Exact Inference for Nested Discrete Probabilistic Programs

Francesco Pontiggia
francesco.pontiggia@tuwien.ac.at
TU Wien
Vienna, Austria

Ezio Bartocci
ezio.bartocci@tuwien.ac.at
TU Wien
Vienna, Austria

Michele Chiari
michele.chiari@tuwien.ac.at
TU Wien
Vienna, Austria

1 Introduction

Conditioning in probabilistic programming allows for representing various human reasoning patterns, such as expressing beliefs, intentions, desires in uncertain circumstances, or adding evidence of facts. Stuhlmüller and Goodman [13] have introduced nested queries (i.e., nested probabilistic programs) as a tool to model *theory of mind*, i.e. metareasoning patterns. An example are Schelling coordination games: two agents wish to meet in a cafè in town but cannot communicate. However, they know perfectly each other's preferences. Each agent samples both a location according to its preference, and a location obtained simulating the behaviour of the other agent, and finally conditions on the two being equal. More applications are in game theory, linguistics, multi-agent planning [10] and sequential decision-making [5].

To express nested queries, probabilistic programming languages (PPLs) must support conditioning via ordinary operators that are part the language, and not merely applied to programs, such as `query()` and `observe()`. Modern PPLs comprise them because they have the expressive power of universal computation, hence they allow for (potentially infinite) recursion, a straightforward way to formalize and implement conditioning and nesting of models. Examples are WebPPL [6], Church , and Anglican.

Nested inference for such infinite-state models is a very challenging task. Existing solutions are dynamic programming [12] and Nested Monte Carlo estimation [9]. In this work, we propose to perform exact inference on nested probabilistic programs through an operational semantics based on probabilistic Pushdown Automata (pPDA). pPDA can be seen as an extension of ordinary finite-state Markov Chains with an unbounded stack. They model programs with discrete probability distributions and all variables having finite domain, but with unrestricted function calls, unbounded recursion and unbounded iteration. As we will show, with some tailoring of the pPDA model, the stack can also track the recursive nesting of queries, thus enabling exact nested inference.

Related Works. `Dice` [7] and `PERPL` [3] are two related PPLs devised for exact inference. `Dice` supports discrete bounded variables and a limited form of function calls and iteration. `PERPL` extends `Dice` with unbounded recursion, while [14] extends `Dice` with unbounded iteration. None of these PPLs supports nested queries.

2 Probabilistic Pushdown Automata

We present probabilistic Operator Precedence Automata (pOPA), a class of pPDA devised to model nested queries by giving a rejection sampling semantics to hard conditioning. We describe first all the components informally, and then assemble them in Def 2.2. We refer to [8, Sec. 4] for the presentation of a reference programming language, cprGCL, and an operational semantics in terms of pOPA. In cprGCL, if condition `b` is not satisfied, a `observe(b)` statement is interpreted as rejecting immediately the current execution, and re-instantiating a new trial of the current probabilistic program (query). If `b` is satisfied, `observe(b)` is skipped. Note that, since we limit ourselves to the discrete setting, we are allowed to consider only hard conditioning.

	call	**ret**	**qry**	**obs**	**stm**
call	$\lessdot$	$\doteq$	$\lessdot$	$\gtrdot$	$\lessdot$
ret	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
qry	$\lessdot$	$\doteq$	$\lessdot$	$\lessdot$	$\lessdot$
obs	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
stm	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$

Figure 1: Operator Precedence Matrix $M_{\mathbf{call}}$. For example, precedence between call and ret is $\doteq$.

Let Q be the set of automaton states. In a pOPA, each $u \in Q$ is labeled with a symbol corresponding to a statement in the original program. These symbols play a crucial role in the management of the stack (as we will describe soon). Therefore, we focus on an alphabet called $\Sigma_{\mathbf{call}}$ with only five symbols: $\Sigma_{\mathbf{call}} = \{\mathbf{call}, \mathbf{ret}, \mathbf{qry}, \mathbf{obs}, \mathbf{stm}\}$. Symbols **call** and **ret** represent resp. procedure calls and returns, and can be decorated with the name of the procedure to be invoked. **qry** marks the beginning of a probabilistic program in their nesting, while **obs** is a triggered rejection (i.e., a non satisfied observation). Label **stm** denotes all other statements (e.g., assignments, sampling from a primitive distribution, while-loop instructions ...).

