

# Polymorphisme

`Luc.Maranget@inria.fr`

`http://www.enseignement.polytechnique.fr/profs/  
informatique/Luc.Maranget/TLP/`

A Polymorphisme.

B Inférence des types polymorphes.

## Limitation des types simples

L'identité **Fun**  $x \rightarrow x$  a plusieurs types.

Type principal  $X \rightarrow X$ .

Mais **Let**  $id = \mathbf{Fun} \ x \rightarrow x \ \mathbf{In} \ id \ id$  n'est pas typable.

$$\frac{[id : X \rightarrow X] \vdash id \rightsquigarrow X \rightarrow X, \emptyset \quad [id : X \rightarrow X] \vdash id \rightsquigarrow X \rightarrow X, \emptyset}{[id : X \rightarrow X] \vdash id \ id \rightsquigarrow Y, (X \rightarrow X) = ((X \rightarrow X) \rightarrow Y)}$$

Ce qui conduit finalement à l'équation  $X = X \rightarrow X$ , qui n'a pas de solution.

Et pourtant, puisque  $id$  possède tous les types  $A \rightarrow A$ , les types suivants sont possibles pour la première et la seconde occurrence de  $id$ .

$$(Z \rightarrow Z) \rightarrow (Z \rightarrow Z) \qquad Z \rightarrow Z$$

Et l'application  $id \ id$  a pour type  $Z \rightarrow Z$ .

## Solution

Autoriser des instances différentes  $\sigma(A)$  d'un même type.

Enrichir les types :

$$\text{Nat} \in \mathcal{T} \quad X \in \mathcal{T} \quad \frac{A_1 \in \mathcal{T} \quad A_2 \in \mathcal{T}}{A_1 \rightarrow A_2 \in \mathcal{T}} \quad \frac{A \in \mathcal{T}}{\forall X [A] \in \mathcal{T}}$$

Ajouter une règle « d'élimination » de  $\forall$ .

$$\frac{E \vdash t : \forall X [A]}{E \vdash t : A[X \mapsto B]}$$

On parle aussi *d'instanciation* (de la variable liée  $X$ ).

(Il nous faut aussi une règle « d'introduction » de  $\forall$ , mais laissons cela de côté pour le moment).

## En passant...

Les types contiennent maintenant un lieu ( $\forall$ ), comme les termes (**Fun** etc.)

Ceci complique la définition de la substitution qui doit éviter les captures des variables libres.

$$\mathcal{F}(X) = \{ X \} \quad \mathcal{F}(\mathbf{Nat}) = \emptyset \quad \mathcal{F}(A \rightarrow B) = \mathcal{F}(A) \cup \mathcal{F}(B)$$

$$\mathcal{F}(\forall X [A]) = \mathcal{F}(A) \setminus \{ X \}$$

Et pour la substitution

$$(\forall X [A])[X \mapsto B] = \forall X [A]$$

$$(\forall X [A])[Y \mapsto B] = \forall X [A[Y \mapsto B]] \text{ avec } X \notin \mathcal{F}(B) \text{ (Gros bug !)}$$

## Identité polymorphe

Dans l'environnement  $E = \text{id} : \forall X[X \rightarrow X]$ , on souhaite typer l'application  $\text{id id}$ .

On y arrive par exemple ainsi.

$$\frac{\frac{E \vdash \text{id} : \forall X[X \rightarrow X]}{E \vdash \text{id} : (Z \rightarrow Z) \rightarrow (Z \rightarrow Z)} \quad \frac{E \vdash \text{id} : \forall X[X \rightarrow X]}{E \vdash \text{id} : Z \rightarrow Z}}{E \vdash (\text{id id}) : Z \rightarrow Z}$$

Les deux occurrences de  $\text{id}$  donnent lieu à deux *instanciations* différentes.

►  $(Z \rightarrow Z) \rightarrow (Z \rightarrow Z) = (X \rightarrow X)[X \mapsto (Z \rightarrow Z)]$

►  $Z \rightarrow Z = (X \rightarrow X)[X \mapsto Z]$

## Intérêt du polymorphisme

- ▶ Définir des fonctions dans une bibliothèque, une fois pour toutes.
- ▶ Les utiliser avec des types divers.

