MDPI

Article

GraphPPL.jl: A Probabilistic Programming Language for Graphical Models

Wouter W. L. Nuijten [1,*], Dmitry Bagaev [1] and Bert de Vries [1,2]

[1] Department of Electrical Engineering, Eindhoven University of Technology, 5612 AE Eindhoven, The Netherlands
[2] GN Hearing, 5612 AB Eindhoven, The Netherlands
* Correspondence: w.w.l.nuijten@tue.nl

Abstract: This paper presents GraphPPL.jl, a novel probabilistic programming language designed for graphical models. GraphPPL.jl uniquely represents probabilistic models as factor graphs. A notable feature of GraphPPL.jl is its model nesting capability, which facilitates the creation of modular graphical models and significantly simplifies the development of large (hierarchical) graphical models. Furthermore, GraphPPL.jl offers a plugin system to incorporate inference-specific information into the graph, allowing integration with various well-known inference engines. To demonstrate this, GraphPPL.jl includes a flexible plugin to define a Constrained Bethe Free Energy minimization process, also known as variational inference. In particular, the Constrained Bethe Free Energy defined by GraphPPL.jl serves as a potential inference framework for numerous well-known inference backends, making it a versatile tool for diverse applications. This paper details the design and implementation of GraphPPL.jl, highlighting its power, expressiveness, and user-friendliness. It also emphasizes the clear separation between model definition and inference while providing developers with extensibility and customization options. This establishes GraphPPL.jl as a high-level user interface language that allows users to create complex graphical models without being burdened with the complexity of inference while allowing backend developers to easily adopt GraphPPL.jl as their frontend language.

Keywords: Bayesian inference; factor graphs; nested models; probabilistic programming

Citation: Nuijten, W.W.L.; Bagaev, D.; de Vries, B. GraphPPL.jl: A Probabilistic Programming Language for Graphical Models. *Entropy* **2024**, *26*, 890. https://doi.org/10.3390/e26110890

Received: 19 September 2024
Revised: 10 October 2024
Accepted: 12 October 2024
Published: 22 October 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Probabilistic programming languages (PPLs) are designed to simplify the complexities of probabilistic inference. They offer a high-level modeling language and an inference engine that automates various inference algorithms, including Bayesian methods. While PPLs often facilitate Bayesian inference, they can also support a broader range of probabilistic models and inference algorithms. Effective use of these PPLs usually requires extensive knowledge of the underlying principles, but recent advances in PPLs, such as NumPyro [1] and Turing.jl [2], aim to reduce this burden on the user. These languages have significantly advanced the field of probabilistic reasoning, enabling applications across diverse domains such as toxicology [3], geophysics [4], and cognitive sciences [5]. They provide users with powerful inference results without the need to customize inference software, allowing researchers without deep knowledge of probabilistic information processing to effortlessly analyze observed data. With the help of PPLs, users only need to interact with a high-level interface and define the generative models that govern data generation.

Once the model has been defined, the extra difficulty lies in achieving feasible inference within limited computational resources. Numerous methods to automate the inference process have been suggested, such as sampling-based methods [2,6,7] or variational optimization-based methods [8–10]. However, most automated inference solutions are intertwined with their model specification component. This makes it challenging to use

the same probabilistic model with different inference methods and necessitates redefining the model for each specific inference method.

This paper addresses this issue by introducing GraphPPL.jl, a PPL that enables users to define a probabilistic model without making assumptions about a specific inference process. The frontend can generate a model as a graph and provides a plugin system to add inference-specific information if needed. To illustrate this capability, GraphPPL.jl includes a variational inference plugin, which injects additional information into the standard model structure and enables support for variational inference methods, such as the Constrained Bethe Free Energy (CBFE) optimization process in RxInfer [10].

The selection of CBFE minimization to demonstrate the capabilities of GraphPPL.jl is deliberate. It has been demonstrated that numerous traditional inference approximation methods can be expressed as the minimization of a CBFE functional. Nonetheless, GraphPPL.jl does not impose any particular assumptions regarding the inference process and can potentially act as a frontend for various other inference techniques, including HMC [11], NUTS [12], and more. GraphPPL.jl outputs a data structure representing the generative model without relying on any inference backend. As far as we know, no other PPL currently provides this feature.

Furthermore, GraphPPL.jl facilitates the definition of nested models, allowing large and complex models to be built from smaller, simpler ones. This is crucial because the exponential increase in computing power has made Bayesian inference, once extremely costly, feasible on standard computer chips. This advancement has broadened the range of generative models, with recent techniques scaling to thousands of variables [2,13], making the specification of such large models even more challenging. When dealing with complex systems, dividing them into separate independent modules (e.g., functions or classes in programming languages) is common practice. However, a concept similar to modularity in probabilistic programming is still underdeveloped, and GraphPPL.jl aims to bridge this gap.

The structure of this paper is outlined as follows:

- Section 2 reviews factor graphs, variational Bayesian inference, and CBFE minimization. Additionally, we discuss why CBFE minimization on a factor graph is considered a general, customizable inference method.
- Section 3 discusses related works.
- Section 4.2 delves into the core design philosophy of GraphPPL.jl and explains the rationale for implementing the CBFE minimization plugin as the default in GraphPPL.jl.
- Section 4.3 introduces the `@model` macro and syntax, which serves as the primary entry point for creating any GraphPPL.jl model.
- Section 4.4 showcases how models defined with the `@model` macro can be reused in larger models, thereby adding modularity to the language.
- Section 4.7 presents the `@constraints` macro, which specifies factorization and functional-form constraints on the variational posterior for the inference engine. We define a clear and intuitive constraint language. Both `@model` and `@constraints` exemplify the integration of models specified within GraphPPL.jl with a particular inference backend.
- Section 5 demonstrates the integration of GraphPPL.jl in the RxInfer.jl inference ecosystem with an inference example.

2. Background

This section aims to provide an overview of Bayesian inference, variational inference, factor graphs, and CBFE minimization. We regard understanding these concepts as essential to appreciating the language design of GraphPPL.jl. However, this is not meant to review the topics fully. Instead, we refer interested readers to [14] where these concepts are discussed in more detail.

2.1. Bayesian Inference

Consider a factorized generative model $p(\boldsymbol{x}, \boldsymbol{z}) = p(\boldsymbol{x} \mid \boldsymbol{z})p(\boldsymbol{z})$ with observations $\boldsymbol{x}$ and unobserved latent states $\boldsymbol{z}$. Note that the factorization of the model $p(\boldsymbol{x}, \boldsymbol{z})$ can be reversed by Bayes' rule as

$$p(\boldsymbol{x}, \boldsymbol{z}) = \underbrace{p(\boldsymbol{x} \mid \boldsymbol{z})}_{\text{likelihood}} \underbrace{p(\boldsymbol{z})}_{\text{prior}} = \underbrace{p(\boldsymbol{z} \mid \boldsymbol{x})}_{\text{posterior}} \underbrace{p(\boldsymbol{x})}_{\text{evidence}}, \tag{1}$$

where the evidence can be computed by

$$p(\boldsymbol{x}) = \int p(\boldsymbol{x}, \boldsymbol{z}) \mathrm{d}\boldsymbol{z}. \tag{2}$$

In a probabilistic modeling context, the first factorization postulates the model as a product of likelihood and prior. Bayesian inference aims to derive the second factorization as the product of posterior and evidence. Both the Bayesian posterior and the model evidence are very interesting quantities, as the posterior describes the distribution over latent states $\boldsymbol{z}$ after observing data $\boldsymbol{x}$, and the model evidence quantifies the model's performance in explaining the observed data. Unfortunately, the integration over latent states in Equation (1) quickly becomes intractable, as the computational complexity of the integration is exponential in the dimensionality of $\boldsymbol{z}$. Therefore, much effort has been devoted to finding a manageable and accurate approximation of Bayesian inference when computing the model evidence, which is computationally not feasible with available resources. One of the solutions involves further factorizing the likelihood and prior distributions, a concept that we will explore next.

