Daphne: Multi-Pass Compilation of Probabilistic Programs into Graphical Models and Neural Networks

Anonymous authors
Paper under double-blind review

Abstract

Daphne is a probabilistic programming system that provides an expressive syntax to denote a large, but restricted, class of probabilistic models. Programs written in the Daphne language can be compiled into a general graph data structure of a corresponding probabilistic graphical model with simple link functions that can easily be implemented in a wide range of programming environments. Alternatively Daphne can also further compile such a graphical model into understandable and vectorized PyTorch code that can be used to train neural networks for inference. The Daphne compiler is structured in a layered multi-pass compiler framework that allows independent and easy extension of the syntax by adding additional passes, while leveraging extensive partial evaluation to reduce all syntax extensions to the graphical model at compile time.

1 Introduction

Probabilistic modeling is integral for modern machine learning and statistics. The recent advent of generative AI has situated a large part of deep learning applications in an approximate Bayesian modeling framework (Brown et al., 2020; Rombach et al., 2022) and the statistics communities around structured probabilistic programming systems equally are expanding quickly (Štrumbelj et al., 2023). Such probabilistic programming systems allow to specify models and inference problems in programming languages with different expressivity and tractability trade-offs. Turing-complete languages are universal and can be very expressive at the expense of rendering inference harder. But systems with restricted languages still can cover large classes of problems.

An interesting position in the language design space are languages that can be readily translated into a representation for which powerful inference algorithms exist, e.g. Stan (Carpenter et al., 2017). Such systems compile an **expressive input syntax** into a lower-level model that can be **efficiently evaluated** with the numerical primitives of the inference engine. Furthermore it should be possible for users to **extend the language** with new syntax if needed. Finally the host environment in which the compiler and runtime operates ideally should be built in a **rich programming environment** with probabilistic primitives and access to modern machine learning primitives. This combination of properties renders a probabilistic programming system also particularly well suited for teaching and research. Following these design principles we introduce Daphne. Compared to other probabilistic programming systems we are aware of, Daphne is set apart by its expressive higher-order syntax and the extensive use of partial evaluation inside of its compiler (Section 4).

2 Motivation for Daphne

To demonstrate the use cases of Daphne we highlight two applications in Figure 1 and Figure 2. The implementation details of these papers are not covered here, they are shown to highlight the benefit of having Daphne as an inference problem specification interface to succinctly describe probabilistic graphical models. The graphical models are then used in both cases to sparsify neural network architectures for amortized inference learning tasks. Daphne is geared to ease the extension of its language to provide extraction of rich, but easy to process, compute graphs of probabilistic programs for such applications.

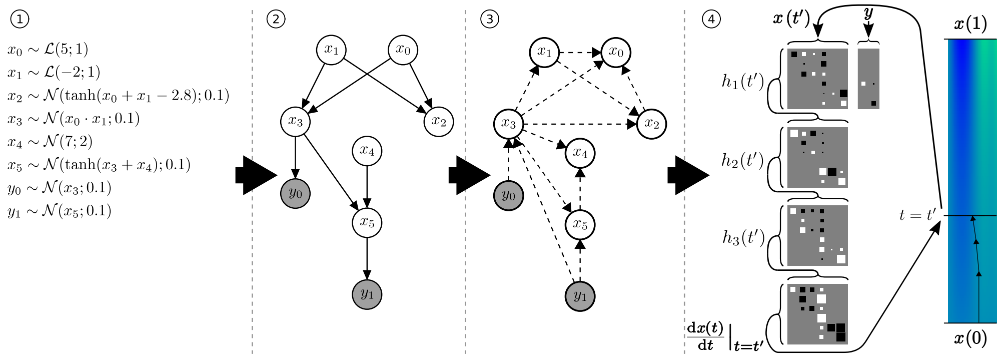
①
$x_0 \sim \mathcal{L}(5; 1)$
$x_1 \sim \mathcal{L}(-2; 1)$
$x_2 \sim \mathcal{N}(\tanh(x_0 + x_1 - 2.8); 0.1)$
$x_3 \sim \mathcal{N}(x_0 \cdot x_1; 0.1)$
$x_4 \sim \mathcal{N}(7; 2)$
$x_5 \sim \mathcal{N}(\tanh(x_3 + x_4); 0.1)$
$y_0 \sim \mathcal{N}(x_3; 0.1)$
$y_1 \sim \mathcal{N}(x_5; 0.1)$
②
③
④
$x(t')$
y
$h_1(t')$
$h_2(t')$
$h_3(t')$
$\frac{dx(t)}{dt}\big|_{t=t'}$
$x(1)$
$t = t'$
$x(0)$