This class of automata takes its name from the fact that their alphabets are equipped with three *Precedence Relations* between symbols. Let Σ be an alphabet. For $a, b \in \Sigma$, if $a \lessdot b$ we say *a yields precedence* to b, if $a \doteq b$ we say a is *equal in precedence* to b, and if $a \gtrdot b$ then *a takes precedence* from b. Precendence relations are expressed concisely in a matrix.

Definition 2.1. An Operator Precedence Matrix (OPM) M is a total function $M : \Sigma^2 \to \{\lessdot, \doteq, \gtrdot\}$ If M is an OPM on a finite alphabet Σ, (Σ, M) is an *OP alphabet.*

pOPA have a fixed set of stack symbols $\Gamma_\bot = \Gamma \cup \{\bot\}$: $\bot$ is the initial stack symbol, and $\Gamma = \Sigma \times Q$ (i.e., stack symbols are pairs of alphabet labels and pOPA states). We indicate with $top(A)$ the topmost symbol of stack A and $smb([a, r]) = a$. $smb(\bot) = \#$, a special symbol that yields precedence to all the others.

A pOPA can perform three types of moves: adding a symbol on top of stack (**push**), updating the current topmost stack symbol (**shift**), or removing the topmost stack symbol (**pop**). Three transition functions reflect them: $\delta_{push} : Q \to Distr(Q)$, $\delta_{shift} : Q \to Distr(Q)$, and $\delta_{pop} : (Q \times Q) \to Distr(Q)$. The precedence relation between the label on top of the stack and the label $\Lambda(u)$ of current state u determines the next move.

Table 1: Results on two nested inference queries. For `POPAlyzer`, we report also the number of equations solved. ✗ indicates failure of the inference process due to a Javascript heap out-of-memory error (with memory limit 16 GB). For Schelling, we found significant differences between multiple executions of rejection sampling. Namely, out of 11 executions, WebPPL run out of memory in seven cases; in the remaining four, it returned 2-outcome distributions $(0.59, 0.41)$, $(0.62, 0.38)$, $(0.61, 0.39)$, $(0.52, 0.48)$**. The exact values computed by our tool are** $(0.610, 0.390)$**.**

Experiment	`POPAlyzer`			WebPPL	
	# equations	time	memory	enumerate	rejection
Schelling	1236	< 1 sec	80 MB	✗	either a few secs or ✗
DMP	3453	< 1 sec	80 MB	✗	✗

For any stack $A \in \Gamma^*\{\bot\}$ and state $u \in Q$:

$$\begin{aligned}
\textbf{push}: \quad & (u, A) \xrightarrow{x} (v, [\Lambda(u), u]A) \\
& \text{if } smb(top(A)) \lessdot \Lambda(u) \text{ and } \delta_{push}(u)(v) = x; \\
\textbf{shift}: \quad & (u, [a, s]A) \xrightarrow{x} (v, [\Lambda(u), s]A) \\
& \text{if } a \doteq \Lambda(u) \text{ and } \delta_{shift}(u)(v) = x; \\
\textbf{pop}: \quad & (u, [a, s]A) \xrightarrow{x} (v, A) \\
& \text{if } a \gtrdot \Lambda(u) \text{ and } \delta_{pop}(u, s)(v) = x.
\end{aligned}$$

where $\xrightarrow{x}$ denotes a transition with probability x.

In pop moves, the state recorded on top of the stack is used as an information to determine where the pOPA transitions to.

Definition 2.2. A pOPA is a tuple $\mathcal{A} = (\Sigma, M, Q, u_0, \delta, \Lambda)$ where: (Σ, M) is an OP alphabet; Q is a finite set of states (disjoint from Σ); u_0 is the initial state; $\Lambda : Q \to \Sigma$ is a state labelling function; and δ is a triple of transition functions $\delta_{push} : Q \to Distr(Q)$, $\delta_{shift} : Q \to Distr(Q)$, and $\delta_{pop} : (Q \times Q) \to Distr(Q)$.

