

Filtrer sans s’appauvrir : inférer les paramètres constants des modèles réactifs probabilistes

Guillaume Baudart,^{1,2} Grégoire Bussone,^{2,3}
Louis Mandel,⁴ et Christine Tasson³

¹ Inria Paris

² École normale supérieure – PSL university

³ Sorbonne Université

⁴ IBM Research

Résumé

ProbZelus étend le langage synchrone Zelus pour permettre de décrire des modèles probabilistes synchrones. Là où la programmation synchrone implémente des fonctions de suites, un langage probabiliste synchrone permet de décrire des modèles qui calculent des suites de distributions. On peut par exemple estimer la position d’un objet en mouvement à partir d’observations bruitées ou estimer l’incertitude d’un capteur à partir d’une suite d’observations. Ces problèmes mêlent des flots de variables aléatoires – les *paramètres d’état* qui changent au cours du temps – et des variables aléatoires constantes – les *paramètres constants*.

Pour estimer les paramètres d’état, les algorithmes d’inférence bayésienne de Monte Carlo séquentiels reposent sur des techniques de filtrage. Le filtrage est une méthode approchée qui consiste à perdre volontairement de l’information sur l’approximation actuelle pour recentrer les estimations futures autour de l’information la plus significative. Malheureusement, cette perte d’information est dommageable pour l’estimation des paramètres constants qui n’évoluent pas au cours du temps. Ce phénomène s’appelle l’appauvrissement.

Inspirés de la méthode d’inférence *Assumed Parameter Filter* (APF), nous proposons (1) une analyse statique, (2) une passe de compilation et (3) une nouvelle méthode d’inférence modulaire pour ProbZelus qui évite l’appauvrissement pour les paramètres constants tout en gardant les performances des algorithmes de Monte Carlo séquentiels pour les paramètres d’états.

1 Introduction

La programmation synchrone flot de données [5] est un paradigme dans lequel les fonctions, appelées noeuds, manipulent des suites infinies, appelées flots. Ces flots progressent au même rythme, de manière synchrone. L’expressivité des langages synchrones est volontairement restreinte, ce qui permet à des compilateurs spécialisés de générer du code efficace et correct par construction avec de fortes garanties sur le temps d’exécution et sur l’utilisation de la mémoire [19]. Le langage industriel Scade est notamment utilisé pour la conception de systèmes embarqués critiques [13].

Les langages probabilistes [6, 14, 17, 18] étendent des langages de programmation généralistes avec des constructions probabilistes. Suivant la méthode bayésienne, un *modèle* probabiliste décrit une distribution de probabilité (la distribution *a posteriori*) à partir d’une croyance initiale (la distribution *a priori*) et d’observations concrètes (les entrées).

Dans la même lignée de langages, ProbZelus [2] est une extension probabiliste du langage synchrone flot de données Zelus [8]. Ce langage permet de décrire des modèles réactifs probabilistes. Là où la programmation synchrone implémente des fonctions de suites, un langage probabiliste synchrone permet de décrire des modèles qui calculent des suites de distributions.

Un exemple est le problème de localisation et cartographie simultanées (SLAM pour *Simultaneous Localization and Mapping*) [20]. Un robot cherche à inférer à la fois sa position et une carte de son environnement. La position est un *paramètre d'état* représenté par un flot de variables aléatoires. À chaque instant, une nouvelle position doit être estimée à partir de la position précédente et des nouvelles observations. La carte est un *paramètre constant* représenté par une variable aléatoire dont la valeur est progressivement affinée à chaque nouvelle observation à partir d'une croyance initiale, la distribution *a priori*. Ce type de problème qui mêle paramètres constants et paramètres d'état qui changent au cours du temps est appelé *State-Space Models* (SSM) [10]. Tout programme probabiliste synchronre peut s'exprimer comme un SSM.

Pour estimer les paramètres d'état, les algorithmes d'inférence de Monte Carlo séquentiels (SMC pour *Sequential Monte Carlo*) reposent sur des techniques de filtrage [10, 15]. Le filtrage est une méthode approchée qui consiste à perdre volontairement de l'information sur l'approximation actuelle pour recentrer les estimations futures autour de l'information la plus significative. Ces méthodes sont donc particulièrement bien adaptées au contexte réactif où un système en interaction avec son environnement ne s'arrête jamais et doit donc s'exécuter avec des ressources bornées. Toutes les méthodes d'inférence de ProbZelus appartiennent à cette famille [1–3]. Malheureusement, cette perte d'information est dommageable pour l'estimation des paramètres constants qui n'évoluent pas au cours du temps. Ce phénomène s'appelle l'appauvrissement.

Pour éviter ce problème, l'algorithme d'inférence *Assumed Parameter Filter* (APF) [16] propose, à chaque instant, de décomposer l'inférence en deux temps : (1) estimation des paramètres d'état, et (2) mise à jour des paramètres constants. Cette méthode suppose que les paramètres constants sont bien identifiés et que le modèle est écrit sous une forme qui permet de mettre à jour les distributions des paramètres constants sachant la valeur des paramètres d'état.

Dans cet article, nous proposons une nouvelle méthode d'inférence fondée sur APF pour ProbZelus. Nous présentons en particulier les contributions suivantes : Section 3 une analyse statique pour identifier les paramètres constants, Section 4 une passe de compilation pour générer un modèle exploitable par APF, Section 5 une implémentation de l'algorithme APF pour les modèles ProbZelus. Nous évaluons cette nouvelle méthode d'inférence sur un ensemble de modèles ProbZelus. Le code et une annexe contenant la preuve du théorème principal sont disponibles à l'adresse suivante : <https://github.com/rpl-lab/jfla23-apf>.

2 Motivation

Pour motiver notre approche, considérons l'exemple d'un radar adapté de [10, Section 2.4.1] illustré Figure 1. On cherche à estimer la position x_t d'un bateau au cours du temps à partir d'une mesure de distance (par exemple à l'aide d'un sonar), et d'une mesure angulaire. On suppose que le bateau dérive à vitesse constante θ et on cherche également à estimer cette vitesse. Ce modèle est donc un *State-Space Model* (SSM). On s'intéresse à la distribution d'un paramètre d'état (le flot de variables aléatoires x_t) et d'un paramètre constant (la variable aléatoire θ).

2.1 Un modèle de radar en ProbZelus

Un premier modèle probabiliste décrit le mouvement du bateau, c'est-à-dire à chaque instant $t > 0$, la distribution *a priori* des variables θ et x_t en fonction de la position précédente x_{t-1} .

```
proba move x0 = theta, x where
  rec init theta = sample(mv_gaussian(zeros, i2))
  and x = x0 → sample(mv_gaussian(pre x +@ theta, 0.5 *@ i2))
```

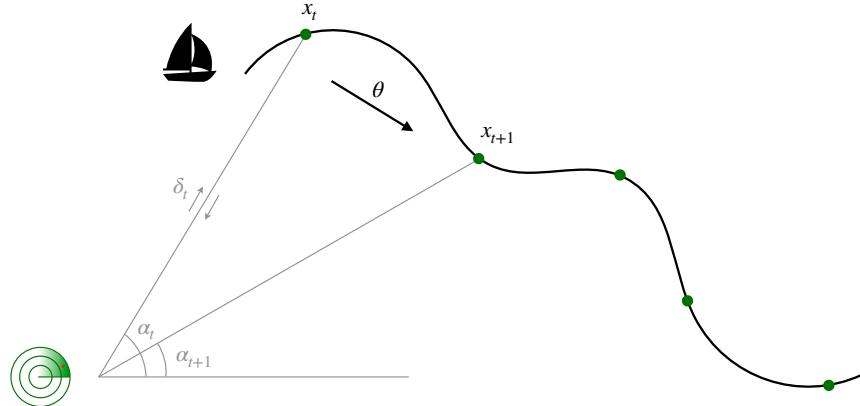


FIGURE 1 – Le radar cherche à estimer la position courante du bateau x_t et sa vitesse de dérive constante θ à partir du délai de rebond d'un sonar δ_t et d'une observation angulaire α_t .

Le mot-clé `proba` introduit un modèle probabiliste appelé `move` qui prend en argument la position initiale $x0$ et renvoie, à chaque instant, la vitesse `theta` et la position courante `x`. Les variables `x0`, `theta`, et `x` sont des vecteurs à deux dimensions dans \mathbb{R}^2 . Le paramètre constant `theta` est introduit par le mot-clé `init`. Ici, on suppose *a priori*, que `theta` est échantillonnée dans une gaussienne multivariée centrée sur $(0, 0)^T$ et de covariance I_2 , la matrice identité de dimension 2 (on suppose que les deux composantes de la vitesse ne sont pas corrélées). La définition de la position `x` utilise les opérateurs d'initialisation `→` et de délai unitaire `pre`. Au premier instant, $x_0 = x0$. Puis, aux instants $t > 0$ suivants, x_t est échantillonnée dans une gaussienne centrée sur la valeur précédente x_{t-1} translatée de `theta` et de covariance $0.5 * I_2$. Le paramètre d'état `x` est donc un flot de variables aléatoires.