Figure 1: Translation from a Daphne program (here written in statistical syntax) into its compute graph, its faithful inverse and then into a sparsity mask of a multilayer perceptron (MLP) guiding a continuous normalizing flow (Grathwohl et al., 2018). The resulting sparsified MLP can be trained and run more efficiently. Figure taken with permission form Weilbach et al. (2020).

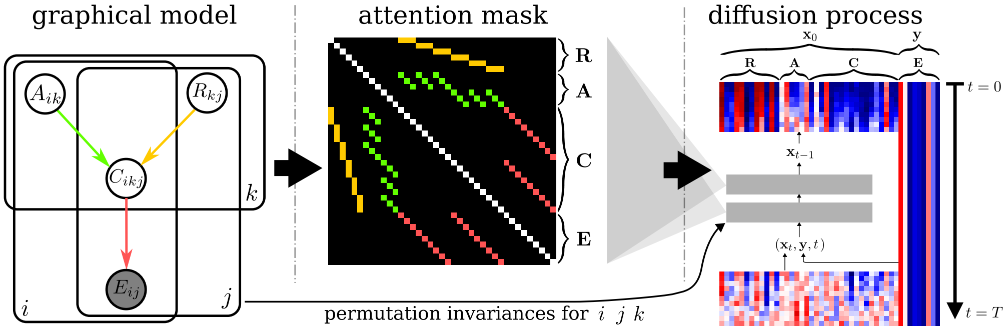

Figure 2: Integration of a graphical model that can be extracted from Daphne into a transformer-based diffusion model that is trained for amortized inference tasks. On the left is the graphical model, its adjacency matrix is then extracted from Daphne's output (or hand-written) and it structures the sparse self-attention transformer layers that guide a diffusion process. Figure taken with permission from Weilbach et al. (2023b).

Daphne provides these graphical model translations in a modular translation process that eases adjustments to the specification language and to the properties that can be captured. As an example of an extension that would be easy to add to Daphne, the permutation invariances projected in Figure 2 could also be extracted automatically since they are statically induced by independence of link functions from plate indices (Weilbach et al., 2023b).[1] Both papers require the translation to a graphical model before they are trained with traditional parametric variational families and hence the ability of Daphne to reduce an expressive language to a simple data structure of a graphical model is particularly useful for such integrations.

3 Graphical Probabilistic Programming Language

A *graphical* probabilistic programming language (GPPL) is a language in which all random variables can be identified without evaluating the stochastic expressions in the program, a prominent example being Stan (Carpenter et al., 2017). Such a language allows to unroll all loops and control flow a priori at compile time and represents all random variables explicitly in a traditional probabilistic graphical model (PGM) (Koller & Friedman, 2009). This is in contrast to probabilistic programming languages (PPLs) which support loop and recursion constructs that cannot be simplified before evaluating the stochastic expressions of the program, e.g. because some loop or recursion termination condition depends on a sample of a random variable. This distinction is made in different terms in van de Meent et al. (2018) as first-order (FOPPL)

---

[1] This was not done for this paper since the neural network was directly implemented in PyTorch.

vs. higher-order (HOPPL) languages. The FOPPL language there is syntactically restricted such that termination guarantees are enforced through using bounded loop constructs and prohibiting higher-order functions and recursion to prohibit unbounded computation. We found these restrictions unnecessarily limiting and therefore arrived at GPPLs. Nonetheless a FOPPL is a valid subset of our GPPL (Section 5).

