KI - Künstliche Intelligenz
https://doi.org/10.1007/s13218-019-00580-7

TECHNICAL CONTRIBUTION

On the Applicability of Probabilistic Programming Languages for Causal Activity Recognition

Stefan Lüdtke[1] · **Maximilian Popko**[1] · **Thomas Kirste**[1]

Received: 15 August 2018 / Accepted: 27 March 2019

Abstract
Recognizing causal activities of human protagonists, and jointly inferring context information like location of objects and agents from noisy sensor data is a challenging task. Causal models can be used, which describe the activity structure symbolically, e.g. by precondition-effect actions. Recently, probabilistic programming languages (PPLs) arose as an abstraction mechanism that allow to concisely define probabilistic models by a general-purpose programming language, and provide off-the-shelf, general-purpose inference algorithms. In this paper, we empirically investigate whether PPLs provide a feasible alternative for implementing causal models for human activity recognition, by comparing the performance of three different PPLs (Anglican, WebPPL and Figaro) on a multi-agent scenario. We find that PPLs allow to concisely express causal models, but general-purpose inference algorithms that are typically implemented in PPLs are outperformed by an application-specific inference algorithm by orders of magnitude. Still, PPLs can be a valuable tool for developing probabilistic models, due to their expressiveness and simple applicability.

Keywords Bayesian filtering · Causal model · Probabilistic programming language · Anglican · WebPPL · Figaro · Particle filter

1 Introduction

Recognizing human activities and inferring context information from sensor data is a challenging task of high relevance, e.g. for providing automatic assistance. Specifically, the objective we are concerned with in this paper is to reconstruct structured causal activity sequences (i.e. individual actions have preconditions and effects), as well as context information (the state of the environment, like location and state of objects). This setting is illustrated by the following example [12] (see Fig. 1):

Example 1 Multiple persons act in an office environment. They can move between locations, carry objects (like coffee or paper), and perform activities like brewing coffee or printing documents. The actions have a causal structure, e.g. to make a coffee, the coffee machine must have been filled with water and ground coffee. Each room is equipped with a presence sensor that signifies whether one or more persons are present at the corresponding room. The goal is to estimate the performed activities and environment state from the sensor observations. A more detailed description can be found in Sect. 3.1.

Causal symbolic models, like *Computational Causal Behaviour Models* [11], can elegantly solve this task. They model the system dynamics symbolically by a set of probabilistic precondition-effect actions, and perform Bayesian filtering using that symbolic description: That is, they repeatedly predict the posterior state distribution at time $t + 1$ from the prior at time t (by using the symbolic description, i.e. the *transition model*) and then update the estimate by incorporating observed sensor data (using another probabilistic model, the *observation model*).

Probabilistic Programming Languages (PPLs) arose as an abstraction mechanism, that allow to represent probabilistic models concisely in terms of a *general-purpose programming language* (like Javascript or Scala), and perform inference over probabilistic runs of such programs. An advantage of these approaches is that they separate the inference algorithm from the model specification, allowing

✉ Stefan Lüdtke
stefan.luedtke2@uni-rostock.de

[1] Institute of Computer Science, University of Rostock, Rostock, Germany

Published online: 02 April 2019

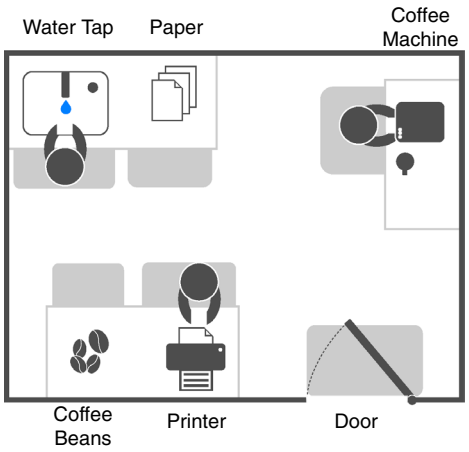

Fig. 1 Scenario used for evaluation of PPL inference. The environment consists of multiple locations, and multiple agents can move between locations, carry objects or perform activities like printing objects. Light grey rectangles denote floor pressure sensors

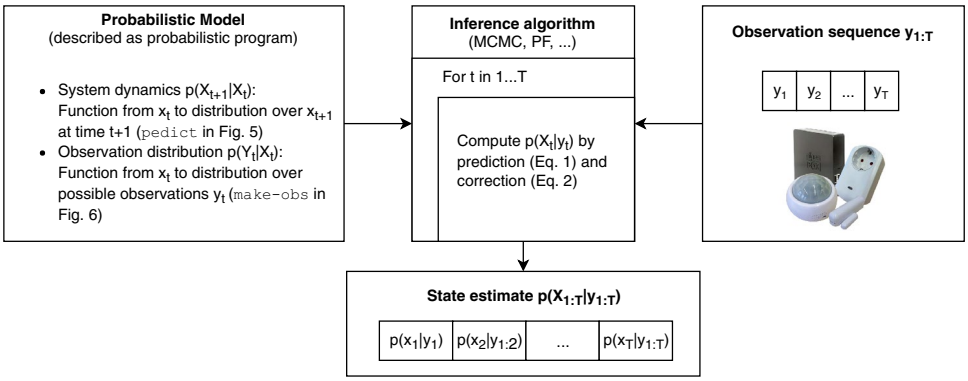

Fig. 2 Illustration of how PPLs can be used for recognition of causal activities. Probabilistic models of the system dynamics and the observation distribution are described in terms of a PPL program. Inference is performed on this program (given a sequence of observations) to estimate the distribution of state sequences explaining these observations. The parts are explained in more detail in Sect. 2

to re-use existing implementations of inference algorithms for different scenarios. PPLs have for example been used for scene perception [13] (e.g. generating a 3D model from a 2D image), or for skill rating systems in online multiplayer games [3]. In principle, PPLs should directly allow to specify causal probabilistic models, and perform inference in these models (see Fig. 2).