Exemple : composition de fonctions.

- ▶ Définir :

```
let (>>>) f g = fun x -> f (g x)
comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

- ▶ Emploi :

```
let f =
  string_of_int >>> (fun x -> x+2) >>> int_of_string
f : string -> string
```

```
f "5"
- : string = "7"
```

## Structures de données polymorphes

La liste de  $X$  est un type noté  $(\forall 'a.) 'a \text{ list}$  en Caml. Défini par deux « constructeurs »

- ▶ Nil (`[]`), de type  $(\forall 'a.) 'a \text{ list}$ .
- ▶ Cons (`::`), de type  $(\forall 'a.) 'a \rightarrow 'a \text{ list} \rightarrow 'a \text{ list}$ .

Toutes les fonctions qui ne « regardent pas » directement les éléments (de type  $'a$ ) sont polymorphes.

```
let rec fold_right f y0 xs = match xs with
| [] -> y0
| x::xs -> f x (fold_right f y0 xs)
fold_right : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b
```

Calcule  $f(x_1, f(x_2, \dots f(x_n, y_0)))$ .

Ou (plus classique) un tri :

```
sort : 'a list -> ('a -> 'a -> bool) -> 'a list
```



## Intérêt théorique du polymorphisme

Des constructions classiques du  $\lambda$ -calcul (PCF réduit à **Fun**, application et variables) sont typables.

Par exemple, l'encodage suivant des entiers, montre que nos constantes et opérateurs sur les entiers sont inutiles (dumoins en théorie).

- Les entiers (dits de Church)

**Let** zero = **Fun** f -> **Fun** x -> x

**Let** un = **Fun** f -> **Fun** x -> f x

...

L'entier  $n$  est le terme

$$\mathbf{Fun} \ f \ -> \ \mathbf{Fun} \ x \ -> \ \underbrace{f \ (f \ \dots \ (f \ x))}_{n \text{ applications}}$$

- Le type des entiers est  $N = \forall X [(X \rightarrow X) \rightarrow X \rightarrow X]$ .

Type des entiers  $N = \forall X[(X \rightarrow X) \rightarrow X \rightarrow X]$

Certaines primitives (toutes ?) sur les entiers sont alors typables:

► Successeur

```
Let succ = Fun n -> Fun f -> Fun x -> n f (f x)
```

Type  $N \rightarrow N$ .

► Addition et multiplication.

