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

Es macht so glücklich, Computer zu sein:
alle Scherenein
verwandeln sich in Rechnerei
und gehn in Millionstel Sekunden vorbei.
In wenigen “bit”
kriegst du die ganze Weltordnung mit
im Grund
heißt die Frage ja immer “Sein oder Nichtsein”,
die erledigst du sogar ohne Und,
den ganzen Moder
mit einem einzigen Oder.

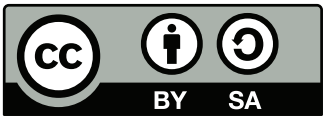
Andreas Okopenko

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 modeled 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, $f_{X,Y|Z}$ 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.

$\mathbb{P}[X \in \mathrm{d}x] = P_X(\mathrm{d}x) = p_X(x) \, \mathrm{d}\mu(x)$

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 explicitly 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 distinguished 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.
$\mathbb{1}(z_n = k)$	Indicator function. The value is 1 when the predicate inside holds, and 0 otherwise.
$[x, y, z] = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$	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.
$\mathrm{D}F(x, y) = (\Delta_1, \Delta_2) \mapsto \partial_1 F(x, y)\,\Delta_1 + \partial_2 F(x, y)\,\Delta_2$	Derivatives are written using a capital D for a total derivative operator (like the Fréchet derivative), and ∂_i for the conventional partial derivatives with respect to the i-th argument.
`f(x) = rand(x)`	Julia code (including identifiers mention in the text) is always typeset in typewriter font.
`IRTracker.jl`	Julia packages are set in the same font as Julia code. In the electronic version, a hyperlink to their source code on Github is automatically added.

*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. It is based on Gabler et al. (2019), which presents a preliminary version of this work.

In machine learning, several methods work on computation graphs of programs that represent mathematical expressions. One example are is automatic differentiation (AD), which derives new expressions from an expression that usually represents a loss function, to calculate its gradient (Gebremedhin & Walther 2020; Griewank & Walther 2008). AD is a special case of more general message passing algorithms (Minka 2005; Minka 2019; Ruozzi 2011), which all require a graph as basic data structure for the operation they perform: there, values are passed between nodes, representing variables in a domain that depend on each other. Moreover, in other fields, such as program analysis or program transformation (cf. Aho, Sethi & Ullman 1986; Muchnick 1997; Singer 2018), the same requirements might occur through the need to derive abstract graphs of program flow from a given program.

There are several options how to extract the computation graph in question, many of which are already established in the AD community (see Baydin et al. (2018) for a survey on AD methods). For one, graphs can be required to be written out explicitly by the user, by providing a library to build graphs "by hand" (e.g. Chewxy et al. (2020) and Jia et al. (2014)) in a more low-level application programming interface (API). Alternatively, there are higher-level APIs like PyTorch (Paszke et al. 2017) or AutoGrad (Maclaurin, Duvenaud & Adams 2015), where the graphs are defined implicitely from a forward program written in more declarative style (TensorFlow (Abadi, Agarwal, et al. 2015) is somewhere between these). Such APIs are called *operator overloading* in AD language, because they extend existing operations to additionally track the computation graph at run-time on so-called tapes or Wengert lists (Bartholomew-Biggs et al. 2000). 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, while it also makes the applicability limited to library functions and not easily extendable. This notably happens for control statements, which can rarely be "overloaded". Furthermore, there are additional run-time 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 program transformation techniques to extract graphs from the input program. Such meta-programs, known as *source transformations*, can in turn operate on plain source code (cf. Tapenade, Tapenade developers (2019)), 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 optimizations. Whereas the user is not restricted to the domain specific language provided by a library, and can use regular language constructs, data structures, and custom functions rather freely, in this approach, usually, no records of the actual execution paths are constructed explicitly. Only purely static information is used only at compile time, and cannot be accessed for further analysis or transformation during execution. (See section 2.4 for a more in-depth treatment of automatic differentiation techniques.)