3.1 Syntax

The core language syntax supported by the compiler is defined by the following grammar in Backus-Naur form (BNF),

$s ::=$ symbol (indicating variables)
$c ::=$ constant value or primitive operation (syntactic atoms)
$f ::=$ procedure
$e ::= c \mid s \mid (\texttt{let}\ [s_1\ e_1\ \dots\ s_n\ e_n]\ e_b) \mid (\texttt{if}\ e_1\ e_2\ e_3) \mid (e_f\ e_1\ \dots\ e_n)$
$\mid (\texttt{sample}\ e) \mid (\texttt{observe}\ e_1\ e_2)$
$q ::= e \mid (\texttt{defn}\ f\ [s_1\ \dots\ s_n]\ e)\ q$

Language 1: Daphne GPPL.

The basic syntax is derived from the Clojure programming language (Hickey, 2008; 2020). For the reader unfamiliar with Lisp syntax it can be thought of as an extended form of JSON where "code is data". Lisp expressions are denoted with a list based meta-syntax, i.e. explicitly grouped in lists starting with "(" and ending with ")". As can be seen in the grammar composition rule for e, expressions can contain symbols. For this reason such expressions are also referred to as symbolic expressions (sexps). The language supports lexical binding with the `let` form and control flow through conditional expressions with `if`. Function applications are denoted in prefix notation, i.e. the function is positioned in the first element in the list and all the arguments follow, e.g. `(+ 1 2)`, and e_f needs to evaluate either to a previously defined procedure or a primitive operation. `defn` provides a means to define reusable functions that can be referred to by name f. The last expression is the global entry point into the program having access to all defined functions. Like in Anglican (Tolpin et al., 2016), the probabilistic programming primitives for this language are denoted as `sample` for drawing samples from a random variable and `observe` to condition it on data.

A Bayesian linear regression example in Daphne can be seen in Program 2. A normal prior is defined for `slope` and `bias` and then the `reduce` iterates a normal likelihood `observe-data` over 6 xy data pairs before returning the posterior parameters. In contrast to van de Meent et al. (2018) Daphne does not rule out recursion or higher-order functions as can be seen in our simple implementation of `reduce`.

4 Compiler

The Daphne compiler runs ahead-of-time (AOT) before inference is conducted. It receives the full input syntax including input data and all needed function definitions as shown in Program 2 and translates it to a dictionary describing a probabilistic graphical model during standard compilation. Daphne provides JSON export for its compiler passes through its command line interface.

4.1 Compiler passes

The compilation process happens in multiple passes, where each pass defines the translation of some language features into a simpler syntax before finally only the graphical model remains (van de Meent et al., 2018). This approach allows to build extensible compilers with towers of languages that deal with single language features (Keep & Dybvig, 2013). The compiler has the following standard passes: `desugar`, `symbolic-simplify` and `partial-evaluation` (described in Section 4.2). The `desugar` pass factorizes one big `let` binding into nested single bindings and `symbolic-simplify` applies operations on syntactic objects if possible, e.g. `(first [sample1])` is translated to `sample1` since the argument to `first` is a syntactically represented vector and can be evaluated symbolically. The implementation of operational semantics in the

```
(defn reduce [f acc s]
  (if (> (count s) 0)
    (reduce f (f acc (first s)) (rest s))
    acc))

(defn observe-data [acc data]
  (let [slope (first acc)
        bias  (second acc)
        xn    (first data)
        yn    (second data)
        zn    (+ (* slope xn) bias)]
    (observe (normal zn 1.0) yn)
    [slope bias]))

;; global entry point
(let [slope (sample (normal 0.0 10.0))
      bias  (sample (normal 0.0 10.0))
      data  [[1.0 2.1] [2.0 3.9] [3.0 5.3] [4.0 7.7] [5.0 10.2] [6.0 12.9]]]
  (reduce observe-data [slope bias] data)
  [slope bias])
```

Program 2: Daphne GPPL Example - Linear regression

compiler also uses an explicit substitution pass `substitute` that allows it to substitute symbols for values while respecting the binding structure of the language.

