Coinductive Trees for Exact Inference of Probabilistic Programs

ALEXANDER BAGNALL, Ohio University, United States
GORDON STEWART, Ohio University, United States
ANINDYA BANERJEE, IMDEA Software Institute, Spain

We propose a coinductive variant of Knuth-Yao trees (variously, Discrete Distribution Generating trees) as an intermediate representation supporting exact inference for probabilistic programs which may contain loops with probabilistic termination conditions. We provide a prototype implementation of a probabilistic programming language, Zar, written in Haskell, exemplifying both Knuth-Yao trees as well as exact inference on the trees.

1 INTRODUCTION

Probabilistic programmers define probabilistic models by writing conventional imperative programs extended with primitives for random sampling and conditioning. Inference is the problem of computing an explicit representation of the probability distribution implicitly specified by a probabilistic program in order to support queries such as "What is the probability of event e?", where e is a predicate on program states or a possible return value. One can infer a probabilistic program's posterior distribution either approximately (e.g., by sampling) or exactly using symbolic methods (e.g., compilation to binary decision diagrams, or BDDs). When computationally viable, exact inference is preferable to approximate techniques because it is deterministic, trustworthy, and does not propagate errors to subsequent analyses.

Symbolic methods based on decision diagrams like BDDs have seen most currency within the probabilistic model checking community (cf. [Miner and Parker 2004] for a survey). In this work, we revisit the problem of exact inference for probabilistic programs and propose a new intermediate representation based in Knuth-Yao (KY) trees (alternatively, Discrete Distribution Generating trees [Knuth and Yao 1976]). In contrast to existing recent work such as that of Holtzen *et al.* [Holtzen et al. 2019] who compile only loop-free programs to BDDs, our KY tree representation enables exact inference even of programs with almost-surely terminating loops. We have a prototype implementation of our technique, Zar [1], that uses our Knuth-Yao IR to do exact inference of programs in a conventional probabilistic programming language.

2 KNUTH-YAO IR FOR ALMOST-SURE TERMINATION

As motivation, consider the probabilistic programs of Listings 1 and 2, the first of which terminates in all executions (it contains no loops) while the second, which simulates a fair coin using a biased one, terminates only with probability 1. If we understood the program of Listing 1 as a random process mapping inputs bits to outputs, we can therefore place an upper bound on the number of bits required for the program to produce an output y (in this case, 3). That the program terminates absolutely leads to a natural interpretation as a finite function of boolean variables, which can be compiled to a symbolic formula or BDD.

```
x <~ flip(1/2)
y <~ if x then flip(1/2) else flip(1/4)
return y
```

Listing 1. A probabilistic program with no loops

```
x, y <- false, false
while (x = y):
  x, y <~ flip(1/3), flip(1/3)
return x
```

Listing 2. Simulating a fair coin with a biased one

```
data TreeF  a b = LeafF a | SplitF b b | NeverF
type Cotree a   = Fix (TreeF a)
data Tree   a   = Leaf a
                | Split (Maybe Label) (Tree a) (Tree a)
                | Hole Label
phi :: Tree a -> TreeCoalgebra a
phi = ...

generate :: Tree a -> Cotree a
generate t = (unfold . phi) t (Hole 0)
```

Listing 3. Knuth-Yao trees in Haskell

The program of Listing 2, which simulates a fair coin by flipping the biased coins x and y until $x = y$, differs fundamentally from that of Listing 1 in that the number of bits required to produce a sample cannot be bounded above by any fixed constant. No finite number of loop iterations suffices to guarantee termination—we can say only that the program *almost surely terminates*, that is, does so with probability 1.

We compile such programs to KY trees, possibly infinite decision diagrams such as the one of Figure 1 in which nodes represent binary decisions and leaves, results. The tree of Figure 1, for example, returns True with probability 1/3, corresponding to the binary expansion of 1/3 as $0.\overline{01}$. To indicate the cycle, we use T_0 to label the root of the tree and $\boxed{0}$ to indicate a labeled "hole", a tree expansion point that unfolds as T_0.

