# TAS TD 3 - Typeur de Lambda-Calcul

## 1 Préliminaires

L'objectif de ce TD est d'écrire, dans un langage libre*, un typeur pour un $\lambda$-calcul contenant différentes fonctionnalités.

### 1.1 Langage

Le choix du langage est plutôt libre, mais:

- les langages **fonctionnels** avec *pattern-matching* (comme `Haskell` ou `OCaml`) sont fortement conseillés. Leur utilisation facilite grandement la rédaction de fonction "par cas" et le travail sur l'arbre syntaxique des termes ou des types.
- les langages avec des fonctionnalités **objet** (comme `Python` ou `Java`) sont plus compliqués à utiliser dans ce cadre, mais l'héritage et les tests d'instance peuvent être utilisés pour faciliter l'écriture de fonction récursive sur les arbres syntaxiques.
- les langages de **bas-niveau** (comme `C` ou `Rust`) sont assez fastidieux à utiliser pour cette tâche, car ils nécessitent de manipuler des structures et des enchainements pointeurs pour parcourir les arbres syntaxiques.
- `Prolog` et `Go` sont interdits. Le premier langage contient un algorithme de résolution qui rend l'écriture d'une fonction d'unification superflue. Le second est le langage utilisé comme exemples dans l'énoncé et la correction (et c'est un langage bas-niveau).

## 2 $\lambda$-calcul simplement typé

Dans une première partie, on construira un typeur reconnaissant le lambda calcul simplement typé.

### 2.1 Syntaxe

On rappelle la syntaxe de $\lambda$:

$$M, N ::= x \ \mid \ \lambda x.M \ \mid \ M \ N$$

1. Donner une syntaxe inductive du lambda-calcul à l'aide d'un type somme, d'une classe ou d'une *struct*

```
type lterme struct {
        est string // nature
        // V -> Variable
        // A -> Application
        // L -> Abstraction
        vari string // la variable du terme variable ou celle liee dans l'abstraction
        fpos *lterme // membre gauche d'une application
        apos *lterme // membre droit d'une application
        corps *lterme // corps d'une fonctions
}
```

2. Ecrire un *pretty printer* pour cette syntaxe (qui convertit un $\lambda$-terme dans une chaîne de caractère lisible par un être humain).

```
func print_lterme(t lterme) string
```

3. On ne demande pas d'écrire *lexer* ou *parser*. Les exemples seront construits directement avec des *constructeurs* (ou directement avec les constructeurs de types, dans le cas des langages fonctionnels). Ecrire (si nécessaire) des constructeurs pour les variables, les abstractions et les applications.

```
func cvar(v string) lterme
func clam(v string, c lterme) lterme
func capp(f lterme, a lterme) lterme
```

## 2.2 Sémantique (Facultatif)

Donner une sémantique au terme n'est pas nécessaire pour écrire un typeur. On pourra sauter les sections correspondantes.

1. Ecrire une fonction qui alpha-convertit un terme en le transformant en un terme qui respecte une convention de Barendregt (les variables liées sont distinctes deux à deux et distinctes des variables libres). Cette fonction parcourt récursivement le terme, et remplace toutes les variables liées par des $\lambda$ par des variables fraiches.

```
func fresh_var() string
func barendregt_rec(l lterme, remp map[string]string) lterme
func barendregt(l lterme) lterme
```

2. Ecrire une fonction qui substitue une variable par un terme dans un autre terme.

```
func instantie(l lterme, x string, a lterme) lterme
```

3. Ecrire une fonction qui réalise une étape d'évaluation selon la stratégie *Left-to-Right Call-by-Value*, c'est à dire que dans une application $M\ N$, on évalue d'abord la fonction $M$, puis l'argument $N$, et ensuite si la fonction est une abstraction $\lambda x.M'$, on $\beta$-réduit le redex correspondant pour obtenir $M'[N/x]$. On ne réduit pas sous les lambdas. (Cette stratégie préserve les divergences qui ne sont pas sous des $\lambda$)

```
func ltrcbv_etape(l lterme) lterme
```

4. Ecrire une fonction qui imprime sur la sortie standard les réduits d'un terme (on pensera à ajouter un *timeout* avant de la tester sur $\Omega$).