Le modèle de radar suivant retourne, à chaque instant, la distribution *a posteriori* de la vitesse du bateau et de sa position à partir des distributions *a priori* générées par `move`, et de deux *observations* bruitées : `delta` le délai de rebond du sonar et `alpha` la mesure angulaire.

```
proba radar (delta, alpha) = theta, x where
  rec theta, x = move x0
  and d = sqrt((vec_get x 0) ** 2. + (vec_get x 1) ** 2.)
  and a = atan(vec_get x 1 /. vec_get x 0)
  and () = observe(gaussian(2. *. d /. c, delta_noise), delta)
  and () = observe(gaussian(a, alpha_noise), alpha)
```

À partir de la vitesse `theta` et de la position courante $x = (x_1, x_2)^T$ estimée, on calcule la distance $d = \sqrt{x_1^2 + x_2^2}$, et l'angle $a = \text{atan}(x_2/x_1)$. La construction `observe` permet ensuite de contraindre le modèle en supposant (1) que le temps de rebond observé `delta` a été échantillonné dans une gaussienne centrée sur $2d/c$ (où `c` est la vitesse du signal envoyé par le sonar), et (2) que la mesure angulaire `angle` a été échantillonnée dans une gaussienne centrée sur `a`.

Finalement, le nœud `main` prend en argument les flots d'observations `delta` et `alpha` et lance l'inférence sur le modèle `radar` pour renvoyer à chaque instant une approximation `d_theta` de la distribution de la vitesse `theta` et une approximation `d_x` de la distribution de la position `x`.

```
node main (delta, alpha) = d_theta, d_x where
  rec d = infer radar (delta, alpha)
  and d_theta, d_x = Dist.split d
```

Données : le modèle `model`, l'entrée y_t et le résultat précédent μ_{t-1} .
Résultat : μ_t une approximation de la distribution de p_t .

pour chaque particule $i = 1$ à N faire

```

    |    $p_{t-1}^i = \text{sample}(\mu_{t-1})$ 
    |    $p_t^i, w_t^i = \text{model}(y_t \mid p_{t-1}^i)$ 
    |    $\mu_t = \mathcal{M}(\{w_t^i, p_t^i\}_{1 \leq i \leq N})$ 
    retourner  $\mu_t$ 
```

Algorithme 1 : Filtre particulaire.

Le mot-clé `node` introduit un nœud déterministe classique. La fonction `Dist.split` sépare une distribution sur des paires (ici `theta`, et `x`) en une paire de distributions. On peut ensuite analyser ces distributions en calculant moyennes, quantiles, etc. ou en affichant un nuage d'échantillons tirés dans la distribution comme dans la Figure 2.

2.2 Le filtrage et ses limites

Les méthodes d'inférence proposées par ProbZelus [1–3] appartiennent à la famille des algorithmes de Monte Carlo séquentiels [10, 15].

Ces méthodes reposent sur un ensemble de simulations indépendantes, appelées *particules*. Chaque particule renvoie une valeur de sortie associée à un score. Le score représente la qualité de la simulation. Un grand nombre de particules permet d'approximer la distribution recherchée. La construction `sample(d)` tire aléatoirement une valeur dans la distribution `d`, et la construction `observe(d, x)` modifie le score courant de la particule en lui ajoutant la *vraisemblance* de l'observation `x` par rapport à la distribution `d`. À chaque instant, l'opérateur `infer` accumule les valeurs calculées par chaque particule pondérées par leurs scores pour approximer la distribution *a posteriori*.

Filtre particulaire. Si le modèle fait appel à l'opérateur `sample` à chaque instant, par exemple pour estimer la position du bateau, la méthode précédente implémente une simple marche aléatoire pour chaque particule. À mesure que le temps avance, il devient de plus en plus improbable qu'une des marches aléatoires coïncide avec le flot d'observations. Le score associé à chaque particule dégringole rapidement vers 0.

Pour régler ce problème, les méthodes de Monte Carlo séquentielles ajoutent une étape de filtrage. L'Algorithme 1 décrit ainsi l'exécution d'un instant pour un filtre particulaire. À chaque instant t , une particule $1 \leq i \leq N$ correspond à une valeur possible des paramètres (*i.e.*, variables aléatoires) p_i^t du modèle. On commence par échantillonner un nouvel ensemble de particules dans la distribution obtenue à l'instant précédent. Les particules les plus probables sont ainsi dupliquées et les moins probables sont éliminées. On recentre ainsi l'inférence autour de l'information la plus significative tout en conservant le même nombre de particules tout au long de l'exécution. Sachant l'état précédent p_{t-1}^i , chaque particule exécute ensuite un pas du modèle pour obtenir un échantillon des paramètres p_t^i associé à un score w_t^i . À la fin de l'instant, on construit une distribution μ_t où chaque particule est associée à son score. $\mathcal{M}(\{w_t^i, p_t^i\}_{1 \leq i \leq N})$ est une distribution multinomiale, où la valeur p_t^i est associée à la probabilité w_t^i ¹.

Appauvrissement. Malheureusement, cette approche engendre une perte d'information dommageable pour l'estimation de paramètres constants. Sur l'exemple du radar, au premier

1. On note $\overline{w_t^i} = w_t^i / \sum_{i=1}^N w_t^i$ les scores normalisés.

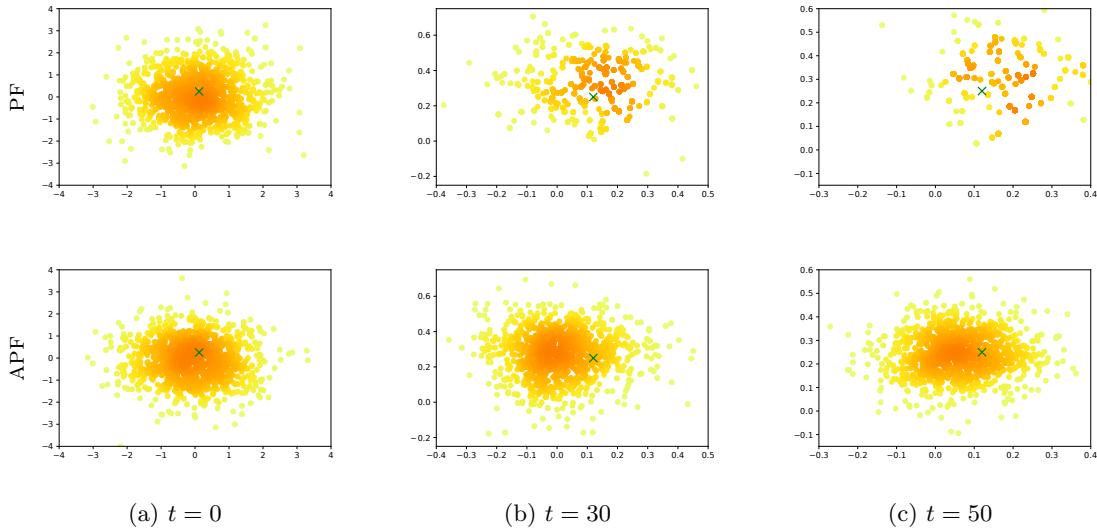


FIGURE 2 – Estimations du paramètre theta de l'exemple du radar au cours du temps avec un filtre particulier (PF, en haut) et *Assumed Parameter Filter* (APF, en bas). La vraie vitesse de dérive est indiquée par une croix verte. Le gradient de couleur représente la densité de points. L'échelle change au cours du temps. Les résultats peuvent différer d'une exécution à l'autre.

instant, chaque particule tire une valeur aléatoire pour le paramètre θ . À chaque instant, les particules dupliquées partagent la même valeur pour θ . La quantité d'information utile pour estimer θ décroît donc à chaque nouveau filtrage et, au bout d'un certain temps, il ne reste plus qu'une seule valeur possible. La partie haute de la Figure 2 illustre graphiquement ce phénomène.

¹ Plus formellement, on peut montrer que le filtre particulaire souffre d'appauvrissement sur les paramètres constants quel que soit le modèle.

Proposition 1 (Appauvrissement). Considérons un filtre particulaire avec $N \in \mathbb{N}^*$ particules pour estimer un paramètre constant θ à valeur dans V . À chaque instant, on note $\Theta_t \in V^N$ l'estimation obtenue à l'instant $t \in \mathbb{N}$, par définition un ensemble de valeurs possibles pour θ . Il existe presque sûrement un instant $T \in \mathbb{N}$ tel que $\forall t > T, |\Theta_t| = 1$.

