Filtrer sans s'appauvrir

Inférer les paramètres constants des modèles réactifs probabilistes

Guillaume Baudart, Grégoire Bussone, Louis Mandel, Christine Tasson
Synchronous Programming

Synchronous data-flow languages
- Inputs/Outputs: streams of values
- Functions: stream processors

\[ u = 1 \quad \text{radar} = 10 \]
\[ u = 1 \quad \text{radar} = 14 \]
\[ u = -1 \quad \text{radar} = 13 \]
\[ u = 1 \quad \text{radar} = 15 \]

\[ \text{tracker} \]
\[ t = 0 \quad x = 11 \]
\[ \text{tracker} \]
\[ t = 1 \quad x = 15 \]
\[ \text{tracker} \]
\[ t = 2 \quad x = 12 \]
\[ \text{tracker} \]
\[ t = 3 \quad x = 16 \]

boat

controller

radar

\[ \text{tracker} \]

\[ t = 0 \quad x = 11 \]
\[ \text{tracker} \]
\[ t = 1 \quad x = 15 \]
\[ \text{tracker} \]
\[ t = 2 \quad x = 12 \]
\[ \text{tracker} \]
\[ t = 3 \quad x = 16 \]
Synchronous Probabilistic Programming

Synchronous data-flow languages
- Inputs/Outputs: streams of values
- Functions: stream processors

ProbZelus: add support to deal with uncertainty
- Extend a synchronous language
- Parallel composition: deterministic/probabilistic
- Inference-in-the-loop
- Streaming inference
Synchronous Probabilistic Programming

\[ p(x_0|y_0) \]
\[ p(x_1|y_0, y_1) \]
\[ p(x_2|y_0, y_1, y_2) \]

\( t = 0 \)
\( t = 1 \)
\( t = 2 \)
Bayesian Inference

From prior to posterior distribution

\[ P(\text{params}|\text{data}) \propto P(\text{params})P(\text{data}|\text{params}) \]

Probabilistic constructs

- **sample** \(d\) (* Prior: sample from distribution \(d\) *)
- **observe** \((d, y)\) (* Condition: assume \(y\) was sampled from distribution \(d\) *)
- **infer** model \(y\) (* Infer: compute the posterior of model given \(y\) *)
Example: Boat

Use a radar to infer the position of a boat

- Inputs: $\alpha_t$ (angle) and $\delta_t$ (ping)
- Outputs: distribution over $x_t$ (position) and $\theta$ (speed)
Example: Boat

\[
\text{proba \ radar (delta, alpha) = theta, x where} \\
\text{rec init \ theta = \text{sample}(mv\_gaussian(zeros, i2))} \\
\text{and \ x = x0 \to \text{sample}(mv\_gaussian(pre x +@ theta, 0.5 *@ i2))} \\
\text{and \ d = sqrt((vec\_get x 0)**2. +. (vec\_get x 1)**2.)} \\
\text{and \ a = atan(vec\_get x 1/. vec\_get x 0)} \\
\text{and () = observe(gaussian(2. * d /. c, delta\_noise), delta)} \\
\text{and () = observe(gaussian(a, alpha\_noise), alpha)}
\]

\[
\text{node \ main (delta, alpha) = d\_theta, d\_x where} \\
\text{rec \ d = infer \ radar (delta, alpha)} \\
\text{and \ d\_theta, d\_x = Dist.split \ d}
\]
Approximate Inference: Basic Monte Carlo

Importance Sampling
- Run a set of $n$ independent executions, called particles
- Sample: draw a sample from a distribution
- Observe: associate a score to the current execution
- Gather output values and scores to approximate the posterior distribution

E.g., estimate the boat position from noisy measurements
Approximate Inference: Sequential Monte Carlo

Filtering: Duplicate most significant particles, kill least significant particles
- Recenter inference on the most significant estimations
- Filtering is required to infer moving parameters with bounded resources

E.g., estimate the boat position from noisy measurements
Particle Impoverishment

Filtering is problematic for fixed parameters. Information loss at each filtering step

E.g., estimate the boat drift (fixed parameter)
Assumed Parameter Filter

Each particle maintains a distribution of fixed parameters. At each step:

- **filter** the set of particles
- **sample** to compute moving parameters
- **update** the distribution of fixed parameters

