Philipp Gabler, BSc

Automatic Graph Tracking in Dynamic Probabilistic Programs via Source Transformations

Master's Thesis

to achieve the university degree of
Master of Science

submitted to
Graz University of Technology

Supervisor
Univ.-Prof. Dipl.-Ing. Dr. mont. Franz Pernkopf

Co-supervisor
Dipl.-Ing. Dr. Martin Trapp, BSc

Institute of Signal Processing and Speech Communication

Faculty of Electrical and Information Engineering

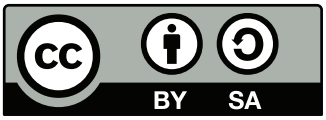
Graz, XXXX 2020

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date

Signature



The LaTeX source of this document is available at
https://github.com/phipsgabler/master-thesis
or upon request from the author*.

*pgabler@student.tugraz.at

Abstract

This thesis presents a novel approach for the implementation of a tracking system to facilitate program analysis, based on program transformations. The approach is then applied to a specific problem in the field of probabilistic programming.

The main contribution is a general system for the extraction of rich computation graphs in the Julia programming language, based on a transformation of the intermediate representation (IR) used by the compiler. These graphs contain a slice of the whole recursive structure of any Julia program in terms of executed IR instructions, including control flow operations. The system is flexible enough to be used for multiple purposes that require dynamic program analysis or abstract interpretation, such as automatic differentiation or dependency analysis.

The second part of the thesis describes the application of this graph tracking system to probabilistic programs written for `Turing.jl`, a probabilistic programming system implemented as an embedded language within Julia. Through this, an executed Turing model can be analyzed, and the dependency structure of involved random variables be extracted from it. Given this structure, analytical Gibbs conditionals can be calculated for a large set of models and passed to Turing's inference mechanism, where they are used in Markov-Chain Monte Carlo samplers approximating the modelled distribution.

Contents

Notation

$\mathbb{P}[\Theta \in A \mid X = x]$	Random variables and their realizations will usually be denoted by upper and lower case letters, respectively (with occasional exceptions for Greek variable names). Sets are also named by uppercase letters.
$\mathbb{E}[X], \mathbb{V}_X[f(X, Y)]$	Expectation and variance; if necessary, the variable with respect to which the moment is taken is indicated as a subscript.
$\phi(x), f_Z(x)$	Density functions are named using letters commonly used for functions, with an optional subscript indicating the random variable they belong to. Densities always come with implied base measures depending on the type of the random variable.
$p(x, y \mid z)$	The usual abuse of notation with the letter "p" standing for any density indicated by the names of the variables given to it is used when no confusion arises (in this case, $f_{X,Y\mid Z}$ is implied). A q may be used as well, mostly for proposal distributions or unnormalized densities.
$\mathbb{P}[X \in A] = P_X(A) = \int_A p_X(x)\,\mathrm{d}\mu(x)$	A capital P with subscript is used for the probability measure associated with a random variable.
$\mathbb{P}[X \in \mathrm{d}x] = p_X(x)\,\mathrm{d}\mu(x)$	Differentials of this form are used to concisely express densities of random variables (i.e., Radon-Nikodym derivatives of the associated probability measure). The example is equivalent to the statement $\frac{\mathrm{d}P_X}{\mathrm{d}\mu} = p_X$.
$X_i \sim \mathrm{Normal}(\mu, \sigma)$	The tilde notation for describing random variables is used throughout, often without explicitely specifying dependence or independence, where understood from context. Named distributions that are not themselves random variables are spelled out in upright script.
$Y \sim q(\cdot, X_{i-1})$	The same notation is used when a random variable is specified to be sampled from a given, possibly unnormalized,

density. In this context and elsewhere, the midpoint is employed to denote anonymous functions of one variable given by partial application.

$y \mapsto p(x \mid y, z)$ — Anonymous functions are distinguised from function evaluation; this is crucial to differentiate between probability densities and likelihoods, for example.

$\int p(x)\,\mathrm{d}x = 1$ — Integrals over the whole domain of a density or measure are written as indefinite integrals, where the usage is clear.

$[x, y, z] = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$ — For consistency with Julia code, vectors (arrays of rank 1) are written in brackets, with elements separated by commas. Thereby, the form written in a row denotes a column vector; actual row vectors are written as transposed column vectors.

$\Theta^{(k)} = [\Theta_1^{(k)}, \ldots, \Theta_N^{(k)}]$ — Superscript indices in parentheses are used for series or sequences of variables, and subscript indices for components of multivariate variables.

$z_{-i} = [z_1, \ldots, z_{i-1}, z_{i+1}, \ldots, z_N]$ — Negative indices denote all components of a variable without the negated one.

$f.(x, 1) = [f(x_1, 1), \ldots, f(x_N, 1)]$ — Function application with a period indicates vectorized application, as in Julia code*: the function is applied over all elements of the input arrays individually, whereby arrays of lower rank or scalars are "broadcasted" along dimensions as necessary.

`f(x) = rand(x)` — Julia code (including identifiers mention in the text) is always typeset in typewriter font.

*See https://docs.julialang.org/en/v1/manual/functions/#man-vectorized-1

1 Introduction

This chapter gives an overview over the scope of the thesis and existing approaches in the literature. A preliminary version of this work has already been presented in **gabler2019graph**, which forms the basis of the introduction.

Many methods in the field of machine learning work on computation graphs of programs that represent mathematical expressions. One example are forms of automatic differentiation (AD) which derive an "adjoint" expression from a expression that usually represents a loss function, to calculate its gradient (**griewank2008evaluating**; **gebremedhin2020introduction**). Another one are message passing algorithms (**minka2005divergence**), which use the graph as the basic data structure for the operation they perform: passing values between nodes, representing random variables that depend on each other (in fact, message passing generalizes various other methods, including AD). But also in more or less unrelated fields, such as program analysis or program transformation (**muchnick1997advanced**; **singer2018static**; **aho1986compilers**), the same requirements might occur through the need to derive abstract graphs of program flow from a given program for the purpose of abstract interpretation.