Démonstration. On note à chaque instant $t \in \mathbb{N}$, $\Theta_t = \{\theta_t^i\}_{1 \leq i \leq N}$ l'ensemble des valeurs possibles et $w(\theta_t^i) \in]0, \infty[$ le score associé à la valeur θ_t^i . À chaque filtrage, θ_t^i est dupliquée avec la probabilité $w(\theta_t^i)$ (le score normalisé de θ_t^i). Pour $t \in \mathbb{N}$, s'il reste au moins deux particules distinctes $|\Theta_t| \geq 2$, on note A la valeur de Θ_t associée au score minimal qui a donc le plus de chances d'être éliminée : $A = \operatorname{argmin}_{\substack{a \in \Theta_t \\ \theta_t^i = a}} w(\theta_t^i)$. On a donc $\forall 1 \leq i \leq N$, $\mathbb{P}(\theta_{t+1}^i = A) \leq \frac{1}{2}$ et :

$$\mathbb{P}(\exists a.a \in \Theta_t \text{ et } a \notin \Theta_{t+1}) \geq \mathbb{P}(A \notin \Theta_{t+1}) = \prod_{i=1}^N \mathbb{P}(\theta_{t+1}^i \neq A) \geq \frac{1}{2^N} = \epsilon > 0.$$

Comme ϵ ne dépend que de N , $\forall n \in \mathbb{N}$, on a : $\mathbb{P}(\Theta_{t+n} = \Theta_t) \leq (1 - \epsilon)^n$. Il existe donc presque sûrement $t' > t$ tel que $\Theta_{t'} \not\subseteq \Theta_t$. Par récurrence sur la taille de Θ_0 , il existe presque sûrement $T \in \mathbb{N}$ tel que $|\Theta_T| = 1$. \square

Données : le modèle `model`, l'entrée y_t et le résultat précédent μ_{t-1} .
Résultat : μ_t une approximation de la distribution de x_t (état) et θ (constant).

pour chaque particule $i = 1$ à N faire

```

     $x_{t-1}^i, \Theta_{t-1}^i = \text{sample}(\mu_{t-1})$ 
     $\theta^i = \text{sample}(\Theta_{t-1}^i)$ 
     $x_t^i, w_t^i = \text{model}(y_t | \theta^i, x_{t-1}^i)$ 
     $\Theta_t^i = \text{Update}(\Theta_{t-1}^i, \lambda\theta. \text{model}(y_t | \theta, x_{t-1}^i, x_t^i))$ 
 $\mu_t = \mathcal{M}(\{w_t^i, (x_t^i, \Theta_t^i)\}_{1 \leq i \leq N})$ 
retourner  $\mu_t$ 
```

Algorithme 2 : Assumed Parameter Filter [16]

2.3 Assumed Parameter Filter

Plutôt que d'échantillonner au début de l'exécution un ensemble de valeurs pour les paramètres constants qui s'appauvrira à chaque filtrage, dans l'algorithme *Assumed Parameter Filter* (APF) [16], chaque particule calcule une distribution symbolique de paramètres constants. Lors de l'exécution, l'inférence alterne ensuite entre une passe d'échantillonnage pour estimer les paramètres d'état, et une passe d'optimisation qui permet de mettre à jour les paramètres constants. On évite ainsi l'appauvrissement pour l'estimation des paramètres constants. La partie basse de la Figure 2 illustre les résultats de APF sur l'exemple du radar.

Plus formellement, l'Algorithme 2 décrit l'exécution d'un instant de APF. À chaque instant t , une particule $1 \leq i \leq N$ correspond maintenant à une valeur possible des paramètres d'état x_t^i et une distribution de paramètres constants Θ_t^i . Comme pour le filtre particulier, on commence par échantillonner un ensemble de particules dans la distribution obtenue à l'instant précédent. On échantillonne ensuite une valeur θ^i dans Θ_{t-1}^i . Sachant la valeur des paramètres constants θ^i et l'état précédent x_{t-1}^i , on peut exécuter un pas du modèle pour obtenir un échantillon des paramètres d'état x_t^i associé à un score w_t^i . Il reste alors à calculer Θ_t^i en explorant les autres valeurs possibles pour θ sachant que la particule a choisi la transition $x_{t-1}^i \rightarrow x_t^i$. À la fin de l'instant, comme pour le filtre particulier de l'Algorithme 1, on construit une distribution μ_t où chaque particule est associée à son score.

La difficulté principale est qu'à chaque instant, l'Algorithme 2 doit ré-exécuter plusieurs fois le modèle en fixant la valeur de certaines variables aléatoires. D'abord pour calculer les paramètres d'état sachant la valeur échantillonnée θ^i , puis pour mettre à jour la distribution Θ_t^i en explorant plusieurs valeurs possibles pour θ , sachant la transition $x_{t-1}^i \rightarrow x_t^i$.

Pour implémenter APF pour des modèles ProbZelus arbitraires, nous proposons donc une analyse statique pour identifier les paramètres constants et leur distributions *a priori* (Section 3) et une passe de compilation qui transforme ces paramètres en entrée supplémentaire du modèle (Section 4).

3 Analyse Statique

Dans cette section, nous présentons un noyau de ProbZelus. L'ensemble du langage, y compris les structures de contrôle de haut niveau comme les automates hiérarchiques, peut être compilé vers ce noyau [12]. Nous présentons ensuite l'analyse statique qui permet d'identifier les paramètres constants et leur distribution *a priori* sur ce noyau.

```

 $d ::= \text{let } x = e \mid \text{node } f x = e \mid \text{proba } f x = e \mid d \ d$ 
 $e ::= c \mid x \mid (e, e) \mid op(e) \mid f^\alpha(e) \mid \text{last } x$ 
 $\mid e \text{ where rec } \text{init } x = e \text{ and } \dots \text{ and } \text{init } x = e \text{ and } p = e \text{ and } \dots \text{ and } p = e$ 
 $\mid \text{present } e \rightarrow e \text{ else } e \mid \text{reset } e \text{ every } e \mid \text{sample}(e) \mid \text{observe}(e, e) \mid \text{infer}(f(e))$ 

```

FIGURE 3 – Syntaxe du noyau de ProbZelus

3.1 Un noyau de ProbZelus

La syntaxe du noyau de ProbZelus est présentée dans la Figure 3. Un programme est une série de déclarations d . Une déclaration peut donner un nom au résultat d'une expression (`let` $x = e$) ou définir un noeud déterministe (`node` $f x = e$) ou probabiliste (`proba` $f x = e$). Le nom de chaque déclaration est unique. Une expression peut être une constante (c), une variable (x), une paire ((e, e)), l'application d'une primitive ($op(e)$), un appel de noeud ($f^\alpha(e)$), un délai (`last` x), un ensemble d'équations locales mutuellement récursives (e `where rec` E), une structure de contrôle (`present` $e \rightarrow e$ `else` e), une structure de réinitialisation (`reset` e `every` e) ou une construction probabiliste (`sample`(e), `observe`(e, e) ou `infer`($f(e)$)).

On impose que toutes les variables initialisées soient définies par une équation, quitte à ajouter $x = \text{last } x$, et que les équations soient ordonnancées. Dans une expression de la forme e `where rec` (`init` $x_i = e_i$) $_{1 \leq i \leq k}$ `and` $(y_j = e'_j)$ $_{1 \leq j \leq n}$, on a donc $\{x_i\}_{1 \leq i \leq k} \subset \{y_j\}_{1 \leq j \leq n}$ et e'_j ne peut dépendre de $y_{j'}$ avec $j \leq j'$ qu'à travers l'opérateur `last`.

Ce noyau est proche de celui utilisé pour définir la sémantique formelle de ProbZelus [2]. La différence principale est qu'il est possible d'initialiser une variable avec une expression arbitraire là où le noyau original limitait la construction `init` à des constantes : `init` $x = c$. On autorise ainsi `init` $x = \text{sample}(d)$ `and` $x = \text{last } x$ pour définir des paramètres constants. Par ailleurs, on autorise la définition des variables globales dans les déclarations. Enfin, l'appel de noeud $f^\alpha(e)$ est ici annoté avec un nom d'instance α ajouté automatiquement par le compilateur qui permet de différencier les appels à une même fonction. Par exemple, l'expression $f(0) + f(1)$ devient dans ce noyau $f^{\alpha_1}(0) + f^{\alpha_2}(1)$.

Sémantique. La sémantique formelle de ProbZelus est définie dans [2]. Dans un environnement γ qui associe les noms de variables à leur valeur, les expressions déterministes sont des machines à états caractérisées par un état initial $\llbracket e \rrbracket_\gamma^{\text{init}}$ de type S et une fonction de transition $\llbracket e \rrbracket_\gamma^{\text{step}}$ de type $S \rightarrow O \times S$. La fonction de transition prend en entrée un état et renvoie une sortie et l'état suivant. On obtient un flot de données en itérant la fonction de transition depuis l'état initial.

Les expressions probabilistes sont elles aussi caractérisées par un état initial de type S , mais la fonction de transition $\llbracket e \rrbracket_\gamma^{\text{step}}$ de type $S \rightarrow \Sigma_{O \times S} \rightarrow [0, \infty)$ renvoie une mesure qui associe un score positif à chaque ensemble mesurable de paires (sortie, état suivant) $\in \Sigma_{O \times S}$.