The contribution of this paper is to investigate whether PPLs are a suitable tool for this task. Surprisingly, there has been little work on experimental comparison of multiple PPLs. We do this by implementing a causal model (describing the scenario illustrated in Example 1) in three different PPLs (Anglican [24], WebPPL [9] and Figaro [20], see Sect. 2), and comparing their performance regarding runtime and convergence speed (Sect. 3). As a baseline, we use a custom inference algorithm [17] that is tailored towards categorical states and provides an upper bound on performance.

In this work, we only consider what has been called *expressive PPLs* [25], as they naturally allow to specify causal constraints, and they provide suitable inference algorithms. Other types of PPLs are discussed in more detail in Sect. 5.

We find that causal models can be expressed concisely in PPLs. However, the different available general-purpose inference algorithms vary strongly in their suitability for the problem, and thus their performance. Even the most suitable algorithm (particle filtering) is outperformed by the custom inference algorithm by orders of magnitude (Sect. 4). Thus, PPLs do not release the user from the burden of choosing appropriate inference algorithms. Furthermore, some problem domains can require the user to implement domain-specific algorithms. Still, PPLs can be a valuable tool for developing probabilistic models and research prototypes, due to their expressiveness and simple applicability.

2 Background

The general workflow for using PPLs for causal activity recognition is visualized in Fig. 2: The PPL inference algorithm works on (i) a probabilistic model (in our case, the transition model and observation model of a Bayesian filter), defined in terms of a probabilistic program, and (ii) evidence on the probabilistic model (in our case, the sensor data), to produce an estimate of the random variables of interest (the system state, in our case). In the following, we introduce the basic concepts underlying this approach in more detail: PPLs and probabilistic inference algorithms.

2.1 Probabilistic Programming Languages

The general idea of PPLs is to take a general-purpose (often functional) programming language and enhance it by probabilistic constructs: Methods to define distributions, sample from a distribution (in the following called elementary random procedure) and methods for conditioning distributions on observed values $\mathbf{y}$. A program that uses these constructs is then a generative probabilistic process: Each run of the

program can be seen as a sample of the distribution $p(\mathbf{x}|\mathbf{y})$, where $\mathbf{y}$ are the observed values, and $\mathbf{x}$ are the execution traces of the program, i.e. the values of all variables of the program. Such a program is not executed directly, but passed to an inference algorithm that uses some strategy to compute the distribution of program traces.

Example 2 We illustrate this concept with a simple generative probabilistic model, involving three binary random variables A, B and C. In PPLs, we specify the *process* that defines the distribution of these variables. Suppose that in our case, $p(A{=}1) = 0.5$. B is either sampled uniformly at random (when $A = 1$) or is always 0 (when $A = 0$), and a similar relationship holds for C and B. We are interested in the probability $p(A|C{=}0)$. The situation can be modeled in Anglican (one of the three PPLs investigated in this paper) as follows (Lisp keywords printed in blue, Anglican keywords printed in green):

```
1 (defquery example []
2   (let [A (sample (flip 0.5))
3         B (sample (flip (if A 0.5 0)))
4         f (flip (if B 0.5 0))
5         C (sample f)]
6   (observe f false)
7   A))
```

Note that in the example, we are only interested in the distribution of a single variable A, instead of the distribution of program traces $p(A, B, C)$. However, $p(A)$ can easily obtained from $p(A, B, C)$ by marginalization, and thus, in the following, we still consider the general case, where the distribution of program traces is of interest.

In the example, lines 2–5 define the generative model, line 7 specifies the distribution that we are interested in, and line 6 (the observe statement) specifies conditions on the distribution of interest. Directly executing lines 2–7 would result in a single sample of $p(A|C{=}0)$. In PPLs, however, such a program is not called directly, but passed to an inference algorithm, that uses a more elaborate strategy to compute the distribution of program traces.

For example, in Anglican, 1000 samples of $p(A|C{=}0)$ are produced via a specific inference algorithm (the Metropolis–Hastings algorithm) as follows:

```
(take 1000 (doquery :lmh example []))
```

This way, the model specification and inference algorithm are separated, which has the following advantages: (i) The user of a PPL can quickly develop probabilistic models, without being concerned with complex probabilistic inference algorithms, and (ii) the inference algorithm can be exchanged easily, without changing the model.

2.2 Inference Algorithms

In the following, we give a brief overview of different probabilistic inference algorithms that are typically implemented in PPLs. The goal of the inference algorithms is to estimate the distribution $p(\mathbf{x})$ of program traces $\mathbf{x}$. Typically, the distribution cannot be calculated exactly. Instead, a distribution $\tilde{p}(\mathbf{x})$ that *approximates* $p(\mathbf{x})$ is estimated.

2.2.1 Rejection Sampling

Rejection sampling is a very simple inference algorithm. It evaluates the program (i.e. simulates the process described by the program) multiple times. Whenever an observation operation occurs, the program sample is either accepted (when the current values of the sample match the evidence), or discarded (when they do not match). The set of all samples that are not discarded (i.e. that match the observations) form the approximate distribution $\tilde{p}(\mathbf{x})$.

Proceeding like this can be very inefficient: In cases where the evidence is very unlikely, most of the samples are discarded at some point, and thus the program must be run many times to obtain a given number of samples.

2.2.2 Markov Chain Monte Carlo

The idea of Markov Chain Monte Carlo (MCMC) algorithms is to construct a Markov Chain of samples $\mathcal{M} = \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots$ that has $p(\mathbf{x})$ as its stationary distribution, i.e. the sequence of samples represents the distribution $\tilde{p}(\mathbf{x})$.

In the following, we describe a specific MCMC algorithm, the Metropolis–Hastings (MH) algorithm [10], in more detail. The samples are produced iteratively by employing a proposal distribution $p(\mathbf{x}^{(i+1)}|\mathbf{x}^{(i)})$ that proposes a move to the next sample $\mathbf{x}^{(i+1)}$, given the current sample $\mathbf{x}^{(i)}$. The proposed sample is accepted with a probability that is chosen such that $p(\mathbf{x})$ really is the stationary distribution of the chain (see [10] for more details).

The reasoning is that by making the next sample depend on a previous (accepted) sample, inference can be more efficient than by rejection sampling: The previous sample already provides an idea on which variable assignments have a high probability, and based on that, it can be much easier to find a new sample with high probability, instead of needing to construct a sample "from scratch".