```
Let add =
```

```
  Fun n1 -> Fun n2 ->
```

```
    Fun f -> Fun x -> n1 f (n2 f x)
```

```
Let mul =
```

```
  Fun n1 -> Fun n2 ->
```

```
    Fun f -> Fun x -> n1 (n2 f) x
```

De types  $N \rightarrow N \rightarrow N$ .

## Introduction du $\forall$

Revenons sur le typage de `succ`

$$N = \forall X[(X \rightarrow X) \rightarrow X \rightarrow X]$$

$$\begin{array}{c}
 E \vdash n : N \\
 \hline
 E \vdash n : (Z \rightarrow Z) \rightarrow Z \rightarrow Z \quad E \vdash f : Z \rightarrow Z \quad \vdots \\
 \hline
 E \vdash (n \ f) : Z \rightarrow Z \quad E \vdash (f \ x) : Z \\
 \hline
 [n : N; f : Z \rightarrow Z; x : Z] \vdash n \ f \ (f \ x) : Z \\
 \hline
 [n : N; f : Z \rightarrow Z] \vdash (\mathbf{Fun} \ x \rightarrow n \ f \ (f \ x)) : Z \rightarrow Z \\
 \hline
 [n : N] \vdash (\mathbf{Fun} \ f \rightarrow \mathbf{Fun} \ x \rightarrow n \ f \ (f \ x)) : (Z \rightarrow Z) \rightarrow Z \rightarrow Z
 \end{array}$$

On veut conclure

$$[n : N] \vdash (\mathbf{Fun} \ f \rightarrow \mathbf{Fun} \ x \rightarrow \dots) : \forall Z[(Z \rightarrow Z) \rightarrow Z \rightarrow Z]$$

Et donc (variable muette), `succ` de type  $N \rightarrow N$ .

## Introduction du $\forall$

La règle de *généralisation*.

$$\frac{E \vdash t : A}{E \vdash t : \forall X [A]} \quad X \notin \mathcal{F}(E)$$

Pour succ la généralisation est légitime, car  $Z$  est quelconque.

Pensons à la locution « Soit  $n$  un entier quelconque ».

Par exemple, pour  $a$  et  $b$  quelconques,  $(a + b)^2 = a^2 + 2ab + b^2$  et donc

$$\forall a, b \in \mathbb{N}^2, (a + b)^2 = a^2 + 2ab + b^2$$

Pour exprimer «  $X$  quelconque » :  $X$  n'apparaît pas dans  $E$ .

En déduction, cela correspond à l'absence d'hypothèses.

Par ex. de  $n$  pair alors  $n^2$  pair, on ne peut *pas* déduire que tous les carrés sont pairs.

## Et si on ne fait pas attention

On peut typer `Obj.magic : 'a -> 'b`.

$$\frac{\frac{\frac{[x : X] \vdash x : X}{[x : X] \vdash x : \forall X [X]}}{[x : X] \vdash x : Y}}{\vdash (\mathbf{Fun} \ x \ -> \ x) : X \ -> \ Y}}$$

En effet,

- ▶ En combinait généralisation (abusive) de `X`, et instanciation de `X` en `Y`, on donne n'importe quel type à `x`.
- ▶ Et le tour est joué..