3.2 Identifier les paramètres constants

L'analyse statique est définie par un jugement $\emptyset, \emptyset \vdash prog : H, C$ qui, pour un programme $prog$, construit un environnement H associant à chaque noeud probabiliste un type ξ qui identifie les paramètres constants du noeud et leur distribution *a priori* (l'environnement C contient les variables globales constantes). Pour un noeud `proba` $f x = e$, le type ξ est une fonction telle que chaque variable x dans $dom(\xi)$ est un paramètre constant dans e de distribution *a priori* $\xi(x)$.

$$\begin{array}{c}
\frac{}{C \vdash c : \emptyset} \quad \frac{}{C \vdash x : \emptyset} \quad \frac{C \vdash e_1 : \xi_1 \quad C \vdash e_2 : \xi_2}{C \vdash (e_1, e_2) : \xi_1 \cup \xi_2} \quad \frac{C \vdash e : \xi}{C \vdash op(e) : \xi} \\
\\
\frac{C \vdash e : \xi}{C \vdash f^\alpha(e) : \xi \cup \{\alpha \leftarrow f_prior\}} \quad \frac{}{C \vdash \text{last } x : \emptyset} \quad \frac{C \vdash e_1 : \xi}{C \vdash \text{present } e_1 \rightarrow e_2 \text{ else } e_3 : \xi} \\
\\
\frac{C \vdash e_2 : \xi}{C \vdash \text{reset } e_1 \text{ every } e_2 : \xi} \quad \frac{C \vdash e : \xi}{C \vdash \text{sample}(e) : \xi} \quad \frac{C \vdash e_1 : \xi_1 \quad C \vdash e_2 : \xi_2}{C \vdash \text{observe}(e_1, e_2) : \xi_1 \cup \xi_2} \\
\\
\frac{C \vdash e : \xi \quad \forall 1 \leq i \leq k, C \vdash e_i : \xi_i \quad \forall 1 \leq j \leq n, C \vdash e'_j : \xi'_j}{\xi' = \{x_i \leftarrow d_i \mid x_i = y_j \wedge e'_j = \text{last } y_j \wedge \vdash^p e_i : d_i \wedge C \vdash^l d_i\}} \\
\frac{}{C \vdash e \text{ where rec } (\text{init } x_i = e_i)_{1 \leq i \leq k} \text{ and } (y_j = e'_j)_{1 \leq j \leq n} : \xi \cup (\cup_{i=1}^k \xi_i) \cup (\cup_{j=1}^n \xi'_j) \cup \xi'} \\
\\
\frac{C \vdash e : \xi}{H, C \vdash \text{proba } f x = e : H \cup \{f \leftarrow \xi\}, C} \quad \frac{}{H, C \vdash \text{node } f x = e : H \cup \{f \leftarrow \emptyset\}, C} \\
\\
\frac{C \vdash^l e}{H, C \vdash \text{let } x = e : H, C \cup \{x\}} \quad \frac{H, C \vdash d_1 : H_1, C_1 \quad H_1, C_1 \vdash d_2 : H', C'}{H, C \vdash d_1 d_2 : H', C'}
\end{array}$$

FIGURE 4 – Identification des paramètres constants avec leurs distributions *a priori*.

$$\frac{}{\vdash^p \text{sample}(d) : d} \quad \frac{\vdash^p e : d}{\vdash^p e \text{ where rec } E : d \text{ where rec } E}$$

FIGURE 5 – Distribution échantillonnée.

Le jugement de typage des expressions est de la forme $C \vdash e : \xi$ et repose sur deux jugements auxiliaires : le jugement $\vdash^p e : d$ qui extrait la distribution d échantillonnée par e , et le jugement $C \vdash^l e$ qui garantit que l'expression e est constante et ne dépend que des variables constantes définies dans l'ensemble C .

Paramètres constants. Le jugement principal $C \vdash e : \xi$ est défini Figure 4. Pour simplifier l'analyse, on suppose que toutes les variables et les instances ont des noms différents. Une passe de renommage des variables permet facilement de satisfaire cette précondition. Toutes les règles parcourrent les sous-expressions pour collecter les paramètres constants. La règle pour $\text{reset } e_1 \text{ every } e_2$ parcourt uniquement e_2 car l'expression e_1 peut être réinitialisée et donc n'est pas constante. Pour une raison similaire, la règle pour $\text{present } e_1 \rightarrow e_2 \text{ else } e_3$ ne parcourt que e_1 . La règle pour $f^\alpha(e)$ associe dans ξ le nom d'instance α avec la distribution *a priori* de f que l'on suppose définie dans la variable f_prior (qui sera ajoutée par la compilation). La règle pour l'expression $e \text{ where rec } (\text{init } x_i = e_i)_{1 \leq i \leq k} \text{ and } (y_j = e'_j)_{1 \leq j \leq n}$ ajoute dans ξ les variables x_i qui respectent les conditions suivantes.

1. Elle n'est pas modifiée par la fonction de transition ($x_i = y_j \wedge e'_j = \text{last } y_j$).
2. Elle échantillonne une distribution d_i ($\vdash^p e_i : d_i$).
3. La distribution d_i est constante et ne dépend que des constantes dans C ($C \vdash^l d_i$).

$$\begin{array}{c}
\frac{}{C \vdash^l c} \quad \frac{x \in C}{C \vdash^l x} \quad \frac{C \vdash^l e_1 \quad C \vdash^l e_2}{C \vdash^l (e_1, e_2)} \quad \frac{C \vdash^l e}{C \vdash^l op(e)} \quad \frac{C \vdash^l e_1 \quad C \vdash^l e_2 \quad C \vdash^l e_3}{C \vdash^l \text{present } e_1 \rightarrow e_2 \text{ else } e_3} \\
\\
\frac{C \vdash^l e_1 \quad C \vdash^l e_2}{C \vdash^l \text{reset } e_1 \text{ every } e_2} \quad \frac{C \cup \{x_i\}_{1 \leq i \leq k} \vdash^l e \quad C \cup \{x_j\}_{1 \leq j < i} \vdash^l e_i}{C \vdash^l e \text{ where rec } (\text{init } x_i = e_i)_{1 \leq i \leq k} \text{ and } (x_j = \text{last } x_j)_{1 \leq j \leq k}}
\end{array}$$

FIGURE 6 – Expressions constantes.

L’analyse d’un nœud probabiliste f type le corps e du nœud et ajoute ce type dans l’environnement H associé au nom f . Les nœuds déterministes n’ont pas de variables aléatoires par définition. Les déclarations globales doivent être constantes, leurs noms sont ajoutés dans l’ensemble C des variables constantes.

Distributions. Le jugement $\vdash^p e : d$, défini Figure 5, garantit que l’expression e échantillonne une distribution d . La règle principale est donc celle de `sample(d)` qui renvoie la distribution d . La règle pour `where rec` autorise l’utilisation de variables locales pour définir la distribution échantillonnée.

Constantes. Le jugement $C \vdash^l e$, défini Figure 6, garantit que l’expression e est constante en vérifiant que toutes les sous-expressions sont constantes et ne dépendent que de variables constantes.

Exemple. Illustrons l’analyse statique sur l’exemple du nœud `move` de la Section 2. L’équation `theta = last theta` est ajoutée automatiquement par le compilateur.

```
proba move x0 = theta, x where
  rec init theta = sample(mv_gaussian(zeros, i2))
  and x = x0 → sample(mv_gaussian(pre x +@ theta, 0.5 *@ i2))
  and theta = last theta
```

La règle `proba` type le corps du nœud en utilisant la règle `where rec`. Le typage de chaque sous-expression renvoie un ensemble vide (aucun paramètre constant local n’est défini). Le système détecte que `theta` est un flot constant (`init theta = ... and theta = last theta`), donc l’expression `sample(mv_gaussian(zeros, i2))` est analysée avec le jugement \vdash^p qui renvoie la distribution `mv_gaussian(zeros, i2)`. Cette distribution est typée avec succès par le jugement \vdash^l dans l’environnement C qui contient les constantes `zeros` et `i2`. Ainsi le corps du nœud `move` a le type `{ theta ← mv_gaussian(zeros, i2) }` et le nœud a le type `{ move ← { theta ← mv_gaussian(zeros, i2) } }`.

4 Compilation

Pour exécuter l’algorithme APF présenté en Section 2.3, il faut maintenant compiler les modèles probabilistes pour que les paramètres constants deviennent une entrée supplémentaire. On pourra ainsi ré-exécuter le modèle en explorant différentes valeurs pour mettre à jour la distribution des paramètres constants. La compilation est définie Figure 7 par induction sur la syntaxe du noyau. On commence par typer le programme complet, c’est-à-dire une liste de déclarations $p = d_1 \dots d_n$ dans un environnement vide : $\emptyset, \emptyset \vdash p : H, C$. La fonction de compilation \mathcal{C} est ensuite paramétrée par l’environnement de typage H pour les déclarations, et le type ξ pour les expressions (voir Section 3).