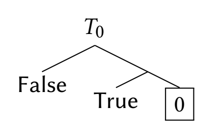


Fig. 1. Knuth-Yao representation of Bernoulli(1/3)

KY trees may have arbitrary structure (consider a representation of the Bernoulli distribution with $p = \pi - 3$). To do exact inference, we target the subclass of KY trees that are finitely representable, as sketched in Haskell in Listing 3. The main data type, `Tree a`, defines labeled trees with leaves of type `a` and `Hole`s, or corecursive expansion points. We define a mapping from `Tree`s to `Cotree`s, the final coalgebra generated by the functor `TreeF`, as the function `generate t`,

[1] https://github.com/OUPL/Zar

Authors' addresses: Alexander Bagnall, Ohio University, United States, ab667712@ohio.edu; Gordon Stewart, Ohio University, United States, gstewart@ohio.edu; Anindya Banerjee, IMDEA Software Institute, Spain, anindya.banerjee@imdea.org.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

$$
\begin{array}{lll}
e & ::= & \ldots \\
c & ::= & \mathsf{skip} \\
 & & x := e \\
 & & x \leftsquigarrow e \\
 & & c_1 \;;\; c_2 \\
 & & \mathsf{if}\ e\ \{c_1\}\ \{c_2\} \\
 & & \mathsf{observe}\ e \\
 & & \mathsf{while}\ e\ c
\end{array}
$$

(a) Syntax

$$
\begin{array}{rcl}
[\![\mathsf{skip}]\!](T) & \triangleq & T \\
[\![x := e]\!](T) & \triangleq & \mathsf{bind}\ T\ (\lambda\sigma.\ \mathsf{Leaf}\ \sigma[x := [\![e]\!]_\sigma]) \\
[\![x \leftsquigarrow e]\!](T) & \triangleq & \ldots // \mathsf{Bernoulli}(p)\ \textit{and}\ \mathsf{Uniform}([\ldots]) \\
[\![c_1 \;;\; c_2]\!](T) & \triangleq & [\![c_2]\!]([\![c_1]\!](T)) \\
[\![\mathsf{if}\ e\ \{c_1\}\ \{c_2\}]\!](T) & \triangleq & \mathsf{bind}\ T\ (\lambda\sigma.\ \mathsf{if}\ [\![e]\!]_\sigma\ \mathsf{then}\ [\![c_1]\!](\mathsf{Leaf}\ \sigma)\ \mathsf{else}\ [\![c_2]\!](\mathsf{Leaf}\ \sigma)) \\
[\![\mathsf{observe}\ e]\!](T) & \triangleq & \mathsf{bind}\ T\ (\lambda\sigma.\ \mathsf{if}\ [\![e]\!]_\sigma\ \mathsf{then\ Leaf}\ \sigma\ \mathsf{else\ Hole}\ 0) \\
[\![\mathsf{while}\ e\ c]\!](T) & \triangleq & \mathsf{bind}\ T\ F \\
\text{where } F(\sigma) \mid & \neg[\![e]\!]_\sigma \triangleq & \mathsf{Leaf}\ \sigma \\
F(\sigma) \mid & [\![e]\!]_\sigma \triangleq & \mathsf{Lbl}\ \ell\ (\mathsf{bind}\ ([\![c]\!](\mathsf{Leaf}\ \sigma))\ (\lambda\sigma'.\ \mathsf{if} [\![e]\!]_{\sigma'}\ \mathsf{then\ Hole}\ \ell\ \mathsf{else\ Leaf}\ \sigma')),\ \ell\ \textit{fresh}
\end{array}
$$

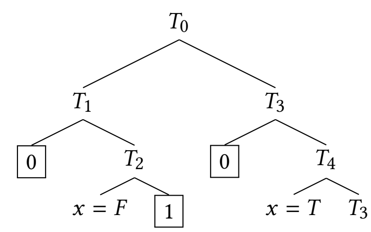
(b) Semantics

Fig. 2. Zar syntax and semantics

the anamorphism `unfold` applied to the tree coalgebra produced by `phi t`.