```
let magic = Fun x -> x In (magic 1) 2
```

Se réduit en `1 2`, c-à-d une erreur de type à l'exécution.

## Le système F

$$\frac{E \vdash t : \forall X [A]}{E \vdash t : A[X \mapsto B]} \qquad \frac{E \vdash t : A \quad X \notin \mathcal{F}(E)}{E \vdash t : \forall X [A]}$$

$$\frac{E(x) = A}{E \vdash x : A} \qquad \frac{E \vdash t_1 : A \rightarrow B \quad E \vdash t_2 : A}{E \vdash (t_1 \ t_2) : B}$$

$$\frac{E \oplus [x : A] \vdash t : B}{E \vdash (\mathbf{Fun} \ x \rightarrow t) : A \rightarrow B}$$

De Girard (1970, logicien) et Reynolds (1975, informaticien).

Notre présentation est emprunté à Wells (1994), qui a prouvé que la synthèse de type est indécidable.

## Règles supplémentaires

Inutiles, si on considère que les constantes  $c$  ont des types présents dans l'environnement de départ.

- ▶ **Let**  $x = t_1$  **In**  $t_2$  se type comme  $(\mathbf{Fun} \ x \ -> \ t_2) \ t_1$ .
- ▶ Le type de tout  $n$  est **Nat**, autrement dit.

$$E \vdash n : \mathbf{Nat}$$

- ▶ Types des opérations, par ex.  $+$ , vu comme une fonction  $(+)$  de type  $\mathbf{Nat} \ -> \ \mathbf{Nat} \ -> \ \mathbf{Nat}$ .  
Alors,  $t_1 + t_2$  se comprend comme  $(+) \ t_1 \ t_2$ .
- ▶ Plus sioux **ifz** de type  $\forall X[\mathbf{Nat} \ -> \ X \ -> \ X]$ .  
Alors, **Ifz**  $t_1$  **Then**  $t_2$  **Else**  $t_3$  se comprend comme **ifz**  $t_1 \ t_2 \ t_3$
- ▶ Même le **Fix** peut se voir ainsi comme une règle dérivée...

## La récursion

On suppose donné, un combinateur de point fixe **fix** de type  $\forall X [(X \rightarrow X) \rightarrow X]$ .

Alors **Fix**  $f \rightarrow t$  se type comme **fix** appliqué à (**Fun**  $f \rightarrow t$ ).

Au lieu d'ajouter une règle du style :

$$\frac{E \oplus [f : A] \vdash t : A}{E \vdash (\mathbf{Fix} f \rightarrow t) : A}$$

On se contente des règles du **Fun** et de l'application.

$$\frac{\frac{E \vdash \mathbf{fix} : \forall X [(X \rightarrow X) \rightarrow X]}{E \vdash \mathbf{Fix} : (A \rightarrow A) \rightarrow A} \quad \frac{E \oplus [f : A] \vdash t : A}{E \vdash (\mathbf{Fun} f \rightarrow t) : A \rightarrow A}}{E \vdash \mathbf{fix} (\mathbf{Fun} f \rightarrow t) : A}$$

Une telle simplification n'est pas négligeable, quand on code l'inférence.



## Retrouver l'inférence

L'inférence de type du système F n'est pas décidable.

► Limiter l'instanciation.

- ▷ Le  $\forall$  ne peut se trouver que dans les types de l'environnement  $E$ .
- ▷  $\forall$  doit se trouver en tête  $\forall X_1. \forall X_2 \dots \forall X_n. A$
- ▷ Toutes les variables sont instanciées à la fois.

► Limiter la généralisation.

- ▷ La généralisation est aussi complète que possible.

Pour généraliser le type  $A$  dans l'env.  $E$  on applique la fonction de généralisation :  $\text{gen}(A, E)$

$$\text{gen}(A, E) = \forall X_1 \dots X_n [A], \quad \text{où } \{ X_1, \dots, X_n \} = \mathcal{F}(A) \setminus \mathcal{F}(E)$$

- ▷ Et ne s'applique qu'aux types des variables liées par **Let**.

## Le typage de ML : (Damas Milner) Types et schémas

Types ( $A, B$  etc.), comme avant !

Nat                       $A \rightarrow B$                        $X$

Schémas de type ( $S$ )

$\forall X_1 \cdots X_n [A]$

Les schémas sont réservés aux environnements de typage

$[\cdots; x : S; \cdots]$