There are several options how to provide the computation graph in question to an application, many of which are already established in the AD community (see **baydin2018automatic** for a survey on AD methods). For one, graphs can be required to be written out explicitely by the user, by defining providing a library to build graphs "by hand" (e.g. **chewxy2020gorgonia**; **jia2014caffe** – these interfaces tend to be more low level) or through a higher-level API (e.g. PyTorch (**paszke2017automatic**) or TensorFlow (**abadi2015tensorflow**)). Such APIs are called *operator overloading* in AD language, because they extend existing operations to additionally track the computation graph at runtime on so-called tapes or Wengert lists (**bartholomew-biggs2000automatic**). This kind of tracking is dynamic, in the sense that a new tape is recorded for every execution. However, being implemented on a library level, it usually requires the programmer to use non-native constructs instead of language primitives, leading to cognitive overhead. This notably happens for control statements, which can rarely be "overloaded". Furthermore, there are additional runtime costs due to the separate interpretation of derivatives stored on the tape.

Alternatively, an implementation can allow the user to write out computations

as a "normal" program in an existing programming language (or possibly a restricted subset of it), and use metaprogramming techniques to extract graphs from the input program. Such metaprograms, known under the name *source transformations*, can in turn operate on plain source code (cf. Tapenade (**tapenadedevelopers2019tapenade**)), or on another, more abstracted notion used by the programming language infrastructure, like the abstract syntax tree (AST), or an intermediate representation (IR). They operate on the syntactic structure of the whole program, during or before compilation. Unlike in operator overloading, it is hence possible to inspect and exploit control structures directly. This can lead to more efficient results, compared to operator overloading, since the transformation is done only once per program and eligible for compiler optimisations. Additionally, the user is not restricted to the domain specific language provided by a libray, and can use regular language constructs, data structures, and custom functions rather freely. But in this approach, usually, no records of the actual execution paths are constructed explicitly – purely static information is used only at compile time, and cannot be accessed for further analysis or transformation during execution.

In a variety of domains, though, the execution path of programs can drastically change at each run. Examples of this from machine learning are models with non-uniform data, such as parse trees (**socher2011parsing**) or molecular graphs (**bianucci2000application**), Bayesian nonparametric models (**hjort2010bayesian**), or simply the occurrence of stochastic control flow in any probabilistic model. Such programs we call dynamic models. The lack of an explicit, unique graph structure makes it impossible, or at least diffult, to apply source transformation approaches on them. Operator overloading is the more direct way for supporting dynamic models, since it automatically records a new tape for each input. In fact, many of the already mentioned state-of-the-art machine learning libraries are based on dynamic graphs using operator overloading in some form.

However, relying on operator overloading makes it impossible to take advantage of the benefits of source transformations, such as utilizing information about the control flow, integrating with optimizations at compile time, or exploiting the source model structure. The source transformation approach based on intermediate compiler representations has recently gained popularity in machine learning; see **bradbury2018jax**; **lattner2020mlir**. While the main focus of these efforts has been optimization of linear algebra/tensor calculations and automatic differentiation, other use cases start to emerge, for example automatic detection of sparsity patterns (**gowda2019sparsity**).

In this thesis, I present a novel variant of automatic extraction of computation graphs suitable for static and dynamic models, using source transformation instead of operator overloading. Inspired by recent work on differential programming (**innes2018don**), the approach transforms the intermediate representation used by the compiler of the Julia programming language. This system can be used to dynamically track computation graphs of any Julia program, including machine learning models and probabilistic programming systems, without having to ex-

plicitely declare graph structures. The transformation is implemented as a custom part of the compilation process. Its result is passed on to the compiler, where it can be optimised further. At run time, both data and control path are tracked alongside the original calculations, in the form of a nested data structure. This data structure contains all functions called during execution, enriched by recorded control flow decisions and meta information that can be used to analyse the execution. Thus, the system combines advantages of a source transformation with a tape-based runtime approach.

1.1 Related Work

The topic of this thesis crosses several disciplines – at least automatic differentiation, compiler and programming language theory, and probabilitstic programming. Since these have not always worked together, similar principles may be have been found or (re-)introduced in each of them.

Automatic differentiation has a long history, in which different styles becoming more or less fashionable depending on the dominating use-case and available languages and infrastructure. Traditionally, numerical code in Fortran or C was differentiated by whole-source transfomation systems like Tapenade (**tapenadedevelopers2019tapenade**). In recent years, after phase of many library-based (or "operator overloading") systems that were driven by the rise of deep learning (**abadi2015tensorflow**; **paszke2017automatic**; **neubig2017dynet**; **tokui2015chainer**), compiler-based approaches have regained popularity lately. There are ongoing efforts to add built-in automatic differentiation to the Swift programming language in Swift for TensorFlow (**swift2019**), and work in Julia for Zygote (**innes2018don**) has started to apply source transformation to the intermediate representation of the compiler, which enables differentiating through complex control flow, custom data types, and nested functions. A similar approach to Zygote is taken in Python with Tangent (**vanmerrienboer2018tangent**).