In the context of PPLs, each sample represents a run of the program, i.e. a program trace. The proposal function works by changing the value of a single elementary random procedure in the program—which then requires

to re-evaluate all random procedures that depend on the changed value—in the extreme case, *all* "later" random procedures. How the value of the random procedure is changed exactly depends on the implementation: It is always possible to re-evaluate the random procedure (i.e. draw a new sample), but often, better results can be obtained by performing *local* moves: For example, a sample drawn from a normal distribution can be shifted by a certain amount [8].

The MH algorithm is problematic when there are many dependencies in the random variables. In these cases, sampling from the proposal distribution can be difficult, to the point where it resorts to rejection sampling (as all random procedures that depend on a changed value must be re-evaluated). As an illustration, consider Example 2. Suppose that the value of A is changed. As B and C depend on A, it is necessary to re-evaluate the whole program, and thus the inference algorithm resorts to rejection sampling.

2.2.3 Particle Filtering

Particle filters (PF), or Sequential Monte Carlo (SMC) methods [6], are sampling-based approximate algorithms that implement the Bayesian filtering (BF) framework.

BF assumes that a sequence of observations $y_{1:T}$ is made, and each observation y_t depends on the (hidden) state $\mathbf{x}_t$ of the system. Given such a sequence $y_{1:T}$, BF iteratively computes the distribution of states $\mathbf{x}_t$ for each t by decomposing the computation in the following two steps: The *predict* step calculates the distribution after applying the *transition model* $p(\mathbf{x}_{t+1}|\mathbf{x}_t)$ (that models the system dynamics), i.e.

$$p(\mathbf{x}_{t+1}|y_{1:t}) = \sum_{\mathbf{x}_t} p(\mathbf{x}_t|y_{1:t})\, p(\mathbf{x}_{t+1}|\mathbf{x}_t). \tag{1}$$

The *update* step calculates the posterior distribution, by employing the *observation model* $p(y_{1:t+1}|\mathbf{x}_{t+1})$ (that models how observations depend on the state), i.e.

$$p(\mathbf{x}_{t+1}|y_{1:t+1}) = \frac{p(\mathbf{x}_{t+1}|y_{1:t})\, p(y_{t+1}|\mathbf{x}_{t+1})}{p(y_{t+1}|y_{1:t})}. \tag{2}$$

In the context of PPLs, the program trace (i.e. the sequence of variables drawn from the random procedures) can be seen as a state sequence $\mathbf{x}_{1:T}$. Each conditioning operation corresponds to an update step, and the sampling of random procedures occurring between observations correspond to the transition step. In a probabilistic program, there can be multiple random procedures between two conditioning operations. We can define the transition model as being simply the composition of the individual random procedures between two observations. More details are provided by [26].

The particle filter maintains a set of weighted samples (called *particles*), that represent the approximation of the true joint distribution of states up to time t, i.e. $\tilde{p}(\mathbf{x}_{1:t})$. The prediction (Eq. 1) is approximated as follows: For each particle $\mathbf{x}_t^i$, a sample from $p(\mathbf{x}_{t+1}|\mathbf{x}_t^i)$ is drawn, i.e. the particles are propagated through the transition model. Afterwards, each particle is reweighed by the observation model.

This method can lead to situations where the particle weights *degenerate*, or where particles vanish completely: For example, an observation that is very unlikely (or impossible) for a large portion of the particles lead to a situation where a small number of particles concentrate a large proportion of the total weight. *Resampling* is used to avoid this problem, making sure that particles with large weight are duplicated and particles with very low weight are removed (with a high probability). Thus, PF can avoid the problem of the MH algorithm in PPLs: Even when the random variables in the program are highly correlated, PF can efficiently generate samples from the distribution $p(\mathbf{x}_{1:T})$ of program traces.

2.2.4 Particle MCMC

Particle Markov chain Monte Carlo (PMCMC) methods [1] are a combination of MCMC methods like the MH algorithm and particle filtering. The PMCMC variant discussed in the following is particle Gibbs (PG). PG is a MH algorithm that uses particle filtering for the proposal: A new sample $p(\mathbf{x}_{1:T}^{(i+1)}|y_{1:T}, \mathbf{x}_{1:T}^{(i)})$ is proposed by running a particle filter, where the run of the particle filter is conditioned on the previous sample $\mathbf{x}_{1:T}^{(i)}$ (also resulting from a run of a PF): At each step of the particle filter, "retained" particle traces (from previous particle filter runs) are inserted into the particle set.

This way, as opposed to the standard MH proposal, it is feasible to sample from the proposal distribution, and the proposal can easily change more than one variable at a time. It has been shown [1] that this approach can achieve good results, even when only few particles are used in the underlying particle filter.

2.2.5 Marginal Filtering

The marginal filter (MF) [17] is an inference algorithm tailored towards inference in categorical domains, where it can outperform conventional inference algorithms like particle filtering.

As explained above, the particle filter approximates the joint distribution $p(\mathbf{x}_{1:t})$. The idea of the MF is to instead only maintain the *marginal* distribution $p(\mathbf{x}_t)$—which is the quantity we are eventually interested in. This is achieved as follows: Instead of sampling from the prediction distribution (Eq. 1), we exactly represent the prediction $p(\mathbf{x}_{t+1}|y_t)$ by computing $p(\mathbf{x}_{t+1}|\mathbf{x}_t^i)$ for all $\mathbf{x}_{t+1}$ and all particles $\mathbf{x}_t^i$. This is only possible when all random variables in $\mathbf{x}_{t+1}$ are categorical, as this allows us to enumerate the complete categorical

Fig. 3 Sensor data sequence for the office scenario. White: No person present, black: one or more persons present at the location

distribution. Identical successor states $\mathbf{x}_{t+1}$ are merged and their probability is summed. This way, the particles represent only the marginal distribution $p(\mathbf{x}_{t+1}|y_t)$ instead of the joint distribution.

Although this computation is exact, it is based on the approximation $\hat{p}(\mathbf{x}_t|y_t)$ from the previous time step. The only approximation for each time step is then to limit the number of particles that represent $p(\mathbf{x}_{t+1}|y_t)$ by an operation called *pruning*. Different pruning strategies have been devised. Here, we use the stratified resampling scheme proposed in [16], which guarantees unbiased pruning.

