Inférence parallèle pour un langage réactif probabiliste

Guillaume Baudart, Louis Mandel, Marc Pouzet, Reyyan Tekin

Inférence parallèle pour un langage réactif probabiliste

Guillaume Baudart,[1,2] Louis Mandel,[3]
Marc Pouzet,[2,1] Reyyan Tekin[1,2]

[1] Inria Paris
[2] École normale supérieure – PSL university
[3] IBM Research

Résumé

ProbZelus est un langage synchrone probabiliste qui permet de décrire des modèles probabilistes réactifs en interaction avec un environnement observable. Des méthodes d'inférences réactives permettent d'apprendre en ligne les distributions associées aux paramètres non-observés du modèle à partir d'observations statistiques. Ce problème n'a, en général, pas de solution analytique simple. Pour obtenir des estimations précises, une méthode classique consiste à analyser les résultats obtenus par de nombreuses exécutions indépendantes du modèle. Ces méthodes sont coûteuses, mais ont l'avantage de pouvoir être massivement parallélisées.

Nous proposons d'utiliser JAX pour paralléliser les méthodes d'inférence réactive de ProbZelus. JAX est une bibliothèque récente qui permet de paralléliser automatiquement du code Python qui peut ensuite être exécuté de manière efficace et transparente sur CPU, GPU, ou TPU.

Dans cet article, nous décrivons un nouveau moteur d'inférence réactive parallèle implémenté en JAX et le nouveau *backend* JAX associé pour ProbZelus. Nous montrons sur des exemples existants que notre nouvelle implémentation surpasse l'implémentation séquentielle originale pour un nombre de particules élevé.

1 Introduction

Les langages synchrones [4] ont été introduits pour concevoir et implémenter des systèmes embarqués temps réel. Ils permettent de décrire une spécification exécutable précise qui est utilisée comme référence pour la simulation, le test et la vérification formelle. Le compilateur génère ensuite le code embarqué qui est *correct par construction*, c'est à dire qu'il préserve la sémantique de la spécification initiale. Le langage synchrone Scade [13] est ainsi utilisé pour le logiciel certifié d'avions et de trains, par exemple.

La plupart des systèmes embarqués évoluent dans un environnement ouvert et incertain qu'il ne perçoivent qu'à travers des capteurs imparfaits et bruités (accéléromètres, caméras ou GPS). Les interactions avec des agents autonomes (animaux, humains, robots) ajoutent encore à cette incertitude. Les langages synchrones classiques ne permettent pas de manipuler ces incertitudes.

La programmation probabiliste est un paradigme de programmation qui a connu un essor important ces dernières années. Les langages de programmation probabilistes permettent de décrire des modèles qui manipulent l'incertitude de manière explicite et proposent des méthodes automatiques pour inférer les paramètres du modèle à partir d'observations statistiques. Ces langages reposent sur la méthode Bayésienne qui permet de raffiner une croyance a priori sur la distribution des paramètres d'un modèle à partir d'observations concrètes [6, 18, 20, 24, 25]. Dans la lignée de ces travaux, nous avons récemment proposé ProbZelus, un langage synchrone probabiliste [2]. ProbZelus combine les constructions d'un langage réactif synchrone (temps logique synchrone et automates hiérarchiques) et les constructions probabilistes (`sample`, `observe` et `infer`) pour concevoir des applications réactives probabilistes [1].

L'exécution d'un programme probabiliste nécessite la résolution d'un problème d'inférence. Ce problème n'a en général, pas de solution analytique simple. Pour obtenir des estimations précises, une méthode classique consiste à analyser les résultats obtenus par de nombreuses exécutions indépendantes du modèle, appelées *particules* [15]. Ces méthodes sont coûteuses mais ont l'avantage de pouvoir être massivement parallélisées.

Le moteur d'inférence de ProbZelus est implémenté en OCaml qui se prête mal à la parallélisation massive. Dans cet article nous proposons d'utiliser JAX pour paralléliser le moteur d'inférence de ProbZelus. JAX est une bibliothèque récente qui permet de compiler du code purement fonctionnel qui peut ensuite être exécuté de manière transparente sur CPU, GPU, ou TPU de manière massivement parallèle [10]. JAX est associé à une impressionnante bibliothèque de calcul numérique qui permet de ré-implementer efficacement le moteur d'inférence. D'autres fonctionnalités, telle que l'auto-différenciation, pourront également être utilisée dans le futur pour implémenter des méthodes d'inférence plus avancées.

Dans cet article, nous montrons comment exécuter des modèles probabilistes réactifs de manière massivement parallèle grâce à JAX. Nous présentons, en particulier, les contributions suivantes : Section 3 un nouveau moteur d'inférence parallèle efficace implémenté en JAX, Section 4 un compilateur de ProbZelus vers JAX , Section 5 une évaluation sur un ensemble de modèles réactifs existants. Le code est disponible à l'adresse suivante : https://github.com/rpl-lab/jfla22-zlax.

2 Contexte

Dans cette section nous rappelons les éléments fondamentaux de la programmation synchrone et de la programmation probabiliste à travers un exemple simple (voir [1] pour une introduction plus complète).

2.1 Programmation réactive probabiliste

Programmation synchrone. ProbZelus est une extension du langage Zelus [1] [9] avec des constructions probabilistes. Zelus est lui-même un descendent de Lustre [19] dont il reprend les principes et le style de programmation *flot de données*. Un programme Zelus est défini par un ensemble de fonctions de suites, appelées *nœuds*. Les entrées et les sorties d'un nœuds sont des suites infinies, appelées des *flots*. Toutes ces suites progressent au même rythme, de manière *synchrone*. Ce type de programmation permet de décrire naturellement les schémas-blocs de l'automatique, la notation classique utilisée dans la conception du logiciel réactif embarqué [21].

Par exemple, le nœud suivant lève une alarme dès que l'entrée Booléenne devient vraie.

```
node watch x = alarm where
  rec automaton
  | Wait → do alarm = false unless x then Ring
  | Ring → do alarm = true done
```

Le nœud `watch` calcule le flot de sortie `alarm` à partir du flot d'entrée `x`. Le comportements de `watch` est défini par un automate à deux états. Dans l'état initial `Wait` la valeur de `alarm` est `false` à tous les instant. Dès que `x` devient `true`, la transition `unless x` est activée, et l'automate passe dans l'état `Ring`. La valeur de `alarm` devient alors `true` à tous les instants. La Figure 1 montre un exemple de trace d'exécution du nœud `watch`.

1. https://zelus.di.ens.fr

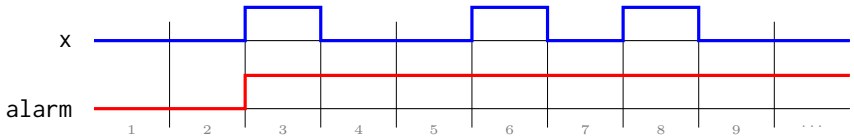


FIGURE 1 – Exemple de chronogramme pour le nœud `edge`.

Programmation probabiliste. L'inférence Bayésienne permet de calculer la distribution d'un paramètre θ sachant une série d'observations $\mathbf{x}$ (distribution *a posteriori* $p(\theta \mid \mathbf{x})$) à partir d'une croyance initiale (distribution *a priori* $p(\theta)$).

$$p(\theta \mid \mathbf{x}) = \frac{p(\theta)\, p(\mathbf{x} \mid \theta)}{p(\mathbf{x})} \qquad \text{(Bayes, 1763)}$$

Les langages de programmation probabilistes introduisent donc trois constructions pour décrire les modèles : `theta = sample(d)` introduit une variable aléatoire `theta` de distribution a priori `d`, $p(\theta)$; `observe(d, x)` mesure la vraisemblance de l'observation `x` par rapport à une distribution `d` (i.e., la valeur de la densité de `d` en `x`), $p(x \mid \theta)$; `infer m x` calcule la distribution a posteriori des valeurs de sortie d'un modèle `m` étant donné les observations `x`, $p(\theta \mid x)$.

Un exemple introductif classique consiste à déterminer le biais d'une pièce à partir d'observations indépendantes. À chaque instant $n \in \mathbb{N}$, on observe le résultat d'un lancé : $x_n = \mathit{True}$ (pile) ou $x_n = \mathit{False}$ (face). On suppose que chacun de ces lancés suit une loi de Bernoulli de paramètre θ : $p(x_n \mid \theta) = \theta$ et $p(\overline{x_n} \mid \theta) = 1 - \theta$. On cherche à estimer le biais θ à partir des observations $(x_n)_{n\in\mathbb{N}}$, i.e., $p(\theta \mid x_0, x_1, x_2, \dots)$.