Generalizations of the kinds of analyses and transformations found in these systems can be found under multiple terms in the compiler literature. Data- and control-flow analysis problems, as well as other forms of information propagation in programs, can be described*abstract interpretation* (**muchnick1997advanced**; **singer2018static**). These methods, which find fixed points in relations defined over a program, can in turn be seen as a form of message passing, under which not only a variety of learning algorithms can be summarized (**minka2005divergence**), but also automatic differentiation (**minka2019automatic**) and gradient based optimization (**dauwels2005steepest**). Other forms of abstract analysis can serve sparsity detection (**gowda2019sparsity**) or the detection of program parts that need not to be reevaluation after input changes (**becker2020dynamic**). In many of these methods, not the original form of the program is used, but a syntactically simplified lowered form. Such forms can be dependency graphs as used in compiler theory, or

As for the trade-off between transformation-based and library-based implementations, several hybrid graph tracking approaches between source transformation

and graph tracking exist. Among AD systems, recent TensorFlow versions have introduced AutoGraph[1], which rewrites regular Python functions to traced TensorFlow implementations by replacing control flow statements by TensorFlow combinators which. Such functions still need to be re-traced whenever a non-tensor input argument changes. Its predecessor TensorFlow Fold (**looks2017deep**) follows a similar, but more explicit style and provides many of these combinators as "dynamic batching operators" to define static graphs emulating dynamic operations. The "dynamicity" problem can be approached in other ways as well: *stochastic memoization* is employed in the probabilistic programming languages Church (**goodman2012church**) and Venture (**mansinghka2014venture**) to produce what in the latter is called "probabilistic execution traces": multiple different traces are dynamically stored as alternative paralles paths in the execution trace, with possible interconnections.

[1] https://www.tensorflow.org/api_docs/python/tf/autograph, visited on 2020-10-26

2 Background

This chapter provides the background for the concepts used later in chapters 3 and 4. Initially, it gives a quick overview of Baysian inference and probabilistic programming in general, necessary to understand the requirements and usual approaches of probabilistic programming systems.

Consequently, the machinery and language used to develop the graph tracking system forming the main part of the work are described. This consists firstly of a short introduction to graph tracking and source-to-source automatic differentiation, which contain many ideas and terminology that will be used later, and often provided inspiration. Secondly, the basic notions and techniques of the Julia compilation process as well as the language's metaprogramming capabilities are described, which form the basis of the implementation.

2.1 Bayesian Inference and MCMC methods

Generative modelling is an approach for modelling phenomena based on the assumption that observables can be fully described through some stochastic process. When we assume this process to belong to a specified family of processes, the estimation of the "best" process is a form of learning: if we have a good description of how obserations are generated, we can make summary statements about the whole population (descriptive statistics) or predictions about new observations. When observations come in pairs of independent and dependent variables, learning the conditional model of one given the other solves a regression or classification problem.

Within a Baysian statistical framework, we assume that the family of processes used is specified by random variables related through conditional distributions with densities, which describe how the observables would be generated: some *unobserved variables* are generated from *prior distributions*, and the *observed data* are generated conditionally on the unobserved variables. The goal is to learn the *posterior distribution* of the parameters given the observations, which is a sort of "inverse" of how the problem is specified.

As an example, consider image classification: if we assume that certain percentages of an image data set picture cats and dogs, respectively, the distribution of these labels forms the prior. Given the information which kind of animal is depicted on it, an image can then be generated as a matrix of pixels based on a distribution of images conditioned on labels. The posterior distribution is then conditional distribution of

the label given an image. When we have this information, we can, for example, build a Baysian classifier, by returning for a newly observed image that label which has the highest probability under the posterior.

This kind of learning is called Bayesian inference since, in the form of densities, the form of the model can be expressed using Bayes' theorem as the conditional distribution with density[1]

$$\overbrace{p(\theta \mid x)}^{\text{posterior}} = \frac{\overbrace{p(x \mid \theta)}^{\text{likelihood}} \overbrace{p(\theta)}^{\text{prior}}}{p(x)}, \tag{2.1}$$

where x are the observed data, and θ are the unobserved parameters. The posterior represents the distribution of the unobserved variables as a combination of the prior belief updated by what has been observed (**congdon2006bayesian**). (In practice, not all of the unobserved variables have to be model parameters we are actually interested in; these can be integrated out).

Going beyond simple applications like the classifier mentioned above, handling the posterior gets difficult, though. Simply evaluating the posterior density $\theta \mapsto p(\theta \mid x)$ at single points is not enough in a Baysian setting for usages such as prediction, parameter estimation, or evaluation of probabilities of continuous variables. The problem is that almost all of the relevant quantities depend on some sort of expectation over the posterior density, an integral of the form

$$\mathbb{E}[f(\Theta) \mid X = x] = \int f(\theta) p(\theta \mid x) \, \mathrm{d}\mu(\theta), \tag{2.2}$$

for some measurable function f (with the base measure μ depending on the type of Θ). This in turn involves calculating the normalizing marginal

$$p(x) = \int p(x, \theta) \, \mathrm{d}\mu(\theta). \tag{2.3}$$

in equation 2.1, often called the "evidence".

When the distributions involved form a sufficiently "nice" combination, e.g., a conjugate pair (**marin2007bayesian**; **murphy2012machine**), the integration can be performed analytically, since the posterior density has a closed form for a certain known distribution, or at least is a known integral. In general, however, this is not tractable, not even by standard numerical integration methods, and approximations have to be made. Even for discrete variables, the applicability of simple summation is limited by combinatorial explosion.

Different techniques for posterior approximation are available: among them are distribution-based approaches for general graphical models, such as variational

[1]Note the abuse of notation regarding $p(\cdot)$; see page xi on notation.

inference (**murphy2012machine**) and other methods generalized under the framework of message passing (**minka2005divergence**). The methods described in this thesis, however, fall into the category of Monte Carlo methods, and are based on sampling (**murphy2012machine**; **viholа2020lectures**). Their fundamental idea is to derive, for a specified density of $\Theta \sim \pi$, a sampling procedure with a consistent estimator for expectations:

$$I^{(k)}(f) \to \mathbb{E}[f(\Theta)] = \int f(\theta)\pi(\theta)\,\mathrm{d}\mu(\theta), \quad \text{as} \quad k \to \infty \tag{2.4}$$

in some appropriate stochastic convergence (usually convergence in probability is enough). We leave out the conditional dependency on X here for simplicity in notation, and since the data are usually fixed in inference problems.