Input: data $y_t$, previous particles distribution $\mu_{t-1}$
Output: current particles distribution $\mu_t$

For each particle $i = 1$ to $N$ do

$$
\begin{align*}
    x_{t-1}^i, \theta_{t-1}^i &= \text{sample}(\mu_{t-1}) \\
    \theta^i &= \text{sample}(\theta_{t-1}^i) \\
    x_t^i, w_t^i &= \text{model}(y_t | \theta^i, x_{t-1}^i) \\
    \theta_t^i &= \text{update}(\theta_{t-1}^i, \lambda \theta. \text{model}(y_t | \theta, x_t^i, x_{t-1}^i)) \\
    \mu_t &= \mathcal{M}(\{w_t^i, (x_t^i, \theta_t^i)\}_{i=1}^{N})
\end{align*}
$$
No more impoverishment for $\theta$
Problem: APF requires the ability to fix the value of some parameters at runtime

Input: data $y_t$, previous particles distribution $\mu_{t-1}$
Output: current particles distribution $\mu_t$

For each particle $i=1$ to $N$ do

1. $x_{t-1}^i, \theta_{t-1}^i = \text{sample}(\mu_{t-1})$
2. $\theta^i = \text{sample}(\theta_{t-1}^i)$
3. $x_t^i, w_t^i = \text{model}(y_t | \theta^i, x_{t-1}^i)$
4. $\theta_t^i = \text{update}(\theta_{t-1}^i, \lambda \theta_t \cdot \text{model}(y_t | \theta, x_t^i, x_{t-1}^i))$

$\mu_t = \mathcal{M}\{w_t^i, (x_t^i, \theta_t^i)\}_{1 \leq i \leq N}$

Sample $\theta$ outside the model
Replay a transition multiple times

Contributions: APF for ProbZelus

- Static analysis to identify fixed parameters
- Compilation pass required by APF
- New APF runtime for ProbZelus
Static Analysis and Compilation
Static Analysis

```plaintext
proba radar (delta, alpha) = theta, x where
  rec init theta = sample(mv_gaussian(zeros, i2))
  and x = x0 -> sample(mv_gaussian prez x +@ theta, 0.5 *@ i2))
  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)

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

Can we define a type system to identify fixed parameters and their priors?
```

\[ \theta \sim \mathcal{N}(0, I_2) \text{ fixed} \]
\[ x_{t+1} \sim \mathcal{N}(x_t \theta, 0.5I_2) \text{ moving} \]
Static Analysis

Extract prior distribution
\[ \vdash^p \text{sample}(e_d) : e_d \]

Only depends on globals
\[ C \vdash^l e_d \]

Global constants

Extract fixed parameters
\[ C \vdash e : \xi \]

Parameters map

\[ H, C \vdash d : H', C' \]

Global constants

Declaration

New parameters map

New global constants

Parameters map
proba radar (delta, alpha) = theta, x

rec init theta = sample(mv_gaussian(zeros, i2))
and x = x0 -> sample(mv_gaussian(pre x + @ theta, 0.5 * i2))
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)

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

{} ⊨ prog : {radar ← {theta ← N(0, I_2)}}}, {0, I_2}
let radar_prior = mv_gaussian(zeros, i2) (* Define priors *)

proba radar (delta, alpha) = theta, x where
proba radar_model (theta, (delta, alpha)) = theta, x where
  rec init theta = sample (mv_gaussian(zeros, i2))
  rec x = x0 -> sample (mv_gaussian (pre x +@ theta, 0.5 *@ i2))
  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)

node main (delta, alpha) = d_theta, d_x where
rec d = infer radar (delta, alpha)
rec d = APF.infer radar_prior radar_model (delta, alpha) (* Change inference *)
and d_theta, d_x = Dist.split d

\emptyset, \emptyset \vdash prog : \{\text{radar} \leftarrow \{\text{theta} \leftarrow \mathcal{N}(0, I_2)\}\}, \{0, I_2\}
Runtime
Zelus Compilation Pipeline