3 Experimental Evaluation

The overall goal of the experiments is to evaluate the suitability of PPLs to specify causal models for behavior and context (environmental state) recognition, and to investigate the performance of different inference algorithms for this task. In the following, we describe the experimental design in more detail.

3.1 Evaluation Scenario

We consider the office scenario described in Example 1.

The office environment consists of six distinct locations: door, printer, coffee machine, paper stack, water tap, and coffee jar. Pressure mats on the floor at every location signify whether at least one person is present at the corresponding location. The sensor data is always correct, i.e. no false negatives or false positives are produced (see Fig. 3). Also, there is a seventh unobserved location (outside of the office). The persons can perform one of the following activities:

- move from a location to another (all locations can be reached from one another, except the "outside" location is only connected to the "door" location)
- take an object (paper, ground coffee, water, coffee)
- refill an object (printer with paper, coffee machine with water or ground coffee)
- make coffee
- repair the printer
- print a document at the printer

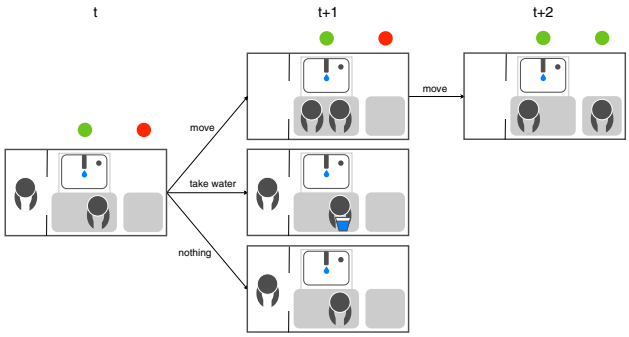

Fig. 4 Example of the state distribution estimated from sensor readings. A green/red circle denotes activity/inactivity of the respective sensor. At $t+1$, multiple states can explain the observation, but only one of them has a successor state at $t+2$ that is consistent with the observation

To perform any of these actions, specific preconditions have to be satisfied. For example, to refill ground coffee at the coffee machine, a person must be standing at the coffee machine, holding ground coffee. The task is to estimate which actions have been performed, as well as the state of the environment form a sequence of sensor data.

The challenge is that despite the fact that the sensor data is *correct*, it does not provide complete information about the system state (i.e. the number of agents at each location, the performed action, and handled objects). Instead, the action and state distribution must be reconstructed by incorporating knowledge about the causal structure. Figure 4 illustrates this on a simplified version of the scenario, consisting of three locations (two of which are observed by presence sensors), and three agents. At time t, we assume that one agent is at the sink, and one is outside. At time $t+1$, three successor states are consistent with the observation, corresponding to different actions. At time $t+2$, both sensors are active, which can only be explained by a single state that can be reached from the states at time t. In general, the number of possible state can quickly grow very large.

If the assumption that the sensor data is correct does not hold, only the observation model needs to be adapted, but not the model of system dynamics—here, we see one of the main benefits of generative probabilistic models. However, this would lead to an increased support of the state distribution, and thus more challenging inference. In the example, all three states at $t+1$ would have successor states, but those states that are inconsistent with the observation would have a different (lower) weight.

We simulated an observation sequence from the symbolic description of the scenario (assuming uniform sampling from the applicable actions at each time step), which is used for the empirical evaluation.

```
(defm predict [x_t]
  (let [statelist (concat
    ;not all actions, for illustration
    (put-weight (print x_t) 1)
    (put-weight (make-coffee x_t) 1)
    (put-weight (take-water x_t) 1))]
    (categorical statelist)))
```

Fig. 5 Definition of the `predict` function: Takes a state, returns a categorical distribution of states

```
(defquery abc [x_0 y_1T]
  (loop [x_t x_0
         y_tT y_1T]
    (let [y_t (first y_tT)
          px_t+1 (predict x_t)
          x_t+1 (sample px_t+1)]
      (observe (make-obs x_t+1) y_t)
      (if (= (count y_tT) 1)
        x_t+1
        (recur x_t+1 (rest y_tT))))))
```

Fig. 6 Anglican query definition. The implementation of `make-obs` (returning a categorical distribution) is not shown

3.2 Implementation

We implemented the scenario in three different PPLs. We considered recent PPLs that are actively developed and support SMC inference—other, non-sequential inference algorithms are unsuitable for our domain (as shown in Sect. 4). Specifically, the following PPLs have been selected:

- **Anglican** [24] is based on Clojure, a functional Lisp dialect running in the Java Virtual Machine. It supports rejection sampling, MCMC inference, and SMC inference. Furthermore, it is the only PPL considered by us that implements the recently proposed PMCMC inference.
- **WebPPL** [9] is based on Javascript, and can be executed locally via node.js or in a web browser. It implements rejection sampling, variational inference, MCMC inference, and SMC inference.
- **Figaro** [20] is based on Scala, a functional, object-oriented language. Figaro implements a large number of inference algorithms. In this paper only SMC inference was considered.

The implementations of the scenario in these PPLs, as well as the simulated observation sequence, is publicly available [22].

Furthermore, we use a custom implementation of the marginal filter [17], implemented in Haskell and compiled by the Glasgow Haskell Compiler (GHC). In this implementation, states $\mathbf{x}_t$ are maintained explicitly, as maps from state properties to values. This implementation is used for two purposes: (i) It is applied without the pruning step, thus calculating the exact state estimate, that is used as a ground truth to assess the convergence of PPL inference, and (ii) it is used as a baseline for assessing the PPL performance.

Figures 5 and 6 provide an intuition on how the scenario has been implemented, by showing the central part of the Anglican implementation. The function `predict` calculates the distribution $p(\mathbf{X}_{t+1}|\mathbf{x}_t)$ for a specific state $\mathbf{x}_t$ and returns it as a categorical distribution. This function actually encodes the causal model, i.e. it checks which actions can be executed, and applies the effects to the posterior states. The function `make-obs` (not shown) computes the categorical distribution $p(Y_{t+1}|\mathbf{x}_{t+1})$ for a specific state $\mathbf{x}_{t+1}$.