Examples of such methods are rejection sampling, importance sampling, and particle filters. Many Monte Carlo methods are defined in a form that directly samples a sequence of individual random variables $(Y^{(k)})_{k\geq 1}$, called a *chain*, for which the estimator is given by the arithmetic mean, such that a law of large numbers (LLN) holds:

$$I^{(k)}(f) = \frac{1}{k}\sum_{i=1}^{k} f(Y^{(i)}) \to \mathbb{E}[f(\Theta)] \tag{2.5}$$

If we can sample $Y^{(k)} \sim \pi$ exactly, they are i.i.d. and the LLN holds trivially; such samplers exist, but might also be difficult to derive or not possess good enough convergence properties (especially in high dimensions). Another large class of samplers is formed by *Markov Chain Monte Carlo* (MCMC) methods, which, instead of sampling exactly from the density, define $Y^{(k)}$ via a (time-homogeneous) Markov chain:

$$\begin{aligned} &\mathbb{P}[Y^{(k+1)} \in \mathrm{d}y \mid Y^{(k)} = y^{(k)}, \ldots, Y^{(1)} = y^{(1)}] \\ &\quad = \mathbb{P}[Y^{(k+1)} \in \mathrm{d}y \mid Y^{(k)} = y^{(k)}] \\ &\quad = K(dy \mid y^{(k)}) \end{aligned} \tag{2.6}$$

for all $k \geq 1$. By constructing the *transition kernel*, K, in the right way, the resulting chain is ergodic with the target density π as the unique stationary distribution, i.e., for all measurable sets A,

$$\int \pi(y)K(A \mid y)\,\mathrm{d}\mu(y) = \int_A \pi(y)\,\mathrm{d}\mu(y), \tag{2.7}$$

and the LLN for Markov chains holds. (For discrete spaces, this relation is more familiarly written as a left eigenvalue equation on a stochastic matrix: $\pi K = \pi$.) The advantage of MCMC methods is that they apply equally well to many structurally complex models, and treat densities in a uniform way, without requiring special knowledge about the specific distribution in question. I refer to **viholа2020lecturesrobert1999montemurphy2012machine** as introductions to MCMC theory and practice.

Algorithm 1 General scheme for the Metropolis-Hastings algorithm.

1. Start from an arbitrary $Y^{(1)} = y^{(1)}$ with $\pi(y^{(k)}) > 0$.
2. For each $k \geq 1$:
    1. Sample a proposal $\hat{Y}^{(k)} \sim q(Y^{(k-1)}, \cdot)$.
    2. With probability $\alpha(\hat{Y}^{(k)}, Y^{(k-1)})$, set $Y^{(k)} = \hat{Y}^{(k)}$; else, keep $Y^{(k)} = Y^{(k-1)}$.

nicer algorithm formatting

Frequently, MCMC methods are variations of the *Metropolis-Hastings algorithm* (MH), which splits the general definition of the transition kernel into two parts: a proposal distribution, given by a conditional density q that needs to be easy to sample from, and an acceptance rate α. Subsequent samples are then produced by proposing values from q given the previous element of the chain, and incorporating them into the chain with a probability given through α (see algorithm 1). There exist many MH-based schemes with different properties and requirements: from the classical random-walk Metropolis algorithm with Gaussian proposals, over Reversible Jump MCMC for varying dimensions (**green1995reversible**), to gradient-informed methods like Metropolis Adjusted Langevin and Hamiltonian Monte Carlo (HMC) (**betancourt2018conceptual**; **girolami2011riemann**).

For multi-component structures, of the form $\Theta = [\Theta_1, \ldots, \Theta_N]$, a good proposal distribution can be hard to find, though. One way to break down the problem is to use a family of componentwise updates, given by conditional distributions q_i operating on only one component of Θ, with the others fixed:

$$\begin{aligned} \hat{Y}_{-i}^{(k)} &= Y_{-i}^{(k-1)} \\ \hat{Y}_{i}^{(k)} &\sim q_i(Y_i^{(k-1)}, \cdot \mid Y_{-i}^{(k-1)}) \end{aligned} \tag{2.8}$$

The components can be scalar or multivariate blocks, and the kernel may itself be any valid transition kernel (**vihola2020lectures**). This allows one to freely mix different MCMC methods suitable for each variable in a problem.

This so-called "within-Gibbs" sampler bears its name because it is a generalization of the classical *Gibbs sampling* algorithm (**geman1984stochastic**): often, the simplest available set of transition kernels is given by the conditional densities $\theta_i \mapsto p(\theta_i \mid \theta_{-i}, x)$. They can directly be used as component proposals for a within-Gibbs sampler, leading to a cancelling acceptance rate of $\alpha \equiv 1$. This approach has the advantage of being very algorithmic, which makes it rather easy to apply, even by hand, to many models, and simply to express algorithmically. Hence, the method is a popular starting point for general probabilistic programming systems, most prominently BUGS (**lunn2000winbugs**; **lunn2009bugs**) and JAGS (**plummer2003jags**).

In many real-world models, the factorization structure is quite sparse and results in small Markov blankets. Algorithms to derive Gibbs samplers exploit this large independency between variables. In short, they "trim" the dependency graph of the

model to the local Markov blankets of each target variable, and derive either a full conditional from it, where possible (for discrete or conjugate variables), or otherwise approximate it through appropriate local sampling (e.g., slice sampling).