$$\begin{aligned}
\mathcal{C}_\xi(c) &= c \\
\mathcal{C}_\xi(x) &= x \\
\mathcal{C}_\xi((e_1, e_2)) &= (\mathcal{C}_\xi(e_1), \mathcal{C}_\xi(e_2)) \\
\mathcal{C}_\xi(op(e)) &= op(\mathcal{C}_\xi(e)) \\
\mathcal{C}_\xi(\text{last } x) &= \text{last } x \\
\mathcal{C}_\xi(\text{present } e_1 \rightarrow e_2 \text{ else } e_3) &= \text{present } \mathcal{C}_\xi(e_1) \rightarrow \mathcal{C}_\xi(e_2) \text{ else } \mathcal{C}_\xi(e_3) \\
\mathcal{C}_\xi(\text{reset } e_1 \text{ every } e_2) &= \text{reset } \mathcal{C}_\xi(e_1) \text{ every } \mathcal{C}_\xi(e_2) \\
\mathcal{C}_\xi(\text{sample}(e)) &= \text{sample}(\mathcal{C}_\xi(e)) \\
\mathcal{C}_\xi(\text{observe}(e_1, e_2)) &= \text{observe}(\mathcal{C}_\xi(e_1), \mathcal{C}_\xi(e_2)) \\
\mathcal{C}_\xi \left(e \text{ where } \begin{array}{l} \text{rec } (\text{init } x_i = e_i)_{1 \leq i \leq k} \\ \text{and } (y_j = e'_j)_{1 \leq j \leq n} \end{array} \right) &= \mathcal{C}_\xi(e) \text{ where } \begin{array}{l} \text{rec } (\text{init } x_i = \mathcal{C}_\xi(e_i))_{1 \leq i \leq k, x_i \notin \text{dom}(\xi)} \\ \text{and } (y_j = \mathcal{C}_\xi(e'_j))_{1 \leq j \leq n, y_j \notin \text{dom}(\xi)} \end{array} \\
\mathcal{C}_\xi(f^\alpha(e)) &= \begin{cases} f(\mathcal{C}_\xi(e)) & \text{si } f \text{ est déterministe} \\ f_model(\alpha, \mathcal{C}_\xi(e)) & \text{si } \alpha \in \text{dom}(\xi) \\ f_model(\alpha, \mathcal{C}_\xi(e)) \text{ where} \\ \quad \text{rec init } \alpha = \text{sample}(f_prior) \\ \quad \text{and } \alpha = \text{last } \alpha & \text{sinon} \end{cases} \\
\mathcal{C}_\xi(\text{infer}(f(e))) &= APF.infer(f_prior, f_model, e) \\
\mathcal{C}_H(\text{let } x = e) &= \text{let } x = e \\
\mathcal{C}_H(\text{node } f x = e) &= \text{node } f x = \mathcal{C}_\emptyset(e) \\
\mathcal{C}_H(\text{proba } f x = e) &= \text{let } f_prior = img(\xi) \quad \text{avec } H(f) = \xi \\ &\quad \text{proba } f_model(\text{dom}(\xi), x) = \mathcal{C}_\xi(e)
\end{aligned}$$

FIGURE 7 – Fonction de compilation d'un programme ProbZelus pour APF. La fonction de compilation est paramétrée par le type ξ pour les expressions et l'environnement de typage H pour les déclarations (voir Section 3).

Modèles. Pour un modèle $\text{proba } f x = e$, si après l'analyse statique $H(f) = \xi$ (i.e., $C \vdash e : \xi$), alors f est compilé en deux déclarations :

- `let f_prior = img(ξ)` : la distribution *a priori* des paramètres constants dans e , et
- `proba f_model (dom(ξ), x) = $\mathcal{C}_\xi(e)$` : un nouveau modèle qui prend en argument supplémentaire $\text{dom}(\xi)$ pour les paramètres constants.

Expressions. La fonction de compilation d'une expression est paramétrée par le type ξ du noeud qui la contient. Cette fonction supprime les définitions de paramètres constants, $x \in \text{dom}(\xi)$, qui sont passés en entrée du nouveau modèle. Les premiers cas dans la Figure 7, de `c` à `observe`, implémentent une simple descente récursive sur les sous-expressions. Le cas `where/rec` supprime effectivement les paramètres constants en ne gardant que les équations définissant les variables x telles que $x \notin \text{dom}(\xi)$.

Exemple. Après compilation, le nœud `move` de l'exemple de la Section 2 devient :

```

let move_prior = mv_gaussian(zeros, i2)
proba move_model (theta, x0) = theta, x where
  rec x = x0 → sample(mv_gaussian(pre x +@ theta, 0.5 *@ i2))

```

Appel de noeud. La difficulté principale est de faire remonter les paramètres constants introduits par les appels de noeud. On distingue trois cas.

- Si le noeud est déterministe, il n'y a pas de paramètre constant possible.
- Si les paramètres constants de l'appel de noeud sont également des paramètres constants dans le contexte d'appel, on a $\alpha \in \text{dom}(\xi)$ et on se contente de remplacer l'appel à f^α par un appel à f_model en utilisant le nom frais d'instance pour les paramètres constants.
- Sinon, il existe des paramètres constants dans f qui ne sont pas constants dans le contexte d'appel car l'instance f^α est utilisée à l'intérieur d'un `reset/every` ou d'un `present/else`. Dans ce cas, on redéfinit localement ces paramètres en échantillonnant leur distribution *a priori* f_prior .

Par ailleurs, dans les noeuds déterministes, on substitue l'appel à `infer(f(e))` par un appel à `APF.infer(f_prior, f_model, e)`.

Exemple. Le noeud `radar` de la Section 2 devient :

```
let radar_prior = move_prior
proba radar_model (i_move_0, (delta, alpha)) = theta, x where
  rec theta, x = move_model (i_move_0, x0)
  and d = sqrt((vec_get x 0) ** 2. +. (vec_get x 1) ** 2.)
  and a = atan(vec_get x 1 /. vec_get x 0)
  and () = observe(gaussian(2. *. d /. c, delta_noise), delta)
  and () = observe(gaussian(a, alpha_noise), alpha)
```

La variable `i_move_0` est le nom frais associé à l'instance de `move` appelée dans `radar`. Il faut enfin substituer l'appel à `infer` par un appel à `APF.infer` dans le noeud main.

```
node main (delta, alpha) =
  let d = APF.infer(radar_prior, radar_model, (delta, alpha)) in
    let theta, x = split d in plot theta; plot x
```

Correction. La propriété de correction établit que l'inference appliquée sur un modèle et son compilé définissent les mêmes flots de distributions pour la sémantique idéale de ProbZelus.

Pour tout programme `prog` tel que $\emptyset, \emptyset \vdash prog : H, C$, pour tout modèle probabiliste m de `prog`,

$$\llbracket \text{infer}(m(obs)) \rrbracket \equiv \llbracket \text{APF.infer}(m_prior, m_model, obs) \rrbracket$$

En pratique, le calcul de la distribution *a posteriori* est un problème intractable, on utilise donc des méthodes approchées. Nous montrons ici que la compilation préserve la sémantique idéale du modèle, mais permet d'utiliser APF pour calculer une meilleure approximation de la distribution définie par le modèle.

Notations. Pour une mesure μ , on note $\bar{\mu}$ la distribution normalisée $\mu/\mu(\top)$ où \top représente l'ensemble du domaine de définition de μ . On note $f_*(\mu)$ la mesure image par une fonction mesurable f (par exemple, les mesures images par les fonctions de projection π_1 et π_2 permettent de séparer une distribution sur des paires en une paire de distributions). On note $\lambda U. \delta_v(U)$ la mesure de Dirac au point v ($\delta_v(U) = 1$ si $v \in U$ et 0 sinon). Enfin on note $\int \mu(dt)f(t)$ l'intégrale de la fonction f le long de la mesure μ où t est la variable d'intégration.

La sémantique de `APF.infer` est définie en appliquant `infer` après avoir échantilloné le paramètre constant α au début de l'exécution.

$$\begin{aligned} \llbracket \text{APF.infer}(m_prior, m_model, obs) \rrbracket &= \\ \llbracket \text{infer}(m_model(\alpha, obs) \text{ where } \text{rec init } \alpha = \text{sample}(m_prior) \text{ and } \alpha = \text{last } \alpha) \rrbracket \end{aligned}$$

Deux expressions probabilistes sont équivalentes si elles définissent les mêmes flots de distributions. La sémantique étant exprimée en terme d'état initial et de fonction de transition, l'équivalence s'exprime avec une relation de bisimilarité [22] initialisée sur les états initiaux et propagée à travers les fonctions de transition [9]. La bisimilarité sur les environnements γ_1 et γ_2 met en rapport les nœuds probabilistes avec leur version compilée et garantit que toutes les autres valeurs sont identiques.

Définition 1 (Bisimilarité d'environnements). Pour tout programme $prog$, pour tous γ_1, γ_2 , $\gamma_1 \equiv \gamma_2$ si pour toute variable $x \in \text{dom}(\gamma_1) \cap \text{dom}(\gamma_2)$, $\gamma_1(x) = \gamma_2(x)$ et pour tout nœud probabiliste $\text{proba } m \ x = e$ tel que $C \vdash e : \xi$ dans $prog$.

$$\left\{ \begin{array}{l} \gamma_1(m_{\text{init}}) = \llbracket e \rrbracket_{\gamma_1}^{\text{init}} \\ \gamma_1(m_{\text{step}}) = \lambda v. \llbracket e \rrbracket_{\gamma_1[x \leftarrow v]}^{\text{step}} \end{array} \right. \quad \text{et} \quad \left\{ \begin{array}{l} \gamma_2(m_{\text{model_init}}) = \llbracket \mathcal{C}_\xi(e) \rrbracket_{\gamma_2}^{\text{init}} \\ \gamma_2(m_{\text{model_step}}) = \lambda(\theta, v). \llbracket \mathcal{C}_\xi(e) \rrbracket_{\gamma_2[\text{dom}(\xi) \leftarrow \theta][x \leftarrow v]}^{\text{step}} \\ \gamma_2(m_{\text{prior}}) = \text{img}(\xi) \end{array} \right.$$

La bisimilarité sur les expressions probabilistes met en rapport les flots de mesures renvoyées à chaque instant par les fonctions de transition.