Le programme suivant implémente ce modèle en ProbZelus.

```
proba coin x = theta where
  rec init theta = sample (uniform_float (0., 1.))
  and () = observe (bernoulli theta, x)
```

Initialement, on ne sait rien sur la pièce, tous les biais sont équivalents. La distribution a priori est donc uniforme sur $[0, 1]$ ($\theta = 0.5$ correspond à une pièce parfaitement équilibrée, $\theta = 0$ est une pièce qui tombe toujours sur face). On suppose que le paramètre θ est constant (mot-clé `init`). À chaque instant on utilise la construction `observe` pour faire l'hypothèse que l'observation `x` suit une loi de Bernoulli de paramètre θ.

2.2 Inférence dans la boucle

La construction `infer` est un nœud d'ordre supérieur qui prend ici en argument le nœud probabiliste `coin` et un flot d'entrées. Il renvoie à chaque instant, la distribution de sortie du modèle. L'inférence est un processus synchrone qui ne s'arrête jamais et qui peut être exécutée *dans la boucle*, i.e., en parallèle, avec d'autres nœuds synchrones déterministes classiques.

Par exemple, le programme suivant combine le modèle `coin` et le nœud `watch` pour signaler une pièce suspecte.

```
node cheater_detector x = cheater where
  rec theta_dist = infer coin x
  and m, s = stats_float theta_dist
  and cheater = watch ((m < 0.2 || 0.8 < m) && (s < 0.01))
```

Le flot `theta_dist` correspond à la distribution estimée du paramètre `theta` sachant les premières observations $x_0, x_1, \dots, x_n$. À chaque instant, on calcule la moyenne et l'écart type de

cette distribution. Le nœud `watch` définit ci-dessus surveille ces valeurs et lève l'alarme `cheater` dès que la condition `(m < 0.2 || m > 0.8) && (s < 0.01)` est vérifiée.

2.3 ProbZelus

La syntaxe, le typage, et la sémantique formelle d'un noyau de ProbZelus sont formellement définit dans [2]. Nous en rappelons ici l'essentiel.

Syntaxe. La syntaxe du noyau de ProbZelus est la suivante. Les constructions manquantes comme les automates hiérarchiques peuvent être compilé vers ce noyau par une série de transformations source à source.

$$
\begin{array}{rcl}
d & ::= & \texttt{let node } f\ x\ \texttt{=}\ e \mid \texttt{let proba } f\ x\ \texttt{=}\ e \mid d\ d \\
e & ::= & c \mid x \mid \texttt{(}e\texttt{,}e\texttt{)} \mid op\texttt{(}e\texttt{)} \mid f\texttt{(}e\texttt{)} \mid \texttt{last } x \mid e\ \texttt{where rec } E \\
 & & \mid \texttt{present } e\ \texttt{->}\ e\ \texttt{else } e \mid \texttt{reset } e\ \texttt{every } e \\
 & & \mid \texttt{sample(}e\texttt{)} \mid \texttt{observe(}e\texttt{,}e\texttt{)} \mid \texttt{infer(}f\texttt{(}e\texttt{))} \\
E & ::= & x\ \texttt{=}\ e \mid \texttt{init } x\ \texttt{=}\ c \mid E\ \texttt{and}\ E
\end{array}
$$

Un programme est une suite de déclaration de fonctions de flots déterministes (`node`) et probabilistes (`proba`). Le corps d'une fonction de flot est défini par une expression. Un expression de base peut être une constante (c), une variable (x), une paire ((e,e)), l'application d'un opérateur primitif (opérateur arithmétique, distribution, etc.), un appel de fonction de flot, un délai (`last` x) qui renvoie la valeur de x à l'instant précédent, ou une définition locale (e `where rec` E). Une équation x `=` e définit la variable x à partir de l'expression e et une équation `init` x `=` e définit la valeur initiale de x. Une expression peut également être une structure de contrôle : `present` x `->` e_1 `else` e_2 active l'expression e_1 ou e_2 selon la valeur de x, `reset` e_1 `every` e_2 réinitialise la valeur des expressions `last` x avec la valeur des équations `init` x `=` e correspondantes. Enfin, le noyau déterministe du langage est étendu avec les expressions probabilistes `sample(`e`)`, `observe(`e_1`,`e_2`)`, et `infer(`f`(`e`))`.

Ordonnancement. En ProbZelus, dans l'expression e `where rec` E, E est un ensemble d'équations mutuellement récursives. Au moment de la compilation, les équations sont simplifiées et ordonnées selon leurs dépendances. Les initialisations `init` x_j `=` c_j sont regroupées au début et l'équation $x_j = e_j$ doit apparaître après l'équation $x_i = e_i$ si l'expression e_j utilise x_i en dehors d'un opérateur `last`. Le compilateur peut introduire des équations supplémentaires pour relâcher les contraintes d'ordonnancement. Les programmes qui ne peuvent pas être ordonnancés statiquement sont rejetés (cette décision est prise par le compilateur de Zelus à l'aide d'un système de types dédiés qui construit une signature exprimant les dépendances entre entrées/sorties d'un noeud [3]). Dans la suite, on fera donc l'hypothèse que le noyau ProbZelus est ordonnancé. L'expression e `where rec` E prend donc la forme suivante :

e `where rec init` x_1 `=` c_1 `...` `and init` x_k `=` c_k
`and` y_1 `=` e_1 `...` `and` y_n `=` e_n

Pour simplifier, on suppose aussi que toutes les variables introduites par `init` sont également définies par une équation, i.e., $\{x_i\}_{1..k} \cap \{y_j\}_{1..n} = \{x_i\}_{1..k}$. Il est toujours possible d'ajouter des équations x_i `=` `last` x_i si ce n'est pas le cas.

Sémantique. La sémantique idéale de ProbZelus est définie de manière co-itérative [11]. Les expressions déterministes sont caractérisées par un état initial de type S et une fonction de

transition de type $S \to T \times S$ qui prend en argument l'état courant, renvoie une sortie et l'état suivant. Le flot correspondant est obtenu en itérant la fonction de transition à partir de l'état initial. Pour un environnement γ qui associe les noms de variables à leur valeur et une expression déterministe e, on note $[\![e]\!]^{\text{init}}_\gamma$ l'état initial et $[\![e]\!]^{\text{step}}_\gamma$ la fonction de transition. Par exemple, l'accès à une variable et la construction `present` sont définis de la manière suivante :

$$
\begin{array}{lcl}
[\![x]\!]^i_\gamma & = & () \\
[\![x]\!]^s_\gamma & = & \lambda s.\ (\gamma(x), s) \\
[\![\texttt{present}\ e\ \texttt{->}\ e_1\ \texttt{else}\ e_2]\!]^{\text{init}}_\gamma & = & ([\![e]\!]^{\text{init}}_\gamma, [\![e_1]\!]^{\text{init}}_\gamma, [\![e_2]\!]^{\text{init}}_\gamma) \\
[\![\texttt{present}\ e\ \texttt{->}\ e_1\ \texttt{else}\ e_2]\!]^{\text{step}}_\gamma & = & \\
\quad \lambda(s, s_1, s_2).\ \mathit{let}\ v, s' = [\![e]\!]^{\text{step}}_\gamma(s)\ \mathit{in} & & \\
\quad\quad \mathit{if}\ v\ \mathit{then}\ \mathit{let}\ v_1, s'_1 = [\![e_1]\!]^{\text{step}}_\gamma(s_1)\ \mathit{in}\ (v_1, (s', s'_1, s_2)) & & \\
\quad\quad\quad \mathit{else}\ \ \mathit{let}\ v_2, s'_2 = [\![e_2]\!]^{\text{step}}_\gamma(s_2)\ \mathit{in}\ (v_2, (s', s_1, s'_2))
\end{array}
$$

La fonction de transition d'une variable renvoie à chaque instant la valeur stockée dans l'environnement. La construction `present` e `->` e_1 `else` e_2 renvoie la valeur de e_1 quand e est vrai, et la valeur de e_2 dans le cas contraire. L'état contient l'état des trois sous-expressions. La fonction de transition exécute paresseusement l'une ou l'autre branche en fonction de la valeur de e.