2.2. Factor Graphs

In this section, we explore factor graphs and their application in expressing generative models. For simplicity, we write our generative model as $p(\boldsymbol{s})$ without explicitly decomposing the variables $\boldsymbol{s}$ into $\boldsymbol{z}$ and $\boldsymbol{x}$. Assume that we can further factorize our model as follows:

$$p(\boldsymbol{s}) = \prod_{i}^{m} f_i(\boldsymbol{s}_i), \tag{3}$$

meaning p adheres to a factorization into m factor functions $\{f_a, f_b, \cdots f_N\}$, where each of the factor functions is a function over a subset of the variables $\boldsymbol{s}$. For example, we could factorize a distribution over four variables as follows:

$$p(s_1, s_2, s_3, s_4) = f_a(s_1) f_b(s_1, s_2, s_3) f_c(s_2, s_4) f_d(s_3) f_e(s_4). \tag{4}$$

If the generative model adheres to a similar factorization, we can draw a computation graph of the generative model, where the vertices represent the factor functions and the edges represent the variables. This visual representation of a factorized generative model is called a Forney-style factor graph (FFG) [15]. An FFG is mathematically represented as a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of vertices, and $\mathcal{E}$ is a set of edges. We use $(a, b, c \ldots)$ as indices for vertices and $(i, j, k \ldots)$ as indices for edges. An example of the FFG corresponding to Equation (4) can be seen in Figure 1.

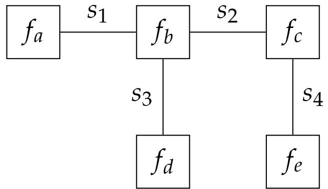

Figure 1. Forney-style factor graph corresponding to the factorization in Equation (4).

Computing the Bayesian posterior and marginal distributions in a factorized distribution is computationally easier than the integration problem in Equation (1). This is because we can use the factorization in Equation (4) to reduce the dimensionality of the integration as follows:

$$\begin{aligned} p(s_2) &= \int p(s_1, s_2, s_3, s_4) \mathrm{d}s_1 \mathrm{d}s_3 \mathrm{d}s_4 \\ &= \int f_a(s_1) f_b(s_1, s_2, s_3) f_c(s_2, s_4) f_d(s_3) f_e(s_4) \mathrm{d}s_1 \mathrm{d}s_3 \mathrm{d}s_4 \\ &= \underbrace{\int f_a(s_1) \int f_b(s_1, s_3, s_3) f_d(s_3) \mathrm{d}s_3 \mathrm{d}s_1}_{\overrightarrow{\mu}(s_2)} \cdot \underbrace{\int f_c(s_2, s_4) f_e(s_4) \mathrm{d}s_4}_{\overleftarrow{\mu}(s_2)} . \end{aligned} \tag{5}$$

Here, we can see that the integral to compute $p(s_2)$ can be split into two separate integration operations. We can also visually represent this separation in an FFG, as illustrated in Figure 2. Here, we see that $\overrightarrow{\mu}(s_2)$ corresponds to computing the marginal distribution of s_2 in the subgraph enclosed by the left dotted box. Similarly, $\overleftarrow{\mu}(s_2)$ corresponds to marginalizing over s_4 in the subgraph enclosed by the right dotted box. This simple example illustrates that we can group individual factor nodes together. After marginalization over the internal variables in this group, we obtain the marginal distributions over the variables that connect to nodes outside of this group. We call these "boundary-crossing" edges the Markov blanket of a group of factors. Visually, we can represent this by drawing a box around a group of factor nodes, as shown in Figure 2. This process is called "closing the box" [15] and allows us to represent any FFG as a hierarchy of nested models. By iteratively defining FFGs and their corresponding Markov blankets, we can use previously defined FFGs as nodes in new factor graphs, giving us a modular composition of the final FFG.

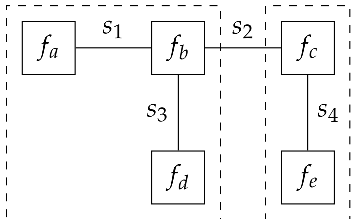

Figure 2. FFG representation of the computation of the marginal distribution over s_2 by computing and multiplying the marginal distributions of two submodels.

2.3. Variational Inference

In Bayesian inference, we are interested in determining posterior distributions over unobserved latent states given observations, adhering to some generative model. Variational inference is concerned with approximating $p(\boldsymbol{z} \mid \boldsymbol{x})$ by introducing a candidate distribution $q(\boldsymbol{z})$ and minimizing the variational free energy (VFE):

$$\begin{aligned} F[q] &\triangleq \int q(\boldsymbol{z}) \log \frac{q(\boldsymbol{z})}{p(\boldsymbol{x}, \boldsymbol{z})} \mathrm{d}\boldsymbol{z} \\ &= \underbrace{\int q(\boldsymbol{z}) \log \frac{q(\boldsymbol{z})}{p(\boldsymbol{z}|\boldsymbol{x})} \mathrm{d}\boldsymbol{z}}_{D_{KL}[q(\boldsymbol{z}) \mid\mid p(\boldsymbol{z} \mid \boldsymbol{x})]} - \underbrace{\log p(\boldsymbol{x})}_{\text{log evidence}} . \end{aligned} \tag{6}$$

Here, D_{KL} is the Kullback–Leibler divergence [16], which is a measure of how one probability distribution q is different from a reference probability distribution p. In other words, it measures how much information is lost when q is used to approximate p, making it a

suitable measure for approximate inference. By choosing a variational family of tractable distributions $\mathcal{Q}$, we can define the variational posterior distribution as follows:

$$q^* = \arg\min_{q \in \mathcal{Q}} F[q] \quad (7)$$

From the VFE in Equation (6), we can see that q^* will approximate $p(\boldsymbol{z} \mid \boldsymbol{x})$, and $F[q^*]$ will approximate $\log p(\boldsymbol{x})$. In this way, variational inference transforms the integration problem in Equation (1) into an optimization problem in Equation (7). We can then apply traditional (un)constrained optimization methods to $\mathcal{Q}$ to find an approximate posterior distribution. We refer readers to [17] for an extensive review of variational inference.

2.4. Constrained Bethe Free Energy

The factorization of Equation (3) also has consequences for the VFE since the denominator in Equation (6) now becomes a factorized distribution. It is common practice to factorize the candidate distribution $q(\boldsymbol{s})$ similarly to the generative model. This means that when we draw the FFG representation of the generative model, we assign candidate functions, which we call beliefs, to the nodes and edges of the graph, which, when multiplied, correspond to our candidate distribution. Therefore, we define the family $\mathcal{Q}_{\mathcal{G}}$ as follows:

$$q(\boldsymbol{s}) \in \mathcal{Q}_{\mathcal{G}} \implies q(\boldsymbol{s}) = \prod_{a \in \mathcal{V}(\mathcal{G})} q_a(\boldsymbol{s}_a) \left(\prod_{i \in \mathcal{E}(\mathcal{G})} q_i(s_i) \right)^{-1}. \quad (8)$$

Here, $q_a(\boldsymbol{s}_a)$ are the node-wise marginal beliefs defined for every node a, and $q_i(\boldsymbol{s}_i)$ are the edge-wise marginal beliefs defined for every edge i. The division of all edge beliefs is an artifact of the fact that we always count every edge twice in the first multiplication; hence, we need a correcting term [18]. Suppose both the generative model and variational distribution follow these factorizations. In that case, it can be shown that the VFE can be rewritten as the Bethe Free Energy (BFE) [19], which is defined as

$$F[q] \triangleq \sum_{a \in \mathcal{V}(\mathcal{G})} \underbrace{\int q_a(\boldsymbol{s}_a) \log \frac{q_a(\boldsymbol{s}_a)}{f_a(\boldsymbol{s}_a)} \mathrm{d}\boldsymbol{s}_a}_{F[q_a]} + \sum_{i \in \mathcal{E}(\mathcal{G})} \underbrace{\int q_i(s_i) \log \frac{1}{q_i(s_i)} \mathrm{d}s_i}_{H[q_i]}. \quad (9)$$