For probabilitic programming languages, there exist mainly two paradigms for handling program structure (van de Meent et al. 2018). In the case of evaluation-based systems (e.g., `Turing.jl` (Ge, Xu & Ghahramani 2018), `Gen.jl`'s dynamic interface (Cusumano-Towner 2020), Church (Goodman, Mansinghka, et al. 2012), Anglican (Wood, van de Meent & Mansinghka 2015), Pyro (Bingham et al. 2018)), no structure is extracted at all. The interaction between the system and user programs consists only of a sequence of messages (in an abstract sense), indicating "events" that can be taken up by inference algorithms. In graph-based systems, an static representation of the model is first constructed and then passed to the inference algorithm. This representation can be close to probabilistic graphical models (`ForneyLab.jl`, (Cox, van de Laar & de Vries 2018), Venture (Mansinghka, Selsam & Perov 2014), PyMC3 (Salvatier, Wiecki & Fonnesbeck 2016)), symbolic (`Gen.jl`'s static interface, `Soss.jl` (Scherrer 2019)), or more compiler-oriented as in Stan (Carpenter, Gelman, et al. 2017), BUGS (Lunn, Thomas, et al. 2000), or JAGS (Plummer 2003).

In a variety of modelled domains, though, the execution path of programs can drastically change at each run. Examples of this are implementations of models containing non-uniform data, such as parse trees (Socher et al. 2011) or molecular graphs (Bianucci et al. 2000), of Bayesian nonparametric models (Hjort et al. 2010), or simply the occurrence of stochastic control flow in any probabilistic model. We call such models dynamic. The lack of an explicit, unique graph structure makes it impossible, or at least difficult, to apply source transformation approaches on their implementation. 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.

Reliance 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. A source transformation approach based on intermediate compiler representations has recently gained popularity in machine learning, and promises to resolve this dilemma; see Bradbury et al. (2018) and Lattner et al. (2020). 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 (Gowda et al. 2019).

In this thesis, I present a novel variant of automatic extraction of computation graphs suitable for static and dynamic models, using IR-based source transformation instead of operator overloading. Inspired by recent work on differential programming (Innes 2018), 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 explicitly declare graph structures. The transformation is implemented as a custom part of the compilation process. Its result is passed back to the compiler, where it can be optimized 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 possibly meta-information that can be used to analyse the execution. Thus, the system combines advantages of a source transformation with a tape-based run-time approach.

The extracted graphs can be used for various purposes. It is possible to implement automatic differentiation on top of it, as well as other algorithms that can be formulated via message-passing (Minka 2005; Minka 2019), and methods that operate on run-time dependency graphs, from simple debugging to concolic execution (Sen, Marinov & Agha 2005; Zeller et al. 2019). As a specific use case in the field of Bayesian inference, a dependency tracking system for a class of models in the Julia-base probabilistic programming language `Turing.jl` has been implemented. On top of this, Gibbs conditionals can automatically be derived for the models and used in compound MCMC algorithms, similar to JAGS and BUGS.

1.1 Related Work

The topic of this thesis crosses several disciplines – at least automatic differentiation, compiler and programming language theory, and probabilistic 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 became 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 transformation systems like Tapenade (Tapenade developers 2019). After phase of many operator overloading systems that were driven by the

rise of deep learning (Abadi, Agarwal, et al. 2015; Neubig et al. 2017; Paszke et al. 2017; Tokui et al. 2015), compiler-based approaches have regained popularity. More recently, there have been efforts to add built-in automatic differentiation to the Swift programming language in Swift for TensorFlow (TensorFlow Developers 2018), and work in Julia for `Zygote.jl` (Innes 2018). Both 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.jl` is taken in Python with Tangent (van Merrienboer, Moldovan & Wiltschko 2018).

Generalizations of the kinds of analyses and transformations used in these systems can be found under multiple terms in the compiler literature: data- and control-flow analysis, information propagation, or abstract interpretation (Muchnick 1997; Singer 2018). There, the program structure is always assumed to be known statically, though, as compilers fundamentally are source transformers. Closest to an evaluation-based analysis are concolic execution methods (Sen, Marinov & Agha 2005; Zeller et al. 2019), in which a given program is "instrumented" through additional instructions that, next to the concrete evaluation, track the execution in symbolic form (hence the name). The symbolic information can then be used in formal methods to, for example, find sets of program input that ensure complete test coverage of all branches. There already exists a proof-of-concept implementation of a concolic fuzzer in Julia (Churavy 2019), which applies the same kind IR-based transformations as `Zygote.jl`.