Les expressions probabilistes sont définies par un état initial et une fonction de transition de type $S \to \Sigma_{T\times S} \to [0, \infty)$ qui prend en argument l'état courant et renvoie une *mesure* qui associe un score positif à tous les ensembles mesurables de couples (sortie, état suivant). [2] Pour un environnement γ et une expression probabiliste e, on note $\{\!|e|\!\}^{\text{init}}_\gamma$ l'état initial, et $\{\!|e|\!\}^{\text{step}}_\gamma$ la fonction de transition.

L'opérateur `infer` permet d'obtenir une valeur déterministe (une distribution) à partir d'un modèle probabiliste f et d'un flot d'observations e. À chaque instant, cet opérateur calcule une distribution de résultats et une distribution d'états suivants possibles.

$$
\begin{array}{lcl}
[\![\texttt{infer(}f\texttt{(}e\texttt{))}]\!]^{\text{init}}_\gamma & = & \lambda U.\ \delta_{[\![f(e)]\!]^i_\gamma}(U) \\
[\![\texttt{infer(}f\texttt{(}e\texttt{))}]\!]^{\text{step}}_\gamma & = & \lambda\sigma.\ \mathit{let}\ \mu = \lambda U.\ \int_S \sigma(ds)\{\!|f(e)|\!\}^{\text{step}}_\gamma(s)(U)\ \mathit{in} \\
 & & \quad \mathit{let}\ \nu = \lambda U.\ \mu(U)/\mu(\top)\ \mathit{in} \\
 & & \quad (\pi_{1*}(\nu), \pi_{2*}(\nu))
\end{array}
$$

Pour un environnement γ, l'état initial de $[\![\texttt{infer(}f\texttt{(}e\texttt{))}]\!]^{\text{init}}_\gamma$ est la mesure de Dirac sur l'état initial de f`(`e`)`. La fonction de transition intègre la mesure définit par f`(`e`)`, notée $\{\!|f\texttt{(}e\texttt{)}|\!\}^{\text{step}}_\gamma$, sur tous les états possibles pour obtenir une mesure μ. Cette mesure est ensuite normalisée pour obtenir une distribution $\nu : T \times S$ `dist` ($\top$ représente l'espace entier). Cette distribution est ensuite séparée en une paire de distribution marginales sur les résultats et les états suivants avec les mesures images par les projections π_1 et π_2.

3 Moteur d'inférence

En général, estimer la distribution a posteriori d'un modèle n'admet pas de solution analytique simple. Les langages de programmation probabiliste reposent donc sur des méthodes d'inférence approximative. Les méthodes de Monte-Carlo consistent à accumuler les résultats d'un grand nombre de simulations indépendantes pour obtenir une estimation précise [15].

2. $\Sigma_{T\times S}$ est la σ-algèbre sur $T \times S$, c'est à dire l'ensemble des ensembles mesurables de $T \times S$.

$$
\begin{aligned}
&\{\!|c|\!\}^{\text{step}}_{\gamma} &&= \lambda s, w.\ (c, s, w)\\
&\{\!|x|\!\}^{\text{step}}_{\gamma} &&= \lambda s, w.\ (\gamma(x), s, w)\\
&\{\!|\texttt{sample(}e\texttt{)}|\!\}^{\text{step}}_{\gamma} &&= \lambda s, w.\ \mathit{let}\ d, s', w' = \{\!|e|\!\}^{\text{step}}_{\gamma}(s, w)\ \mathit{in}\ (\text{draw}(d), s', w')\\
&\{\!|\texttt{observe(}e_1\texttt{,}e_2\texttt{)}|\!\}^{\text{step}}_{\gamma} &&= \lambda s, w.\\
&&&\quad \mathit{let}\ \mu, s_1, w_1 = \{\!|e_1|\!\}^{\text{step}}_{\gamma}(s, w)\ \mathit{in}\\
&&&\quad \mathit{let}\ v, s_2, w_2 = \{\!|e_2|\!\}^{\text{step}}_{\gamma}(s_1, w_1)\ \mathit{in}\ ((), s_2, w_2 * \mu_{\text{pdf}}(v))\\
&\{\!|\texttt{present}\ e\ \texttt{->}\ e_1\ \texttt{else}\ e_2|\!\}^{\text{step}}_{\gamma} &&= \lambda(s, s_1, s_2), w.\\
&&&\quad \mathit{let}\ v, s', w' = \{\!|e|\!\}^{\text{step}}_{\gamma}(s, w)\ \mathit{in}\\
&&&\qquad \mathit{if}\ v\ \mathit{then}\ \mathit{let}\ v_1, s'_1, w_1 = \{\!|e_1|\!\}^{\text{step}}_{\gamma}(s_1, w')\ \mathit{in}\ (v_1, (s', s'_1, s_2), w_1)\\
&&&\qquad\quad \mathit{else}\ \ \mathit{let}\ v_2, s'_2, w_2 = \{\!|e_2|\!\}^{\text{step}}_{\gamma}(s_2, w')\ \mathit{in}\ (v_2, (s', s_1, s'_2), w_2)\\
&\{\!|\texttt{reset}\ e_1\ \texttt{every}\ e_2|\!\}^{\text{step}}_{\gamma} &&= \lambda(s_0, s_1, s_2), w.\\
&&&\quad \mathit{let}\ v_2, s'_2, w_2 = \{\!|e_2|\!\}^{\text{step}}_{\gamma}(s_2, w)\ \mathit{in}\\
&&&\quad \mathit{let}\ v_1, s'_1, w_1 = \{\!|e_1|\!\}^{\text{step}}_{\gamma}(\mathit{if}\ v_2\ \mathit{then}\ (s_0, w_2)\ \mathit{else}\ (s_1, w_2))\ \mathit{in}\\
&&&\quad (v_1, (s_0, s'_1, s'_2), w_1)
\end{aligned}
$$

$$
\left\{\!\!\left|\begin{array}{l} e\ \texttt{where}\\ \quad \texttt{rec init}\ x_1\ \texttt{=}\ c_1\ \ldots\\ \quad \texttt{and init}\ x_k\ \texttt{=}\ c_k\\ \quad \texttt{and}\ y_1\ \texttt{=}\ e_1\ \ldots\\ \quad \texttt{and}\ y_n\ \texttt{=}\ e_n \end{array}\right|\!\!\right\}^{\text{step}}_{\gamma} =
$$

$$
\begin{array}{l}
\lambda((m_1, \ldots, m_k), (s_1, \ldots, s_n), s), w.\\
\quad \mathit{let}\ \gamma_1 = \gamma[m_1/x_1\texttt{\_last}]\ \mathit{in}\ \ldots\\
\quad \mathit{let}\ \gamma_k = \gamma_{k-1}[m_k/x_k\texttt{\_last}]\ \mathit{in}\\
\quad \mathit{let}\ v_1, s'_1, w_1 = \{\!|e_1|\!\}^{\text{step}}_{\gamma_k}(s_1, w)\ \mathit{in}\ \mathit{let}\ \gamma'_1 = \gamma_k[v_1/y_1]\ \mathit{in}\ \ldots\\
\quad \mathit{let}\ v_n, s'_n, w_n = \{\!|e_n|\!\}^{\text{step}}_{\gamma'_{n-1}}(s_n, w_{n-1})\ \mathit{in}\ \mathit{let}\ \gamma'_n = \gamma'_{n-1}[v_n/y_n]\ \mathit{in}\\
\quad \mathit{let}\ v, s', w' = \{\!|e|\!\}^{\text{step}}_{\gamma'_n}(s)\ \mathit{in}\\
\quad (v, ((\gamma'_n[x_1], \ldots, \gamma'_n[x_k]), (s'_1, \ldots, s'_n), s'), w')
\end{array}
$$

FIGURE 2 – Sémantique opérationnelle des expressions probabilistes.

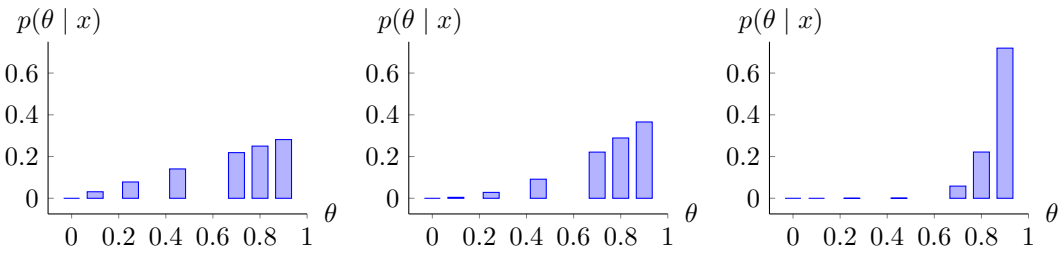
3.1 Échantillonneur

Pour ce type d'algorithme d'inférence, la sémantique opérationnelle d'un modèle probabiliste est un échantillonneur. Les états initiaux ne changent pas, mais la fonction de transition produit un échantillon aléatoire associé à un score qui mesure la qualité de l'échantillon.