The BFE calculates a local VFE term for each node in the FFG, allowing us to minimize the VFE at each node individually. However, because the BFE divides the global optimization problem into multiple local optimization problems, it is necessary to impose constraints on the variational posterior $q(\boldsymbol{s})$ to ensure that, after optimizing the BFE, the result is still a valid probability distribution. The required constraints are normalization and marginalization constraints:

$$\begin{aligned} &\int q_a(\boldsymbol{s}_a) \mathrm{d}\boldsymbol{s}_a = 1, \text{for all } a \in \mathcal{V}(\mathcal{G}) \\ &\int q_a(\boldsymbol{s}_a) \mathrm{d}\boldsymbol{s}_{a \setminus i} = q_i(s_i), \text{for all } a \in \mathcal{V}(\mathcal{G}), i \in \mathcal{E}(\mathcal{G}). \end{aligned} \quad (10)$$

Minimization of the BFE over a constrained variational family is called Constrained Bethe Free Energy (CBFE) minimization, and it can be shown that the stationary points of the BFE correspond to the local minima of the belief propagation algorithm [20,21]. By introducing additional constraints on the variational posterior, many popular inference algorithms are recognized as instances of CBFE minimization [14]. For example, we obtain the mean-field algorithm by further constraining the node beliefs $q_a(s_a)$ to be fully factorized. Therefore, a PPL that allows users to specify a CBFE of their generative model gives the user a powerful language to define generative models and influence the inference results.

3. Related Works

Probabilistic programming languages aim to provide users with an intuitive way to create probabilistic models for inference. In general, existing PPLs are designed with the models on which we can perform efficient inference in mind. Although this places constraints on the expressiveness of the modeling language, this approach has resulted in successful languages such as BUGS [6,22], STAN [7,23], and CHURCH [24]. In some languages, such as IBAL [25], inference details are part of the model specification, further intertwining model definition and inference implementation. Over time, numerous probabilistic programming libraries have emerged, each contributing unique perspectives to the field [24–28]. In recent years, the increasing popularity of *Python* and its rich deep learning community have paved the way for PPLs in *Python*, using a deep learning engine as a backend, such as Pyro [27] and TensorFlow Probability [29]. For a more comprehensive review of probabilistic programming languages, we refer readers to [30].

An effort was made to construct a universal PPL that does not depend on inference details through Turing.jl [2], a PPL in the *Julia* [31] language aimed at providing universal inference through sampling. With the implementation of DynamicPPL.jl language, considerable effort was made to ensure the separation between the modeling language and the inference process. The increasing popularity of Turing.jl suggests the need for a universal modeling language that is not restricted by the details of inference. Turing.jl also makes an effort toward modularity; with the `@submodel` macro, existing Turing.jl models can be called within larger models. However, we can only fit the submodel in the forward generative direction, so this is not a general solution for modular probabilistic programming.

The utility of functions in traditional programming languages is well recognized, primarily due to their ability to facilitate code reuse, reduce errors in program definitions, and minimize the number of code lines. This concept parallels the realm of nested probabilistic models within PPLs. The necessity for nested PPLs arises from their potential to conserve lines of code, decrease errors in model definitions, and enhance reusability, mirroring the benefits offered by functions in conventional programming languages. This notion of modularity and reusability in programming is not just a matter of convenience or good software engineering practice; it also resonates with our understanding of intelligent systems, both biological and artificial.

4. The GraphPPL.jl Engine

In this section, we elaborate on the philosophy and details of the GraphPPL.jl (https://github.com/ReactiveBayes/GraphPPL.jl (accessed on 18 September 2024), documentation available at https://reactivebayes.github.io/GraphPPL.jl/stable/(accessed on 18 September 2024)) package, explaining the design choices and highlighting its distinguishing features. We also explain the rationale behind focusing specifically on the CBFE for the inference backend integration.

We implemented GraphPPL.jl in the *Julia* [31] programming language. The selection of *Julia* as the programming language is primarily due to two key factors. First, *Julia* offers high performance tailored for scientific computing. Given the computational demands of Bayesian inference, a performance-centric language is essential. While the PPL itself may not be heavily impacted by this computational load, having both the user language and the inference backend in the same language provides considerable benefits. Second, *Julia*'s meta-programming capabilities enable us to create a custom syntax that is converted into standard *Julia* code. This allows us to enhance the *Julia* language with the specific features and operators required for probabilistic programming.

4.1. Representing Graphical Models

GraphPPL.jl offers a frontend for defining probabilistic models, providing the flexibility to accommodate various inference backends. These probabilistic models are represented as graphs, enabling visualization and inspection of their properties without committing to a particular inference method. Models can be stored in memory or transmitted over

the internet to a standalone device for later use. The graph-based representation allows models to be broken down into smaller components, or submodels, which can be reused in more complex hierarchical probabilistic models. The choice of a factor graph representation allows for an analysis of the created generative model, and inference backends can match subgraphs against known models to employ faster and more tractable inference procedures [14].

Nonetheless, GraphPPL.jl recognizes that the main objective is to perform inference on the specified model. To facilitate the integration of inference methods, GraphPPL.jl employs a plugin system that allows for the addition of extra information to the graph required by specific inference backends. For instance, this information might include specific hyperparameters for sampling-based approaches like HMC or constraints for variational inference methods. As an example of such seamless integration, GraphPPL.jl integrates with a particular inference method called the Constrained Bethe Free Energy (CBFE) framework. In Section 4.7, we will showcase the integration of variational constraints in GraphPPL.jl, allowing CBFE definition.

4.2. Language Philosophy

In designing GraphPPL.jl, we adopted a usability-centric approach that focuses primarily on providing high-level model specification capabilities that integrate with various inference operations without depending on a specific inference backend. With this usability-centric approach, we aim to balance expressiveness, ease of use, and learnability, ensuring that novice and experienced users can leverage the language effectively without being burdened by a steep learning curve during their initial acquaintance.

The philosophy of GraphPPL.jl rests on a few requirements:

- Creating probabilistic models with GraphPPL.jl should resemble drafting a mathematical description of a generative model. Similar to Turing.jl [2] and BUGS [22], the core language of a GraphPPL.jl model should closely match the mathematical representation of the generative model.
- GraphPPL.jl models should be as readable as *Julia* programs. Drawing inspiration from PyTorch [32] and their approach of treating 'deep learning models as *Python* programs', we want GraphPPL.jl models to have the same readability and feel as *Julia* programs. As a result, any GraphPPL.jl model should be usable as a component within a larger GraphPPL.jl model.
- A materialized GraphPPL.jl model should be compatible with various inference backends; for example, CBFE minimization. The model should be extendable to inject all required information for any inference backend to perform Bayesian inference.

4.3. The Model Specification Language

To specify a model in a language that is as close as possible to the mathematical representation of a model, we employ *Julia*'s [31] powerful meta-programming functionality to create our own syntax. By creating our own syntax using the `@model` macro, we encapsulate all the necessary logic for constructing a GraphPPL.jl model. A consequence of using *Julia*'s macro functionality is that we extend the syntax of *Julia*, meaning our language accepts any regular *Julia* code while extending the *Julia* syntax to allow probabilistic modeling. We begin by constructing a model that characterizes a sequence of coin tosses to introduce the model syntax. The model for n coin tosses is defined in Equation (11).

$$\begin{aligned} \theta &\sim \mathrm{Beta}(1,1) \\ y_i &\sim \mathrm{Bernoulli}(\theta),\ \text{for all } i \in [1\,..\,n] \end{aligned} \tag{11}$$

The `@model` syntax accepts a *Julia* function. The function's arguments are all data structures external to the model, such as the data, but we could also include priors or hyperparameters. These arguments are called interfaces because they connect the model's

exterior with its interior. Since our model receives data y, we begin our model definition by creating a function: `@model function coin_toss(y)`.

The `@model` macro exposes a GraphPPL.jl-specific operator: the `~`. This operator creates a new factor node and a variable in the factor graph. For example, the statement `θ~ Beta(1, 1)` creates a `Beta` factor node and a θ variable, along with two variables representing the constant `1`. It connects these to the `Beta` node. The subgraph created by this statement can be seen in Figure 3.