Semantics. We map programs in a conventional probabilistic command language (Figure 2a) to KY trees using the semantics of Figure 2b. Commands are interpreted as tree transformers in which input trees T defining prior distributions over `States` σ are mapped to output trees defining posterior distributions over results. The function `bind :: ∀a b. Tree a -> (a -> Tree b) -> Tree b` applies a continuation to the leaves of a tree. When interpreting an observe statement, we bind to the leaves of the tree the continuation that returns `Leaf σ` if the observe condition is satisfied, and `Hole 0` otherwise, indicating failure (a return to the distinguish root labeled 0). The interpretation of while loops does not take a fixed point, instead generating a finite `Tree` describing the infinite process that repeatedly executes the interpretation of the loop body $[\![c]\!](\mathsf{Leaf}\ \sigma)$. Sampling a discrete distribution expression $x \leftsquigarrow e$ (implementation elided) generates a tree that assigns x the outcomes dictated by e, with the appropriate probabilities (our prototype currently supports simple discrete distributions like Bernoulli and Uniform).

Exact Inference. Figure 3 shows the `Tree` generated by the semantics of Figure 2b for the fair coin program of Listing 2. To perform exact inference, we fix a return predicate `f : State -> ℝ` whose expected value we wish to compute, then construct and solve the system of equations induced by mapping `f` over the tree. The Figure 3 tree, for example, has leaves corresponding to `f(σ) ≜ σ(x)` (projection of x, corresponding to `return x`). The resulting system of linear equations (RHS of Figure 3) computes the weight of each subtree, the probability that x = True. The weight of the root T_0 gives the total weight of x. The system of linear equations generated in this way has a unique solution iff the program interpreted by the tree terminates with probability 1. The inferred distribution is normalized despite the presence of observe statements in the source language because branches inconsistent with observations are made to loop to the root T_0. An alternative representation sets such branches to 0, yielding an unnormalized distribution.



$$
\begin{aligned}
T_0 &= \tfrac{1}{2}T_1 + \tfrac{1}{2}T_3 \\
T_1 &= \tfrac{1}{2}T_0 + \tfrac{1}{2}T_2 \\
T_2 &= \tfrac{1}{2}T_1 \\
T_3 &= \tfrac{1}{2}T_0 + \tfrac{1}{2}T_4 \\
T_4 &= \tfrac{1}{2}T_3 + \tfrac{1}{2}
\end{aligned}
$$

Fig. 3. Knuth-Yao tree (left) for the fair coin program of Listing 2 together with the corresponding system of equations (right)

3 RELATED WORK AND DISCUSSION

Both approximate and exact inference are theoretically hard [Roth 1996], but approximate techniques tend to perform better at scale. Nonetheless, as [Holtzen et al. 2019] show, exact inference is often successfully applied in practice by exploiting repeating and compositional structure that appears in many problem instances.

Holtzen *et al.* perform exact inference on finite-domain probabilistic Boolean programs without loops by compiling them to Boolean formulae (represented by BDDs) and applying weighted model-counting (WMC) techniques on the resulting representations. Compilation to BDDs, in contrast to path-based enumeration inference methods (e.g., [Gehr et al. 2016]), exploits duplication and conditional independence in order to minimize the size of the representation, which in turn improves efficiency. Coinductive KY trees, in addition to supporting probabilistic loops, lie in between full path enumeration and compilation to BDDs. Since they are finitely represented, duplication and otherwise unnecessary structure in KY trees can be eliminated to improve performance.

In other closely related work, [Claret et al. 2013] use dataflow analysis techniques to perform exact inference directly on the syntax of finite-domain probabilistic programs, supporting inference on probabilistic loops via computation of fixpoints. Zar supports programs over (possibly infinite) discrete domains, but requires finite support, a restriction our implementation enforces by doing dataflow analysis on Zar loops to ensure the absence of loop-carried dependences (for example, between the values of integer i in an almost surely terminating loop that increments i at each step).

REFERENCES

Guillaume Claret, Sriram K Rajamani, Aditya V Nori, Andrew D Gordon, and Johannes Borgström. 2013. Bayesian inference using data flow analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 92–102.

Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*. Springer, 62–83.

Steven Holtzen, Todd Millstein, and Guy Van den Broeck. 2019. Symbolic Exact Inference for Discrete Probabilistic Programs. arXiv:cs.PL/1904.02079

Donald Knuth and Andrew Yao. 1976. The complexity of nonuniform random number generation. *Algorithm and Complexity, New Directions and Results* (1976), 357–428.

Andrew Miner and David Parker. 2004. Symbolic representations and analysis of large probabilistic systems. In *Validation of Stochastic Systems*. Springer, 296–338.

Dan Roth. 1996. On the hardness of approximate reasoning. *Artificial Intelligence* 82, 1-2 (1996), 273–302.