As an example, consider a simple Gaussian mixture model with equal weights, specified as follows:

$$\begin{aligned} \mu_k &\overset{\text{iid}}{\sim} \text{Normal}(m, s) \quad \text{for } 1 \leq k \leq K, \\ Z_n &\overset{\text{iid}}{\sim} \text{Categorical}(K) \quad \text{for } 1 \leq n \leq N, \\ X_n &\overset{\text{iid}}{\sim} \text{Normal}(\mu_{Z_n}, \sigma) \quad \text{for } 1 \leq n \leq N \end{aligned} \tag{2.9}$$

To derive the conditional distribution of Z_n given the remaining variables, we start by writing down the factorization of the joint density:

$$p(z_{1:N}, \mu_{1:K}, x_{1:N}) = \prod_k p(\mu_k) \prod_n p(z_n) \prod_n p(x_n|\mu_{z_n}). \tag{2.10}$$

From this, we can derive an unnormalized density proportional to the conditional by removing all factors not including the target variable:

$$p(z_n \mid z_{-n}, \mu_{1:K}, x_{1:N}) \propto p(z_n)p(x_n|\mu_{z_n}) \tag{2.11}$$

This is equivalent to finding the Markov blanket of Z_n: only those conditionals relating the target variable to its children and parents remain. Since the clusters are drawn from a categorical distribution, the support is discrete, and we can find the normalization constant by summation:

$$\begin{aligned} &p(z_n \mid z_{-n}, \mu_{1:K}, x_{1:N}) \\ &\quad = \frac{\text{Categorical}(z_n \mid K)\, \text{Normal}(x_n \mid \mu_{z_n}, \sigma)}{\sum_{k \in \text{supp}(Z_n)} \text{Categorical}(k \mid K)\, \text{Normal}(x_n \mid \mu_k, \sigma)}, \end{aligned} \tag{2.12}$$

which can be expressed as a general discrete distribution over $\text{supp}(Z_n) = \{1, \ldots, K\}$, with the unnormalized weights given by the numerator. Next, the conditionals of the μ_k have the form

$$\begin{aligned} &p(\mu_k \mid z_{1:N}, \mu_{-k}, x_{1:N}) \\ &\quad \propto p(\mu_k) \prod_n p(x_n|\mu_k)^{\mathbb{1}(z_n = k)} \\ &\quad = \prod_n \left(\text{Normal}(\mu_k \mid m, s)\, \text{Normal}(x_n \mid \mu_k, \sigma)\right)^{\mathbb{1}(z_n = k)} \end{aligned} \tag{2.13}$$

which we recognize as a product of conjugate pairs of normal distributions. More examples are extensively covered in **murphy2012machine**.

2.2 Probabilistic Programming

Probabilistic programming is a structured way implementing generative models, as described in the previous section, through the syntax of a programming language. It is beneficial to consider probabilistic programs not only as syntactic sugar for denoting the implementation of a joint probability density over some set of variables, but as organized objects in their own right: they open up possibilities that "black box" density functions cannot automatically provide. In more concise terms of **vandemeent2018introduction**:

> Probabilistic programming is largely about designing languages, interpreters, and compilers that translate inference problems denoted in programming language syntax into formal mathematical objects that allow and accommodate generic probabilistic inference, particularly Bayesian inference and conditioning.

A probabilistic program differs from a regular program (that may also contain stochastic parts) through the possibility of being conditioned on: some of the internal variables can be fixed to observed values, from outside. As such, the program denotes on the one hand a joint distribution, that can be *forward sampled* from by simply running the program top to bottom and producing (pseudo-) random values. But at the same time, it also represents a conditional distribution, in form on the unnormalized conditional density, which together with an inference algorithm can also be *backward sampled* from. (Other terms, such as "evaluation" and "querying", are used as well.) Consider the model (2.9) from above: its mathematical description might be translated into a program in `Turing.jl` syntax (**ge2018turing**; **tarek2020dynamicppl**) as written in listing 2.1. We can then sample from the model in several ways:

more prominent turing introduction

Fix floated listings layout

```
julia> m = normal_mixture(x_observations, K, m, s, σ);
julia> forward = sample(m, Prior(), 10);
julia> chain = sample(m, MH(), 1000);
```

The value of `forward` will be an dataframe-like object containing 10 values for each variable sampled from the forward (i.e., joint) distribution, matching the size of `x_observations`. Similarly, `chain` will contain a length 1000 sample from a Markov chain targetting the posterior, conditially on `x_observations`, created using the MH algorithm. If we were to write out these two functionalities manually, in idiomatic Julia, we would end up with at least the following two functions:

```
function normal_mixture_sampler(N, K, m, s, σ)
    μ = rand(Normal(m, s), K)
    z = rand(Categorical(K), N)
    x = rand.(Normal.(μ[z], s))
    return μ, z, x
end
```

Listing 2.1: `Turing.jl` implementation of a Gaussian mixture model with prior on the cluster centers, equal cluster weights, and all other parameters fixed.

```
@model function normal_mixture(x, K, m, s, σ)
    N = length(x)

    μ = Vector{Float64}(undef, K)
    for k = 1:K
        μ[k] ~ Normal(m, s)
    end

    z = Vector{Int}(undef, N)
    for n = 1:N
        z[n] ~ Categorical(K)
    end

    for n = 1:N
        x[n] ~ Normal(μ[z[n]], σ)
    end

    return x
end
```

```
function normal_mixture_logpdf(μ, z, x, K, m, s, σ)
    N = length(x)
    ℓ = 0.0
    ℓ += sum(logpdf(Normal(m, s), μ[k]) for k = 1:K)
    ℓ += sum(logpdf(Categorical(K), z[n]) for n = 1:N)
    ℓ += sum(logpdf(Normal(μ[z[n]]), x[n]) for n = 1:N)
    return ℓ
end
```

And still, with these, we would lack much of the flexibility that models written in `DynamicPPL.jl` form: no general interface for sampling algorithms to automatically detect all latent and observed variables; no possility for other, nonstandard execution forms as are needed for Variational Inference or gradient computation for HMC; no automatic dataframe construction for chains. All these points highlight the advantages of dedicated probabilistic programming languages (PPLs) over manual likelihood functions.