La fonction de transition d'une expression probabiliste de type $S \times [0, \infty) \to T \times S \times [0, \infty)$ prend en argument un état et un score et renvoie l'état suivant, la sortie, et le nouveau score. Les fonctions de transition des expressions probabilistes de ProbZelus sont présentées en Figure 2. Les constantes, et l'accès à une variable (initialisée ou non) renvoient la valeur attendue et laissent l'état et le score inchangés. `sample(d)` tire une valeur aléatoire dans la distribution `d` sans modifier le score. `observe(d, x)` multiplie le score par la vraisemblance de l'observation `x` par rapport à la distribution `d` et renvoie une valeur de type `unit`. [3] Pour les autres constructions, l'état contient les états de toutes les sous-expressions. La fonction de transition se contente de calculer l'état suivant et de propager le score en suivant l'ordre des sous-expressions.

L'état d'un ensemble d'équations récursives (ordonnancé) e `where rec` E contient, en plus de l'état des sous-expressions, la valeur des variables locales à l'instant précédent $(m_1, \ldots, m_k)$. La fonction de transition commence par mettre à jour l'environnement γ avec un ensemble de variables fraîches x_i`_last` initialisées avec les valeurs m_i. Cet environnement est ensuite étendu par les définitions des variables y_i en exécutant toutes les sous-expressions tout en propageant le score. Enfin, l'expression e est exécutée dans l'environnement final. L'état suivant contient la nouvelle valeur des variables initialisées qui sera utilisée pour démarrer l'instant suivant.

3. On note D_{pdf} la densité de la distribution D.



(a) instant 1 (b) instant 2 (c) instant 10

FIGURE 3 – Échantillonnage préférentiel avec 7 particules. On observe pile/`true` à tout les instants. Au cours du temps, les valeurs de θ proches de 1 deviennent de plus en plus probables.

3.2 Échantillonnage préférentiel

À partir d'un échantillonneur, la méthode d'inférence la plus simple possible lance N exécutions indépendantes, appelés *particules*. À chaque instant, chaque particule exécute un pas de l'échantillonneur pour calculer un triplet (résultat, état suivant, score). Les scores sont ensuite normalisés pour obtenir une distribution catégorique, i.e., une distribution discrète sur les paires (résultat, état).

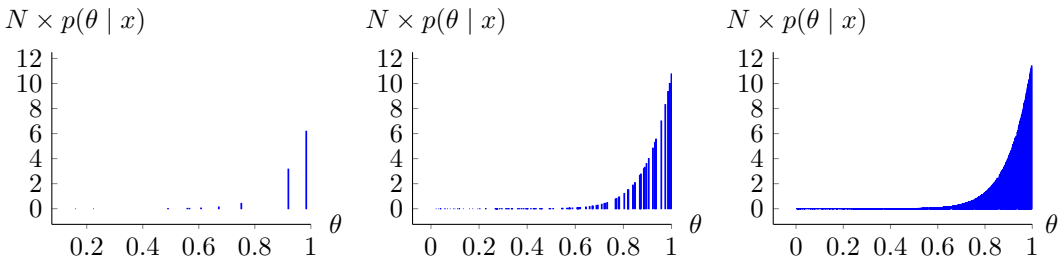
$$
\begin{aligned}
[\![\texttt{infer}(f(e))]\!]^{\text{init}}_{\gamma} &= [([\![f(e)]\!]^{\text{init}}_{\gamma}, 1)]_{1\leq i\leq N} \\
[\![\texttt{infer}(f(e))]\!]^{\text{step}}_{\gamma} &= \lambda s.\ let\ \left[(o_i, s'_i, w'_i) = let\ s_i, w_i = s[i]\ in\ \{\!|f(e)|\!\}^{\text{step}}_{\gamma}(s_i, w_i)\right]_{1\leq i\leq N}\ in \\
&\quad let\ \mu = \lambda U.\ \textstyle\sum_{1\leq i\leq N} \overline{w'_i} * \delta_{o_i}(U)\ in \\
&\quad \mu, [(s'_i, w'_i)]_{1\leq i\leq N}
\end{aligned}
$$

L'état de l'opérateur `infer` est donc un tableau initialisé avec N copies de l'état initial de l'échantillonneur associé au score initial 1. À chaque instant, chaque particule récupère son état courant et son score dans le tableau pour exécuter un pas de l'échantillonneur. Les scores sont ensuite normalisés pour obtenir la distribution de résultats μ, et le tableau est mis à jour pour l'instant suivant. On note $\overline{w_i} = w_i / \sum_{i=1}^{N} w_i$ les poids normalisés. Comparé à la sémantique idéale de `infer` de la Section 2.3, l'échantillonnage préférentiel approche l'intégrale incalculable par une somme discrète sur le tableau de particules.

Reprenons l'exemple `coin` de la Section 2.1. À l'instant initial, la première équation `init theta = sample (uniform_float (0., 1.))` tire un ensemble de valeurs possibles pour `theta`. Puis, à chaque instant, la première équation ne change plus (opérateur `init`), mais la seconde équation `() = observe (bernoulli theta, x)` met à jour le score pour chacune des valeurs possibles pour `theta` avec la formule suivante $w' = w * Bernoulli(\theta)_{\text{pdf}}(x)$. On peut enfin normaliser ces scores pour obtenir la distribution de sorties à chaque instant.

La Figure 3 illustre une exécution possible dans le cas où les valeurs observées sont toujours pile/`true`. On constate que les valeurs de `theta` les plus proche de 1 sont les plus probables, ce qui correspond bien aux premières observations.

La Figure 4 illustre la précision de l'échantillonnage préférentiel pour un nombre croissant de particules sur l'exemple précédent. On observe que la précision de la distribution obtenue dépend du nombre de particules utilisées. Sur cet exemple simple, 1000 particules suffisent pour obtenir une approximation raisonnable de la valeur théorique : $Beta(1, 10 + 1)$. Pour des modèles plus compliqués, où les paramètres cherchés sont de plus grande dimension, le nombre de particules nécessaires peut croître très rapidement. On peut cependant exploiter le fait que les particules sont indépendantes les unes des autres pour paralléliser le moteur d'inférence.



(a) 10 particules (b) 100 particules (c) 1000 particules

FIGURE 4 – Précision de l'échantillonnage préférentiel en fonction du nombre de particules N après 10 instants. On observe pile/`true` à tous les instants.

3.3 Parallélisation avec JAX

JAX [4] est une bibliothèque récente qui permet d'exécuter du code écrit dans un sous-ensemble de Python purement fonctionnel sur CPU, GPU ou TPU de manière transparente pour l'utilisateur [10]. Au moment de l'exécution, un compilateur *juste-à-temps* (JAT) spécialise les fonctions Python qui sont ensuite envoyées à XLA, [5] le compilateur haute-performance de Google pour GPU et TPU.

Opérateur `vmap`. JAX offre en particulier un opérateur `vmap` qui permet de vectoriser automatiquement une fonction pour l'exécuter de manière massivement parallèle. Pour les structures de données arborescentes simples, cet opérateur préserve la structure des entrées/sorties et ne vectorise que les feuilles. Par exemple, pour une fonction telle que `f(0) = ((1, 2), 3)` on aura :

```
vmap(f)([0, 0, 0]) = (([1, 1, 1], [2, 2, 2]), [3, 3, 3]).
```

Plus généralement, si les entrées sont décrites par le type `float t_in` et les sorties par le type `float t_out`, le type de `vmap` est le suivant :

```
val vmap: (float t_in → float t_out) → float array t_in → float array t_out
```

Dans notre contexte, cette propriété est fondamentale pour vectoriser l'état des particules qui stocke l'état interne de toutes les sous-expressions sous la forme de tuples imbriqués (cf. Section 2.3). En utilisant l'opérateur `vmap`, on peut ainsi implémenter un nœud ProbZelus d'ordre supérieur `zmap` qui exécute en parallèle N instances d'un nœud de la manière suivante.

```
from jax import vmap
class zmap(Node):
  def __init__(self, f, n):
      self.f = f()
      self.n = n

  def init(self)
      s_init = vmap(lambda _: init(self.f))(np.empty(self.n))
      return s_init
```

4. https://github.com/google/jax
5. https://www.tensorflow.org/xla

```
    def step(self, s, i):
        o, s = vmap(step(self.f))(s, i)
        return s, o
```

Les nœuds ProbZelus implémentent la classe `Node` qui impose la définition des deux méthodes `init` et `step`. L'état initial est obtenu en vectorisant sur un tableau de taille n une fonction qui renvoie l'état initial de `f` quel que soit son argument. La fonction de transition de `zmap` vectorise l'exécution de la fonction de transition de `f` sur l'état courant `s` et un tableau d'entrées `i`. `init` et `step` sont des fonctions génériques qui appellent les méthodes correspondantes.