The query repeats the following steps for each observation y_{t+1}: Obtain a posterior state distribution $p(\mathbf{X}_{t+1}|\mathbf{x}_t)$ via `predict`; sample a state $\mathbf{x}_{t+1}$ from this distribution; obtain the distribution $p(Y_{t+1}|\mathbf{x}_{t+1})$ via `make-obs`; weigh the probability of the sample $\mathbf{x}_{t+1}$ with $p(Y_{t+1}{=}y_{t+1}|\mathbf{x}_{t+1})$, where y_{t+1} is the actual observation at time $t+1$. Figure 2 shows how this model and query definition relates to the overall inference process: Different inference algorithms can be used to perform inference for this query and a given sequence of sensor observations, to estimate an approximation $\tilde{p}(\mathbf{X}_T|y_{1:T})$ of the state distribution.

Note that in this paper, we are not concerned with *learning* the probabilistic model from data. Instead, the correct probabilities (that were used for calculating the exact distribution, see below) are encoded in each probabilistic program. This means that all inference algorithms will eventually converge to the exact state distribution (although they can have different convergence speed).

3.3 Experimental Design

To evaluate the PPLs, we use a factorial design. Factors and levels are shown in Table 1. For each PPL, different inference algorithms are considered (as different inference algorithms are provided by the different PPLs). The approximate marginal state distribution at time t, $\tilde{p}(\mathbf{X}_t|y_{1:t})$, is computed for each $t = 1, \ldots, 26$ and different numbers of samples of each of the inference algorithms. Furthermore, the number of persons that are present in the scenario is varied, which varies the size of the state space. The reasoning here is that state space size influences inference performance – some algorithms might show their strength only for very large state spaces (e.g. because they have a large overhead), while others might work better for a small state space. Overall, we perform 11,700 experiments (one for each viable combination of factor levels).

Table 1 Factors and levels of experimental design

Factor	Levels	Description
Language	Anglican, WebPPL, Figaro	
Algorithm	MH (Anglican, WebPPL), PF (Anglican, WebPPL, Figaro), PIMH (Anglican), PGIBBS (Anglican), MF (custom Haskell-based implementation)	PPLs in bracktets denote which PPL supports each of the inference algorithms
Samples	1000, …, 10000 (Anglican, WebPPL) 1000, …, 30000 (Figaro)	Number of samples/particles drawn by the inference algorithm
Agents	3, 5, 8, 13, 21	Number of persons that move around the environment in the simulated scenario
Timesteps	1, …, 26	Length of the observation sequence, i.e. number of predict and update steps that are performed

3.4 Performance Measures

To assess the convergence of the inference algorithms, we use the *Jensen-Shannon divergence* (JSD). Let p_t be the exact marginal state distribution at time t, and let $\tilde{p}_t^n$ be the distribution estimated by the inference algorithm by drawing n samples. The JSD at time t when drawing n samples, JSD_t^n, is then

$$\mathrm{JSD}_t^n(p_t \parallel \tilde{p}_t^n) = \frac{1}{2} D(p_t \parallel m_t^n) + \frac{1}{2} D(\tilde{p}_t^n \parallel m_t^n))$$

where $m_t^n = \frac{1}{2}(p_t + \tilde{p}_t^n)$ and D is the Kullback-Leibler divergence

$$D(p \parallel q) = -\sum_x p(x) \log \frac{q(x)}{p(x)}.$$

The JSD measures the distance between the true distribution p_t and the estimated distribution $\tilde{p}_t^n$, and is 0 when $p_t = \tilde{p}_t^n$. The *mean JSD* over time is calculated as

$$\mathrm{JSD}^n = 1/T \sum_t \mathrm{JSD}_t^n.$$

We also assess the *mixing time* $\tau(\epsilon)$, that measures how many samples need to be drawn until the mean JSD falls below a threshold ϵ:

$$\tau(\epsilon) = \min\{m \mid \mathrm{JSD}^n \leq \epsilon \text{ for all } n \geq m\}$$

The mixing time is an indicator of the number of samples that need to be drawn (and thus also of the runtime of the algorithm) until a "good" approximation is achieved. Furthermore, we measure the runtime of each experiment (as different PPLs or inference algorithms might need a different time to draw a given number of samples). All experiments were performed on a PC with 4 ∗ 2.6 GHz CPUs and 16 GB of RAM.

4 Results

We find that the causal model describing the application domain can be concisely expressed in all three PPLs (less than 300 lines of code in each language, including specification of inference methods, in/output and comments).

Figure 7 shows the runtime of the MH and PF inference algorithms of Anglican and WebPPL, in relation to the length of the observation sequence that is processed (the number of timesteps). The runtime of the MH algorithm increases exponentially after the first few timesteps. This is not surprising, given the considerations about the working of the MH proposal function above: The MH algorithm needs to draw exponentially many samples to get a fixed number of samples with non-zero probability. The MH algorithm is infeasible for inference with more than 7 timesteps, and thus not suited for inference in this domain—at least, using the off-the-shelf proposal function used in the PPLs. Therefore, in the following, we limit the evaluation to the other inference algorithms.

Figures 8 and 9 show the evaluation results for the other inference algorithms (for a fixed complexity of the scenario).

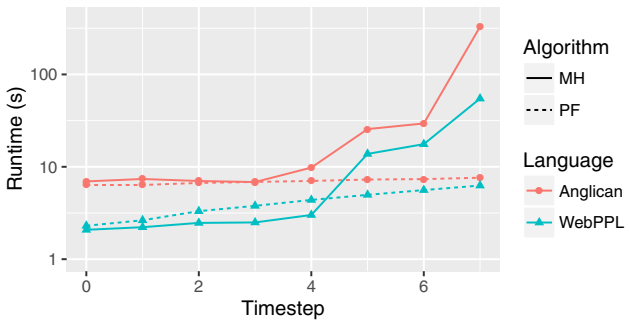

Fig. 7 Runtime of MH and PF inference, relative to the number of timesteps that are processed. Note the logarithmic y axis. The MH algorithm was infeasible for timesteps > 7. Other parameters are set to: 5 agents, 1000 samples

