# M2 MPRI - Programmation Probabiliste - BYO-RPPL

Guillaume Baudart

14 janvier 2022

**Objectif.** Le but de cet exercice est d'ajouter les construction probabilistes `sample`, `factor` et `infer` à un langage synchrone : Zelus. Le compilateur Zelus génère du code OCaml, on peut donc implémenter notre extension probabiliste comme une bibliothèque OCaml.

Le dépôt git `https:github.com/gbdrt/byo-ppl` contient un repertoire `rppl` avec le code à compléter :

- `rppl.ml` : l'implémentation de la bibliothèque à compléter.
- `rppl.zli` : le fichier d'interface pour le compilateur Zelus
- `coin.zls, hmm.zls` : deux exemples à compléter
- `dune` : le fichier dune pour construire le projet.

## 1 Installation

Pour commencer, installer le projet `byo-ppl` et le compilateur Zelus.

```
$ git clone https:github.com/gbdrt/byo-ppl
$ cd byo-ppl
$ opam install . --deps-only
$ opam install zelus
```

Vous pouvez tester votre installation sur un exemple :

```
$ dune exec ./examples/coin.exe
```

## 2 Échantillonnage Préférentiel

L'implémentation de l'*échantillonnage préférentiel* est similaire au module `basic` de `byo-ppl`. Les opérateurs `sample`, `factor` et `observe` prennent un argument supplémentaire : l'accumulateur probabiliste `prob` dont le type est à définir.

**Question 1.** Compléter l'implémentation de `prob`, `sample`, `factor` et `observe`.

*Note.* Pour garantir l'allocation statique, Zelus restreint l'utilisation de l'ordre supérieur. Tous les arguments sont donc groupés dans un tuple.

Les nœuds Zelus sont introduits par le mot-clé `node` dans le code source. Contrairement aux fonctions instantanées classiques, les nœuds sont compilés vers la structure de données suivante.

```
type ('a, 'b) cnode = (* type 'a -D-> 'b *)
    Cnode:
      { alloc : unit -> 's;        (* allocate the state *)
        copy : 's -> 's -> unit;   (* copy the source into the destination *)
        step : 's -> 'a -> 'b;     (* compute a step *)
        reset : 's -> unit;        (* reset/inialize the state *)
      } -> ('a, 'b) cnode
```

La fonction `alloc` permet d'allouer l'état d'un nœud. La fonction `reset` initialise l'état initial (par exemple après un appel à `alloc`). La fonction `step` est la fonction de transition du nœud : à chaque instant, étant donné l'état courant et les entrées, `step` met à jour l'état (par effet de bord) et retourne une sortie.

Le nœud `infer_importance` est un nœud d'ordre supérieur qui prend un argument un nœud : le modèle probabiliste dont le type est donné dans le fichier de signatures `rppl.zli` :

```
val infer_importance : int -S-> ((prob * 'a) -D-> 'b) -S-> 'a -D-> 'b distribution
```

Le premier argument est le nombre de particule. Le second argument est un nœud probabiliste : le modèle dont le premier argument doit être de type `prob`. Les lettres sur les flèches precisent les *sortes* : `S` pour statique (le nombre de particule et le modèle), `D` pour discret (le nœud modèle, et le nœud d'ordre supérieur `infer_importance`).

L'état de `infer_importance` contient un tableau de particules (l'état de chaque particule) et un tableau de scores.

```
type 'a infer_state = { particles : 'a array; scores : float array }
```

**Question 2.** Compléter l'implémentation des fonctions `infer_alloc` et `infer_reset` du nœud `infer_importance`. On peut utiliser la fonction d'allocation du modèle pour initialiser le tableau de particules.

La fonction de transition de `infer_importance` est similaire à l'implémentation du module `basic`. Il faut appliquer la fonction de transition du modèle sur chacun des états contenu dans le tableau de particules et l'entrée courante pour mettre à jours ces états et les scores associés.

**Question 3.** Compléter la fonction `infer_step` du nœud `infer_importance`.

# 3 Exemples

Il est maintenant possible d'essayer `infer_importance` sur des exemples simples.

**Question 4.** Compléter le fichier `coin.zls` pour implémenter un modèle flot-de-données de la pièce biaisée. L'entrée $x$ est un flots d'entiers (0 ou 1). Le biais de la pièce $z$ doit être constant et partagé par toutes les observations.

$$z \sim \mathit{Uniform}(0,1)$$
$$x_t \sim \mathit{Bernoulli}(z)$$

La sortie de `infer_importance` est un flot de distributions $d$ : l'estimation courante du biais sachant les premières observations $d_t = p(z \mid x_0, \ldots, x_t)$.

*Note.* Zelus ne supporte pas les arguments nommés (comme dans `gaussian ~mu ~sigma`). On a donc redéfinit dans `rppl.ml` quelques distributions avec des arguments classiques.

Pour compiler le programme `coin.zls` on précise à Zelus le nœud qu'on veut simuler (typiquement `main` : option `-s main` dans le fichier `dune`). Le compilateur génère deux fichiers : `coin.ml` (code compilé) et `coin_main.ml` (code de simulation). Il est ensuite possible de compiler `coin_main.ml` pour obtenir un executable.

Avec dune, on peut directement compiler et exécuter la simulation avec :

```
$ dune exec ./rppl/hmm_main.exe
```

**Question 5.** Compléter le fichier `hmm.zls` pour implémenter un modèle flot-de-données de tracker. À l'instant $t$, $x_t$ est la position cherchée, et $y_t$ l'observation bruitée, et on suppose $\forall t > 0$ :

$$\begin{aligned} x_{t+1} &\sim \mathcal{N}(x_t, 2) \\ y_t &\sim \mathcal{N}(x_t, 4) \end{aligned}$$

*Note.* `infer_importance` doit échouer sur cet exemple (comme pour l'implémentation en CPS).

# 4 Filtre Particulaire

Le *filtre particulaire* se comporte quasiment comme `infer_importance` mais, à chaque pas, avant de renvoyer la distribution courante, il faut ré-échantillonner le tableau de particules.

**Question 6.** Compléter la fonction `resample` qui ré-échantillonne l'état de `infer_pf`. Étant donné un état `{particles; scores}`, `resample` tire aléatoirement un nouveau tableau de particules en suivant les `scores`. Vous pourrez utiliser la fonction `alloc` pour initialiser le nouveau tableau de particules, et `copy src dst` pour copier une particule source `src` dans une destination `dst` (c'est une *deep copy*, il faut copier l'ensemble de la structure de donnée).

**Question 7.** Compléter la définition de `infer_pf` pour implémenter le filtre particulaire. L'exemple `hmm` doit maintenant renvoyer le résultat attendu.

**Visualisation.** Pour visualiser graphiquement les résultat obtenus, vous pouvez installer feedgnuplot. Il est alors possible de rediriger la sortie standard vers gnuplot en rafraîchissant l'affichage toutes les 0.05s.

```
$ dune exec ./rppl/hmm_main.exe | feedgnuplot --stream 0.05
```

*Note.* Par défaut la simulation ne s'arrête jamais. Il est possible de rediriger la sortie de l'exécutable dans `head -n 100` pour n'executer que les 100 premiers instants.

```
$ dune exec ./rppl/hmm_main.exe | head -n 100 | feedgnuplot --stream 0.05
```