L'implémentation de l'échantillonnage préférentiel suit le même schéma. L'état initial contient N copies de l'état initial de `f` associées à un score initial de 1. La fonction de transition déconstruit l'état courant en une paire (états, scores) qui permet de vectoriser l'exécution de l'échantillonneur. Les résultats sont ensuite normalisés pour obtenir une distribution.

```
class infer_importance(Node):
    def __init__(self, f, n):
        self.f = f()
        self.n = n

    def init(self):
        s_init = vmap(lambda _: (init(self.f), 1.0))(np.empty(self.n))
        return s_init

    def step(self, s, obs):
        s_in, w_in = s
        o, s_out, w_out = vmap(step(self.f))(s_in, w_in, obs)
        mu = normalize(o, w_out)
        return mu, (s_out, w_out)
```

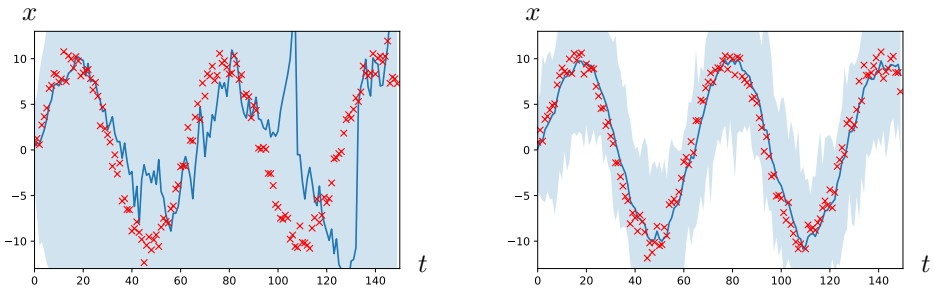
Remarque. Le générateur aléatoire de JAX qui permet d'échantillonner une valeur dans une distribution (utilisé pour la construction `sample`) prend explicitement une source aléatoire en argument : la *clé*. Nous avons ignoré ces clés pour simplifier la présentation. En pratique, les fonctions de transition prennent un argument supplémentaire *key* qui peut être regroupé avec le score pour former un argument unique *proba* qui contient toute l'information nécessaire pour les fonctions de transition probabilistes (cf. Section 4.1).

3.4 Filtre particulaire

L'échantillonnage préférentiel est une méthode relativement efficace pour inférer des paramètres constants à partir d'une série d'observations, comme dans l'exemple de la pièce `coin`. Malheureusement cette méthode montre rapidement ses limites pour des modèles où les paramètres peuvent changer au cours du temps. Par exemple, le modèle suivant implémente un modèle de Markov caché qui estime la position courante `x` à partir d'observations bruitées `obs`.

```
proba hmm obs = x where rec x = sample (gaussian (0. → pre x), speed)
                      and () = observe (gaussian (x, noise), obs)
```

À chaque instant, on suppose que la position courante n'est pas trop loin de la position précédente, i.e., suit une distribution normale centrée sur `0. → pre x` (la position précédente initialisée à 0). On impose également à chaque instant que la position estimée ne soit pas trop loin de l'observation courante, i.e., l'observation `obs` suit une distribution normale centrée sur `x`. `speed` et `noise` sont des constantes globales.



(a) échantillonnage préférentiel (b) filtre particulaire

FIGURE 5 – Résultat de l'inférence pour le modèle `tracker` avec 100 particules sur un sinus bruité pour 150 instants. Les points rouges sont les observations, la ligne bleue est la moyenne, la zone bleue contient les valeurs de toutes les particules (intervalle min/max à chaque instant).

Sur ce type de modèle, l'échantillonnage préférentiel correspond à une marche aléatoire : à chaque instant, la valeur courante de `x` est tirée aléatoirement à partir de la valeur précédente. La probabilité qu'une trajectoire aléatoire correspond à une série d'observations tend rapidement vers 0. Les estimations sont donc inexploitables après 2 ou 3 instants (Figure 5a).

Ré-échantillonnage. Pour résoudre ce problème, un *filtre particulaire* introduit une étape de ré-échantillonnage. Le nombre de particules reste constant au cours de l'exécution, mais à chaque instant, les particules les moins probables sont écartés et les particules les plus probables sont dupliqués. En d'autres termes, le tableaux de particules est recentré sur les observations.

$$
\begin{aligned}
[\![\texttt{infer}(f(e))]\!]^{\text{init}}_{\gamma} &= [(\{\!|f(e)|\!\}^{\text{init}}_{\gamma}, 1)]_{1\leq i\leq N} \\
[\![\texttt{infer}(f(e))]\!]^{\text{step}}_{\gamma} &= \lambda s.\ let\ \left[(o_i, s'_i, w'_i) = let\ s_i, w_i = s[i]\ in\ \{\!|f(e)|\!\}^{\text{step}}_{\gamma}(s_i, w_i)\right]_{1\leq i\leq N}\ in \\
&\quad let\ \mu = \lambda U.\ \textstyle\sum_{1\leq i\leq N} \overline{w'_i} * \delta_{(o_i,s_i)}(U)\ in \\
&\quad \pi_{1*}(\mu), [(\text{draw}(\pi_{2*}(\mu)), 1)]_{1\leq i\leq N}
\end{aligned}
$$

La sémantique de l'opérateur `infer` suit de prêt celle de l'échantillonnage préférentiel. À chaque instant, chaque particule exécute un pas de l'échantillonneur. Les scores sont ensuite normalisés pour obtenir une distribution sur les paires (résultat, nouvel état). Comme pour la sémantique idéale présentée en Section 2.3, on sépare ensuite cette distribution pour obtenir une paire de distributions. Le nouvel état est obtenu en ré-échantillonnant N valeurs dans la distribution d'états possibles associé à un score réinitialisé à 1.

L'implémentation de cette méthode d'inférence est très proche du nœud `infer_importance`. On utilise à nouveau l'opérateur `vmap` pour vectoriser l'exécution de l'échantillonneur. La fonction `normalize` renvoie une distribution catégorique qui peut être échantillonnée en utilisant la méthode `sample`.

```
class infer_pf(Node):
    def __init__(self, f, n):
        self.f = f()
        self.n = n

    def init(self):
        s_init = vmap(lambda _: (init(self.f), 1.0))(np.empty(self.n))
        return s_init
```

```
def step(s, obs):
    o, s_out, w_out = vmap(f_step)(s, np.ones(self.n), obs)
    mu = normalize(o, w_out)
    s_out = normalize(s_out, w_out).sample(sample_shape=self.n)
    return mu, s_out
```

La Figure 5b montre les résultats du filtre particulaire sur le modèle `hmm`. On observe qu'on obtient cette fois des résultats satisfaisants pour un modèle probabiliste réactif, où les flots de paramètres peuvent évoluer au cours du temps.

Il reste maintenant à comprendre comment compiler les modèles ProbZelus vers du code Python/JAX qui peut être appelé par les nœuds `infer_importance` et `infer_pf`. Dans la section suivante nous présentons un nouveau *backend* JAX pour ProbZelus.

4 Compiler ProbZelus vers JAX

Traditionnellement, les compilateurs de langages synchrones transforment des programmes à flot de données en programmes impératifs. La mémoire nécessaire à l'exécution du programme est allouée statiquement avant l'exécution. Les mises à jour sont réalisées par effet de bord. Ce schéma de compilation, bien adapté aux langages bas-niveau utilisés pour les systèmes embarqués, garantit une exécution en mémoire bornée. Malheureusement, JAX n'accepte que du code purement fonctionnel, plus facilement parallélisable.