Program flow:
- zeluc
- Parser → Analyses → Rewrites → Scheduling → OBC

zls

Embedded code (OCaml)
Imperative updates to the state

Program files:
- program.zls
- program.byte
Synchronous State Machines

type \((\alpha, \beta)\) cnode = Cnode: {
  alloc : unit -> \sigma;
  reset : \sigma -> unit;
  step : \sigma -> \alpha -> \beta;
} -> (\alpha, \beta) cnode

\[
\begin{array}{c}
\text{S} \\
\rightarrow \\
\text{step} \\
\rightarrow \\
\text{S} \\
\end{array}
\]

\[
\begin{array}{c}
\alpha \\
\rightarrow \\
\text{step} \\
\rightarrow \\
\beta \\
\end{array}
\]
Probabilistic Synchronous State Machines

type $(\alpha, \beta) \text{cnode} = \text{Cnode:} \{\text{alloc : unit }\rightarrow \sigma;\}
\quad \text{reset : } \sigma \rightarrow \text{unit;}\n\quad \text{step : } \sigma \rightarrow \alpha \rightarrow \beta;\n\quad \text{copy : } \sigma \rightarrow \sigma \rightarrow \text{unit;}\n\} \rightarrow (\alpha, \beta) \text{cnode}

type $(\alpha, \beta) \text{pnode} = (\text{prob }\ast\alpha, \beta) \text{cnode}
Infer: Sequential Monte Carlo

```ocaml
val infer_smc : int -> (α, β) pnode -> (α, β dist) cnode

type σ infer_state = { mutable sigma : σ dist }

let infer_smc n (Cnode { alloc; reset; step; copy }) =
  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
    let values = Array.mapi
      (fun i s -> step s ({ id = i; scores }, obs)) particles
    in
    state.sigma <- multinomial particles scores;
    multinomial values scores
  in ...
```

23
Particle Filter

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

let infer_pf n model = infer_smc n model
Assumed Parameter Filter

Each particle maintains a distribution of fixed parameters. At each step:

- **filter** the set of particles
- **sample** to compute moving parameters
- **update** the distribution of fixed parameters

**Input:** data $y_t$, previous particles distribution $\mu_{t-1}$

**Output:** current particles distribution $\mu_t$

For each particle $i = 1$ to $N$ do

- $x_{t-1}^i, \theta_{t-1}^i = \text{sample}(\mu_{t-1})$
- $\theta^i = \text{sample}(\theta_{t-1}^i)$
- $x_{t}^{i'}, w_{t}^{i'} = \text{model}(y_t | \theta^i, x_{t-1}^i)$
- $\theta_{t}^i = \text{update}(\theta_{t-1}^i, \lambda \theta. \text{model}(y_t | \theta, x_{t}^{i'}, x_{t-1}^i))$

$\mu_t = M(\{w_{t}^{i'}, (x_{t}^{i'}, \theta_{t}^i)\}_{1 \leq i \leq N})$

We need to replay the same transition multiple times.

Problem: Each call to sample returns a different value.
Replay with sample as a Node

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 }
Assumed Parameter Filter

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

let apf_particle m_prior (Cnode \{ alloc; reset; step; copy \} as m_model) =
  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 (* Sampling *)
    let o = step p_state (\{ prob with replay = false \}, (theta, obs)) in (* Sampling *)
    let tmp_state = alloc () in (* Update *)
    state.d_theta <- update
      (fun theta ->
        copy start_state tmp_state;
        let prob = \{ id = 0; replay = true; scores = [\(|0.\|] \} in
        let _ = step tmp_state (prob, (theta, obs)) in
        prob.scores.(prob.id))
    d_theta;
  o
  in ...
Assumed Parameter Filter

```ocaml
let APF.infer n m_prior m_model =
  let particle = apf_particle m_prior m_model in
  infer_smc n particle

(* Sampling and update *)

(* Filtering *)
```
Evaluation
Evaluation

**Precision**
Metrics: Mean Square Error

**Impoverishment**
Metrics: Effective Sample Size

**Time cost:** between 1.3x and 3x slower
Takeaway

Assumed Parameter Filter (APF)
- Problem: particle impoverishment for fixed parameters
- APF: split moving and fixed parameters
- Requires a program transformation

Static Analysis and Compilation
- Identify fixed parameters in ProbZelus models
- Move fixed parameters as inputs of the model

Runtime
- Sequential Monte Carlo for probabilistic state machines
- Sample with replay using stratified memory states

Also in the paper
- Evaluation on existing models
- Simplified correctness proof of the compilation

https://github.com/rpl-lab/jfla23-apf