# TP2 - Rapport

Philippe Gabriel
Yan Zhuang

21 juin 2021

# 1 Premier aperçu

Comme première étape de travail, nous avions tenter d'apparier les données de ce travail avec celui du premier travail pratique. Nous avions donc commencer par supposer que les relations `infer` et `check` devrait ressembler à aux fonctions du même nom dans le premier travail, et que les relations `expand` et `coerce` ressemblait en quelque sorte à `s2l`. Nous avions vite remarqué que de telles suppositions étaient fausses à prendre, surtout dans la méthode d'implémentation car on remarque dans le travail sur le langage `psil` que l'on peut diviser le travail en quatre phases distinctes qui est même reflété par l'implémentation. Cependant, pour le langage $\mu$`pts`, les relations communiquent toutes entre elles en quelque sorte. Cela a rendu la tâche assez difficile au début. La technique que l'on a employé par après pour ce travail est une sorte de "Test-driven development" (tdd). On roulait les tests à l'aide de `test_samples` en appliquant l'outil de déboguage `trace` qu'offre l'environnement de `prolog`. Cela nous permet de situer l'emplacement des échecs d'exécution et nous permet par la suite de corriger ceux-ci.

# 2 Initialiser l'environnement

La première erreur qui apparaissait après le démarrage des tests était l'échec de l'inférence du type `type`. Nous regardons donc la figure de la donnée indiquant les règles de typage bidirectionnelles du langage surface et on s'apeçoit que la règle suivante semble s'appliquer à notre situation:

$$\frac{\Gamma(x) = e}{\Gamma \vdash x \Rightarrow e}$$

Nous avions donc rajouté une règle d'inférence supplémentaire comme-ci:

```
% Regle 1
infer(Env, X, X, T) :-
    member(X : T, Env).
```

Cela a donc permis l'élaboration de l'environnement des types `int`, `float` et `bool`. Nous avions continué cette approche pour les autres types en ajoutant, au fur et à mesure, les règles nécessaires pour le passage de ces tests. Pour le type du `if`, il nous a été nécessaire d'ajouter un cas supplémentaire dans nos règles d'expansion comme suit:

```
expand(forall(X, B), forall(X, _, B)).
```

Cela nous a permis de faire passer le test pour `if`. Un problème est ensuite survenu pour `nil`, où on avait affair pour la première fois à une construction de `functor`. Il nous fallait la décomposer en langage interne en ajoutant un cas supplémentaire à `expand`. Il nous a été nécessaire d'écrire plusieurs versions de ce cas après plusieurs tests individuelles pour finalement arriver à cette forme:

```
expand(Fa, app(Fb, AL)) :-
    Fa =.. [N | AS],
    \+ member(N, [fun, app, arw, forall, (->), (:), let, [], (.)]),
    length(AS, L),
    L \= 0,
    functor(Fa, N, L),
    last(AS, AL),
    append(AI, [AL], AS),
    Fb =.. [N | AI].
```

Après cela, le cas particulier du type de `cons`, qui prend une liste de paramètres nous a donc inciter à modifier notre définition d'expansion d'un `forall` de sorte à l'élaborer en des `forall` currifiés:

```
expand(forall(X, B), F) :-
    (X = [T | TS], TS \= [] ->
           F = forall(T, _, forall(TS, B));
           (X = [A] ->
               F = forall(A, _, B);
               F = forall(X, _, B))).
```

Suite à cette modification et à d'autres ajouts de règles de typages, l'initialisation de l'environnement était complète.

# 3 Test d'expressions