4.2 Partial evaluation

Partial evaluation (Jones et al., 1993) is the process of interpreting parts of a program ahead of time. It is an effective method to implement compilation simply by running the interpreter. The interpreter is able to take sub-expressions and evaluate them as soon as sufficient information is available. For example `(+ 1 2)` can be evaluated to `3` and replaced in the program code since all information to evaluate it is available. The Daphne language is, like Clojure (Hickey, 2020), a functional, stateless Lisp and hence naturally provides substitution semantics which is particularly well-suited for partial evaluation (Jones et al., 1993).

Daphne applies partial evaluation in a fixed point operator together with the other passes until no further simplification is possible. It does so by applying Clojure's `eval` with a properly scoped environment top-down on smaller and smaller sub-expressions of the input syntax until it is able to fully evaluate a sub-expression. Code without random variables can be fully evaluated by the partial evaluator, while expressions depending on random variables are simplified as much as possible (van de Meent et al., 2018) as can be seen in the resulting graph for linear regression in Program 3).

Note that these fixed point iterations entail that the compiler does not have a constant number of passes. Furthermore, it is not trivial to determine ahead of time how many fixed point iterations are needed. Note also that functions defined by `defn` naturally allow bounded recursion during partial evaluation as long as their inputs shrink in every recursion step.[2]

4.3 Graphical model

After iteratively substituting, expanding and reducing the input syntax a graph data structure with vertices `:V`, adjacency `:A`, link functions `:P` and observed nodes `:Y` remain,

---

[2] This is not enforced currently and the compiler will not terminate on violations. Termination checks could be implemented in the partial evaluator similar to the runtime contracts in Racket (Nguyen et al., 2019), but would provide guarantees at translation time here.

```
{:V #{sample1 sample2 observe3 observe4 observe5 observe6 observe7 observe8},
 :A
 {sample2 #{observe3 observe4 observe5 observe6 observe7 observe8},
  sample1 #{observe3 observe4 observe5 observe6 observe7 observe8}},
 :P
 {sample1 (sample* (normal 0.0 10.0)),
  sample2 (sample* (normal 0.0 10.0)),
  observe3 (observe* (normal (+ (* sample1 1.0) sample2) 1.0) 2.1),
  observe4 (observe* (normal (+ (* sample1 2.0) sample2) 1.0) 3.9),
  observe5 (observe* (normal (+ (* sample1 3.0) sample2) 1.0) 5.3),
  observe6 (observe* (normal (+ (* sample1 4.0) sample2) 1.0) 7.7),
  observe7 (observe* (normal (+ (* sample1 5.0) sample2) 1.0) 10.2),
  observe8 (observe* (normal (+ (* sample1 6.0) sample2) 1.0) 12.9)},
 :Y
 {observe3 2.1, observe4 3.9,  observe5 5.3,
  observe6 7.7, observe7 10.2, observe8 12.9}}
```

Program 3: Compiled Graphical Model - Linear regression

The link functions of `:P` are expressed in a language that requires no binding or loop support, e.g. `(observe* (normal (+ (* sample1 1.0) sample2) 1.0) 2.1)`. Its evaluation can be readily implemented with numpy or PyTorch arithmetic and probability primitives in Python, e.g. to implement ancestral sampling in Appendix A.1. Many compilers simplify code to A-normal form (ANF) or single static assignment (SSA) form, which are very close to this graphical format. This allows the compiler to map it to low-level languages without garbage collection, in particular CUDA. In correspondence with the syntax primitives in the GPPL language, `observe*` and `sample*` refer to low-level implementations of sampling and log-probability evaluations.

4.4 Inference runtime

Daphne also provides optional runtime support for inference. A differentiable subset of the GPPL is supported with source to source reverse-mode automatic differentiation (Baydin et al., 2017) and can be plugged into a simple Hamiltonian Monte Carlo (HMC) implementation. A core set of derivatives such as those for normal distributions, arithmetic and a set of scalar functions is provided, but might require additional definitions of derivatives for wider classes of programs. There is also support for Metropolis within Gibbs sampling. Both of these implementations have been used for testing and teaching and provide a starting point for further exploration, but should not yet be expected to perform competitively with mature probabilistic programming systems. Daphne builds on well tested Anglican (Tolpin et al., 2016) primitives though and and can be extended to a wide range of MCMC methods if needed.[3]