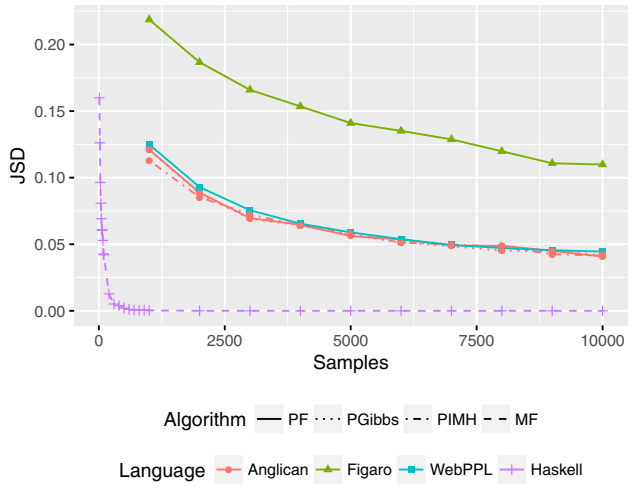

Fig. 8 Mean JSD of the different inference algorithms, for different number of samples and eight agents

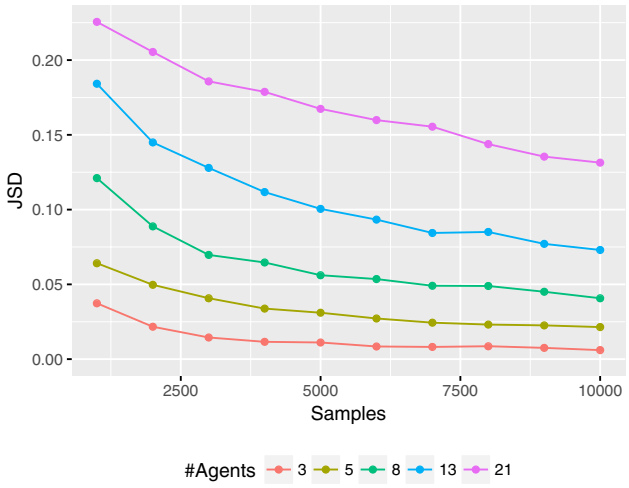

Fig. 10 Mean JSD the Anglican PF for different scenario complexities

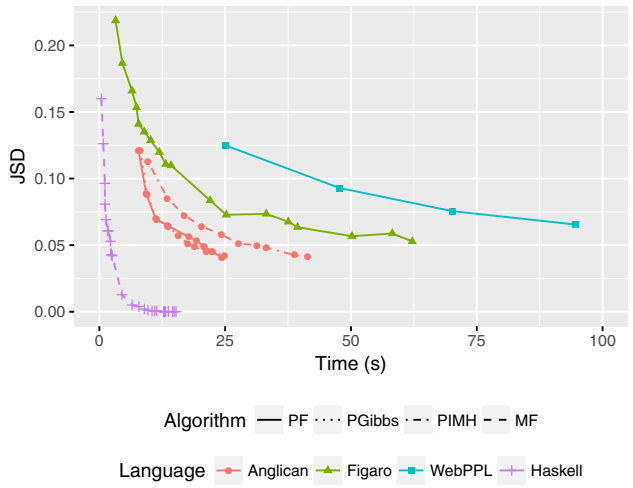

Fig. 9 Mean JSD of the different inference algorithms, in comparison to algorithm runtime (for eight agents)

In Figure 8, the mean JSD of the algorithms in relation to the number of particles is shown. Naturally, the JSD is lower the more particles are available, and in all cases converges towards 0. Interestingly, we see the same JSD trajectory for all inference algorithms of Anglican and WebPPL, and we see that Figaro's particle filter has a higher JSD for the same number of particles. This indicates that PMCMC algorithms (like PGibbs and PIMH) do not have an advantage over particle filtering in our domain—they basically reduce to particle filtering. This is due to the fact that the estimated distribution is *categorical*, and thus no "local" moves can be made.

The inferior performance of Figaro's particle filter can be explained, for example, by a different resampling strategy that is used internally. Note that still, the estimated state distribution eventually converges to the true distribution, i.e. our implementation of the scenario in Figaro is correct.

The MF converges much faster than the other inference algorithms—thus confirming that the MF is better suited to inference in categorical domains than the other inference algorithms. Furthermore, it is the only algorithm that can perform exact inference (i.e. achieve JSD = 0), given sufficiently many particles. For eight agents, the posterior estimate is exact when 2000 particles are available.

Figure 9 shows the JSD in relationship to the *runtime* of the algorithms (instead of the number of samples), to investigate the efficiency of the different PPL implementations. Figaro is faster than the other PPL-based algorithms in producing a given number of samples, i.e. it is implemented more efficiently. However, due to the poorer approximation quality for a given number of samples, it is still inferior to Anglican for a given runtime. The MF is again superior to the other inference algorithms, regarding the convergence with respect to algorithm runtime: Given a specific runtime, the MF achieves a more accurate estimation of the true posterior than the PPL-based algorithms.

Next, we investigate the effect that the scenario complexity (in terms of states space size due to varying the number of agents) has on algorithm performance. Figure 10 illustrates the different convergence speeds for the Anglican-based PF. Naturally, a larger state space leads to slower convergence (i.e. a given number of samples relates to a lower approximation quality). In Fig. 11, this effect is illustrated for all inference algorithms (for a fixed number of 10,000 samples). We observe that for all inference algorithms, the JSD grows linearly with the number of agents.[1] As seen

[1] Interestingly, the state space size relates exponentially to the number of agents (due to the exponential number of agent permutations).