Définition 2 (Bisimilarité probabiliste). Soient e_1 et e_2 des expressions probabilistes, et une bisimulation $\mathcal{P} \subseteq S \text{ dist} \times S \text{ dist}$ entre les *distributions* d'états. $\llbracket e_1 \rrbracket \equiv_{\mathcal{P}} \llbracket e_2 \rrbracket$ lorsque :

— pour tout $\gamma_1 \equiv \gamma_2$, $(\delta_{\llbracket e_1 \rrbracket_{\gamma_1}^{\text{init}}}, \delta_{\llbracket e_2 \rrbracket_{\gamma_2}^{\text{init}}}) \in \mathcal{P}$ et

— pour tout $\gamma_1 \equiv \gamma_2$, si $(\sigma_1, \sigma_2) \in \mathcal{P}$

$$\left\{ \begin{array}{l} \mu_1 = \lambda U. \int \sigma_1(ds_1) \llbracket e_1 \rrbracket_{\gamma_1}^{\text{step}}(s_1)(U), o_1 = \pi_{1*}(\overline{\mu_1}), \sigma'_1 = \pi_{2*}(\overline{\mu_1}) \\ \mu_2 = \lambda U. \int \sigma_2(ds_2) \llbracket e_2 \rrbracket_{\gamma_2}^{\text{step}}(s_2)(U), o_2 = \pi_{1*}(\overline{\mu_2}), \sigma'_2 = \pi_{2*}(\overline{\mu_2}) \end{array} \right. \text{ alors } \left\{ \begin{array}{l} (\sigma'_1, \sigma'_2) \in \mathcal{P} \\ o_1 = o_2 \end{array} \right.$$

Nous prouvons la correction de la compilation dans un cadre simplifié.

Théorème 1. Pour tout programme $prog$ tel que $\emptyset, \emptyset \vdash prog : H, C$, pour tout modèle probabiliste m de $prog$ sous la forme $\text{proba } m \ x = e \text{ where rec init } \theta = \text{sample}(d) \text{ and } \theta = \text{last } \theta$, avec $H(m) = \{\theta \leftarrow d\}$, où e ne contient pas d'appel de nœud, alors $\llbracket \text{infer}(m(obs)) \rrbracket \equiv \llbracket \text{APF.infer}(m_{\text{prior}}, m_{\text{model}}, obs) \rrbracket$.

Les contraintes sur m ne sont pas excessives : e peut contenir des définitions locales, θ peut être un tuple de paramètres constants, et on peut inliner les appels de nœuds. De plus, on peut imaginer une analyse statique qui permet de vérifier que m est sous la bonne forme.

Démonstration. Vue la sémantique de `infer` et `APF.infer`, il suffit de montrer que : $\llbracket m(obs) \rrbracket \equiv_{\mathcal{P}} \llbracket m_{\text{model}}(\alpha, obs) \text{ where rec init } \alpha = \text{sample}(m_{\text{prior}}) \text{ and } \alpha = \text{last } \alpha \rrbracket$ pour une bisimulation \mathcal{P} . Comme $H(m) = \{\theta \leftarrow d\}$, alors $C \vdash e : \emptyset$ car le seul paramètre constant, θ , est défini à l'extérieur de e et donc $\mathcal{C}_{\emptyset}(e) = e$. On pose $\mathcal{P} \sigma_1 \sigma_2$ ssi il existe σ tel que $\sigma_1 = \lambda U. \int \sigma(dm, ds) \delta(((), (m, (), s))(U)$ et $\sigma_2 = \lambda U. \int \sigma(dm, ds) \delta(m, (((), ()), s))(U)$. On vérifie alors que \mathcal{P} relie les états initiaux et se propage à travers les fonctions de transition ce qui suppose de dérouler et faire commuter les intégrales imbriquées introduites par les environnements locaux. \square

5 Implémentation

Il reste maintenant à implémenter la construction `APF.infer`. Dans cette section, nous décrivons d'abord comment implémenter les méthodes de Monte Carlo séquentielles (SMC) pour les machines à états produites par la compilation d'un programme synchrone. Nous détaillons ensuite l'implémentation de *Assumed Parameter Filter* comme un cas particulier de SMC.

5.1 Méthodes de Monte Carlo séquentielles

Zelus compile les nœuds en machines à états. Une machine est une structure qui définit les méthodes suivantes : `alloc` alloue la mémoire nécessaire pour stocker l'état, `reset` réinitialise l'état en place, et `step` implémente la fonction de transition qui met à jour l'état en place. La mémoire est donc allouée statiquement avant l'exécution. Pour implémenter le filtrage, le compilateur génère également une méthode `copy` qui copie l'état d'une particule source dans celui d'une particule cible. En OCaml, on a ainsi² :

```
type ('alpha, 'beta) cnode = Cnode: { alloc : unit → 's;
                                         reset : 's → unit;
                                         step : 's → 'alpha → 'beta;
                                         copy : 's → 's → unit; } → ('alpha, 'beta) cnode
```

Les nœuds probabilistes sont également des machines à état mais la fonction `step` prend un argument supplémentaire `prob` qui contient les informations nécessaires à l'inférence :

```
type ('alpha, 'beta) pnode = (prob * 'alpha, 'beta) cnode
```

Les méthodes de Monte Carlo séquentielles utilisent un ensemble de simulations indépendantes, les *particules*, associées à un score pour approximer la distribution recherchée. En pratique, au cours d'une simulation les constructions probabilistes `sample` et `observe` utilisent l'argument supplémentaire `prob` pour mettre à jour le score de la particule.

```
type prob = {id : int; scores : float array}

let sample (prob, d) = draw d
let observe (prob, d, x) = prob.scores.(prob.id) ← prob.scores.(prob.id) +. logpdf d x
```

Le type `prob` contient un identifiant de particule `id`, et le tableau de scores `scores`. `sample` tire un échantillon aléatoire dans la distribution `d`, et `observe` incrémente le score de la particule avec la vraisemblance de l'observation $d_{\text{pdf}}(x)$. On calcule les scores en échelle logarithmique pour des raisons de stabilité numérique.

L'opérateur `infer` est un nœud d'ordre supérieur qui transforme une machine probabiliste de type (α, β) `pnode` en une machine déterministe de type $(\alpha, \beta \text{ dist})$ `cnode` qui calcule un flot de distributions. L'implémentation de `infer` avec une méthode de Monte Carlo séquentielle, approxime les distributions recherchées à l'aide d'un ensemble de particules indépendantes. L'implémentation suit de près l'Algorithme 1 présenté dans la Section 2.

```
type alpha infer_state = {mutable sigma : alpha dist}

let infer n particle =
  let (Cnode {alloc; reset; step; copy}) = particle in
  let i_alloc () = {sigma = Dist.dirac (alloc ())} in
  let i_reset state =
    let s = Dist.draw state.sigma in
    reset s; state.sigma ← Dist.dirac (s)
  in
  let i_step state obs =
    let particles = Array.init n (fun _ → let p = alloc () in
                                              let s = Dist.draw state.sigma in
                                              copy s p; p) in
    let scores = Array.make n 0. in
```

2. Voir le fichier `lib/std/ztype.ml` du compilateur Zelus <https://github.com/inria/zelus>.

```

let values = Array.mapi (fun i s → step s ({id = i; scores}, obs)) particles in
state.sigma ← multinomial particles scores;
multinomial values scores
in
let i_copy src dst = dst.sigma ← src.sigma in
Cnode {alloc = i_alloc; reset = i_reset; step = i_step; copy = i_copy}

```

L'opérateur `infer` prend en arguments un nombre de particules `n` et une machine probabiliste `particle`. L'état mémoire de `infer` est une distribution d'états possibles initialisé par la fonction `i_alloc` avec une distribution de Dirac sur l'état initial des particules. La fonction de transition `i_step` commence par échantillonner un tableau de particules dans la distribution d'états : c'est le filtrage. On itère ensuite la fonction de transition des particules `step` sur le tableau `particles` pour mettre à jour l'état de chaque particule et calculer les valeurs de sortie. On utilise un tableau `scores` pour stocker les scores obtenus. Pour chaque particule l'argument `prob` prend la valeur `{id = i; scores}` qui permet de lier la case `scores.(i)` à la particule `i`. On peut enfin mettre à jour l'état de `infer` avec une distribution multinomiale sur les particules, et renvoyer la distribution de sortie : une distribution multinomiale sur les valeurs.

Un filtre particulaire est un cas particulier simple de SMC où chaque particule se contente d'exécuter le modèle pour obtenir une valeur de sortie et un score.

```
let infer_pf n model = infer n model
```

Remarque. Cette implémentation suit de près la sémantique idéale de ProbZelus [2] mais la fonction de transition alloue de la mémoire à chaque itération. En pratique, le nombre de particules est fixé. On peut donc allouer statiquement toute la mémoire nécessaire dans l'état et utiliser uniquement des modifications en place dans la fonction de transition.

5.2 Assumed Parameter Filter

Assumed Parameter Filter est également un cas particulier de SMC qui utilise des particules plus avancées. La fonction de transition des particules suit l'Algorithme 2 décrit dans la Section 2.3. En plus de l'état et du score, chaque particule maintient une distribution de paramètres constants Θ^i qui est mise à jour à chaque instant.

Pour mettre à jour Θ^i , il faut être capable de mesurer la vraisemblance d'une exécution où seule la valeur des paramètres constants θ a changé : $\Theta_t^i = \text{Update}(\Theta_{t-1}^i, \lambda\theta. \text{model}(y_t | \theta, x_{t-1}^i, x_t^i))$ dans l'Algorithme 2. Il faut donc trouver un moyen de rejouer un pas d'exécution en fixant la valeur de tous les paramètres d'état.