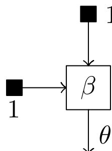

Figure 3. Subgraph created by the statement `θ~ Beta(1, 1)` in the `@model` macro.

The `~` operator in `GraphPPL.jl` checks whether the factor being created is stochastic or deterministic. If it is deterministic and the arguments are known at model construction time, GraphPPL.jl will not create a factor node but will execute the function and return the result. For example, calling `a ~ norm([1, 2, 3])` in `GraphPPL.jl` will transform to `a = norm([1, 2, 3])`, making the constant `a` available for the rest of the model specification.

As the model macro accepts *Julia* syntax, we can combine the `~` operator with regular *Julia* syntax to write the rest of the model. The complete model for the coin tosses defined in Equation (11) can be found in Code Block 1.

```
@model function coin_toss(y)
    θ ~ Beta(1, 1)
    for i in eachindex(y)
        y[i] ~ Bernoulli(θ)
    end
end
```

Code Block 1. GraphPPL.jl code for the coin toss model defined in Equation (11).

The `~` operator is not the only way to create factor nodes and variables in the underlying factor graph; we can use the `:=` operator, which is an alias of the `~` operator, to denote deterministic relations. Furthermore, GraphPPL.jl supports compound statements and will unroll any statement on the right-hand side of the `~` operator to create all factor nodes. To illustrate this, we implement the Gaussian-with-Controlled-Variance (GCV) [33] model, a fundamental building block of the widely used hierarchical Gaussian filter [34]. The GCV model, with inputs `x`, `ω`, `κ`, and `z`, is defined as

$$y \sim \mathcal{N}(x, \exp(\kappa * z + \omega))\,. \tag{12}$$

Two GraphPPL.jl model definitions, both defining the same model, can be seen in Code Blocks 2 and 3. The model definition in Code Block 2 uses a compound statement to create `Normal`, `exp`, `+`, and `*` factor nodes in the same line. The model definition in Code Block 3 uses the `:=` as an alias for the `~` operator to denote deterministic relations. Both model definitions create the same factor graph, as shown in Figure 4. For both models, `x`, `ω`, `κ`, and `z` are inputs to the model, and `y` is the output. We use these five variables as interfaces, which are the arguments of the model function.

```
@model function gcv(y, x, ω, κ, z)
    y ~ Normal(x, exp(κ * z + ω))
end
```

Code Block 2. The GCV model from Equation (12) implemented with compound statements.

```
@model function gcv(y, x, ω, κ, z)
    log_σ :=  κ * z + ω
    σ := exp(log_σ)
    y ~ Normal(x, σ)
end
```

Code Block 3. The GCV model from Equation (12) implemented with deterministic statements.

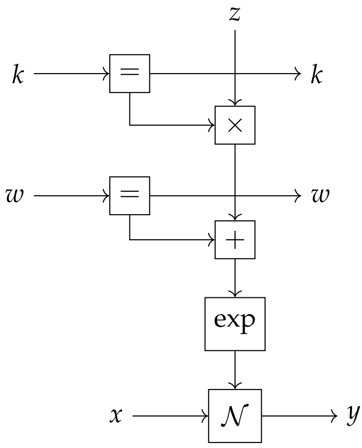

Figure 4. A Forney-style factor graph representation of the Gaussian-with-Controlled-Variance (GCV) model created by the model definitions in Code Blocks 2 and 3.

4.4. Modular Definition and Usage of Models

Generative models for real-world processes can become unwieldy and large. An example can be seen in Figure 5, where we visualize a hierarchical Gaussian filter of depth 3, and the factor graph representation of the generative model for the first three data points is depicted. While we only visualize three data points, we usually want to incorporate hundreds to thousands of data points [14,35]. The three-level HGF is most often used in the literature [34,35]; however, the model is not limited to three layers. Recreating this model in previously discussed PPLs involves manually creating all nodes, a cumbersome and error-prone process that results in unnecessarily long model specifications. In contrast, in GraphPPL.jl, we can define models and use them as submodels of other GraphPPL.jl models. An example of this can be seen in Code Block 4, which uses the `gcv` submodel to chain a Gaussian-with-Controlled-Variance state transition with a Gaussian likelihood model. Note that when invoking `gcv`, we supply all but one interface through named keyword arguments. GraphPPL.jl can recognize which interface to the `gcv` submodel is missing and will attach this interface to the `x_next` variable that is on the left-hand side of the `~` operator. The factor graph representation of this model can be seen in Figure 6. For visual clarity, we have grouped k, w, and x_t as the state variable s_t. Although this model describes a complex state transition and a likelihood model, the FFG is remarkably easy to read.

```
@model function gcv_lm(y, x_prev, x_next, z, ω, κ)
    x_next ~ gcv(x = x_prev, z = z, ω = ω, κ = κ)
    y ~ Normal(x_next, 1)
end
```

Code Block 4. Model chaining a `gcv` submodel with a Gaussian likelihood.

Figure 5. A 3-layer hierarchical Gaussian filter drawn as a Forney-style factor graph. The first 3 time steps are depicted.

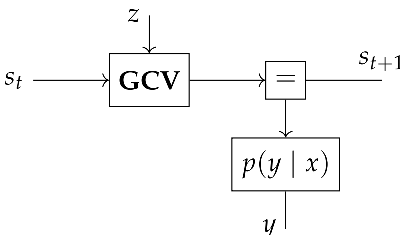

Figure 6. The Gaussian-with-Controlled-Variance Likelihood Model (GCV-LM) submodel.

With the `gcv` and `gcv_lm` models defined, we can now simplify the HGF model shown in Figure 5 to use these submodels. The GraphPPL.jl code for creating the HGF model can be seen in Code Block 5. For each data point in y, we compute state transitions using the `gcv` and `gcv_lm` submodels to fully specify the likelihood model. Note that the number of lines of GraphPPL.jl code we have written so far is still fewer than 20. In this definition of `hgf`, we can pass constants or distributions as arguments ξ, ω, and κ, which will serve as

either parameters or priors to these parameters. Note that in Code Block 5, we have used the `new` syntax, which creates a new variable in a vector of random variables if it does not exist. With this syntax, we can specify an observation through a state transition and create both variables on the same line.

The FFG of the HGF model using our defined submodels can be viewed in Figure 7. In this graph, we obtain a clearer picture of the computation flow and can better communicate to readers the intentions of the model and the modeling decisions made. More importantly, if we wanted to use a different likelihood model instead of the Gaussian likelihood specified in Code Block 4, we would only have to change this submodel, and every instance of `gcv_lm` in the `hgf` model would change accordingly. This contrasts with widely used PPLs, where the likelihood model for every timestep must be changed individually.

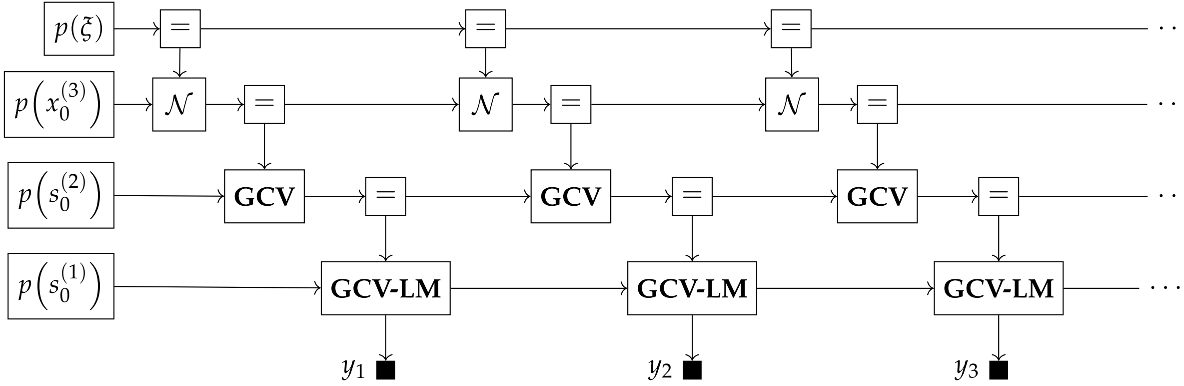

Figure 7. The hierarchical Gaussian filter shown in Figure 5 drawn as a composite model using the **GCV** and **GCV-LM** submodels.