Research with Daphne lead to exploration of variational inference methods for amortized inference (Weilbach et al., 2020). In this work the compiler furthermore provides a sparse inversion of the graphical model structure according to Webb et al. (2018) and a translation of the graphical model to Python code. This additional compilation step provides a human readable PyTorch implementation of sampling and log probability evaluation of prior and likelihood for a given graphical model. The code supports batching and can be used to sample a synthetic data set for training a continuous normalizing flow (Grathwohl et al., 2018) as described in (Weilbach et al., 2020). Follow-up work has extended this approach and leveraged the diffusion model framework to yield reliable and more scalable amortized inference artifacts (Weilbach et al., 2023b;a). Daphne can provide automatic derivation of the attention masks from the GPPL language for the sparse transformer in this line of work (Weilbach et al., 2023b) as shown in Section 2.

$$e ::= \ldots \mid (\texttt{foreach}\ e_c\ [s_1\ e_1\ \ldots\ s_n\ e_n]\ e'_1\ \ldots\ e'_m) \mid (\texttt{loop}\ e_c\ e_{init}\ f_{acc}\ e_1\ \ldots\ e_n)$$

Language 2: Loop extensions.

5 Deep learning primitives

Daphne also optionally supports the two loop forms of van de Meent et al. (2018). These forms do not make the language more expressive, but have been originally used to implement a linear algebra library for Weilbach et al. (2020), which includes matrix multiplication and 2d convolution. Both `foreach` and `loop` require a loop counter e_c expression evaluating to an integer to determine the number of loop iterations. `foreach` binds $s_1, \ldots, s_n$ with each element of the collection yielding expressions $e_1, \ldots, e_n$ in the body expressions $e'_1, \ldots, e'_m$. `loop` iterates an accumulating function f_{init} starting with e_{init} over $e_1, \ldots, e_n$. To illustrate the use of the language we provide an excerpt of the library together with an example 2d convolution invocation at the end.

```
(defn dot-helper [t state a b]
  (+ state
     (* (get a t)
        (get b t))))

(defn dot [a b]
  (loop (count a) 0 dot-helper a b))

(defn row-mul [t state m v]
  (conj state (dot (get m t) v)))

(defn mmul [m v]
  (loop (count m) [] row-mul m v))

(defn row-helper [i sum a b]
  (+ sum
     (dot (get a i)
          (get b i))))

(defn inner-square [a b]
  (loop (count a) 0 row-helper a b))

(defn inner-cubic [a b]
  (apply + (foreach (count a) [n (range (count a))]
                    (inner-square (get a n) (get b n)))))

(defn slice-square [input size stride i j]
  (foreach size [k (range (* i stride)
                          (+ size (* i stride)))]
           (subvec (get input k)
                   (* j stride)
                   (+ size (* j stride)))))

(defn slice-cubic [inputs size stride i j]
  (foreach (count inputs) [input inputs]
           (slice-square input size stride i j)))

(defn conv-kernel [inputs kernel bias stride]
  (let [ic (count (first inputs))
        size (count (first kernel))
```

---

[3] There is also zero-copy access to all of Python through `https://github.com/clj-python/libpython-clj` or `https://github.com/oracle/graalpython`

```
        remainder (- size stride)
        to-cover (- ic remainder)
        iters (int (Math/floor (/ to-cover stride)))]
    (foreach iters [i (range iters)]
             (foreach iters [j (range iters)]
                      (inner-cubic (slice-cubic inputs size stride i j)
                                   kernel)))))

(defn conv2d [inputs kernels bias stride]
  (foreach (count kernels) [ksi (range (count kernels))]
           (conv-kernel inputs (get kernels ksi) (get bias ksi) stride)))

(let [w1 [[[[0.8, 0.9],
            [0.9, 0.6]],
           [[0.0, 0.0],
            [0.1, 0.5]]],
          [[[0.2, 0.4],
            [0.6, 0.1]],
           [[0.4, 0.5],
            [0.1, 0.1]]]]
      b1 [0.1, 0.2]
      x  [[[0.4, 0.5, 0.8, 0.8]
           [0.5, 0.8, 0.6, 0.1]
           [0.9, 0.4, 0.7, 0.2]
           [0.5, 0.0, 0.4, 0.2]],
          [[0.0, 0.8, 0.2, 0.3]
           [0.2, 0.2, 0.8, 0.7]
           [0.1, 0.6, 0.6, 0.3]
           [0.6, 0.7, 0.5, 0.2]]]]
  (conv2d x w1 b1 2))
```

In combination those primitives can be used to implement the deep learning applications in Weilbach et al. (2023b) including a stochastic deconvolution layer and a small convolutional network.