Mémoriser les échantillons. On peut implémenter l'opérateur `sample` comme une machine à état et utiliser l'état mémoire pour mémoriser les valeurs échantillonées lors de la première exécution du modèle : $x_t^i, w_t^i = \text{model}(y_t | \theta^i, x_{t-1}^i)$ dans l'Algorithme 2. Malheureusement, l'exécution du modèle modifie l'état du programme en place. Si on ne fait pas attention, la fonction `Update` utilise donc $\lambda\theta. \text{model}(y_t | \theta, x_t^i, x_t^i)$ au lieu de $\lambda\theta. \text{model}(y_t | \theta, x_{t-1}^i, x_t^i)$.

Pour résoudre ce problème, nous proposons de stratifier la mémoire. Avant l'exécution du modèle, on copie superficiellement l'état qui contient des pointeurs vers l'état *profond* de chaque `sample`. Après l'exécution, on peut ainsi réutiliser cette copie de l'état où seul l'état des `sample` a été modifié. L'état de `sample` ne stocke donc pas directement la valeur échantillonnée, mais un pointeur vers une mémoire partagée. On distingue alors deux modes d'exécution gardés par une variable booléenne `replay`. Si `replay` est faux, l'appel à `sample(d)` tire un nouvel échantillon dans `d` et met à jour la mémoire partagée. Si `replay` est vrai, la valeur est déjà connue et l'appel à `sample(d)` se comporte comme `observe(d, m)` où `m` est le contenu de la mémoire partagée.

```

type prob = {id : int; scores : float array; replay : bool}

let sample =
  let alloc () = ref (ref None) in
  let reset state = !state := None in
  let step state (prob, dist) =
    match prob.replay with
    | false → let v = Dist.draw dist in !state := Some (v); v
    | true → let v = Option.get (!(!state)) in observe (prob, (dist, v)); v in
  let copy src dst = dst := !src in
  Cnode { alloc; reset; copy; step; }

```

La méthode `copy` implémente la copie superficielle qui ne duplique que les pointeurs en préservant l'état profond.

On peut maintenant définir une fonction `apf_particle` qui prend en argument le résultat de la compilation de la Section 4 et qui construit une particule qui implémente l'algorithme APF.

```

type ('alpha, 'beta) apf_state = {p_state: 'alpha ; mutable d_theta : 'beta dist}

let apf_particle m_model m_prior =
  let Cnode {alloc; reset; step; copy} = m_model in
  let p_alloc () = {p_state = alloc (); d_theta = m_prior} in
  let p_reset state = reset state.p_state; state.d_theta ← m_prior in
  let p_step ({p_state; d_theta} as state) (prob, obs) =
    let start_state = alloc () in copy p_state start_state;
    let theta = Dist.draw d_theta in
    let o = step p_state ({prob with replay = false}, (theta, obs)) in
    let tmp_state = alloc () in
    state.d_theta ← update
      (fun theta →
        copy start_state tmp_state;
        let _ = step tmp_state ({prob with replay = true}, (theta, obs)) in
        prob.scores.(prob.id))
      d_theta;
    o
  in
  let p_copy src dst = copy src.p_state dst.p_state; dst.d_theta ← src.d_theta in
  Cnode {alloc = p_alloc; reset = p_reset; step = p_step; copy = p_copy}

```

En plus de l'état du modèle, l'état de chaque particule contient une distribution de paramètres constants `d_theta` initialisée par la fonction `p_alloc` avec `m_prior`. La fonction de transition `p_step` d'une particule se décompose ensuite de la manière suivante :

1. Copier superficiellement l'état de départ `state` dans une variable `start_state`.
2. Échantillonner une valeur `theta` dans `d_theta`.
3. Exécuter la fonction de transition de `m_model`, `step` avec `replay = false` pour mettre à jour l'état et mémoriser les valeurs de tous les paramètres d'état.
4. Copier superficiellement l'état de départ `start_state` dans une variable `tmp_state`.
5. Mettre à jour la distribution `d_theta` en utilisant une fonction qui prend en argument `theta`, copie superficiellement l'état `start_state` dans une variable `tmp_state` et renvoie le score obtenu en re-exécutant `step` à partir de `tmp_state` avec `replay = true` pour rejouer l'instant courant en réutilisant les valeurs des paramètres d'état.
6. Renvoyer la sortie `o` calculée à l'étape 3.

Finalement, APF.infer est un cas particulier de SMC qui utilise apf_particle.

```
let APF.infer n m_prior m_model = infer n (apf_particle m_model m_prior)
```

Modularité. Notre implémentation ne fait pas d'hypothèse sur la fonction *Update* de l'Algorithm 2 qui peut donc implémenter diverses heuristiques. L'article original [16] se concentre sur la méthode de correspondance des moments. À chaque instant, on tire un ensemble d'échantillons aléatoires dans *d_theta* et on mesure les scores obtenus. *d_theta* devient alors la distribution Gaussienne qui approxime le mieux ces échantillons pondérés.

Nous avons également implémenté l'échantillonnage préférentiel, une méthode d'inférence sans filtrage. Au premier instant, on tire un ensemble de valeurs aléatoires pour les paramètres constants. Puis, à chaque instant, on met à jour le score associé à chacune de ces valeurs.

6 Évaluation

Dans cette section, nous validons expérimentalement notre implémentation d'APF sur un ensemble de modèles ProbZelus. Notre évaluation répond aux questions suivantes : (QR1) Quel est l'impact d'APF sur la précision des estimations à ressources fixées ? (QR2) APF résout-il le problème d'appauvrissement sur les estimations de paramètres constants ?

Méthode expérimentale. Pour chacun des exemples, on mesure l'évolution sur 500 pas de l'erreur quadratique moyenne (*Mean Square Error*, MSE) pour les paramètres d'état et les paramètres constants. Pour mesurer l'appauvrissement, on regarde également l'évolution de la taille effective d'échantillonnage (*Effective Sample Size*, ESS), une métrique qui indique le nombre d'échantillons utiles. Les expériences ont été réalisées avec un processeur Intel Core i5-6200U (2.30GHz) avec 8Go de mémoire vive.

On compare les performances d'APF avec deux autres méthodes d'inférence : l'échantillonnage préférentiel (*Importance Sampling*, IS), une méthode de Monte Carlo élémentaire sans filtrage, et le filtre particulaire (*Particle Filter*, PF), une méthode de Monte Carlo séquentielle avec filtrage à chaque instant. Pour chacune des méthodes d'inférence, on fixe les ressources à 10^4 particules. Sauf mention contraire, APF repartit les ressources disponibles avec 100 particules de filtrage et 100 particules d'échantillonnage pour le calcul de la correspondance des moments.

Pour comparer l'ESS d'APF avec celui des autres méthodes, on génère ensuite le même nombre d'échantillons en tirant une valeur aléatoire dans la distribution des paramètres constants pour chaque particule d'échantillonnage. Si n_e est le nombre de particule d'échantillonnage et n_f le nombre de particules de filtrage utilisés par APF, chaque particule génère donc n_e échantillons qui sont tous associés au score de la particule. On obtient donc à chaque instant $n_e * n_f$ échantillons, soit le nombre d'échantillons produit par les autres méthodes.

Nous avons sélectionné des modèles qui illustrent plusieurs types de dépendances entre paramètres d'état et paramètres constants. **Coin** infère le biais d'une pièce à partir d'une série de jets [2]. Ce modèle n'a pas de paramètre d'état. APF utilise 10^4 particules d'échantillonnage. **Kalman** estime la position d'un agent à partir d'observations bruitées [2]. Ce modèle n'a pas de paramètre constant. APF utilise 10^4 particules de filtrage. **Sin** est similaire à Kalman, mais le modèle de mouvement fait intervenir une relation non-linéaire entre un paramètre constant et la position. C'est le modèle utilisé pour motiver APF dans l'article original [16]. **Split** fusionne deux problèmes indépendants : l'inférence d'un paramètre constant à partir d'observations bruitées et un filtre de Kalman. **Radar** est le modèle de radar de la Section 2.