Many PPLs are implemented as external domain-specific languages (DSLs), like Stan (**carpenter2017stan**), JAGS (**plummer2003jags**), and BUGS (**lunn2000winbugs**; **lunn2009bugs**). Others are specified in the "meta-syntax" of Lisp S-expressions, as Church (**goodman2012church**), Anglican (**wood2015new**), or Venture (**mansinghka2014venture**). A third group is embedded into host programming languages with sufficient syntactic flexibility, for example Gen (**cusumano-towner2020gen**) and Soss (**scherrer2019soss**) in Julia (besides the already named Turing), or Pyro (**bingham2018pyro**) and PyMC3 (**salvatier2016probabilistic**) in Python.

The latter approach is advantageous when one wants to enable the use of regular, general-purpose programming constructs or interact with other functionalities of

the host language. There are also a variety of further reasons why one would rather describe an inference problem in terms of a program than in more "mathematical" form, like as a graph or likelihood function. In a good probabilistic programming DSL, models will read as close to textbook model specifications as possible, while allowing to use the host language to:

- define recursive relationships,
- write models using imperative constructs, such as loops, or mutable intermediate computations for efficiency,
- optimize details of th execution, e.g. for memoization, likelihood scaling, or preliminary termination
- use distributions over complex custom data structures, e.g. trees,
- perform inference involving complex transformations from other domains, for which implementations already exist, e.g. neural networks or differential equation solvers , or
- integrate calls to very complex external systems, e.g. simulators or renderers.

See **vandemeent2018introduction** for a general introduction into some common implementation approaches for PPLs. **goodman2014design** gives a detailed overview of the internals of one specific, continuation-based implementation called WebPPL (using a Lisp-based syntax).

cite DPPL paper

Models in `Turing.jl` work by transforming the model code, which is syntactically a valid Julia function definition, through the macro `@model`. The result is new function which produces instances of a structure of type `Model`, that in turn will contain the observations, some metadata, and a nested function with the slightly changed original model code. In the concrete case of the model in listing 2.1, the resulting code would be approximately equal to the code in listing 2.2. The purpose of this is the following: the outer function, the "generator", constructs an instance of the model for given parameters – usually done once per inference problem, to fix the observations and hyperparameters. Subsequently, the `sample` function can be applied to this instance with different values for the sampling algorithm, which in turn will use the `evaluator` function of the instance to run the model with chosen `sampler` and `context` arguments, that are passed to the "tilde functions", to which the statements of the form `expr ~ D` are converted.

observe vs assume

Inside the "tilde functions", the real stochastic work happens. Depending on the `sampler` and the `context`, values may be generated and stored in the `varinfo` object, as happens for most "real" samplers. In this case, one call to the `evaluator` corresponds to one (Gibbs) sampling step. In other situations, model evaluation serves the purpose of density evaluation, in which no new values need to be produced; this use case is needed for probability queries, or density-based inference methods (which might additionally use automatic differentiation on the density evaluation procedure). All shared information for external usage is thereby conventionally

Listing 2.2: Expansion of model 2.1.

```
function normal_mixture(x, K, m, s, σ)
    function evaluator((rng, model, varinfo, sampler, context, x, K, m, s, σ)
        N = length(x)
        μ = Vector{Float64}(undef, K)
        for k = 1:K
            dist_mu = Normal(m, s)
            vn_mu = @varname μ[k]
            inds_mu = ((k,),)
            μ[k] = tilde_assume(
                rng, context, sampler, dist_mu, vn_mu, inds_mu, varinfo
            )
        end
        z = Vector{Int}(undef, N)
        for n = 1:N
            dist_z = Categorical(K)
            vn_z = @varname z[n]
            inds_z = ((n,),)
            z[n] = tilde_assume(
                rng, context, sampler, dist_z, vn_z, inds_z, varinfo
            )
        end
        for n = 1:N
            dist_x = Normal(μ[z[n]], σ)
            vn_x = @varname(x[n])
            inds_x = ((n,),)
            if isassumption(model, x, vn_x)
                x[n] = tilde_assume(
                    rng, context, sampler, dist_x, vn_x, inds_x, varinfo
                )
            else
                tilde_observe(
                    context, sampler, dist_x, x[n], vn_x, inds_x, varinfo
                )
            end
        end
        return x
    end
    return Model(
        :normal_mixture, evaluator,
        (x = x, K = K, m = m, s = s, σ = σ),
        NamedTuple()
    )
end
```

stored in the `varinfo` object, which resembles a dictionary from variable names[2] to values (internal sampler state can also be stored in the `sampler` object). Through the `sample` interface, the resulting values are then stored in a `Chains` object, a data frame containing a value for each variable at each sampling step.

A special distinction is made for variables that are contained in the model arguments. These by default serve as observed variables, and will only contribute to the likelihood, instead of being sampled. But in certain cases, such as in probability evaluation or when using the complete model in a generative way, this behaviour can be different. For this purpose, the variables `x[i]` in the example are wrapped in a conditional statement.

From the point of view of a sampling algorithm, all that it sees is a sequence of tilde statements, consisting of a value, a variable name, and a distribution. Turing, crucially, does not have a representation of model structure. This is sufficient for many kinds of inference algorithms that it already implements – Metropolis-Hastings, several particle methods, HMC and NUTS, and within-Gibbs combinations of these – but does not allow more intelligent usage of the available information. For example, to use a true, conditional, Gibbs sampler, the user has to calculate the conditionals themselves. Optimizations such as partial reevaluation of a model to save calculations, automatic conjugacy detection, or model transformations such as Rao-Blackwellization cannot be performed in this representation.

cite autoconj etc.