These information propagatino methods, most of which find recursive solutions to equations defined over program structure, can in turn be seen as a form of message passing, under which not only a variety of learning algorithms can be summarized (Minka 2005), but also automatic differentiation (Minka 2019) and gradient based optimization (Dauwels, Korl & Loeliger 2005). Other forms of abstract analysis exist for program optimization, e.g., sparsity detection in numerical programming (Gowda et al. 2019), or the detection of parts of probabilistic programs that need not to be reevaluation after input changes (Becker 2020).

Many implementations of these methods do not use the original form of the program, but a syntactically simplified lowered form. Such forms can be dependency graphs as used in compiler theory, or the intermediate languages used by actual compiler implementations. These can take portable, language independent forms as in LLVM (LLVM Project 2019), or be special to a particular compiler implementation, as in Julia (Bezanson, Edelman, et al. 2017) or Swift (Apple 2020). As these two languages illustrate, there are often even multiple layers of intermediate representations used in the same system.

Lately, special intermediate representations for machine learning applications have been introduced. One of them is the machine learning intermediate representation (MLIR, Lattner et al. (2020)), with the purpose of forming a reusable mid-layer between programming languages and run-times, featuring exchangeability between different machine learning frameworks and and "accelerators" (pieces of hardware), while taking advantage of modern compiler technology like LLVM. Another one is

Swift's intermediate representation (SIL) in the Swift for TensorFlow project. JAX (Bradbury et al. 2018) plays a similar role for expression-graph based machine learning systems, by tracing Python functions and compiling their graphs directly to optimized code. This system can interact with XLA ("accelerated linear algebra", TensorFlow Developers (2020)), which allows to compile sequences of numerical Python functions, which would otherwise be slow, to efficiently fused platform code.

As for the trade-off between transformation-based and evaluation-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 with TensorFlow combinators. Such functions still need to be re-traced whenever a non-tensor input argument changes. Its predecessor TensorFlow Fold (Looks et al. 2017) follows a similar, but more explicit style and provides many of these combinators as "dynamic batching operators" to define static graphs emulating dynamic operations. In probabilistic programming, "dynamicity problem" can be approached in other ways as well: a technique called *stochastic memoization* is employed in the probabilistic programming languages Church (Goodman, Mansinghka, et al. 2012) and Venture (Mansinghka, Selsam & Perov 2014) to produce what in the latter is called "probabilistic execution traces", where multiple different traces are dynamically stored as alternative parallel paths in the execution trace, with possible interconnections. `Gen.jl` (Cusumano-Towner et al. 2019; Cusumano-Towner 2020) in Julia is defined over a single abstract interface, for which two implementations are defined: a dynamic one, where dictionary-like traces are recorded at runtime from general programs, and a static one, that converts a restricted, combinator-based model syntax to a fixed graph structure through meta-programming.

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

2 Background

This chapter provides background for concepts used later in chapters 3 and 4. It gives a quick overview of Bayesian 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. First, 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. Second, a short introduction to graph tracking and source-to-source automatic differentiation is given, from which many of the ideas and terminology that will later be used were taken

2.1 Bayesian Inference and MCMC methods

Probabilistic modeling (Winn, Bishop, et al. 2019) is an approach to model phenomena based on the assumption that observable data can be fully described through some generative process that involves randomness. Recovering the details of this process, by estimating which one from a class of processes fits observed data best, is a form of learning: if we have a good description of how observations are generated, we can make summary statements about the whole population (descriptive statistics) or predictions about new observations. For example, learning the specifics a conditional relation between independently observed paired data can be used to solve regression or classification problems.