Nous fondons notre travail sur le compilateur Zelus [8] qui a l'architecture suivante. (1) Le programme source est d'abord analysé et réécrit par des transformations successives dans un sous-ensemble du langage source. Parmi les analyses et transformations il y a, par exemple, le typage de données, l'analyse de causalité, la compilation des automates, la mise sous forme normale des équations, l'élimination de sous-expressions communes, ou la suppression de code mort. (2) Les équations mutuellement récursives sont ensuite ordonnancées pour satisfaire les dépendances de données (cf. Section 2.3) et le programme est traduit en OBC, un langage impératif intermédiaire (3) Enfin, le code OBC est compilé en OCaml impératif.

Pour bénéficier le plus possible de cette chaîne de compilation pour le nouveau *backend* vers JAX, nous remplaçons seulement la dernière étape. À partir du code OBC, nous le traduisons dans un langage μF purement fonctionnel que nous pouvons ensuite compiler en JAX. [6]

4.1 Langages intermédiaires

Le langage OBC. Le langage intermédiaire OBC (pour Object Based Code) a été introduit pour compiler un langage synchrone semblable à Lustre vers du code impératif (e.g., C) [5]. Il est utilisé par les compilateurs Heptagon [16], Vélus [7] et Zelus (avec quelques spécificités pour chacun de ces compilateurs). OBC permet de représenter une fonction de flot par un état et un ensemble de fonctions de transition agissant sur cet état et le modifiant en place. La syntaxe du langage étendue avec les constructions probabilistes est la suivante :

6. Dans [2], μF permet d'exprimer la sémantique mais ne correspond pas au code généré.

$$
\begin{array}{rcl}
\mathit{program} & ::= & d^* \\
d & ::= & \texttt{machine proba}^? m = \\
 & & \quad \texttt{memory}\ (x, \ldots, x) \\
 & & \quad \texttt{instances}\ (o : m, \ldots, o : m, o : \texttt{infer}(m), \ldots, o : \texttt{infer}(m)) \\
 & & \quad \texttt{reset}() = S \\
 & & \quad \texttt{step}(p)\ \texttt{returns}(p) = S \\
S & ::= & \texttt{var}\ x\ \texttt{in}\ S \mid x := e \mid \texttt{state}(x) := e \mid S\ ;\ S \mid \texttt{skip} \\
 & & \mid \texttt{match}\ x\ \texttt{with} \mid C \texttt{ -> } S \ldots \mid C \texttt{ -> } S \mid o.\texttt{reset} \mid p := o.\texttt{step}(e) \\
 & & \mid p := o.\texttt{pstep}(e) \mid p := \texttt{sample}(e) \mid \texttt{observe}(e, e) \\
e & ::= & c \mid x \mid \texttt{state}(x) \mid op(e) \\
p & ::= & x \mid (p, p)
\end{array}
$$

Un programme est une suite de déclarations de machines (ou classes). L'annotation optionnelle `proba` indique un modèle probabiliste. Une machine m est composée de quatre champs: (1) `memory` l'état de la fonction de transition, (2) `instances` les machines utilisées dans m, (3) `reset` la méthode de réinitialisation de l'état, et (4) `step` la fonction de transition qui prend une entrée, met à jour l'état et renvoie une sortie. Les instances sont annotées par le type m de la machine ou `infer`(m) s'il s'agit d'une instance de l'opérateur `infer`.

Une instructions peut être une déclaration de variable mutable locale (`var` x `in` S), une mise à jour d'une variable locale (x `:=` e), une mise à jour d'une variable d'état (`state`(x) `:=` e), une séquence d'instruction (S `;` S), une instruction sans effets (`skip`), une structure de contrôle (`match`/`with`), ou l'appel des méthodes (`step` or `reset`) d'une instance. OBC est étendu avec les instructions probabilistes `sample`, `observe`, et une méthode `pstep` qui correspond à l'appel de la méthode `step` d'une machine probabiliste. Une expression peut être une constantes (c), l'accès à une variable local (x), l'accès à une variable d'état (`state`(x)), ou l'application d'un opérateur primitif.

Le langage μF. μF est un simple langage purement fonctionnel sans ordre supérieur [2]. La syntaxe de μF est définie par la grammaire suivante :

$$
\begin{array}{rcl}
\mathit{program} & ::= & d^* \\
d & ::= & \texttt{val}\ p = e \mid \texttt{val}\ m = \texttt{stream}\ \{\ \texttt{init} = e\ ;\ \texttt{step}(p, p) = e\ \} \\
e & ::= & c \mid x \mid (e, e) \mid op(e) \mid f(e) \mid \texttt{match}\ x\ \texttt{with} \mid C \texttt{ -> } e \ldots \mid C \texttt{ -> } e \\
 & & \mid \texttt{let}\ p = e\ \texttt{in}\ e \mid \texttt{init}(m) \mid \texttt{unfold}(x, e) \\
 & & \mid \texttt{sample}(\mathit{proba}, e) \mid \texttt{observe}(\mathit{proba}, e, e) \mid \texttt{infer}(m)
\end{array}
$$

Un programme est une suite de déclaration de valeurs et de fonctions de suites (`stream`). Une fonction de suite m est composée d'un état initial (`init`) et d'une fonction de transition (`step`). Une fonction de transition prend en argument un état et une entrée et renvoie une sortie et un nouvel état. Une expression peut être une constante, une variable, une paire, l'application d'un opérateur, un appel de fonction, une conditionnelle, ou une définition locale. L'expression `init`(m) renvoie une nouvelle instance m où l'état de l'instance est l'état initial. L'expression `unfold`(x,e) exécute un pas de l'instance x et renvoie l'élément suivant du flot et une nouvelle instance avec un état mis à jour.

La fonction de transition d'une machine probabiliste ajoute dans ses entrées/sorties une valeur *proba* qui représente l'état du modèle probabiliste, par exemple le score pour les méthodes d'inférence particulaire, ou les clés pour le générateur aléatoire de JAX (cf. Section 3.3). Cette valeur *proba* est passée en argument, mise à jour et renvoyée par les opérateurs probabilistes `sample` et `observe`. Enfin, `infer` correspond à l'instanciation de l'opérateur d'inférence sur un modèle probabiliste.

$$
\begin{array}{rcl}
\mathcal{C}_p(\texttt{skip}) & = & p \\
\mathcal{C}_p(x \texttt{ := } e) & = & \texttt{let } x \texttt{ = } \mathcal{C}(e) \texttt{ in } p \\
\mathcal{C}_p(\texttt{state}(x) \texttt{ := } e) & = & \texttt{let } x \texttt{ = } \mathcal{C}(e) \texttt{ in } p \\
\mathcal{C}_p(\texttt{var } x \texttt{ in } S) & = & \texttt{let } (p,x) \texttt{ = } \mathcal{C}_{(p,x)}(S) \texttt{ in } p \\
\mathcal{C}_p(S_1 \texttt{ ; } S_2) & = & \texttt{let } p \texttt{ = } \mathcal{C}_p(S_1) \texttt{ in } \mathcal{C}_p(S_2) \\
\mathcal{C}_p(\texttt{match } e \texttt{ with} & = & \texttt{match } \mathcal{C}(e) \texttt{ with} \\
\quad \texttt{|} C_1 \texttt{ -> } S_1 \ldots \texttt{|} C_n \texttt{ -> } S_n) & & \quad \texttt{|} C_1 \texttt{ -> } \mathcal{C}_p(S_1) \ \ldots \texttt{|} C_n \texttt{ -> } \mathcal{C}_p(S_n) \\
\mathcal{C}_p(o\texttt{.reset}) & = & \texttt{let } o \texttt{ = init}(m) \texttt{ in } p \qquad \text{si } o : m \\
\mathcal{C}_p(o\texttt{.reset}) & = & \texttt{let } o \texttt{ = infer}(m) \texttt{ in } p \qquad \text{si } o : \texttt{infer}(m) \\
\mathcal{C}_p(p' \texttt{ := } o\texttt{.step}(e)) & = & \texttt{let } (p',o) \texttt{ = unfold}(o,\mathcal{C}(e)) \texttt{ in } p \\
\mathcal{C}_p(p' \texttt{ := } o\texttt{.pstep}(e)) & = & \texttt{let } ((p',proba),o) \texttt{ = unfold}(o,(proba,\mathcal{C}(e))) \texttt{ in } p \\
\mathcal{C}_p(\texttt{sample}(proba,e)) & = & \texttt{let } (p',proba) \texttt{ = sample}(proba,\mathcal{C}(e)) \texttt{ in } p \\
\mathcal{C}_p(\texttt{observe}(proba,e_1,e_2)) & = & \texttt{let } proba \texttt{ = sample}(proba,\mathcal{C}(e)) \texttt{ in } p
\end{array}
$$

FIGURE 6 – Compilation de OBC vers μF.

