INF 544 Luc Maranget

Polymorphisme

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

Limitation des types simples

L'identité **Fun** `x -> x` a plusieurs types.

Type principal `X -> X`.

Mais **Let** `id` = **Fun** `x -> x` **In** `id id` n'est pas typable.

$$\frac{[\mathtt{id} : \mathtt{X\ ->\ X}] \vdash \mathtt{id} \rightsquigarrow \mathtt{X\ ->\ X},\ \emptyset \qquad [\mathtt{id} : \mathtt{X\ ->\ X}] \vdash \mathtt{id} \rightsquigarrow \mathtt{X\ ->\ X},\ \emptyset}{[\mathtt{id} : \mathtt{X\ ->\ X}] \vdash \mathtt{id\ id} \rightsquigarrow \mathtt{Y},\ (\mathtt{X\ ->\ X}) = ((\mathtt{X\ ->\ X})\ \mathtt{->\ Y})}$$

Ce qui conduit finalement à l'équation $\mathtt{X} = \mathtt{X\ ->\ X}$, qui n'a pas de solution.

Et pourtant, puisque `id` possède tous les types A `->` A, les types suivants sont possibles pour la première et la seconde occurrence de `id`.

`(Z -> Z) -> (Z -> Z)` `Z -> Z`

Et l'application `id id` a pour type `Z -> Z`.

Solution

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

Enrichir les types :

$$\mathbf{Nat} \in \mathcal{T} \qquad X \in \mathcal{T} \qquad \frac{A_1 \in \mathcal{T} \qquad A_2 \in \mathcal{T}}{A_1\ \mathtt{->}\ A_2 \in \mathcal{T}} \qquad \frac{A \in \mathcal{T}}{\forall X\,[A] \in \mathcal{T}}$$

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 contienent maintenant un lieur ($\forall$), comme les termes (**Fun** etc.)

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

$$\mathcal{F}(X) = \{\, X \,\} \qquad \mathcal{F}(\mathbf{Nat}) = \emptyset \qquad \mathcal{F}(A \texttt{ -> } B) = \mathcal{F}(A) \cup \mathcal{F}(B)$$

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

Et pour la substitution

$$(\forall\, X\,\texttt{[}A\texttt{]})[X \mapsto B] = \forall\, X\,\texttt{[}A\texttt{]}$$

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

Identitité polymorphe

Dans l'environnement $E = \texttt{id} : \forall\,\texttt{X[X -> X]}$, on souhaite typer l'application `id id`.

On y arrive par exemple ainsi.

$$\dfrac{\dfrac{E \vdash \texttt{id} : \forall\,\texttt{X[X -> X]}}{E \vdash \texttt{id} : \texttt{(Z -> Z) -> (Z -> Z)}} \qquad \dfrac{E \vdash \texttt{id} : \forall\,\texttt{X[X -> X]}}{E \vdash \texttt{id} : \texttt{Z -> Z}}}{E \vdash (\texttt{id id}) : \texttt{Z -> Z}}$$

Les deux occurrences de `id` donnent lieu à deux *instanciations* différentes.

- $\texttt{(Z -> Z) -> (Z -> Z)} = (\texttt{X -> X})[\texttt{X} \mapsto (\texttt{Z -> Z})]$
- $\texttt{Z -> Z} = (\texttt{X -> X})[\texttt{X} \mapsto \texttt{Z}]$

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 list` en Caml. Défini par deux « constructeurs »

- Nil (`[]`), de type ($\forall$`'a.`)`'a list`.
- Cons (`::`), de type ($\forall$`'a.`)`'a -> 'a list -> 'a 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 $f(x_1, f(x_2, \ldots f(x_n, y_0)))$.

Ou (plus classique) un tri :

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

Intérêt théorique du polymorphisme

Des constructions classiques du λ-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

  $$\textbf{Fun}\ \texttt{f -> }\textbf{Fun}\ \texttt{x -> }\underbrace{\texttt{f (f}\ \cdots\ \texttt{(f x))}}_{n\ \text{applications}}$$

- Le type des entiers est $N = \forall\,\texttt{X[(X -> X) -> X -> X]}$.

Type des entiers $N = \forall\,\texttt{X[(X -> X) -> X -> X]}$

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

- Successeur

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

  Type `N -> 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 -> N -> N`.

