

$$fak\ \ [\lambda\alpha.\alpha\ \texttt{list}]\ \ \lambda\, x.\ [x]$$