4.2 Compilation

Compilation de OBC vers μF. La fonction de compilation rend explicite la manipulation de l'état pour transformer le code OBC impératif en code fonctionnel.

Considérons la machine m suivante :

$$
\begin{array}{l}
\texttt{machine } m \texttt{ =} \\
\quad \texttt{memory } (x_1, \ldots, x_n) \\
\quad \texttt{instances } (o_1 : m_1, \ldots, o_t : m_t, o_{t+1} : \texttt{infer}(m_{t+1}), \ldots, o_k : \texttt{infer}(m_k)) \\
\quad \texttt{reset}() \texttt{ = } S \\
\quad \texttt{step}(p) \texttt{ returns}(p) \texttt{ = } S
\end{array}
$$

La compilation d'une telle machine m produit une déclaration de fonction de suites (`stream`) et se décompose en deux étapes: la compilation de la méthode `reset` pour produire le champs `init`, et la compilation de la méthode `step` pour produire le champs `step`.

$$
\begin{array}{rcl}
\mathcal{C}(\texttt{reset}() \texttt{ = } S) & = & \texttt{init = } \mathcal{C}_{x_1,\ldots,x_n,o_1,\ldots,o_k}(S) \\
\mathcal{C}(\texttt{step}(p_{input}) \texttt{ returns}(p_{output})) & = & \\
\multicolumn{3}{c}{\texttt{step}(x_1, \ldots, x_n, o_1, \ldots, o_k, p_{input}) \texttt{ = } \mathcal{C}_{(p_{output},x_1,\ldots,x_n,o_1,\ldots,o_k)}(S)}
\end{array}
$$

Pour machine probabiliste, la variable *proba* est ajouté en entrée/sortie de la fonction `step`.

$$
\begin{array}{rcl}
\mathcal{C}(\texttt{step}(p_{input}) \texttt{ returns}(p_{output})) & = & \\
\multicolumn{3}{c}{\texttt{step}(x_1, \ldots, x_n, o_1, \ldots, o_k, (proba, p_{input})) \texttt{ = } \mathcal{C}_{((p_{output},proba),x_1,\ldots,x_n,o_1,\ldots,o_k)}(S)}
\end{array}
$$

La fonction de compilation des instructions $\mathcal{C}_p(S)$ est définie Figure 6. Cette fonction est paramétrée par l'ensemble p des variables potentiellement mises à jour par l'évaluation de l'instruction S et renvoie la valeur de ces variables. La compilation de `skip` renvoie simplement p. La compilation de l'affectation d'une variable ($\mathcal{C}_p(x \texttt{ := } e)$) respecte l'invariant que la variable x fait parti des variables de p et donc l'expression $\texttt{let } x \texttt{ = } \mathcal{C}(e) \texttt{ in } p$ masque la valeur précédente de x et retourne sa nouvelle valeur. La compilation de la méthode $o\texttt{.reset}$ instancie un nouvel état. Selon la déclaration de l'instance o, l'état est alloué avec l'opérateur `init` ou `infer`. La compilation de l'appel de machines probabilistes et des opérateurs `sample` et `observe` rend explicite l'état probabiliste qui est mis à jour par chacune de ces opérations. La fonction de compilation des expressions $\mathcal{C}$ correspond globalement à l'identité. On peut simplement remarquer que les variables d'état et locales sont traitées de manière similaire en μF: $\mathcal{C}(\texttt{state}(x)) = x$.

Remarque. Cette fonction de compilation génère beaucoup de `let` imbriqués et de copies de variables. Une passe par réécriture source à source est appliquée pour simplifier le code généré.

Compilation de μF vers JAX A partir du code μF purement fonctionnel, il est maintenant possible de générer du code Python compatible avec JAX : les `stream`s sont compilés en classes où les champs `init` et `step` deviennent des méthodes.

Afin d'éviter les problèmes de portée de variables en Python, une passe de compilation réécrit le code μF sans définitions imbriquées avant la traduction vers Python. La compilation est ensuite assez directe. La construction `match` est traduit vers des conditionnelles JAX qui prennent en arguments des fermetures pour contrôler leur évaluation. Enfin, pour permettre la parallélisation automatique, les structures de données doivent être sérialisables. Comme l'état des programmes ProbZelus respecte une structure arborescente simple, on peut utiliser le décorateur `@register_pytree_node_class` pour générer automatiquement les fonctions de sérialisation et déserialisation.

Par exemple, le code généré pour le modèle `coin` de la section Section 2 est le suivant :

```
@register_pytree_node_class
class coin(Node):
    def init(self):
        return {"theta": 42.0, "first": True}

    def step(self, *args):
        (state, (proba, obs)) = args
        def _ft(_):
            (proba, theta) = sample(proba, uniform_float(0.0, 1.0))
            return (proba, {**state, "theta": theta})
        def _ff(_):
            return (proba, state)
        (proba, state) = cond(state["first"], _ft, _ff, None)
        state = {**state, "first": False}
        (proba, ()) = observe(proba, bernoulli(state["theta"], obs)
        return (state, (proba, state["theta"]))
```

5 Évaluation

Nous avons évalué les performances de notre nouveau moteur d'inférence pour ProbZelus sur un ensemble d'exemples qui illustrent plusieurs aspects du langage [2]. Notre évaluation se concentre sur les deux questions suivantes : (**QR1**) Quel est l'impact du nouveau moteur d'inférence sur la précision des estimations ? (**QR2**) La parallélisation automatique permet-elle d'améliorer les performances ?

5.1 Méthode expérimentale

Pour chacun des exemples, nous mesurons le temps d'exécution et la précision obtenu après 500 pas d'exécution avec le moteur d'inférence OCaml sur CPU, et le moteur d'inférence JAX sur GPU. La précision est l'erreur quadratique moyenne (*Mean Square Error* MSE) sur l'ensemble des paramètres du modèle. L'ensemble des expériences a été réalisé sur un serveur Intel Xeon E7 avec 64 cœurs, 128Go de RAM et un GPU Nvidia Quadro M6000 avec 24 Go de RAM.

Les exemples retenus sont les suivants : **Coin** correspond au modèle de la pièce présenté en Section 2 où un agent estime le biais d'une pièce à partir d'une série de lancés. On suppose initialement que le biais suit une loi uniforme sur $[0, 1]$. **Gaussian-Gaussian** estime la moyenne et l'écart type d'une loi normale à partir d'un ensemble d'échantillons. On suppose initialement que la moyenne suit une distribution $\mathcal{N}(0, 10)$, et que l'écart type suit une distribution $\mathcal{N}(0, 1)$. **Kalman** correspond au modèle de la Section 3.4 où un agent qui estime sa position à partir d'observations bruitées. À chaque instant, on suppose que la position courante suit une distribution normale autour de la position précédente, et que l'observation courante suit une distribution normale autour de la position courante.

5.2 QR1 : Précision

Les résultats de précision sont présentés en Figure 7. On constate sur l'ensemble des exemples considérés que les deux moteurs d'inférence sont équivalents. Comme attendu, pour tous les exemples, l'erreur diminue quand le nombre de particules augmente avant d'atteindre un plateau lorsque l'estimation se rapproche de la distribution théorique. Les différences mineures sont dues aux divergences d'implémentation des générateurs aléatoires OCaml et JAX.

5.3 QR2 : Performance

Les résultats de performance sont présentés en regard des résultats de précisions en Figure 8. Les performances du moteur d'inférence OCaml font apparaître le compromis classique temps de calcul/précision : plus le nombre de particules est élevé, plus le temps de calcul est important, mais plus l'estimation est précise.

Les performances du moteur d'inférence JAX restent quasiment constantes jusqu'à 100 000 particules. La compilation vers du code GPU optimisé est coûteuse et n'est amortie que pour un nombre élevé de particules (autour de 10 000 pour nos exemples). En revanche, une fois le code compilé, JAX peut facilement exécuter un très grand nombre de particules en parallèle à peu de frais. Pour tous les modèles, les limites du GPU sont atteintes vers 200 000 particules. On retrouve alors des performances qui se dégradent linéairement avec le nombre de particules, mais qui restent significativement supérieurs à celle de l'implémentation en OCaml.