6 Related work

Table 1: Comparison of Probabilistic Programming Systems

Feature	**Daphne**	**Stan**	**PyMC**	**Anglican**	**Pyro**	**Gen**
Turing-complete	✓	✗	✓	✓	✓	✓
Stochastic recursion	✗	✗	✓	✓	✓	✓
Graphical model focus	✓	✓	✗	✗	✗	✗
Higher-order functions	✓	✗	✗	✓	✓	✓
Recursion support	✓	✗	✓	✓	✓	✓
HMC inference	(✓)	✓	✓	✗	✓	✓
Python integration	✓	✓	✓	✗	✓	✗
Compilation to CUDA	✓	✗	✗	✗	✗	✗

Compared to Turing-complete languages such as Anglican (Tolpin et al., 2016), Pyro (Bingham et al., 2019), PyMC (Oriol et al., 2023) or Gen (Cusumano-Towner et al., 2019), the Daphne language only supports programs without stochastic recursion. This restricts it from implementing certain non-parametric models and complex stochastic recursion schemes. Inference in such models can be very challenging and brittle though, rendering the restriction to graphical models an acceptable trade-off for many problems. While Daphne shares most of its syntax with Anglican, Anglican programs cannot be translated into a graphical model ahead of inference time and integrated into other inference systems, such as the deep learning pipelines shown in Section 2. Daphne is geared to first being translated to the graphical model before inference.[4] Stan (Carpenter et al., 2017) has a successful GPPL with programs that are restricted to play well with its HMC inference runtime. Compared to Stan, Daphne provides a much more expressive functional probabilisitic programming syntax, including recursion and higher-order functions. BUGS, which is similar to Stan in syntax, has been translated to the FOPPL language in van de Meent et al. (2018). The Daphne graphical model output could equally be translated back to Stan to use its inference engine with the more expressive language. In contrast to Stan, Daphne is implemented not in C++, but in the high-level interactive programming environment of Clojure, which is naturally geared towards meta-programming and exploration of novel ideas. Its code base is small, the compiler passes and core semantics are at most a few hundreds lines long and self-contained. This makes Daphne quick to change and adapt to new ideas, while providing direct access to the reach of the Clojure ecosystem, e.g. into PyTorch at its runtime through libpython-clj.[5] Exploration of this runtime integration is left for future work.

7 Conclusion and future work

Daphne is a small, yet versatile, probabilistic programming environment that represents a new point in the design space of probabilistic programming languages and compilers. Its goal is to capture as much of higher-order functional programming as possible while being reducible to traditional probabilistic graphical models ahead of inference time. The representations of different compiler passes can be exported to provide fine-grained program information, such as dependencies between random variables, to downstream inference runtimes. These representations have been used in a line of research on structured neural networks for amortized inference and for teaching graduate courses in probabilistic programming.

Limitations For very large sized input programs, such as large deep learning models, partial evaluation can lead to expression swell that significantly slows down compilation as all linear algebra operations need to be syntactically expanded and reduced. Scaling better to such programs requires further research, in particular a more structured approach to expanding and reducing forms during partial evaluation in which reductions are always applied before expansions. Note that many other probabilistic programming systems (Bingham et al., 2019; Tolpin et al., 2016) treat tensors as single objects and do not model the full computational graph with all scalars and intermediate computations in the same way Daphne does, avoiding the problem at the expense of not tracking fine-grained sparse structures. Beyond these improvements, expression swell in full generality cannot be avoided since all branches and loops are unrolled into a graph before execution. Take a markov decision process (MDP) such as an agent navigating a maze for example. For each timestep into the future that is unrolled, each branch needs to expanded with all combinations of further branching. This leads to exponential blow-up and renders the program unsuitable for a GPPL-based system like Daphne or Stan.