The nested model specification in GraphPPL.jl enables researchers to easily build intricate models, facilitating the exploration and creation of sophisticated probabilistic models without sacrificing readability or maintainability. This hierarchical model definition and composition stand out as defining features, representing a significant leap in usability and model organization, aligning with established programming design principles such as the Single Responsibility Principle [36].

```
@model function hgf(y, ξ, ω, κ)

    # Specify priors

    x_1[1] ~ Normal(0, 1)
    x_2[1] ~ Normal(0, 1)
    x_3[1] ~ Normal(0, 1)

    # Specify generative model

    for i in eachindex(y)
        x_3[i+1] ~ Normal(x_3[i], ξ)
        x_2[i+1] ~ gcv(x = x_2[i], z = x_3[i], ω = ω[2], κ = κ[2])
        y[i] ~ gcv_lm(x_prev = x_1[i], x_next = new(x_1[i+1]), z = x_2[i],
            ω = ω[1], κ = κ[1])
    end
end
```

Code Block 5. The hierarchical Gaussian filter defined in GraphPPL.jl using the `gcv` and `gcv_lm` submodels.

4.5. Example: Bayesian Neural Network

Now that we have introduced the core functionality of the language, we can demonstrate an advanced example. Although GraphPPL.jl is not a language specifically designed for neural networks, it is expressive enough to define Bayesian neural networks in a few lines of code. As in the previous section, we build a larger model starting with smaller submodels. First, we start by defining a `neural_dot` submodel that takes a vector `input`

and a vector of weights `w`, returning the dot product between `input` and `w` while applying an activation function, as shown in Code Block 6. This, together with the definition of a weight vector, defines an artificial neuron, as seen in Code Block 7.

```
@model function neural_dot(out, input, w)
    c[1] := input[1] * w[1]
    for i in 2:length(input)
        c[i] := c[i - 1] + input[i] * w[i]
    end
    out := relu(c[end])
end
```

Code Block 6. Dot product with nonlinearity applied as the basic building block of a neural network.

```
@model function neuron(input, out)
    local w
    for i in 1:length(input)
        w[i] ~ Normal(0.0, 1.0)
    end
    out ~ neural_dot(input = input, w = w)
end
```

Code Block 7. Artificial neuron.

We can now define a single fully connected layer consisting of neurons that share the same input and produce a single output element (Code Block 8). With this defined, we can create a multilayer perceptron or fully connected neural network. The code to create a four-layer Bayesian neural network can be seen in Code Block 9. This example again highlights how GraphPPL.jl can construct intricate probabilistic models by combining smaller, simpler ones.

```
@model function neural_network_layer(input, out, n)
    for i in 1:n
        out[i] ~ neuron(input = input)
    end
end
```

Code Block 8. Fully connected neural network layer defined as a composition of neurons.

```
    @model function neural_net(input, out)
        h1 ~ neural_network_layer(input = input, n = 10)
        h2 ~ neural_network_layer(input = h1, n = 16)
        h3 ~ neural_network_layer(input = h2, n = 32)
        out ~ neural_network_layer(input = h3, n = 2)
    end
```

Code Block 9. Example of a multilayer perceptron.

4.6. Extensibility

GraphPPL.jl does not operate only with a predetermined set of factor nodes. Instead, it contains an extensible architecture that relies on *Julia*'s multiple dispatch functionality to determine when to materialize a factor node for a function and when to evaluate a function with the supplied arguments deterministically. By specifying whether a function is deterministic or stochastic, developers of inference backends can introduce new nodes to be integrated into probabilistic modeling.

The power of this design becomes evident when considering the `Normal` distribution used in Code Block 2. Rather than being part of the native GraphPPL.jl, the `Normal` distribution is part of an extension that becomes available upon loading the Distributions.jl package. This extension defines the essential functionality required to create a node for each distribution in Distributions.jl, expanding the range of distributions accessible for probabilistic modeling without explicitly relying on Distributions.jl out of the box.

Some inference backends (such as ReactiveMP.jl [37]) might expose different implementations of well-known distributions for computational efficiency. For example, ReactiveMP.jl exposes four implementations of the Normal distribution: `NormalMeanVariance`, `NormalWeightedMeanVariance`, `NormalMeanPrecision`, and `NormalWeightedMeanPrecision`. Following the mechanism described above, developers of inference backends could create specific factor nodes for each of these implementations. However, this introduces two problems. First, it exposes an implementation detail (the user should not have to care about the parameterization used for the normal distribution), and it breaks the convenient syntax of Code Block 2, where we could call `Normal(x, σ)`. To address these issues, GraphPPL.jl contains an elaborate aliasing system based on keyword arguments that developers can customize to their needs. With this aliasing system, developers can transform `Normal(ν = 1, τ = 1)` to `NormalWeightedMeanPrecision(1, 1)`. With this aliasing system, developers can hide implementation details from users and maintain a high-level intuitive interface.

GraphPPL.jl is a backend-agnostic modeling language; therefore, a GraphPPL.jl model does not contain any information concerning the inference process by default. However, some inference backends, like ReactiveMP.jl, require additional information along with the structure of the model to conduct inference. To accommodate these needs, GraphPPL.jl exposes an elaborate plugin system. This system allows inference backend developers to include custom code and information, effectively enabling GraphPPL.jl to gather all necessary information for inference. The plugin required to run inference using the ReactiveMP.jl backend is the variational constraints plugin, which we elaborate on below.

4.7. Constraint Specification

In Section 2, we explored the utility of CBFE minimization as an effective method for Bayesian inference. Utilizing the CBFE framework necessitates the user to supply a series of constraints on the variational posterior. These constraints can be seen as additional information embedded within the model's graph structure. The variational constraints plugin in GraphPPL.jl is specifically designed to handle such scenarios, providing support for the ReactiveMP.jl backend.

GraphPPL.jl features a constraint specification language that integrates seamlessly with the `@model` macro. This language supports two constraints that can be applied to the variational posterior: Factorization Constraints (FCs) and Functional Form Constraints (FFCs). FCs separate the dependencies among specified variables in the variational posterior [38,39]. For instance, consider a generative model with variables `x`, `y`, and `z`, where we aim to impose a mean-field assumption on these variables in the variational posterior $q(x, y, z)$. The FC `q(x, y, z) = q(x)q(y)q(z)` would represent this assumption and inject this information into the graph, which can later be utilized by an inference backend. FFCs, on the other hand, restrict the marginal distribution of a specified variable in the variational posterior to a particular functional form. For example, if we want the marginal distribution for `x` to follow a Beta distribution, we can include the constraint `q(x) :: Beta` in our constraint specification. This syntax is implemented in the `@constraints` macro, offering an advanced language for specifying constraints on the generated factor graph for a model.

The constraint language implemented in GraphPPL.jl has one additional feature that harmonizes with the rest of the PPL: By opening **`for`** code blocks, one can access the constraints for variables in submodels in the factor graph. For example, a mean-field constraint on the ω, κ, and `u` variables in every `gcv` instance in the `hgf` model of Code Block 5 can be imposed using the constraint specification seen in Code Block 10.

```
@constraints begin
    for q in gcv
        q(ω, κ, z) = q(ω)q(κ)q(z)
    end
    for q in gcv_lm
        for q in gcv
            q(ω, κ, z) = q(ω)q(κ)q(z)
        end
    end
end
```

Code Block 10. Example constraint specification for an HGF model.

With this syntax, we supply a concise constraint language that still offers significant flexibility and control over the family of variational posterior distributions to consider.