Introduction du $\forall$

Revenons sur le typage de `succ`

$$N = \forall\,\texttt{X[(X -> X) -> X -> X]}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{E \vdash \texttt{n} : N}{E \vdash \texttt{n} : \texttt{(Z -> Z) -> Z -> Z}} \qquad E \vdash \texttt{f} : \texttt{Z -> Z}}{E \vdash \texttt{(n f)} : \texttt{Z -> Z}} \qquad \dfrac{\vdots}{E \vdash \texttt{(f x)} : \texttt{Z}}}{[\texttt{n} : N; \texttt{f} : \texttt{Z -> Z}; \texttt{x} : \texttt{Z}] \vdash \texttt{n f (f x)} : \texttt{Z}}}{[\texttt{n} : N; \texttt{f} : \texttt{Z -> Z}] \vdash (\textbf{Fun}\ \texttt{x -> n f (f x)}) : \texttt{Z -> Z}}}{[\texttt{n} : N] \vdash (\textbf{Fun}\ \texttt{f -> }\textbf{Fun}\ \texttt{x -> n f (f x)}) : \texttt{(Z -> Z) -> Z -> Z}}$$

On veut conclure

$$[\texttt{n} : N] \vdash (\textbf{Fun}\ \texttt{f -> }\textbf{Fun}\ \texttt{x -> }\cdots) : \forall\,\texttt{Z[(Z -> Z) -> Z -> Z]}$$

Et donc (variable muette), `succ` de type N `->` N.

Introduction du $\forall$

La règle de *généralisation.*

$$\dfrac{E \vdash t : A}{E \vdash t : \forall\,X\,\texttt{[}A\texttt{]}}\,X \notin \mathcal{F}(E)$$

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`.

$$\dfrac{\dfrac{\dfrac{[\mathtt{x} : \mathtt{X}] \vdash \mathtt{x} : \mathtt{X}}{[\mathtt{x} : \mathtt{X}] \vdash \mathtt{x} : \forall\, \mathtt{X}\,[\mathtt{X}]}}{[\mathtt{x} : \mathtt{X}] \vdash \mathtt{x} : \mathtt{Y}}}{\vdash (\mathbf{Fun}\ \mathtt{x}\ \texttt{->}\ \mathtt{x}) : \mathtt{X}\ \texttt{->}\ \mathtt{Y}}$$

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

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

$$\frac{E(x) = A}{E \vdash x : A} \qquad \frac{E \vdash t_1 : A\ \texttt{->}\ B \qquad E \vdash t_2 : A}{E \vdash (t_1\ t_2) : B}$$

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

De Girard (1970, logicien) et Reynols (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 (**Fun** x -> t_2) t_1.
- Le type de tout `n` est **Nat**, autrement dit.

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

- Types des opérations, par ex. `+`, vu comme une fonction `(+)` de type **Nat** -> **Nat** -> **Nat**.

  Alors, t_1 `+` t_2 se comprend comme `(+)` t_1 t_2.
- Plus sioux `ifz` de type $\forall\,\mathtt{X}\,[\mathbf{Nat}\ \texttt{->}\ \mathtt{X}\ \texttt{->}\ \mathtt{X}]$.

  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 suppse donné, un combinateur de point fixe `fix` de type $\forall\,\mathtt{X}\,[(\mathtt{X}\ \texttt{->}\ \mathtt{X})\ \texttt{->}\ \mathtt{X}]$.

Alors **Fix** f -> t se type comme `fix` appliqué à (**Fun** f -> t).

Au lieu d'ajouter une règle du style :

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

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

$$\dfrac{\dfrac{E \vdash \mathtt{fix} : \forall\,\mathtt{X}\,[(\mathtt{X}\ \texttt{->}\ \mathtt{X})\ \texttt{->}\ \mathtt{X}]}{E \vdash \mathtt{Fix} : (A\ \texttt{->}\ A)\ \texttt{->}\ A} \qquad \dfrac{E \oplus [f : A] \vdash t : A}{E \vdash (\mathbf{Fun}\ f\ \texttt{->}\ t) : A\ \texttt{->}\ A}}{E \vdash \mathtt{fix}\ (\mathbf{Fun}\ f\ \texttt{->}\ t) : A}$$

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 $\forall X_1.\forall X_2 \ldots \forall X_n.A$
  - 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 : $\mathsf{gen}(A, E)$

    $$\mathsf{gen}(A,E) = \forall\, X_1 \cdots X_n\, \texttt{[}A\texttt{]}, \quad \text{où } \{\, X_1, \ldots, X_n \,\} = \mathcal{F}(A) \setminus \mathcal{F}(E)$$
  - 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 !

$$\mathbf{Nat} \qquad\qquad A \texttt{ -> } B \qquad\qquad X$$

Schémas de type (S)

$$\forall\, X_1 \cdots X_n\, \texttt{[}A\texttt{]}$$

Les schémas sont réservés aux environnements de typage

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

Note : un type ordinaire A est représenté par le schéma $\forall\,\emptyset\,\texttt{[}A\texttt{]}$.