Addition of a module system, probably following Clojure, is left to future work. Currently the linear algebra of Section 5 has to be manually concatenated with the program before compilation. The Daphne language also does not yet support definitions of anonymous functions, which would also provide closures, rendering it closer to standard Clojure and more convenient for complex programs. Adding closures is left for a future iteration of the language.

References

Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic Differentiation in Machine Learning: A Survey. *Journal of Machine Learning Research*, 18: 153:1–153:43, 2017.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research*, 20(1):973–978, January 2019. ISSN 1532-4435.

---

[4] There are also some exploratory interpreters for teaching included in the code base, but they are not the main focus of the system.

[5] `https://github.com/clj-python/libpython-clj`

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. *arXiv:2005.14165 [cs]*, July 2020.

Bob Carpenter, Andrew Gelman, Matthew Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A Probabilistic Programming Language. *Journal of Statistical Software, Articles*, 76(1):1–32, 2017. ISSN 1548-7660. doi: 10.18637/jss.v076.i01.

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 *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pp. 221–236, Phoenix, AZ, USA, June 2019. Association for Computing Machinery. ISBN 978-1-4503-6712-7. doi: 10/gf4jwn.

Will Grathwohl, Ricky T. Q. Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. FFJORD: Free-Form Continuous Dynamics for Scalable Reversible Generative Models. In *International Conference on Learning Representations*, September 2018.

Rich Hickey. The Clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages*, Dls '08, pp. 1:1–1:1, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-270-2. doi: 10.1145/1408681.1408682.

Rich Hickey. A history of Clojure. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–46, June 2020. ISSN 2475-1421. doi: 10.1145/3386321.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International Series in Computer Science. Prentice Hall, New York, 1993. ISBN 978-0-13-020249-9.

Andrew W. Keep and R. Kent Dybvig. A nanopass framework for commercial compiler development. *ACM SIGPLAN Notices*, 48(9):343–350, September 2013. ISSN 0362-1340. doi: 10.1145/2544174.2500618.

Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, 2009. ISBN 978-0-262-01319-2.

Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. Size-Change Termination as a Contract. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2019*, pp. 845–859, 2019. doi: 10.1145/3314221.3314643.

Abril-Pla Oriol, Andreani Virgile, Carroll Colin, Dong Larry, Fonnesbeck Christopher J., Kochurov Maxim, Kumar Ravin, Lao Jupeng, Luhmann Christian C., Martin Osvaldo A., Osthege Michael, Vieira Ricardo, Wiecki Thomas, and Zinkov Robert. PyMC: A modern and comprehensive probabilistic programming framework in python. *PeerJ Computer Science*, 9:e1516, 2023. doi: 10.7717/peerj-cs.1516.

Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-Resolution Image Synthesis With Latent Diffusion Models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10684–10695, 2022.

Erik Štrumbelj, Alexandre Bouchard-Côté, Jukka Corander, Andrew Gelman, Haavard Rue, Lawrence Murray, Henri Pesonen, Martyn Plummer, and Aki Vehtari. Past, Present, and Future of Software for Bayesian Inference. September 2023.

David Tolpin, Jan Willem van de Meent, Hongseok Yang, and Frank Wood. Design and Implementation of Probabilistic Programming Language Anglican. *arXiv preprint arXiv:1608.05263*, 2016.

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. *An Introduction to Probabilistic Programming*. September 2018.

Stefan Webb, Adam Golinski, Rob Zinkov, Siddharth N, Tom Rainforth, Yee Whye Teh, and Frank Wood. Faithful Inversion of Generative Models for Effective Amortized Inference. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

Christian Weilbach, Boyan Beronov, Frank Wood, and William Harvey. Structured Conditional Continuous Normalizing Flows for Efficient Amortized Inference in Graphical Models. In *International Conference on Artificial Intelligence and Statistics*, pp. 4441–4451. PMLR, June 2020.