6 Travaux connexes et conclusion

La parallélisation des algorithmes d'inférence pour la programmation probabiliste est un sujet de recherche dynamique. La plupart des langages probabiliste modernes [6, 23, 14, 25] s'appuient sur des systèmes de calcul parallèle efficaces popularisés par l'apprentissage profond. Notre travail est cependant le premier à appliquer ces techniques à un langage réactif comme ProbZelus. Il existe également plusieurs travaux sur la parallélisation de langages synchrones [17, 22, 12]. Mais aucun ne traite les problèmes spécifiques de la programmation probabiliste.

Dans cet article nous avons montré comment utiliser JAX pour paralléliser l'inférence probabiliste sur les modèles réactifs programmés avec ProbZelus. Notre implémentation permettra d'experimenter avec d'autres fonctionnalité de JAX comme l'auto-différentiation qui est un composant clé de plusieurs méthodes d'inférence plus avancées.

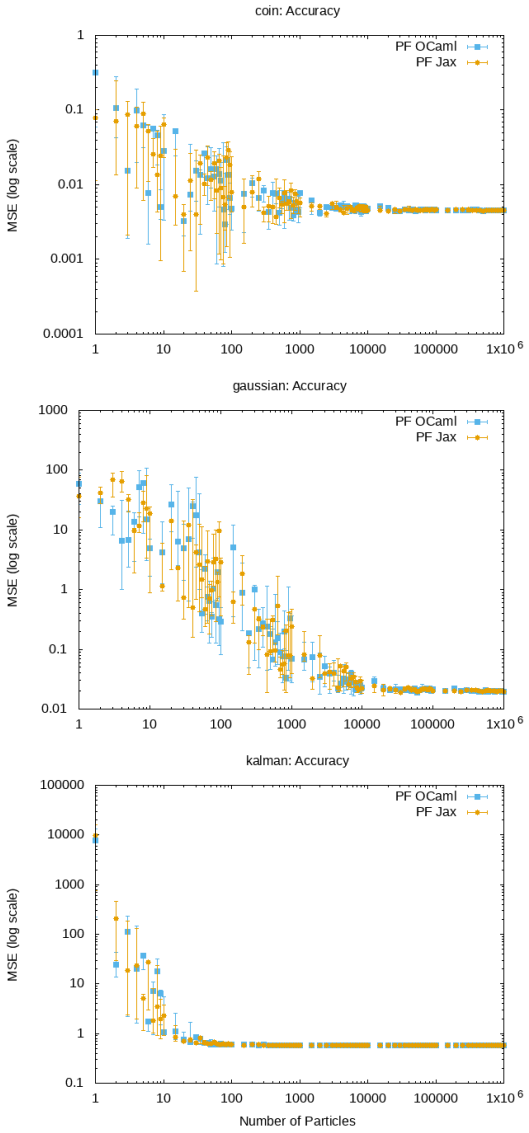


FIGURE 7 – Précision en fonction du nombre de particules.

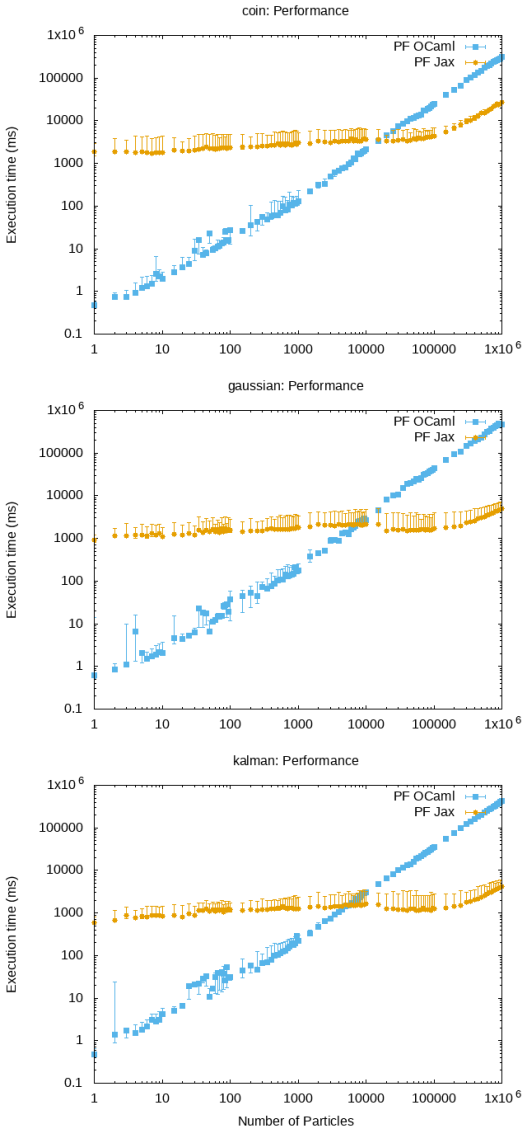


FIGURE 8 – Performance en fonction du nombre de particules.

Références

[1] Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. Programmation d'applications réactives probabilistes. In *JFLA*, Gruissan, France, 2020.

[2] Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. Reactive probabilistic programming. In *PLDI*. ACM, 2020.

[3] Albert Benveniste, Timothy Bourke, Benoît Caillaud, Bruno Pagano, and Marc Pouzet. A type-based analysis of causality loops in hybrid systems modelers. In *HSCC*, pages 71–82. ACM, 2014.

[4] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proc. IEEE*, 91(1):64–83, 2003.

[5] Dariusz Biernacki, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *LCTES*, pages 121–130. ACM, 2008.

[6] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep universal probabilistic programming. *J. Mach. Learn. Res.*, 20:28:1–28:6, 2019.

[7] Timothy Bourke, Lélio Brun, and Marc Pouzet. Mechanized semantics and verified compilation for a dataflow synchronous language with reset. *Proc. ACM Program. Lang.*, 4(POPL):44:1–44:29, 2020.

[8] Timothy Bourke and Marc Pouzet. *Zélus, a Hybrid Synchronous Language*. École normale supérieure, September 2013. Distribution at: `zelus.di.ens.fr`.

[9] Timothy Bourke and Marc Pouzet. Zélus: a synchronous language with ODEs. In *HSCC*, pages 113–118. ACM, 2013.

[10] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[11] Paul Caspi and Marc Pouzet. A co-iterative characterization of synchronous stream functions. In *CMCS*, volume 11 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[12] Albert Cohen, Léonard Gérard, and Marc Pouzet. Programming parallelism with futures in lustre. In *EMSOFT*, pages 197–206. ACM, 2012.

[13] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. SCADE 6: A formal language for embedded critical software development. In *TASE*, pages 1–11. IEEE Computer Society, 2017.

[14] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *PLDI*, pages 221–236. ACM, 2019.

[15] Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. Sequential Monte Carlo samplers. *J. Royal Statistical Society: Series B (Statistical Methodology)*, 68(3):411–436, 2006.

[16] Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet. A modular memory optimization for synchronous data-flow languages: application to arrays in a lustre compiler. In *LCTES*, pages 51–60. ACM, 2012.

[17] Alain Girault. A survey of automatic distribution method for synchronous programs. In F. Maraninchi, M. Pouzet, and V. Roy, editors, *SLAP*, ENTCS, 2005.

[18] Noah D. Goodman and Andreas Stuhlmüller. The design and implementation of probabilistic programming languages, 2014. Accessed April 2020.

[19] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[20] Lawrence M. Murray and Thomas B. Schön. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control*, 46:29–43, 2018.

[21] Bruno Pagano. Le Langage Scade 6 pour les systèmes embarqués De la conception à la compilation. Séminaire du cours de G. Berry au Collège de France, 2013.

[22] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discret. Event Dyn. Syst.*, 21(3):307–338, 2011.

[23] Du Phan, Neeraj Pradhan, and Martin Jankowiak. Composable effects for flexible and accelerated probabilistic programming in numpyro. *arXiv preprint arXiv:1912.11554*, 2019.

[24] David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank D. Wood. Design and implementation of probabilistic programming language Anglican. In *IFL*, pages 6:1–6:12. ACM, 2016.

[25] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. Deep probabilistic programming. In *ICLR (Poster)*. OpenReview.net, 2017.