Damas-Milner : règles

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

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

$$\frac{E \vdash t_1 : A \texttt{ -> } B \qquad E \vdash t_2 : A}{E \vdash (t_1\ t_2) : B}$$

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

Et `if` : $\forall\,\texttt{X[}\mathbf{Nat}\texttt{ -> X -> X]}$, `fix` : $\forall\,\texttt{X[(X -> X) -> X]}$ etc.

L'unification incrémentale

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

Car le language 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 :

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

On a :

$$\frac{E, \sigma \vdash t_1 \rightsquigarrow A_1, \sigma_1 \qquad E, \sigma_1 \vdash t_2 \rightsquigarrow A_2, \sigma_2}{E, \sigma \vdash t_1\ t_2 \rightsquigarrow X, \mathsf{mgu}[A_1 = (A_2\ \texttt{->}\ X), \sigma_2]}(X \text{ frais})$$

La forme générale du jugement est

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

Le composant σ représente la solution courante des équations « vues ».

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

L'unification incrémentale

Soit σ la solution d'un problème d'unification. C'est à dire $\sigma = \{\, X_1 \mapsto A_1, \ldots, X_n \mapsto A_n \,\}$, 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 $\mathsf{mgu}(\mathcal{E})$, mais ça serait dommage d'oublier que σ est déjà résolu.

Une vue incrémentale de l'unification

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

1. Remplacer les X_i par les A_i dans B et C — les X_i n'apparaissent plus.
2. Résoudre la nouvelle équation : $\sigma' = \{\, Y_1 \mapsto B_1, \ldots, Y_m \mapsto B_m \}$ — 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 $\{\, Y_1 \mapsto B_1, \ldots, Y_m \mapsto B_m, X_1 \mapsto C_1, \ldots, X_n \mapsto C_m \,\}$.

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

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

Un exemple

Fix `pow` -> **Fun** `n` -> **Ifz** `n` **Then** `1` **Else** `2*pow (n-1)`

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

- De **Ifz**, il vient d'abord, $\sigma = [\texttt{N} \mapsto \mathbf{Nat}]$, puis.
  - Pour le **Then** $E, \sigma \vdash \texttt{1} \rightsquigarrow \mathbf{Nat}, \sigma$.
  - Et le **Else**
    - D'abord $E, \sigma \vdash \texttt{2} \rightsquigarrow \mathbf{Nat}, \sigma$,
    - Puis `n-1` donne $\mathsf{mgu}[\sigma(\texttt{N}) = \mathbf{Nat}] \circ \sigma$ qui ne change rien
    - Puis surtout (application `pow (n-1)`) $\mathsf{mgu}[\sigma(\texttt{P}) = \sigma(\mathbf{Nat}\ \texttt{->}\ \texttt{Y})] \circ \sigma$. Qui vaut $\sigma' = [\texttt{P} \mapsto (\mathbf{Nat}\ \texttt{->}\ \texttt{Y}), \texttt{X} \mapsto \mathbf{Nat}]$.
    - Et enfin le deuxième argument de « `*` » conduit au calcul

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

$$\sigma'' = [\mathtt{Y} \mapsto \mathbf{Nat}, \mathtt{P} \mapsto (\mathbf{Nat} \texttt{ -> } \mathbf{Nat}), \mathtt{N} \mapsto \mathbf{Nat}]$$

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

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

- Le type de **Fun** est `N ->` **Nat**,
- C'est donc le type de **Fix**, avec à calculer

$$\mathsf{mgu}[\sigma''(\mathtt{P}) = \sigma''(\mathtt{N} \texttt{ -> } \mathbf{Nat})] \circ \sigma''$$