2.3 Compilation and Metaprogramming in Julia

sldk sldks dlks dlksfj dlkfjs lkdjf slkdjls dkf§

```
function foo(x)
    y = zero(eltype(x))
    for i in eachindex(x)
        @show y += sin(x[i])
    end
    return y
end
```

The macro invocation of `@show` will be converted to a function call receiving as input the expression

```
Expr(:+=, :y, Expr(:call, :sin, Expr(:ref, :x, :i)))
```

In this particular case, the nested structure is not taken advantage of or transformed, but simply taken as is and used to print the value of the expression, labelled by it's form in the code:

[2] These `VarName` objects, constructed by the macro `@varname`, simply represent an indexed variable by a symbol and an index tuple.

```
function foo(x)
    y = zero(eltype(x))
    for i = eachindex(x)
        begin
            Base.println(
                "y += sin(x[i]) = ",
                Base.repr(var"#24#value" = (y += sin(x[i]))))
            var"#24#value"
        end
    end
    return y
end
```

After macro expansion, the code of the function is *lowered* into an intermediate representation consisting of only function calls and branches. For example, the for loop in the example is converted into code equivalent to

```
iterable = eachindex(x)
iter_result = iterate(iterable)
while !(iter_result === nothing)
    i, state = iter
    @show y += sin(x[i])
    iter_result = iterate(iterable, state)
end
```

which is then lowered into three blocks related through conditional branching statements.

```
1 — %1  = Main.eltype(x)
    │       y = Main.zero(%1)
    %3  = Main.eachindex(x)
    │       @_4 = Base.iterate(%3)
    %5  = @_4 === nothing
    %6  = Base.not_int(%5)
└──     goto §4 if not %6
2 — %8  = @_4
    │       i = Core.getfield(%8, 1)
    %10 = Core.getfield(%8, 2)
    %11 = y
    %12 = Base.getindex(x, i)
    %13 = Main.sin(%12)
    %14 = %11 + %13
    │       y = %14
    │       value = %14
    %17 = Base.repr(%14)
    │       Base.println("y += sin(x[i]) = ", %17)
    │       value
    │       @_4 = Base.iterate(%3, %10)
    %21 = @_4 === nothing
    %22 = Base.not_int(%21)
└──     goto §4 if not %22
3 —     goto §2
4 —     return y
```

```
1: (%1, %2)
  %3 = Main.eltype(%2)
  %4 = Main.zero(%3)
  %5 = Main.eachindex(%2)
  %6 = Base.iterate(%5)
  %7 = %6 === nothing
  %8 = Base.not_int(%7)
  br 3 (%4) unless %8
  br 2 (%6, %4)
2: (%9, %10)
  %11 = Core.getfield(%9, 1)
  %12 = Core.getfield(%9, 2)
  %13 = Base.getindex(%2, %11)
  %14 = Main.sin(%13)
  %15 = %10 + %14
  %16 = Base.repr(%15)
  %17 = Base.println("y += sin(x[i]) = ", %16)
  %18 = Base.iterate(%5, %12)
  %19 = %18 === nothing
  %20 = Base.not_int(%19)
  br 3 (%15) unless %20
  br 2 (%18, %15)
3: (%21)
  return %21
```

```
1 ─ %1  = Main.eltype(x)::Core.Compiler.Const(Float64, false)
│         (y = Main.zero(%1))
│   %3  = Main.eachindex(x)::Base.OneTo{Int64}
│         (@_4 = Base.iterate(%3))
│   %5  = (@_4 === nothing)::Bool
│   %6  = Base.not_int(%5)::Bool
└──       goto §4 if not %6
2 ─ %8  = @_4::Tuple{Int64,Int64}::Tuple{Int64,Int64}
│         (i = Core.getfield(%8, 1))
│   %10 = Core.getfield(%8, 2)::Int64
│   %11 = y::Float64
│   %12 = Base.getindex(x, i)::Float64
│   %13 = Main.sin(%12)::Float64
│   %14 = (%11 + %13)::Float64
│         (y = %14)
│         (value = %14)
│   %17 = Base.repr(%14)::String
│         Base.println("y += sin(x[i]) = ", %17)
│         value
│         (@_4 = Base.iterate(%3, %10))
│   %21 = (@_4 === nothing)::Bool
│   %22 = Base.not_int(%21)::Bool
└──       goto §4 if not %22
3 ─       goto §2
4 ─       return y
```

```
1 ── %1  = Base.arraysize(x, 1)::Int64
│    %2  = Base.slt_int(%1, 0)::Bool
│    %3  = Base.ifelse(%2, 0, %1)::Int64
│    %4  = Base.slt_int(%3, 1)::Bool
└────       goto §3 if not %4
2 ──        goto §4
```

```
3 ──       goto §4
4 ── %8  = ϕ (§2 => true, §3 => false)::Bool
│    %9  = ϕ (§3 => 1)::Int64
│    %10 = ϕ (§3 => 1)::Int64
│    %11 = Base.not_int(%8)::Bool
└───       goto §22 if not %11
5 ── %13 = ϕ (§4 => 0.0, §21 => %18)::Float64
│    %14 = ϕ (§4 => %9, §21 => %42)::Int64
│    %15 = ϕ (§4 => %10, §21 => %43)::Int64
│    %16 = Base.arrayref(true, x, %14)::Float64
│    %17 = invoke Main.sin(%16::Float64)::Float64
│    %18 = Base.add_float(%13, %17)::Float64
│    %19 = Base.sle_int(1, 1)::Bool
└───       goto §7 if not %19
6 ── %21 = Base.sle_int(1, 0)::Bool
└───       goto §8
7 ──       nothing::Nothing
8 ── %24 = ϕ (§6 => %21, §7 => false)::Bool
└───       goto §10 if not %24
9 ──       invoke Base.getindex(()::Tuple, 1::Int64)::Union{}
└───       $(Expr(:unreachable))::Union{}
10 ─       goto §11
11 ─       goto §12
12 ─       goto §13
13 ─       goto §14
14 ─ %32 = invoke Base.:(var"#sprint#339")(
             nothing::Nothing, 0::Int64, sprint::typeof(sprint),
             show::Function, %18::Float64
           )::String
└───       goto §15
15 ─       goto §16
16 ─       goto §17
17 ─       invoke Base.println("y += sin(x[i]) = "::String, %32::String)::Any
│    %37 = (%15 === %3)::Bool
└───       goto §19 if not %37
18 ─       goto §20
19 ─ %40 = Base.add_int(%15, 1)::Int64
└───       goto §20
20 ─ %42 = ϕ (§19 => %40)::Int64
│    %43 = ϕ (§19 => %40)::Int64
│    %44 = ϕ (§18 => true, §19 => false)::Bool
│    %45 = Base.not_int(%44)::Bool
└───       goto §22 if not %45
21 ─       goto §5
22 ─ %48 = ϕ (§20 => %18, §4 => 0.0)::Float64
└───       return %48
```