**Note :** un type ordinaire  $A$  est représenté par le schéma  $\forall \emptyset [A]$ .

## Damas-Milner : règles

$$\frac{E(x) = \forall X_1 \cdots X_n [A]}{E \vdash x : A[X_1 \mapsto B_1, \dots, X_n \mapsto B_n]}$$

$$\frac{E \vdash t_1 : A \quad E \oplus [x : \text{gen}(A, E)] \vdash t_2 : B}{E \vdash (\mathbf{Let} \ x = t_1 \ \mathbf{In} \ t_2) : B}$$

$$\frac{E \vdash t_1 : A \rightarrow B \quad E \vdash t_2 : A}{E \vdash (t_1 \ t_2) : B}$$

$$\frac{E \oplus [x : \forall \emptyset [A]] \vdash t : B}{E \vdash (\mathbf{Fun} \ x \rightarrow t) : A \rightarrow B}$$

Et  $\text{if} : \forall X [\text{Nat} \rightarrow X \rightarrow X]$ ,  $\text{fix} : \forall X [(X \rightarrow X) \rightarrow X]$  etc.

## L'unification incrémentale

Il devient délicat de procéder à l'inférence en deux phases

Car le langage des équations devient plus riche que  $A = B$  (il doit comprendre des schémas de type).

Mais la résolution immédiate fonctionne.

Revoyons l'idée : résoudre les équations dès qu'elles sont posées.

## Une règle de résolution immédiate

On avait :

$$\frac{E \vdash t_1 \rightsquigarrow A_1, \mathcal{E}_1 \quad E \vdash t_2 \rightsquigarrow A_2, \mathcal{E}_2}{E \vdash t_1 t_2 \rightsquigarrow X, \mathcal{E}_1 \cup \mathcal{E}_2 \cup [A_1 = A_2 \rightarrow X]} (X \text{ frais})$$

On a :

$$\frac{E, \sigma \vdash t_1 \rightsquigarrow A_1, \sigma_1 \quad E, \sigma_1 \vdash t_2 \rightsquigarrow A_2, \sigma_2}{E, \sigma \vdash t_1 t_2 \rightsquigarrow X, \text{mgu}[A_1 = (A_2 \rightarrow X), \sigma_2]} (X \text{ frais})$$

La forme générale du jugement est

$$E, \sigma \vdash t \rightsquigarrow A, \sigma'$$

Le composant  $\sigma$  représente la solution courante des équations

« vues ».

Propriété :  $\sigma'$  étend  $\sigma$  ( $\sigma' = \sigma'' \circ \sigma$ ) et  $\sigma'(E) \vdash t : \sigma'(A)$  (et même type principal).

## L'unification incrémentale

Soit  $\sigma$  la solution d'un problème d'unification. C'est à dire

$\sigma = \{ X_1 \mapsto A_1, \dots, X_n \mapsto A_n \}$ , avec :

- ▶ Les  $X_i$  sont deux à deux distincts.
- ▶ Les  $X_i$  n'apparaissent pas dans les  $A_i$ .

Et soit une nouvelle équation  $B = C$ .

On veut résoudre les équation  $\mathcal{E} = \{ B = C \} \cup \sigma$ .

On pourrait simplement poser  $\text{mgu}(\mathcal{E})$ , mais ça serait dommage d'oublier que  $\sigma$  est déjà résolu.

## Une vue incrémentale de l'unification

Pour résoudre  $\{ B = C, X_1 \mapsto A_1, \dots, X_n \mapsto A_n \}$ .

1. Remplacer les  $X_i$  par les  $A_i$  dans  $B$  et  $C$  — les  $X_i$  n'apparaissent plus.
2. Résoudre la nouvelle équation :  $\sigma' = \{ Y_1 \mapsto B_1, \dots, Y_m \mapsto B_m \}$  — les  $X_i$  n'apparaissent ni dans les  $Y_j$ , ni dans les  $B_j$ .
3. Remplacer les  $Y_i$  par les  $B_i$  dans les  $A_j$ , ce qui donne les  $C_j$  — les  $Y_i$  n'apparaissent plus.
4. Renvoyer  $\{ Y_1 \mapsto B_1, \dots, Y_m \mapsto B_m, X_1 \mapsto C_1, \dots, X_n \mapsto C_m \}$ .

Cela s'abrège en  $\text{mgu}[\sigma(B) = \sigma(C)] \circ \sigma$ .

La notation  $\circ$  exprimant les étapes 3. et 4. mais de façon trop générale (quid si  $X_i = Y_j$  ?).

## Un exemple

**Fix** pow  $\rightarrow$  **Fun** n  $\rightarrow$  **Ifz** n **Then** 1 **Else** 2\*pow (n-1)

pow est de type P, n est de type N ( $E$  est  $[x : X, \text{pow} : P]$ ).