5. Inference Example with the ReactiveMP.jl Backend

In this example, we create a model using GraphPPL.jl with the variational constraints plugin enabled and then perform inference using the ReactiveMP.jl backend. This combination of frontend and backend is accessible in the RxInfer.jl package. We are working with a simple hierarchical state-space model. The FFG of this model is shown in Figure 8. In the FFG diagram, we have omitted constant hyperparameters for distributions to avoid visual clutter. This model describes a random walk with drift, where the random walk also determines the drift. Inside the dotted box is a repeated pattern: two Gaussian distributions are summed, followed by applying another Gaussian distribution, resulting in an observation with a Gaussian likelihood and a Gamma-distributed precision parameter. This model can be used as a nested model in GraphPPL.jl. As seen in Code Block 11, we use the `ssm` submodel twice, which invokes the `ssm_step` submodel on every data point. In this way, we can create a complex factor graph in under 20 lines.

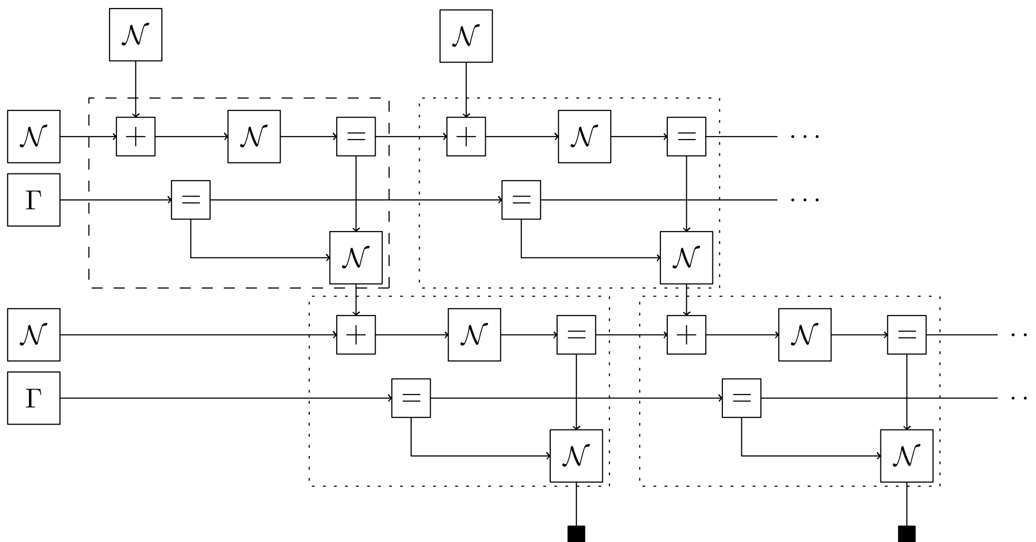

Figure 8. Model used in inference. The dashed box denotes a subgraph that is reused over the entire graph.

Our model assumes a joint dependency between the hidden state drift and the noise in this hidden state drift. However, in our variational posterior distribution, we would like to see these two variables as independent of each other. That is why we impose the factorization constraint `q(x_next, y, precision) = q(x_next, y)q(precision)` in every copy of the `ssm` submodel. Code Block 12 shows how to apply this constraint to all copies of the `ssm` submodel.

```
@model function ssm_step(x_prev, x_next, y, drift, precision)
    x_next_mean := x_prev + drift
    x_next ~ Normal(mean=x_next_mean, variance=10)
    y ~ Normal(mean=x_next, precision=precision)
end

@model function ssm(drift, y)
    observation_precision ~ Gamma(2, 1)
    x[1] ~ Normal(mean=1.0, variance=10.0)
    for i in eachindex(drift)
        y[i] ~ ssm_step(x_prev=x[i], x_next=new(x[i+1]), drift=drift[i],
            precision=observation_precision)
    end
end

@model function hierarchical_ssm(y)
    local upper_drift
    for i in eachindex(y)
        upper_drift[i] ~ Normal(mean=0, variance=1)
    end
    hidden_state_drift ~ ssm(drift=upper_drift)
    y ~ ssm(drift=hidden_state_drift)
end
```

Code Block 11. Simple hierarchical state-space model in GraphPPL.jl.

```
constraints = @constraints begin
    for q in ssm
        for q in ssm_step
            q(x_next, y, o_var) = q(x_next, y)q(o_var)
        end
    end
end
```

Code Block 12. Inference constraints for the hierarchical state-space model.

To run inference on this model, we first have to generate data. We generated 100 data points, as shown in Figure 9a. We then ran inference using RxInfer.jl by calling the `infer` function, as seen in Code Block 13. After running inference with RxInfer.jl, we recovered the hidden state `hidden_state_drift` that drives the change between the hidden states that generate the observations. In Figure 9b, we can see the inference result, showing that we can use GraphPPL.jl with the ReactiveMP.jl backend to automate a sophisticated Bayesian inference process. The code used to generate these images can be found at Github (https://github.com/wouterwln/GraphPPL-demo) (accessed on 18 September 2024). Additional examples can be found in the code blocks in Appendix A or in the RxInfer Examples.

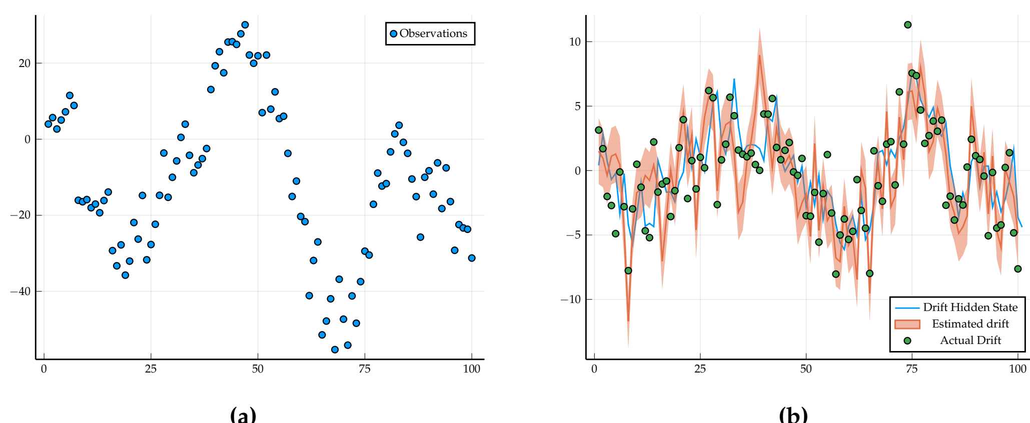

Figure 9. Generated observations and inference result. (**a**) Generated observations. (**b**) Hidden state drift and recovered estimated drift.

```
result = infer(
    model = hierarchical_ssm(),
    iterations = 10,
    data = (y = data,),
    initialization = init,
    constraints = constraints
)
```

Code Block 13. Running inference in the `hierarchical_ssm` model with generated data.

6. Discussion

In this paper, our objective was to design and implement a PPL with the following desiderata/design principles:

- It should be possible to use any GraphPPL.jl model as a submodel in any subsequent model.
- A materialized GraphPPL.jl model should contain all information necessary to perform Bayesian inference. The model should be extendable by backend developers to include additional information.
- A GraphPPL.jl model should look, as much as possible, like the mathematical representation of the generative model, exposing as few implementation details as possible.

In Section 4.4, we introduced modularity, showing how we can use any existing GraphPPL.jl model as a submodel in larger models. This modularity in graphical model construction is versatile and allows for a rich class of models. As seen in Section 5, we can create complex graphical models with readable and interpretable code. The Bayesian Brain Hypothesis [40] suggests that intelligent biological agents are Bayesian reasoning machines, continuously reducing uncertainty about their environments by processing sensory observations. This hypothesis is further supported by the Free-Energy Principle [41], which theorizes that intelligent agents possess a generative model of their environment in their brains. Furthermore, it has been suggested that the main prerequisite for intelligent behavior in biological agents is the hierarchical composition of the agent's generative model into smaller submodels, each aiming to minimize Bayesian surprise independently [42,43]. This hierarchical structure allows for modularity and scalability in cognitive processes, enabling more efficient and flexible adaptation to new information and environments. With the introduction of GraphPPL.jl, we have introduced a language where the nesting of probabilistic models is at the core of its design philosophy, allowing the design of agents inspired by the Free-Energy Principle.