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

L'algorithme $\mathcal{J}$

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

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

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

$$\frac{E, \sigma \vdash t_1 \rightsquigarrow A, \sigma_1 \qquad E, \sigma_1 \vdash t_2 \rightsquigarrow B, \sigma_2}{E, \sigma \vdash t_1\ t_2 \rightsquigarrow X, \mathsf{mgu}[\sigma_2(A) = \sigma_2(B \texttt{ -> } X)] \circ \sigma_2} (X \text{ frais})$$

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

Prenons un exemple

Let `id` **=** **Fun** `x -> x` **In** `id id`

Inférons d'abord le type de **Fun** `x -> x`. Rien de bien sorcier.

$$\frac{[\mathtt{x} : \mathtt{X}], \mathsf{id} \vdash \mathtt{x} \rightsquigarrow \mathtt{X}, \mathsf{id}}{\emptyset, \mathsf{id} \vdash (\mathbf{Fun}\ \mathtt{x} \texttt{ -> } \mathtt{x}) \rightsquigarrow \mathtt{X} \texttt{ -> } \mathtt{X}, \mathsf{id}}$$

On a $\mathsf{gen}(\mathsf{id}(\mathtt{X \texttt{ -> } X}), \mathsf{id}(\emptyset)) = \forall \mathtt{X}\texttt{[X -> X]}$. L'application `id id` est à typer dans $E = [\mathtt{id} : \forall \mathtt{X}\texttt{[X -> X]}]$.

$$\frac{\mathsf{id}, E \vdash \mathtt{id} \rightsquigarrow \mathtt{Y \texttt{ -> } Y}, \mathsf{id} \qquad \mathsf{id}, E \vdash \mathtt{id} \rightsquigarrow \mathtt{Z \texttt{ -> } Z}, \mathsf{id}}{\mathsf{id}, E \vdash (\mathtt{id\ id}) \rightsquigarrow \mathtt{W}, \mathsf{mgu}[\mathsf{id}(\mathtt{Y \texttt{ -> } Y}) = \mathsf{id}((\mathtt{Z \texttt{ -> } Z}) \texttt{ -> } \mathtt{W})] \circ \mathsf{id}}$$

La résolution donne $[\mathtt{Y} \mapsto (\mathtt{Z \texttt{ -> } Z}), \mathtt{W} \mapsto (\mathtt{Z \texttt{ -> } Z})]$.

Prenons un autre exemple

Soit à typer **Fun** `f ->` **Fun** `x ->` **Let** `y = f x` **In** `y+1`.

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

La suite

- $[\ldots, \mathtt{y} = \forall\emptyset\mathtt{[Y]}], \sigma \vdash \mathtt{((+)\ y)\ 1} \rightsquigarrow ?, ?$ avec $\sigma = \{\, \mathtt{F} \mapsto \mathtt{X\ {-}{>}\ Y} \,\}$.
  - Typer `(+)` appliqué à `y` produit l'équation
    Nat `->` (**Nat** `->` **Nat**) = `Y -> Z`, σ devient

    $$\{\, \mathtt{Y} \mapsto \mathbf{Nat}, \mathtt{Z} \mapsto \mathbf{Nat}\ \mathtt{{-}{>}}\ \mathbf{Nat}, \mathtt{F} \mapsto \mathtt{X\ {-}{>}}\ \mathbf{Nat} \,\}$$

    Le type rendu est `Z`.
  - Typer `(+) y` appliqué à `1`, produit l'équation
    Nat `->` **Nat** = **Nat** `-> W`, σ devient

    $$\{\, \mathtt{W} \mapsto \mathbf{Nat}, \mathtt{Y} \mapsto \mathbf{Nat}, \mathtt{Z} \mapsto \mathbf{Nat}\ \mathtt{{-}{>}}\ \mathbf{Nat}, \mathtt{F} \mapsto \mathtt{X\ {-}{>}}\ \mathbf{Nat} \,\}$$

    Le type rendu est `W`.
- Le type de **Let** `y = f x` **In** `y+1` est donc `W`.
- Le type de **Fun** `x ->` $\cdots$ est `X -> W`, celui de **Fun** `f ->` $\cdots$ est `F -> (X -> W)`, soit finalement (`X ->` **Nat**) `-> X ->` **Nat**.

Culture : ML

ML est un language 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 -> x+x` ?).
- Le typage de l'égalité (`=:'a -> 'a -> bool`) n'est pas bien satisfaisant, si `'a` vaut `'b -> 'c`...
- 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.