Figure 1 gives a precedence matrix for the alphabet $\Sigma_{\textbf{call}}$ such that a pOPA on $\Sigma_{\textbf{call}}$ mimics the behaviour of a procedural probabilistic programming language. E.g., **call** $\lessdot$ **call** indicates to push on top of the stack when calling a function inside a function. The precedence relations for **obs** trigger pop moves that unwind the stack until, and no further a symbol with **qry** is reached, in line with rejection sampling.

3 System of Equations

The denotation of the original program, its posterior probability distribution, corresponds in the pOPA to the least non-negative fixed point of the system of polynomial equations $\mathbf{v} = f(\mathbf{v})$, where $\mathbf{v}$ is the vector of triples $[\![u, \alpha \mid v]\!]$ for all $u, v \in Q$, $\alpha \in \Gamma$ and

$$f([\![u, \alpha \mid v]\!]) = \begin{cases} f_{push}([\![u, \alpha \mid v]\!]) & \text{if } \alpha = \bot, \text{ or } \alpha = [a, s] \text{ and } a \lessdot \Lambda(u) \\ f_{shift}([\![u, \alpha \mid v]\!]) & \text{if } \alpha = [a, s] \text{ and } a \doteq \Lambda(u) \\ f_{pop}([\![u, \alpha \mid v]\!]) & \text{if } \alpha = [a, s] \text{ and } a \gtrdot \Lambda(u) \end{cases}$$

$$f_{push}([\![u, \alpha \mid v]\!]) = \sum_{r,t \in Q} \delta_{push}(u)(r)[\![r, [\Lambda(u), u] \mid t]\!][\![t, \alpha \mid v]\!]$$

$$f_{shift}([\![u, \alpha \mid v]\!]) = \sum_{r \in Q} \delta_{shift}(u)(r)[\![r, [\Lambda(u), s] \mid v]\!]$$

$$f_{pop}([\![u, \alpha \mid v]\!]) = \delta_{pop}(u, s)(v)$$

Such equation systems have been extensively studied in the context of pPDA model checking [1, 4, 16] under the form of *termination probabilities*. Informally, $[\![u, \alpha \mid v]\!]$ is the probability that a pOPA in state u with α on top of its stack eventually pops α and reaches state v. When u is the initial state and α is the called `main()` query, if v encodes the return value it is easy to see how these equations represent the posterior distribution.

3.1 Computing solutions

In general, solutions of $\mathbf{v} = f(\mathbf{v})$ are (possibly irrational) algebraic numbers [4], so they cannot always be computed exactly in an explicit form. However, the system can be encoded in the Existential Theory of the Real [4], which is decidable in PSPACE [2]. We note that our expressive setting does not come at any additional complexity cost with respect to `Dice` and `PERPL`, whose system of equations are PSPACE-hard [7]. The relation with the theory of the Reals is only of theoretical interest though, as SMT solvers offer only doubly exponential algorithms for it, with poor performances.

On the other hand, various numerical methods [4, 16] can approximate solutions to any desired level of accuracy [11]. In this work, we propose to exploit recent results on finding certificates for pPDA, and in particular Optimistic Value Iteration [15]. This methods computes firstly a vector of lower bounds to the solution via standard value iteration, and then guesses numerically a vector of upper bounds to the least fixpoint. A key property is that these upper bounds are *inductive* (or self-certifying): there exist an easy algorithm to prove that they are correct. Since the fixpoint lies between the two bounds, this approach ensures soundness of our inference algorithm despite the solutions being irrational. Moreover, these bounds can be made as tight as desired.

4 Preliminary Experiments

We have implemented the method sketched above in a tool called `POPAlyzer`. We report on some preliminary experiments in Table 1. We consider two case studies: the Schelling coordination game of the Introduction, and a generic multi-agent sequential decision making problem called DMP. For Schelling, recursion depends on a biased coin flip, thus being unbounded. As a baseline, we consider WebPPL, a Javascript-based PPL supporting nested queries. We have further optimized our tool by decomposing the system of equations into strongly connected components. For WebPPL, we have increased heap size available to Javascript's runtime to 16GB. We report the default inference method (enumeration), and rejection sampling with default options: 100 samples per query, and incremental mode enabled. While we expect enumeration to fail on Schelling, due to unbounded recursion, the DMP model presents a chain of recursive calls that eventually terminates: an explicit enumeration of all possible paths is in principle feasible.