Christian Dietrich Weilbach, William Harvey, Hamed Shirzad, and Frank Wood. Scaling Graphically Structured Diffusion Models. In *ICML 2023 Workshop on Structured Probabilistic Inference {\&} Generative Modeling*, July 2023a.

Christian Dietrich Weilbach, William Harvey, and Frank Wood. Graphically Structured Diffusion Models. In *Proceedings of the 40th International Conference on Machine Learning*, pp. 36887–36909. PMLR, July 2023b.

A Appendix

A.1 Python export

The following export is done with the help of hy-lang[6] which allows a direct translation between lisps and then into Python syntax. This is the compiled code for Program 2.

```
import hy
import torch
import math
from torch.distributions import Normal, Bernoulli, Laplace, Uniform

class Model:
    dim_latent = 2
    dim_condition = 6
    faithful_adjacency = [[0, 1], [0, 2], [0, 3], [0, 4], [0, 5], [0, 6], [0, 7], [1, 2], [1, 3], [1, 4], [1, 5], [1, 6], [1, 7], [0, 0], [1, 1]]
    src = '((defn\n  reduce\n  [f acc s]\n  (if (> (count s) 0) (reduce f (f acc (first s)) (rest s)) acc))\n (defn\n
observe-data\n  [acc data]\n  (let\n   [slope\n    (first acc)\n    bias\n    (second acc)\n    xn\n    (first data)\n
yn\n    (second data)\n    zn\n    (+ (* slope xn) bias)]\n   (observe (normal zn 1.0) yn)\n   [slope bias]))\n (let\n
[slope (sample (normal 0.0 10.0)) bias (sample (normal 0.0 10.0))]\n  (reduce\n   observe-data\n   [slope bias]\n
[[1.0 2.1] [2.0 3.9] [3.0 5.3] [4.0 7.7] [5.0 10.2] [6.0 12.9]])\n  [slope bias]))\n'

    def sample(self):
        sample_1 = Normal(0.0, 10.0).sample()
        sample_0 = Normal(0.0, 10.0).sample()
        observe_4 = Normal(sample_0 * 3.0 + sample_1, 1.0).sample()
        observe_6 = Normal(sample_0 * 5.0 + sample_1, 1.0).sample()
        observe_5 = Normal(sample_0 * 4.0 + sample_1, 1.0).sample()
        observe_7 = Normal(sample_0 * 6.0 + sample_1, 1.0).sample()
        observe_2 = Normal(sample_0 * 1.0 + sample_1, 1.0).sample()
        observe_3 = Normal(sample_0 * 2.0 + sample_1, 1.0).sample()
        return [torch.tensor([sample_0, sample_1]), torch.tensor([observe_2, observe_3, observe_4, observe_5, observe_6, observe_7])]

    def log_likelihood(self, sample, observe):
        log_likeli = torch.zeros(sample.shape[0])
```

---

[6] https://hylang.org/

```
        log_likeli += Normal(sample[[slice(None), 0]] * 2.0 + sample[[slice(None), 1]], 1.0).log_prob(observe[[slice(None), 1]])
        log_likeli += Normal(sample[[slice(None), 0]] * 5.0 + sample[[slice(None), 1]], 1.0).log_prob(observe[[slice(None), 4]])
        log_likeli += Normal(sample[[slice(None), 0]] * 6.0 + sample[[slice(None), 1]], 1.0).log_prob(observe[[slice(None), 5]])
        log_likeli += Normal(sample[[slice(None), 0]] * 1.0 + sample[[slice(None), 1]], 1.0).log_prob(observe[[slice(None), 0]])
        log_likeli += Normal(sample[[slice(None), 0]] * 3.0 + sample[[slice(None), 1]], 1.0).log_prob(observe[[slice(None), 2]])
        log_likeli += Normal(sample[[slice(None), 0]] * 4.0 + sample[[slice(None), 1]], 1.0).log_prob(observe[[slice(None), 3]])
        return log_likeli

    def log_prior(self, sample):
        log_prior = torch.zeros(sample.shape[0])
        log_prior += Normal(0.0, 10.0).log_prob(sample[[slice(None), 1]])
        log_prior += Normal(0.0, 10.0).log_prob(sample[[slice(None), 0]])
        return log_prior
```