With the variational constraints plugin, GraphPPL.jl can be enabled to fully specify a CBFE. We have seen that many popular inference algorithms can be written as a minimization of a CBFE [14]. In GraphPPL.jl, we realize this by supplying a generative model with a set of inference constraints (Factorization Constraints and Functional Form Constraints) that together fully define a CBFE to be minimized. However, while Factorization Constraints and Functional Form Constraints allow for an expressive constraint language, these are not the only constraints one could apply to the variational posterior. Examples of constraints that are not covered by GraphPPL.jl are Chance Constraints [44] or joint Functional Form Constraints (Functional Form Constraints that constrain the functional form of joint posterior distributions over certain variables instead of constraining the functional form of a single random variable). While an advantage can be gained by supporting a wider class of constraints, we are unaware of any inference backend that supports these constraints, so no effort has been made to expose these constraints in GraphPPL.jl.

One might be interested in performance issues and analysis with GraphPPL.jl, as well as the availability of model statistics such as model comparison with Bayes factors. We point out that GraphPPL.jl is a language for specifying a model with inference constraints but does not execute the inference procedure. In the existing integration with the RxInfer.jl inference engine, we can analyze the performance of the inference procedure, and since

RxInfer.jl approximates the variational free energy, we also have an approximation of the model evidence, which could be used for model comparison.

Future improvements to this work might extend the functionality of the GraphPPL.jl engine, such as having an extended class of constraints available or having an extensive visualization tool with which users can inspect their graphical models.

7. Conclusions

In this paper, we have introduced GraphPPL.jl, a probabilistic programming language for graphical models. GraphPPL.jl represents a probabilistic model as a factor graph and uses a custom syntax to efficiently generate factor graphs from high-level user code. With the plugin system, GraphPPL.jl becomes highly extensible. We have shown the extensibility of GraphPPL.jl by implementing the variational constraints plugin, which allows users to define a Constrained Bethe Free Energy instead of only defining a generative model. With this plugin, we have integrated GraphPPL.jl as the frontend of the ReactiveMP.jl inference backend, which minimizes the Constrained Bethe Free Energy using message-passing. This shows the versatility of GraphPPL.jl and the importance of a backend-agnostic, extensible probabilistic programming language.

GraphPPL.jl introduces a mechanism for model nesting, allowing for modular graphical models and, therefore, greatly reducing the cognitive complexity a user is burdened with when creating large graphical models. In short, GraphPPL.jl is an intuitive, powerful, expressive probabilistic programming language that ensures separation between model definition and inference while still providing extensibility and customizability to developers.

Author Contributions: Conceptualization: W.W.L.N. & D.B., Methodology: W.W.L.N., Software: W.W.L.N. & D.B., Writing, original draft preparation: W.W.L.N., Writing, review and editing: D.B. & B.d.V., Supervision: B.d.V. All authors have read and agreed to the published version of the manuscript.

Funding: This publication is part of the ROBUST project with project number KICH3.LTP.20.006, which is (partly) financed by the Dutch Research Council (NWO), GN Hearing, and the Dutch Ministry of Economic Affairs and Climate Policy (EZK) under the program LTP KIC 2020-2023. This project is also partly financed by Holland High Tech with PPS funding for the AUTO-AR project RVO TKI2112P09.

Institutional Review Board Statement: Not applicable

Data Availability Statement: The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author.

Acknowledgments: The authors want to thank the BIASlab team for the discussions and the testing of the software, and the reviewers for their insightful comments.

Conflicts of Interest: Author Bert de Vries was employed by the company GN Hearing. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Appendix A. Additional Examples

In this appendix, we demonstrate some interesting additional examples of the GraphPPL.jl modeling language.

Appendix A.1. Recursive Generative Model

In this example, we demonstrate a Quadtree-like structure. A Quadtree is a data structure that recursively divides a two-dimensional space into quadrants. Here, we write a model in GraphPPL.jl inspired by a Quadtree to demonstrate the recursive capabilities of GraphPPL.jl. In our model, we recursively use an `aggregate` function that combines the intermediate variables of the four quadrants, where the base case is the leaves.

Here, the depth of the model is determined by the size of the input; an 8×8 input results in a model of depth 4, whereas a 128×128 input results in a model of depth 8. At

every layer, we have the intermediate representation (`i1, i2, i3,` and `i4`), together with all variables that reside in this model, which contains a coarse-grained representation of the output. In this sense, a model like this can be seen as inspired by a Scale-Space [45] method, where every layer represents a coarser, higher-level version of the input.

```
function aggregate end

@model function quadtree(intermediate, output)
    asize = first(size(output))
    subarray_size = div(asize, 2)
    if subarray_size == 1
        output[1, 1] ~ Normal(0, 1)
        output[1, 2] ~ Normal(0, 1)
        output[2, 1] ~ Normal(0, 1)
        output[2, 2] ~ Normal(0, 1)
        intermediate ~ aggregate(output[1, 1], output[1, 2], output[2, 1],
             output[2, 2])
    else
        i1 ~ quadtree(output = output[1:subarray_size, 1:subarray_size])
        i2 ~ quadtree(output = output[1:subarray_size, (subarray_size + 1):
             asize])
        i3 ~ quadtree(output = output[(subarray_size + 1):asize, 1:
             subarray_size])
        i4 ~ quadtree(output = output[(subarray_size + 1):asize, (
             subarray_size + 1):asize])
        intermediate ~ aggregate(i1, i2, i3, i4)
    end
end
```

Code Block A1. Recursively defined generative model. Depth is determined by the input size.

Appendix A.2. Variational Autoencoder

A significant advantage of GraphPPL.jl is that submodels do not have a fixed generative direction in which to be used. This makes the modularity of GraphPPL.jl significantly different from that of Turing.jl. In this example, we demonstrate that by using the neural network architecture defined in Section 4.5, we can easily create a variational autoencoder [46] by making two copies of our encoder–decoder submodel.

```
    @model function enc_dec(input, out)
        h1 ~ neural_network_layer(input = input, n = 256)
        h2 ~ neural_network_layer(input = h1, n = 128)
        h3 ~ neural_network_layer(input = h2, n = 64)
        h4 ~ neural_network_layer(input = h3, n = 32)
        out ~ neural_network_layer(input = h3, n = 16)
    end
```

Code Block A2. Neural network used as both encoder and decoder in the variational autoencoder example.

Materializing this model twice, once in the forward generative direction and once mirrored, gives us a variational autoencoder. Using the modularity of GraphPPL.jl, we can guarantee that the encoder and decoder of this network have the same architecture, and if we change something in this model, we know the that changes will be reflected in both places.

```
    @model function vae(y, y_hat)
        bottleneck ~ enc_dec(input = y)
        y_hat ~ enc_dec(output = y_hat)
    end
```

Code Block A3. A Variational autoencoder, with the decoder implemented as a mirrored version of the encoder.