References

[1] Tomás Brázdil, Javier Esparza, Stefan Kiefer, and Antonín Kucera. 2013. Analyzing probabilistic pushdown automata. *Formal Methods Syst. Des.* 43, 2 (2013), 124–163. https://doi.org/10.1007/s10703-012-0166-0

[2] John F. Canny. 1988. Some Algebraic and Geometric Computations in PSPACE. In *STOC'88*. ACM, 460–467. https://doi.org/10.1145/62212.62257

[3] David Chiang, Colin McDonald, and Chung-chieh Shan. 2023. Exact Recursive Probabilistic Programming. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 665–695. https://doi.org/10.1145/3586050

[4] Kousha Etessami and Mihalis Yannakakis. 2009. Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations. *J. ACM* 56, 1 (2009), 1:1–1:66. https://doi.org/10.1145/1462153.1462154

[5] Owain Evans, Andreas Stuhlmüller, John Salvatier, and Daniel Filan. 2017. Modeling Agents with Probabilistic Programs. http://agentmodels.org

[6] Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org.

[7] Steven Holtzen, Guy Van den Broeck, and Todd D. Millstein. 2020. Scaling exact inference for discrete probabilistic programs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 140:1–140:31. https://doi.org/10.1145/3428208

[8] Francesco Pontiggia, Ezio Bartocci, and Michele Chiari. 2024. Model Checking Recursive Probabilistic Programs with Conditioning (v2). *CoRR* (2024). https://arxiv.org/abs/2404.03515v2

[9] Tom Rainforth. 2018. Nesting Probabilistic Programs. In *UAI 2018*, Amir Globerson and Ricardo Silva (Eds.). AUAI Press, 249–258. http://auai.org/uai2018/proceedings/papers/92.pdf

[10] Iris Rubi Seaman, Jan-Willem van de Meent, and David Wingate. 2018. Modeling Theory of Mind for Autonomous Agents with Probabilistic Programs. *CoRR* abs/1812.01569 (2018). http://arxiv.org/abs/1812.01569

[11] Alistair Stewart, Kousha Etessami, and Mihalis Yannakakis. 2015. Upper Bounds for Newton's Method on Monotone Polynomial Systems, and P-Time Model Checking of Probabilistic One-Counter Automata. *J. ACM* 62, 4 (2015), 30:1–30:33. https://doi.org/10.1145/2789208

[12] Andreas Stuhlmüller and Noah D. Goodman. 2012. A Dynamic Programming Algorithm for Inference in Recursive Probabilistic Programs. In *StaRAI-12*. https://starai.cs.kuleuven.be/2012/accepted/stuhlmuller.pdf

[13] Andreas Stuhlmüller and Noah D. Goodman. 2014. Reasoning about reasoning by nested conditioning: Modeling theory of mind with probabilistic programs. *Cognitive Systems Research* 28 (2014), 80–99. https://doi.org/10.1016/J.COGSYS.2013.07.003

[14] Mateo Torres-Ruiz, Robin Piedeleu, Alexandra Silva, and Fabio Zanasi. 2024. On Iteration in Discrete Probabilistic Programming. In *FSCD (LIPIcs, Vol. 299)*, Jakob Rehof (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 20:1–20:21. https://doi.org/10.4230/LIPICS.FSCD.2024.20

[15] Tobias Winkler and Joost-Pieter Katoen. 2023. Certificates for Probabilistic Pushdown Automata via Optimistic Value Iteration. In *TACAS'23 (LNCS, Vol. 13994)*. Springer, 391–409. https://doi.org/10.1007/978-3-031-30820-8_24

[16] Dominik Wojtczak and Kousha Etessami. 2007. PReMo: An Analyzer for Probabilistic Recursive Models. In *TACAS'07 (LNCS, Vol. 4424)*. Springer, 66–71. https://doi.org/10.1007/978-3-540-71209-1_7