singer2018static

2.4 Computation Graphs and Automatic Differentiation

3 Implementation of Dynamic Graph Tracking in Julia

3.1 Automatic Graph Tracking and Extended Wengert Lists

3.2 Evaluation

4 Graph Tracking in Probabilistic Models

5 Discussion

The history of this project forms a large arc from a general problem in `Turing.jl`, over a digression into compiler technology, back the the implementation of a proof of concept in the form of a very specific inference method. As we have seen, two separate pieces of software have emerged from it: `IRTracker.jl` and `AutoGibbs.jl`. The underlying issue – that `Turing.jl` lacks a structural representation of models – is not at all resolved by them.

The real difficulty is that dynamic models cannot be satisfactorily handled through static snapshots.

compare to autograd, venture, church

(`Cassette.jl` is a package very similar to `IRTools.jl`)

> Using Cassette on code you wrote is a bit like shooting youself with a experimental mind control weapon, to force your hands to move like you knew how to fly a helicopter. Even if it works, you still had to learn to fly the helicopter in order to program the mind-control weapon to force yourself to act like you knew how to fly a helicopter.[1]

5.1 Future Work

Many of the following ideas have already been informally described by me online[2].

Currently, Turing models are very primitive in this respect: a data structure called `VarInfo` contains a map from variable names to values, the accumulated log-likelihood, and some other metadata. During this project, I noticed that retrofitting structure onto this is not ideal, and for proper analysis, it would be nice to begin with a better representation from the start. The two main difficulties were matching of variable names (e.g., subsuming `x[1:10]` under `x[1:3][2]`), and getting rid of array mutations that shadow actual data dependencies (e.g., when one has an array `x`, samples `x[1]`, writes it to `x` with `setindex!`, and then uses `getindex(x, i)` somewhere downstream). A more versatile dictionary structure for variable name keys could improve this situation, but wouldn't satisfactorily solve all of the issues.

From these difficulties that became appearent during the implementation of the Gibbs conditional extraction, together with the knowledge about `DynamicPPL.jl`'s internals, I developed the following understanding of what an ideal representation of

[1] Lyndon White, private communication on https://julialang.slack.com.
[2] https://github.com/phipsgabler/probability-ir

probabilistic models for the purpuse of analysis would be for me. Probably the answer to any confusion I have caused is this: I come from a metaprogramming/analysis perspective, with interest in programming language design. I wanted variable names and dependencies to behave nicely, and primarily a closed, elegant language. Many PPL people probably come from an inference perspective, putting the language design problem second to that. “I want to write all the models” vs. “I want to do all the inference”. But I also try to close a bridge to the mostly theoretical, FP-based approaches of just formalizing probabilistic programs.

he separation between the “specification abstraction” and “evaluator abstraction”, across multiple implementations, would be something that I haven’t really seen before – everyone’s always proposing a complete system, right? The closest thing would be the formalization attempts of probabilistic models with monads and types, but that is more semantic than syntactic. We do have abstracted “pure inference” libraries, that really only take a function and do their work, but they aren’t really a PPL. There’s some “linguae frankae” like the Stan/JAGS syntax, but it’s also somewhat restricted and not independently maintained – the ones coming later just chose to take over the same kind of input format for their own implemenation. What I’m thinking of is a model specification form in its own right, that has more general analysis capabilites, and can then be transformed town to whatever the evaluator requires – into CPS, as a monad, as a DAG, as a factor graph, you name it.

The advantage of this kind of approach, besides solving "compiler domain" problems like the ones I mentioned above, is that it provides a different kind of common abstraction for PPLs. Recently, people have started writing "bridge code" to allow PPL interaction: there is invented a common interface that multiple PPL systems can be fit under, and then models in each can be used from within the other at evaluation. This approach is due to the lack of division of a system into an evalator and a model specification part (DynamicPPL is supposed to be a factored out model system, but currently way too specialized to Turing): they always go totheger. I believe that starting from a common model specification language is in many cases more feasible and general than defining a common interface for evaluators: the latter tends to assume much more about the internals, while model syntax is essentially fixed: the notation of random variables used in model specification by hand, extended through general Julia syntax.

List of Algorithms

Colophon

This document was typeset using the pdfLaTeX typesetting system, with the memoir document class. The body text is set in 11 pt Linux Libertine, enhanced by the microtype package. Other fonts include Biolinum and Inconsolata.

The document source has been written in Emacs with AUCTeX mode, using TeXworks as PDF viewer.