References

1. Phan, D.; Pradhan, N.; Jankowiak, M. Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro. *arXiv* **2019**, arXiv:1912.11554.
2. Ge, H.; Xu, K.; Ghahramani, Z. Turing: A Language for Flexible Probabilistic Inference. In Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics, PMLR, Playa Blanca, Spain, 9–11 April 2018; pp. 1682–1690, ISSN 2640-3498.
3. Semenova, E.; Williams, D.P.; Afzal, A.M.; Lazic, S.E. A Bayesian neural network for toxicity prediction. *Comput. Toxicol.* **2020**, *16*, 100133.
4. Da Silva, S.L.E.; Karsou, A.; Moreira, R.M.; Cetale, M. Bayesian weighted time-lapse full-waveform inversion using a receiver-extension strategy. *IEEE Trans. Geosci. Remote. Sens.* **2024**, *62*, 5921522.
5. Griffiths, T.L.; Chater, N.; Kemp, C.; Perfors, A.; Tenenbaum, J.B. Probabilistic models of cognition: Exploring representations and inductive biases. *Trends Cogn. Sci.* **2010**, *14*, 357–364.
6. Spiegelhalter, D.; Thomas, A.; Best, N.; Gilks, W. BUGS 0.5: Bayesian inference using Gibbs sampling manual (version ii). In *MRC Biostatistics Unit, Institute of Public Health*; Citeseer: Cambridge, UK, 1996; pp. 1–59.
7. Gelman, A.; Lee, D.; Guo, J. Stan: A Probabilistic Programming Language for Bayesian Inference and Optimization. *J. Educ. Behav. Stat.* **2015**, *40*, 530–543. [CrossRef]
8. Cox, M.; van de Laar, T.; de Vries, B. ForneyLab. jl: Fast and flexible automated inference through message passing in Julia. In Proceedings of the International Conference on Probabilistic Programming, Cambridge, MA, USA, 5–6 October 2018.
9. Luttinen, J. BayesPy: Variational Bayesian inference in Python. *J. Mach. Learn. Res.* **2016**, *17*, 1419–1424.
10. Bagaev, D.; Podusenko, A.; Vries, B.d. RxInfer: A Julia package for reactive real-time Bayesian inference. *J. Open Source Softw.* **2023**, *8*, 5161. [CrossRef]
11. Duane, S.; Kennedy, A.D.; Pendleton, B.J.; Roweth, D. Hybrid Monte Carlo. *Phys. Lett. B* **1987**, *195*, 216–222. [CrossRef]
12. Hoffman, M.D.; Gelman, A. The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *arXiv* **2011**, [CrossRef]
13. Bagaev, D.; de Vries, B. Reactive Message Passing for Scalable Bayesian Inference. *arXiv* **2021**. [CrossRef]
14. Şenöz, İ. Message Passing Algorithms for Hierarchical Dynamical Models. Ph.D. Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2022; ISBN 9789038655321.
15. Loeliger, H.A.; Dauwels, J.; Hu, J.; Korl, S.; Ping, L.; Kschischang, F.R. The Factor Graph Approach to Model-Based Signal Processing. *Proc. IEEE* **2007**, *95*, 1295–1322. [CrossRef]
16. Kullback, S.; Leibler, R.A. On Information and Sufficiency. *Ann. Math. Stat.* **1951**, *22*, 79–86. [CrossRef]
17. Blei, D.M.; Kucukelbir, A.; McAuliffe, J.D. Variational Inference: A Review for Statisticians. *J. Am. Stat. Assoc.* **2017**, *112*, 859–877. [CrossRef]
18. Heskes, T. Convexity Arguments for Efficient Minimization of the Bethe and Kikuchi Free Energies. *J. Artif. Intell. Res.* **2006**, *26*, 153–190. [CrossRef]
19. Chertkov, M.; Chernyak, V.Y. Loop calculus in statistical physics and information science. *Phys. Rev. E* **2006**, *73*, 065102. [CrossRef]
20. Pearl, J. Reverend Bayes on inference engines: A distributed hierarchical approach. In Proceedings of the AAAI National Conference on AI, Pittsburgh, PA, USA, 18–20 August 1982; pp. 133–136.
21. Yedidia, J.S.; Freeman, W.; Weiss, Y. Constructing free-energy approximations and generalized belief propagation algorithms. *IEEE Trans. Inf. Theory* **2005**, *51*, 2282–2312. [CrossRef]
22. Lunn, D.J.; Thomas, A.; Best, N.; Spiegelhalter, D. WinBUGS-A Bayesian modelling framework: Concepts, structure, and extensibility. *Stat. Comput.* **2000**, *10*, 325–337. [CrossRef]
23. Carpenter, B.; Gelman, A.; Hoffman, M.D.; Lee, D.; Goodrich, B.; Betancourt, M.; Brubaker, M.; Guo, J.; Li, P.; Riddell, A. *Stan*: A Probabilistic Programming Language. *J. Stat. Softw.* **2017**, *76*, 1. [CrossRef]
24. Goodman, N.; Mansinghka, V.; Roy, D.M.; Bonawitz, K.; Tenenbaum, J.B. Church: A language for generative models. *arXiv* **2014**. [CrossRef]
25. Pfeffer, A. IBAL: A Probabilistic Rational Programming Language. In *IJCAI*; Citeseer: Cambridge, UK, 2001.
26. Murray, L.M. Bayesian State-Space Modelling on High-Performance Hardware Using LibBi. *arXiv* **2013**. [CrossRef]
27. Bingham, E.; Chen, J.P.; Jankowiak, M.; Obermeyer, F.; Pradhan, N.; Karaletsos, T.; Singh, R.; Szerlip, P.; Horsfall, P.; Goodman, N.D. Pyro: Deep Universal Probabilistic Programming. *arXiv* **2018**. [CrossRef]
28. Mansinghka, V.; Selsam, D.; Perov, Y. Venture: A higher-order probabilistic programming platform with programmable inference. *arXiv* **2014**. [CrossRef]
29. Dillon, J.V.; Langmore, I.; Tran, D.; Brevdo, E.; Vasudevan, S.; Moore, D.; Patton, B.; Alemi, A.; Hoffman, M.; Saurous, R.A. TensorFlow Distributions. *arXiv* **2017**. [CrossRef]
30. Krapu, C.; Borsuk, M. Probabilistic programming: A review for environmental modellers. *Environ. Model. Softw.* **2019**, *114*, 40–48. [CrossRef]
31. Bezanson, J.; Edelman, A.; Karpinski, S.; Shah, V.B. Julia: A Fresh Approach to Numerical Computing. *arXiv* **2015**. [CrossRef]
32. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *arXiv* **2019**. [CrossRef]

33. Şenöz, İ.; de Vries, B. Online Variational Message Passing in the Hierarchical Gaussian Filter. In Proceedings of the 2018 IEEE 28th International Workshop on Machine Learning for Signal Processing (MLSP), Aalborg, Denmark, 17–20 September 2018; pp. 1–6, ISSN 1551-2541, [CrossRef]
34. Mathys, C.D. Hierarchical Gaussian filtering: Construction and variational inversion of a generic Bayesian model of individual learning under uncertainty. Ph.D. Thesis, ETH Zurich, Zürich, Switzerland, 2012. [CrossRef]
35. Mathys, C.D.; Lomakina, E.I.; Daunizeau, J.; Iglesias, S.; Brodersen, K.H.; Friston, K.J.; Stephan, K.E. Uncertainty in perception and the Hierarchical Gaussian Filter. *Front. Hum. Neurosci.* **2014**, *8*, 825. [CrossRef]
36. Martin, R.C. *Agile Software Development: Principles, Patterns, and Practices*; Pearson Education: Upper Saddle River, NJ, USA, 2003.
37. Bagaev, D.; van Erp, B.; Podusenko, A.; de Vries, B. ReactiveMP.jl: A Julia package for reactive variational Bayesian inference. *Softw. Impacts* **2022**, *12*, 100299. [CrossRef]
38. Dauwels, J. On Variational Message Passing on Factor Graphs. In Proceedings of the IEEE International Symposium on Information Theory, Nice, France, 24–29 June 2007; pp. 2546–2550. [CrossRef]
39. Winn, J.; Bishop, C. Variational Message Passing. *J. Mach. Learn. Res.* **2005**, *6*, 661–694.
40. Knill, D.C.; Pouget, A. The Bayesian brain: The role of uncertainty in neural coding and computation. *Trends Neurosci.* **2004**, *27*, 712–719. [CrossRef]
41. Friston, K. The free-energy principle: A unified brain theory? *Nat. Rev. Neurosci.* **2010**, *11*, 127–138. [CrossRef]
42. Kirchhoff, M.; Parr, T.; Palacios, E.; Friston, K.; Kiverstein, J. The Markov blankets of life: Autonomy, active inference and the free energy principle. *J. R. Soc. Interface* **2018**, *15*, 20170792. [CrossRef]
43. Annila, A.; Kuismanen, E. Natural hierarchy emerges from energy dispersal. *Biosystems* **2009**, *95*, 227–233. [CrossRef]
44. Dritsas, I. *Stochastic Optimization: Seeing the Optimal for the Uncertain*; BoD—Books on Demand: Norderstedt, Germany, 2011.
45. Koenderink, J.J. The structure of images. *Biol. Cybern.* **1984**, *50*, 363–370. [CrossRef]
46. Kingma, D.P.; Welling, M. Auto-Encoding Variational Bayes. *arXiv* **2022**. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.