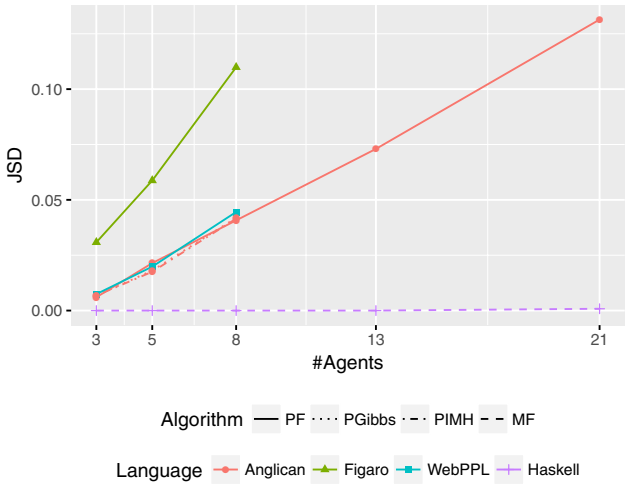

Fig. 11 Mean JSD of the different inference algorithms with respect to scenario complexity (using 10,000 samples)

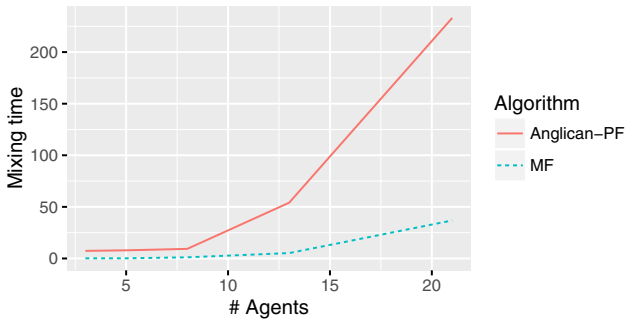

Fig. 12 Empirical mixing time for $\epsilon = 0.1$ of a PPL-based inference algorithm (particle filtering implemented in Anglican) and marginal filtering. The mixing time of the PPL-based algorithm is always higher and grows faster

before, the algorithms of Anglican and WebPPL behave identical (for identical number of samples), and Figaro has a larger JSD for a given number of samples. Furthermore, the marginal filter is able to perform exact inference with 10,000 samples for up to 13 agents (i.e. $JSD = 0$), and has a very low JSD (8e−4) for 21 agents.

Finally, we compare the mixing time of the best-performing PPL inference algorithm (particle filtering implemented in Anglican) and the MF (see Fig. 12). The empirical mixing time for $\epsilon = 0.1$ indicates that the MF will converge faster, regardless of the domain size of the problem. We can even see that the performance difference of the algorithms becomes larger for an increased domain size.

5 Related Work

In this paper, we considered a specific type of PPLs, that has been called *expressive* PPLs [25]. These languages define generative probabilistic models via general-purpose (and, in many cases, functional) programming languages, i.e. model-definition is very flexible, but inference in the general case is difficult. Other languages that fall in this category are Church [8], Probabilistic C [18] and PyMC [19]. Inference in these languages is typically done by MCMC-based or SMC-based algorithms. Another type of inference paradigm supported by some expressive PPLs is *variational inference*. In variational inference, a complicated distribution (e.g. the posterior distribution of the model) is approximated by a simpler distribution, by making assumptions about the parametric form of the distribution, or by making independence assumptions. As for MCMC inference, this is typically only useful for metrical domains. An example of a PPL that relies on this approach is the recently proposed [2], that uses deep neural networks to model the variational distribution.

A different class of PPLs is *logic-based*. They add probabilistic annotations to facts of a probabilistic program. Examples are Problog [7] and Prism [23]. Inference algorithms for logic-based PPLs is based on knowledge compilation (where the inference task is compiled into a weighted model counting problem) or MC-SAT [21] (an MCMC algorithm that calls a SAT solver in the proposal step). Logic-based PPLs are not considered here, as they do not consider prior distributions [25]. This makes them unsuitable for Bayesian filtering tasks.

There are other approaches like Stan [4], Bugs [14] or Factorie [15] that are related to probabilistic programming, but are not turing-complete, general-purpose programming languages themselves: They allow to specify probabilistic graphical models by a domain-specific modeling language, and provide inference algorithms for these models. They do not allow typical programming constructs, like loops or branches—which are necessary in the structured domains we are considering, where actions can have certain preconditions. Markov Logic Networks (MLNs) fall in an intermediate position between these approaches, and logic-based PPLs: They are a first-order logic formalism that is annotated with probabilities – much like logic-based PPLs, but without being turing-complete. For MLNs, tractable inference algorithms that work (partially) in the first-order domain (known as Lifted Inference [5]) can be used in some cases.

An overview of the different types of PPLs, and their advantages and disadvantages for causal activity recognition, is given in Table 2.

Footnote 1 (continued)

Thus, when the scenario complexity is increased exponentially, the estimation error (in terms of JSD) grows only linearly.

Table 2 Overview of different types of PPLs, with advantages and disadvantages for the application domains considered in this work

Type	PPLs	Inference (typical)	Advantages for AR	Disadvantages for AR
Expressive PPLs	Anglican, WebPPL, Figaro, Church, Probabilistic C, PyMC, Pyro, IBAL	MCMC, SMC, PMCMC, Variational inference	Expressiveness, suitable inference algorithms for dynamic systems	Low inference performance (for categorical RVs)
Logic-based PPLs	Problog, Prism, cplint, CLP(BN)	Knowledge compilation, MC-SAT	High inference performance in categorical domains	No continuous RVs, no prior distributions
Graphical model specification	Stan, Bugs, Factorie	MCMC, Belief Propagation	High inference performance	Not Turing-complete

6 Conclusion

In this paper, we investigated the usefulness of probabilistic programming languages (PPLs) for the recognition of causal, structured human activities. We found that expressive PPLs allow to concisely specify causal models. However, some inference algorithms, like MCMC inference, are completely unsuitable to the problem. Even the more suitable algorithms (particle filtering) are outperformed by a domain-specific algorithm. The main reason for this is that the scenario is modeled by categorical states (as usual in causal models), and inference algorithms like particle filtering rely on a metrical structure of the random variables. These results do not only apply to activity and context recognition, but to inferences in categorical states spaces in general. Therefore, users of PPLs must be aware of the different capabilities of the inference algorithms, to be able to select an appropriate algorithm. Providing domain-specific inference algorithms that allow efficient inference in more application domains can be seen as a goal for the further development of PPLs.