► De **Ifz**, il vient d'abord,  $\sigma = [N \mapsto \mathbf{Nat}]$ , puis.

▷ Pour le **Then**  $E, \sigma \vdash 1 \rightsquigarrow \mathbf{Nat}, \sigma$ .

▷ Et le **Else**

★ D'abord  $E, \sigma \vdash 2 \rightsquigarrow \mathbf{Nat}, \sigma$ ,

★ Puis n-1 donne  $\text{mgu}[\sigma(N) = \mathbf{Nat}] \circ \sigma$  qui ne change rien

★ Puis surtout (application pow (n-1))

$\text{mgu}[\sigma(P) = \sigma(\mathbf{Nat} \rightarrow Y)] \circ \sigma$ . Qui vaut

$\sigma' = [P \mapsto (\mathbf{Nat} \rightarrow Y), X \mapsto \mathbf{Nat}]$ .

★ Et enfin le deuxième argument de « \* » conduit au calcul



de  $\text{mgu}[\sigma'(Y) = \mathbf{Nat}] \circ \sigma'$  qui est

$$\sigma'' = [Y \mapsto \mathbf{Nat}, P \mapsto (\mathbf{Nat} \rightarrow \mathbf{Nat}), N \mapsto \mathbf{Nat}]$$

Nous avons donc le bilan du typage du **Ifz**

$$E, \text{id} \vdash (\mathbf{Ifz} \ n \ \mathbf{Then} \ 1 \ \mathbf{Else} \ 2 * \text{pow} \ (n-1)) \rightsquigarrow \mathbf{Nat}, \sigma''$$

- ▶ Le type de **Fun** est  $N \rightarrow \mathbf{Nat}$ ,
- ▶ C'est donc le type de **Fix**, avec à calculer

$$\text{mgu}[\sigma''(P) = \sigma''(N \rightarrow \mathbf{Nat})] \circ \sigma''$$

L'équation est triviale, et la substitution ne change pas.

## L'algorithme $\mathcal{J}$

$$\frac{E, \sigma \vdash t_1 \rightsquigarrow A, \sigma_1 \quad E \oplus [x = \text{gen}(\sigma_1(A), \sigma_1(E))], \sigma \vdash t_2 \rightsquigarrow B, \sigma_2}{E, \sigma \vdash \mathbf{Let} \ x = t_1 \ \mathbf{In} \ t_2 \rightsquigarrow B, \sigma_2}$$

$$\frac{E(x) = \forall X_1 \cdots X_n [A]}{E, \sigma \vdash x \rightsquigarrow A[X_1 \mapsto Y_1, \dots, X_n \mapsto Y_n], \sigma} (Y_1, \dots, Y_n \text{ frais})$$

$$\frac{E \oplus [x = \forall \emptyset [X]], \sigma \vdash t \rightsquigarrow A, \sigma'}{E, \sigma \vdash (\mathbf{Fun} \ x \rightarrow t) \rightsquigarrow X \rightarrow A, \sigma'} (X \text{ frais})$$

$$\frac{E, \sigma \vdash t_1 \rightsquigarrow A, \sigma_1 \quad E, \sigma_1 \vdash t_2 \rightsquigarrow B, \sigma_2}{E, \sigma \vdash t_1 \ t_2 \rightsquigarrow X, \text{mgu}[\sigma_2(A) = \sigma_2(B \rightarrow X)] \circ \sigma_2} (X \text{ frais})$$

Et les constantes `ifz` de type  $\forall X[\mathbf{Nat} \rightarrow X \rightarrow X]$ , `fix` de type  $\forall X[(X \rightarrow X) \rightarrow X]$  etc.

## Prenons un exemple

**Let**  $\text{id} = \mathbf{Fun} \ x \rightarrow x \ \mathbf{In} \ \text{id} \ \text{id}$

Inférons d'abord le type de  $\mathbf{Fun} \ x \rightarrow x$ . Rien de bien sorcier.

$$\frac{[x : X], \text{id} \vdash x \rightsquigarrow X, \text{id}}{\emptyset, \text{id} \vdash (\mathbf{Fun} \ x \rightarrow x) \rightsquigarrow X \rightarrow X, \text{id}}$$

On a  $\text{gen}(\text{id}(X \rightarrow X), \text{id}(\emptyset)) = \forall X[X \rightarrow X]$ . L'application  $\text{id} \ \text{id}$  est à typer dans  $E = [\text{id} : \forall X[X \rightarrow X]]$ .

$$\frac{\text{id}, E \vdash \text{id} \rightsquigarrow Y \rightarrow Y, \text{id} \quad \text{id}, E \vdash \text{id} \rightsquigarrow Z \rightarrow Z, \text{id}}{\text{id}, E \vdash (\text{id} \ \text{id}) \rightsquigarrow W, \text{mgu}[\text{id}(Y \rightarrow Y) = \text{id}((Z \rightarrow Z) \rightarrow W)] \circ \text{id}}$$

La résolution donne  $[Y \mapsto (Z \rightarrow Z), W \mapsto (Z \rightarrow Z)]$ .