## 2.3 Types

La syntaxe des types simples est donnée par:

$$T ::= v \ \mid \ T \to T$$

1. Donner une syntaxe pour les types.

```
type lterme struct {
        est string
        // V -> variable
        // A -> type fleche
        tvari string // variable de type
        targ *stype // type de l'argument
        tres *stype // type du resultat
}
```

2. Ecrire un *prettyprinter* pour les types, et des constructeurs de types pour la variable et la flèche.
3. Ecrire une fonction qui décide quand deux types sont égaux.

```
func stype_egal(t1 stype, t2 stype) bool
```

### 2.4 Génération d'équations

1. Donner un type pour les équations de typage, puis écrire une fonction qui génère des équations de typage à partir d'un terme. Cette fonction parcourt récursivement un terme, munie d'un environnement et d'un type cible $T$, elle procède ainsi:
   - Si le terme est une variable, elle trouve son type $Tv$ dans l'environnement et génère l'équation $Tv = T$.
   - Si le terme est une abstraction, elle prend deux variables de type fraiches $T_a$ et $T_r$, génère l'équation $T = T_a \to T_r$ puis génère récursivement les équations du corps de la fonction avec comme cible le type $T_r$ et en rajoutant dans l'environnement que la variable liée par l'abstraction est de type $T_a$.
   - Si le terme est une application, elle prend une variable de type fraîche $T_a$, puis génère récursivement les équations du terme en position de fonction, avec le type cible $T_a \to T$, et les équations du terme en position d'argument avec le type cible $T_a$, en gardant le même environnement dans les deux cas.

   ```
   func gen_equas(envi map[string]stype, l lterme, t stype) []tequa
   ```

### 2.5 Unification

1. Ecrire une fonction d'*occur check* qui vérifie si une variable appartient à un type.

   ```
   func occur_check(v string, t stype) bool
   ```

2. Ecrire une fonction qui substitue une variable de type par un type à l'intérieur d'un autre type, puis une fonction qui substitue une variable de type par un type partout dans un système d'équation.

   ```
   func substitue(v string, ts stype, t stype) stype
   func substitue_partout(v string, ts stype, eqs []tequa) []tequa
   ```

3. Ecrire une fonction pour réaliser une étape d'unification dans les systèmes d'équations de typage selon un algo d'unification "simple", par exemple:
   - Si les deux types de l'équation sont égaux, on supprime l'équation
   - Si un des deux types est une variable $X$, qu'elle n'apparait pas dans l'autre type $T_d$, on supprime l'équation $X = T_d$ et on remplace $X$ par $T_d$ dans toutes les autres équations.
   - Si les deux types sont des types flèche $T_{ga} \to T_{gr} = T_{da} \to T_{dr}$, on supprime l'équation et on ajoute les équations $T_{ga} = T_{da}$ et $T_{gr} = T_{dr}$
   - Sinon on échoue.

   ```
   func unification_etape(eqs []tequa, i int) unif_res
   ```

4. Ecrire une fonction qui résout un système d'équation (avec un *timeout*) puis une fonction qui infère le type (ou la non-typabilité) d'un terme.

   ```
   func typeur(l lterme) typage_res
   ```

## 3 (un genre de) PCF

### 3.1 Syntaxe

On complexifie un peu le $\lambda$-calcul en ajoutant à la syntaxe:
- des entiers natifs (pas les entiers de Church) avec les opérateurs addition et soustraction.
- des listes natives (i.e. des séquences ordonnées d'éléments) avec les opérateurs tête, queue et cons.
- deux opérateurs *if zero then else* et *if empty then else* permettant de faire des branchements et testant respectivement les entiers et les listes.
- un opérateur de point fixe, permettant de définir récursivement des fonctions.
- un `let x = e1 in e2` natif.

1. Mettre à jour la syntaxe des termes, le *prettyprinter* et les constructeurs pour inclure les nouvelles fonctionnalités.

## 3.2 Sémantique (Facultatif)

1. Mettre à jour la sémantique:
   - pour évaluer `let x = e1 in e2` on évalue d'abord `e1` en `v`, puis on remplace `x` par `v` dans `e2` et on évalue `e2`
   - pour évaluer `fix (phi -> M)`, on remplace `phi` par `fix (phi -> M)` partout dans `M`.
   - on ne peut pas traiter `izte` et `iete` (les branchements) comme des opérateurs "normaux" (contrairement aux autres), parce qu'on veut bloquer l'exécution du conséquent et de l'alternant tant que le branchement n'est pas réduit (ce qui n'est pas possible en *ltr-CbV* si ce sont des opérateurs "normaux"), il faut donc les représenter par des constructeur de termes.

## 3.3 Types

On ajoute aux types :
- un type $\mathbf{N}$ pour les entiers,
- un constructeur de type $[T]$ pour les listes.
- un constructeur de type $\forall X.T$ pour gérer le let-polymorphisme.