Still, given that PPLs allow to model causal models very conveniently, they are a viable alternative to handcrafted inference algorithms. Using PPLs basically trades off a (potentially) increased inference time by faster model development. PPLs are thus well suited for developing prototypical probabilistic inference systems, as at early stages of development, inference performance is probably not of critical importance. Aspects not covered in this paper are the possibility to learn parameters from data (e.g. by expectation maximization), inference in models with mixed discrete and categorical variables (which arise when processing data of, for example, accelerometers and gyroscopes), and exploring PPLs for a real-world activity recognition scenario.

Acknowledgements We are grateful to the anonymous reviewers for their comments and suggestions, which vastly improved the presentation and discussion of our work.

References

1. Andrieu C, Doucet A, Holenstein R (2010) Particle markov chain monte carlo methods. J R Stat Soc Ser B (Stat Methodol) 72(3):269–342
2. Bingham E, Chen J.P, Jankowiak M, Obermeyer F, Pradhan N, Karaletsos T, Singh R, Szerlip P, Horsfall P, Goodman N.D (2018) Pyro: Deep Universal Probabilistic Programming. arXiv preprint arXiv:1810.09538
3. Borgström J, Gordon A.D, Greenberg M, Margetson J, Gael J.V (2011) Measure transformer semantics for Bayesian machine learning. In: European symposium on programming, pp. 77–96. Springer, Berlin. https://doi.org/10.1007/978-3-642-19718-5_5
4. Carpenter B, Gelman A, Hoffman M.D, Lee D, Goodrich B, Betancourt M, Brubaker M, Guo J, Li P, Riddell A (2017) Stan: a probabilistic programming language. J Stat Softw. https://doi.org/10.18637/jss.v076.i01
5. De Raedt L, Kersting K, Natarajan S, Poole D (2016) Statistical relational artificial intelligence: logic, probability, and computation. Synth Lect Artif Intell Mach Learn 10:1–189
6. Doucet A, de Freitas N, Gordon N (2001) Sequential Monte Carlo Methods in Practice. Springer, Berlin
7. Fierens D, Van den Broeck G, Renkens J, Shterionov D, Gutmann B, Thon I, Janssens G, De Raedt L (2015) Inference and learning in probabilistic logic programs using weighted boolean formulas. Theory Pract Logic Programm 15(3):358–401
8. Goodman N, Mansinghka V, Roy D, Bonawitz K, Tenenbaum J (2008) Church: a language for generative models. In: Proceedings of the conference on uncertainty in artificial intelligence
9. Goodman N.D, Stuhlmüller A (2014) The design and implementation of probabilistic programming languages. http://dippl.org. Accessed 29 Mar 2018
10. Häggström O (2002) Finite Markov chains and algorithmic applications, vol 52. Cambridge University Press, Cambridge
11. Krüger F, Nyolt M, Yordanova K, Hein A, Kirste T (2014) Computational state space models for activity and intention recognition. A feasibility study. PLoS One 9(11):e109381. https://doi.org/10.1371/journal.pone.0109381
12. Krüger F, Steiniger A, Bader S, Kirste T (2012) Evaluating the robustness of activity recognition using computational causal behavior models. In: Proceedings of the international workshop on situation, activity and goal awareness held at Ubicomp 2012, pp. 1066–1074. ACM, Pittsburgh, PA, USA. https://doi.org/10.1145/2370216.2370443
13. Kulkarni T.D, Kohli P, Tenenbaum J.B, Mansinghka V (2015) Picture: a probabilistic programming language for scene

perception. In: The IEEE conference on computer vision and pattern recognition (CVPR), pp. 4390–4399
14. Lunn DJ, Thomas A, Best N, Spiegelhalter D (2000) Winbugs—a bayesian modelling framework: concepts, structure, and extensibility. Stat Comput 10(4):325–337
15. McCallum A, Schultz K, Singh S (2009) FACTORIE: probabilistic programming via imperatively defined factor graphs. In: Bengio Y, Schuurmans D, Lafferty JD, Williams CKI, Culotta A (eds) Advances in neural information processing systems 22, Curran Associates, Inc., pp. 1249–1257. http://papers.nips.cc/paper/3654-factorie-probabilistic-programming-via-imperatively-defined-factor-graphs.pdf
16. Nyolt M, Kirste T (2015) On resampling for Bayesian filters in discrete state spaces. In: Proceedings 2015 IEEE 27th international conference on tools with artificial intelligence, pp. 526–533. IEEE computer society. https://doi.org/10.1109/ICTAI.2015.83
17. Nyolt M, Krüger F, Yordanova K, Hein A, Kirste T (2015-06) Marginal filtering in large state spaces. Int J Approx Reason 61:16–32. https://doi.org/10.1016/j.ijar.2015.04.003
18. Paige B, Wood F (2014) A compilation target for probabilistic programming languages. In: Xing EP, Jebara T (eds) Proceedings of the 31st international conference on machine learning, proceedings of machine learning research, vol 32, pp. 1935–1943. PMLR, Beijing, China
19. Patil A, Huard D, Fonnesbeck CJ (2010) Pymc: Bayesian stochastic modelling in python. J Stat Softw 35(4):1
20. Pfeffer A (2016) Practical probabilistic programming, 1st edn. Manning Publications Co., Shelter Island
21. Poon H, Domingos P (2006) Sound and efficient inference with probabilistic and deterministic dependencies. AAAI 6:458–463
22. Popko M, Lüdtke S (2018) On the applicability of probabilistic programming languages for causal activity recognition. https://doi.org/10.5281/zenodo.1635591
23. Sato T, Kameya Y (2008) New advances in logic-based probabilistic modeling by PRISM. Springer, Berlin. pp 118–155 https://doi.org/10.1007/978-3-540-78652-8_5
24. Tolpin D, van de Meert JW, Yang H, Wood F (2016) Design and implementation of probabilistic programming language Anglican. In: Proceedings of the 28th symposium on the implementation and application of functional programming languages, pp. 6:1–6:12. ACM
25. Turliuc CR, Dickens L, Russo A, Broda K (2016-11) Probabilistic abductive logic programming using Dirichlet priors. Int J Approx Reason 78:223–240. https://doi.org/10.1016/j.ijar.2016.07.001
26. Wood F, van de Meert JW, Mansinghka V (2014) A new approach to probabilistic programming inference. In: Artificial intelligence and statistics