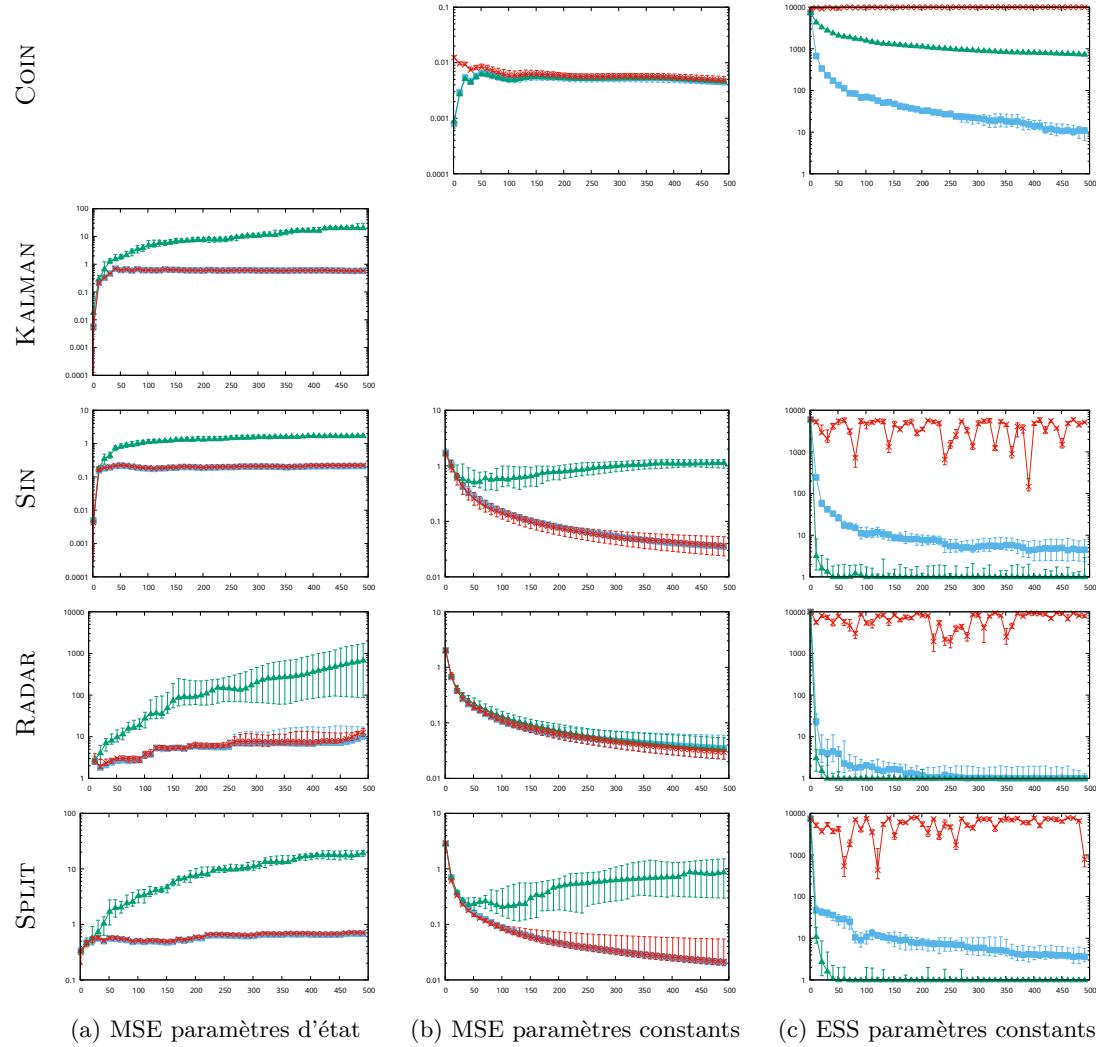


FIGURE 8 – ■ PF, ▲ IS, ✕ APF. Pour chacun des 5 modèles on mesure l'évolution sur 500 pas d'exécution du *mean square error* (MSE) pour les paramètres d'état et les paramètres constants, et le *effective sample size* (ESS) des paramètres constants. Les marqueurs indiquent les valeurs médianes et les barres d'erreur correspondent aux quantiles 90% et 10% sur 10 exécutions.

QR1 : Précision Les résultats de précision sont présentés Figures 8a et 8b. On remarque que, sur tous les exemples, la précision d'APF est équivalente à celle de PF pour l'estimation des paramètres d'état et des paramètres constants. Les courbes bleue et rouge sont quasiment identiques. On note cependant une plus grande variabilité dans les résultats d'APF sur les paramètres constants, notamment sur les derniers pas d'exécution. Ce comportement s'explique par le phénomène d'appauvrissement : après autant de filtrages successifs, l'estimation de PF pour les paramètres constants ne repose plus que sur une unique valeur et on a perdu toute mesure sur l'incertitude de l'estimation.

QR2 : Appauvrissement. Les résultats d'appauvrissement sont présentés Figure 8c. L'ESS pour APF reste quasiment constant autour du nombre d'échantillons total. Tous les échantillons générés ajoutent de l'information sur la distribution recherchée.

Sans surprise, on observe que l'ESS pour PF décroît strictement à chaque nouveau filtrage. L'ESS pour IS s'effondre lui aussi mais pour une raison différente. Aucune particule n'est éliminée, mais la plupart des valeurs tirées initialement deviennent tellement improbables que leur score tombe à 0. Ces particules n'ajoutent plus d'information pour le calcul de la distribution et ne sont donc plus comptées dans l'ESS.

Coût. Sur les trois exemples précédents où APF repartit les ressources entre filtrage et échantillonnage APF est en moyenne 1.43x plus lent que PF (Sin : 1.32x, Radar : 1.65x, Split : 1.33x). Pour les modèles avec un seul type de paramètre où APF utilise toutes les ressources soit en filtrage, soit en échantillonnage, APF est en moyenne 2.63x plus lent que PF. Ce ralentissement s'explique par le fait que APF re-exécute plusieurs fois une même itération du modèle pour mettre à jour la distribution des paramètres constants puis générer les échantillons, là où le filtre particulaire n'exécute le modèle qu'une fois par instant. Ces performances sont correctes si l'appauvrissement n'est pas acceptable et que l'on cherche à estimer précisément l'incertitude sur la valeur des paramètres constants.

7 Conclusion

Nous avons montré comment estimer les paramètres constants des modèles ProbZelus en évitant le problème d'appauvrissement grâce à l'algorithme d'inférence *Assumed Parameter Filter*. Notre implémentation repose sur une nouvelle analyse statique, une nouvelle passe de compilation, et un nouveau moteur d'inférence pour ProbZelus. Nos résultats expérimentaux montrent que cette méthode d'inférence permet d'éviter le problème d'appauvrissement pour les paramètres constants sans pénaliser la précision des estimations pour les paramètres d'état.

Travaux connexes L'analyse statique de la Section 3 est inspirée d'analyses spécifiques aux langages synchrones flot de données, en particulier l'analyse d'initialisation [11], qui vérifie que tous les flots sont bien définis au premier instant, et le typage des arguments statiques de Zelus, qui vérifie que certaines valeurs peuvent être calculées dès la compilation [7].

La compilation présentée en Section 4 se concentre uniquement sur les paramètres constants. C'est le moteur d'exécution qui se charge de fixer la valeur des paramètres d'état à l'aide d'une mémoire partagée pour rejouer un instant. Une alternative possible est de se rapprocher de la compilation des modèles hybrides de Zelus [4] et ajouter des entrées/sorties pour toutes les variables aléatoires introduites dans le modèle. On pourrait ainsi simplifier le moteur d'exécution au prix d'une compilation plus avancée.

La méthode d'inférence privilégiée de ProbZelus est *Delayed Sampling* [2], une méthode de Monte Carlo séquentielle qui calcule autant que possible des solutions analytiques exactes et n'échantillonne une variable aléatoire que lorsque le calcul symbolique échoue [21]. *Delayed Sampling* peut ainsi éviter l'appauvrissement si les paramètres constants n'ont jamais besoin d'être échantillonnés. Malheureusement, le moteur symbolique ne sait exploiter que des relations de conjugaison entre variables aléatoires, ou des relations affines [1]. On pourrait cependant combiner APF et *Delayed Sampling* pour améliorer la précision de ces deux méthodes.

Références

- [1] Eric Atkinson, Charles Yuan, Guillaume Baudart, Louis Mandel, and M. Carbin. Semi-symbolic inference for efficient streaming probabilistic programming. In *OOPSLA*, 2022.
- [2] Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. Reactive probabilistic programming. In *PLDI*. ACM, 2020.
- [3] Guillaume Baudart, Louis Mandel, and Reyyan Tekin. Jax based parallel inference for reactive probabilistic programming. In *LCTES*, San Diego, USA, June 2022.
- [4] Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. Divide and recycle : types and compilation for a hybrid synchronous language. In *LCTES*, April 2011.
- [5] 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.
- [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, Francois Carcenac, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. A synchronous look at the simulink standard library. *ACM Trans. Embed. Comput. Syst.*, 16(5s) :176 :1–176 :24, 2017.
- [8] Timothy Bourke and Marc Pouzet. Zélus : a synchronous language with ODEs. In *HSCC*, pages 113–118. ACM, 2013.
- [9] 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.
- [10] Nicolas Chopin and Omilos Papaspiliopoulos. *An introduction to sequential Monte Carlo*. Springer, 2020.
- [11] Jean-Louis Colaço and Marc Pouzet. Type-based Initialization Analysis of a Synchronous Data-flow Language. *International Journal on Software Tools for Technology Transfer*, 6(3) :245–255, 2004.
- [12] Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Mixing signals and modes in synchronous data-flow systems. In *EMSOFT*, 2006.
- [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] Yusuf Bugra Erol, Yi Wu, Lei Li, and Stuart Russell. A nearly-black-box online algorithm for joint parameter and state estimation in temporal models. In *AAAI*, 2017.
- [17] Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing : A language for flexible probabilistic inference. In *Proceedings of Machine Learning Research*, 2018.
- [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] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. Fastslam : A factored solution to the simultaneous localization and mapping problem. In *AAAI*, 2002.
- [21] Lawrence M. Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas B. Schön. Delayed sampling and automatic rao-blackwellization of probabilistic programs. In *AISTATS*, 2018.
- [22] David Michael Ritchie Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, 1981.