1. Mettre à jour la syntaxe des types, le *prettyprinter* et les constructeurs.

## 3.4 Génération d'équations

- pour traiter un opérateur (par exemple $+$, ou $hd$) dans la génération d'équation pour le type cible $T$, on égalise $T$ et le type de l'opérateur (respectivement $T = \mathbf{N} \to \mathbf{N} \to \mathbf{N}$, ou $T = \forall X.[X] \to X$)
- pour traiter un entier dans la génération d'équation pour le type cible $T$, on génère $T = \mathbf{N}$
- pour traiter un branchement dans la génération d'équation pour le type cible $T$, on génère les équations de la condition (avec pour type cible, le type $\mathbf{N}$ ou $\forall X.[X]$ en fonction du type de branchement), les équations du conséquent avec la cible $T$ et les équations de l'alternant avec la cible $T$.
- pour traiter un `let x = e1 in e2` dans la génération d'équation pour le type cible $T$, on type (il faut ici utiliser la fonction de typage, ce qui induit une récusion mutuelle entre la génération d'équation, l'unification et le typage) `e1`, on récupère son type `T0` (si le typage n'échoue pas) et on génère les équations pour `e2` en ajoutant à l'environnement que `x` a le type $\forall \texttt{X1}, ..., \texttt{X}_\texttt{k}.\texttt{T0}$ (généralisation du type `T0`).

1. Ecrire une fonction qui généralise un type à partir d'un environnement $e$, c'est-à-dire qui ajoute un $\forall X.$ autour du type pour chacune de ses variables libres qui n'est pas dans l'environnement $e$.

   ```
   func generalise(envi map[string]stype, t stype) stype
   ```

2. Mettre à jour la génération d'équations.

## 3.5 Unification

Les modifications à apporter à l'unifications sont les suivantes:
- quand un des deux type d'une équation est un $\forall$, on lui applique une "barendregtisation" (on renomme ses variables de type liées) et on "l'ouvre" (on garde la même équation, mais sans le $\forall$)
- quand les deux termes sont des listes, on égalise le type des éléments.
- si les constructeurs de types (à l'exception du $\forall$) ne sont pas les mêmes à gauche et à droite d'une équation, on échoue.

1. Mettre à jour la génération d'équations.

# 4 Traits impératifs

## 4.1 Syntaxe

On ajoute au langage des traits impératifs, concrétisés par l'apparition d'une valeur *unit* `()`, de *régions* (des cases mémoires) dans la syntaxe et par trois nouveaux opérateurs:

- `!e` pour le déréférencement,
- `ref e` pour la création de région,
- `e1 := e2` pour l'assignement.

1. Mettre à jour la syntaxe, le *pretty printer* et les constructeurs.

## 4.2 Sémantique (Facultatif)

La sémantique est donnée par une relation de réduction pour les couples (terme, état), un état étant une table d'association entre des régions et des termes.

- pour réduire `!e` on réduit d'abord `e`. Si `e` est une région `rho`, le terme `!rho` se réduit en le terme associé à la région `rho` dans l'état.
- pour réduire `ref e` on réduit d'abord `e`. Si `e` est une valeur `v`, on crée une nouvelle région `rho` dans l'état, à laquelle on associe `v`, et `ref v` se réduit en `rho`.
- pour réduire `e1 := e2` on réduit d'abord `e1`, puis `e2`, puis si `e1` est une région `rho`, alors on associe la valeur de `e2` à `rho` dans l'état, et `e1 := e2` se réduit en `()`.

1. Mettre à jour la sémantique.

## 4.3 Types

On ajoute aux types:

- le type `unit`,
- le constructeur de type `Ref T`, qui représente une région dans laquelle on stocke des valeurs de type `T`,

1. Mettre à jour les types.

## 4.4 Génération d'équations

La génération d'équations n'est modifiée que pour le traitement des nouveaux constructeurs (par exemple `assign` est de type `∀X.Ref X -> X -> unit`)

1. Mettre à jour la génération d'équations.

## 4.5 Unification

1. Mettre à jour l'unification pour traiter les nouveaux constructeurs de types.

## 4.6 Polymorphisme faible

Sans surprise, le typeur accepte `let l = ref [] in let _ = l := [(^x.x)] in (hd !l) + 2` alors que le terme ne peut pas se réduire. Pour pouvoir rejeter un tel terme, il faut ajouter la notion de polymorphisme faible, comme vue en cours.

1. Introduire la notion de non-expansivité pour les termes, de polymorphismes faibles pour les types, et modifier le typeur en conséquence pour proposer un système de type correct pour le langage

# 5 Polymorphisme de rangée

Ajouter au langage des *records* et au système de type une notion de *sous-typage de rangée*: le type d'un record est l'ensemble de ses champs, et un record peut être utilisé à la place d'un autre quand il contient au moins tous ses champs.