## Prenons un autre exemple

Soit à typer **Fun**  $f \rightarrow$  **Fun**  $x \rightarrow$  **Let**  $y = f\ x$  **In**  $y+1$ .

- ▶ Le **Let** est à typer dans l'environnement  $E = [f = \forall \emptyset [F], x = \forall \emptyset [X]]$ . On type d'abord  $f\ x$ .
  - ▷ Le typage des variables produit les types  $F$  et  $X$ .
  - ▷ On doit résoudre l'équation  $F = X \rightarrow Y$ , dont la solution est  $\sigma = \{ F \mapsto X \rightarrow Y \}$
  - ▷ Le type de  $f\ x$  est  $Y$ .
- ▶ Généralisé en  $\forall \emptyset [Y]$ , car  $\sigma(E) = [f = \forall \emptyset [X \rightarrow Y], \dots]$ . On type donc  $y+1$  dans  $E' = E \oplus [y = \forall \emptyset [Y]]$ .
- ▶ Rappelons que  $y+1$  se type comme  $(+)$   $y\ 1$  avec le type de  $(+)$   $\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$ .

## La suite

- ▶  $[\dots, y = \forall \emptyset[Y]], \sigma \vdash ((+) y) 1 \rightsquigarrow ?, ?$  avec  $\sigma = \{ F \mapsto X \rightarrow Y \}$ .

- ▷ Typer  $(+)$  appliqué à  $y$  produit l'équation  
 $\mathbf{Nat} \rightarrow (\mathbf{Nat} \rightarrow \mathbf{Nat}) = Y \rightarrow Z$ ,  $\sigma$  devient

$$\{ Y \mapsto \mathbf{Nat}, Z \mapsto \mathbf{Nat} \rightarrow \mathbf{Nat}, F \mapsto X \rightarrow \mathbf{Nat} \}$$

Le type rendu est  $Z$ .

- ▷ Typer  $(+) y$  appliqué à  $1$ , produit l'équation  
 $\mathbf{Nat} \rightarrow \mathbf{Nat} = \mathbf{Nat} \rightarrow W$ ,  $\sigma$  devient

$$\{ W \mapsto \mathbf{Nat}, Y \mapsto \mathbf{Nat}, Z \mapsto \mathbf{Nat} \rightarrow \mathbf{Nat}, F \mapsto X \rightarrow \mathbf{Nat} \}$$

Le type rendu est  $W$ .

- ▶ Le type de  $\mathbf{Let} y = f \ x \ \mathbf{In} \ y+1$  est donc  $W$ .
- ▶ Le type de  $\mathbf{Fun} \ x \rightarrow \dots$  est  $X \rightarrow W$ , celui de  $\mathbf{Fun} \ f \rightarrow \dots$  est  $F \rightarrow (X \rightarrow W)$ , soit finalement  $(X \rightarrow \mathbf{Nat}) \rightarrow X \rightarrow \mathbf{Nat}$ .

## Culture : ML

ML est un langage fondée sur le typage de Damas-Milner. La synthèse de type (et donc l'existence de typage principaux) est fondamentale.

Cela pose des problèmes et freine un peu les extensions possibles

- ▶ La surcharge (quel est le type de **fun**  $x \rightarrow x+x$  ?).
- ▶ Le typage de l'égalité ( $=: 'a \rightarrow 'a \rightarrow \text{bool}$ ) n'est pas bien satisfaisant, si  $'a$  vaut  $'b \rightarrow 'c \dots$
- ▶ Les systèmes avec sous-typage (inférence difficile).

Il y a encore de la recherche sur ces questions, les *type classes* de Haskell proposent une solution intéressante aux deux premiers problèmes.

## Culture : les génériques de Java

Le polymorphisme (paramétrique) existe en Java !

```
class Poly<E> {  
    Poly() {}  
    E id(E x) { return x ; }  
  
    public static void main(String [] args) {  
        int x = new Poly<Integer>().id(10) ;  
        String s = new Poly<String>().id(args[0]) ;  
        Poly<String> p = new Poly<String> () ;  
        Poly<String> q = new Poly<Poly<String>> ().id(p) ;  
    }  
}
```

Mais c'est un peu plus explicite qu'en ML. (En échange, le sous-typage est possible.)

Se montre suffisant pour les structure de données (Set<E> etc.)

- ▶ TP, inférence des types polymorphes.
- ▶ La prochaine fois, PCF avec  $:=$ ,  $!$  etc.