Within the Bayesian framework (Bolstad 2004; Congdon 2006; Gelman, Carlin, et al. 2020), we assume that the generative process is specified by random variables related through conditional distributions with densities, which describe how the observables would be generated: some *latent variables* are generated from *prior distributions*, and the data are generated conditionally on the latent variables. The goal is to learn the *posterior distribution* of the latent variables given the data. Whereas the generative model specifies how we assume data generation to work from latent to observed values in "forward" direction, the posterior estimate allows us to reason "backwards" from given observations to the latent values that have generated them. This can also be described iteratively, as a process of updating prior beliefs through the inclusion of new knowledge given by observations.

As an example, consider a linear regression model, where some output depends linearly on an input variable, with Gaussian noise:

$$Y \sim \text{Normal}(\theta_0 + x\theta_1, \sigma). \tag{2.1}$$

In a traditional approach, θ would be considered fixed, and estimated through least-squares optimization given pairs of observations of x and y. In a Baysian approach, though, we first decide on some prior distribution for the parameters, which are now a thought of as realizations of random variable Θ, and then try to recover the posterior distribution of Θ given the observed data. This estimate allows some more applications, compared to the single point estimate of least-squares regression. With the full posterior distribution, more complex question can be answered, such as the variance or credibility of the estimation. We can also derive a posterior predictive distribution, that models the probability of future values, taking into account the information gained by the values and variation of the already observed values (Marin & Robert 2007).

In the following, we will mostly assume that the involved random variables have densities with respect to a suitable base measure, generally written as μ: the counting measure in the discrete case, and the Lebesgue measure in the finite-dimensional continuous case (Kallenberg 2006). This allows to conveniently unify summation and integration under one notation. See appendix A for more information.

The posterior of the generative model can be expressed as a conditional distribution using Bayes' theorem. In terms of densities, we then have[1]

$$\overbrace{p(\theta \mid x)}^{\text{posterior}} = \frac{\overbrace{p(x \mid \theta)}^{\text{likelihood}} \overbrace{p(\theta)}^{\text{prior}}}{p(x)}, \tag{2.2}$$

where x are the observed data, and θ are the latent values. The posterior represents the distribution of the unobserved variables as a combination of the prior belief updated by what has been observed (Congdon 2006). θ often functions as a parametrization of the likelihood distribution. Also note that in practice, one might not be interested in all of the latent variables, but only a marginal; this corresponds to integrating out some parts of θ.

Going beyond simple applications like the example 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 Bayesian setting for usages such as prediction, certain parameter estimation methods, or exact evaluation of the normalization term $p(x)$. The problem is that almost all of the relevant quantities depend on some sort of expectation over the posterior, an integral of the form

$$\mathbb{E}[f(\Theta) \mid X = x] = \int f(\theta) p(\theta \mid x) \, \mathrm{d}\mu(\theta), \tag{2.3}$$

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

for some (measurable) function f (the integral is understood to range over the whole support of Θ). This in turn involves calculating the marginal

$$p(x) = \int p(x, \theta) \, \mathrm{d}\mu(\theta), \tag{2.4}$$

the normalization term in equation (2.2), often called the "model evidence".

When the involved distributions are of a sufficiently "nice" form, e.g., a conjugate pair (see Marin & Robert 2007, chapter 2.2.2; Murphy 2012, chapter 9.2.5), 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, naive application of summation can lead to combinatorial explosion.

Different techniques for posterior approximation are available: among them are optimization-based approaches for general graphical models, such as variational inference (Murphy 2012, chapter 21 and 22) and other methods generalized under the framework of message passing (Minka 2005). The methods described in this thesis, however, fall into the category of Monte Carlo methods, and are based on sampling (Murphy 2012, chapter 23; Vihola 2020). Their fundamental idea is to derive, for given observations x and a specified density of $\Theta \sim \pi(\cdot \mid x)$, a sampling procedure with a consistent estimator for expectations:

$$I^{(k)}(f) \to \mathbb{E}[f(\Theta) | X = x] = \int f(\theta) \pi(\theta \mid x) \, \mathrm{d}\mu(\theta), \quad \text{as} \quad k \to \infty \tag{2.5}$$

in some appropriate stochastic convergence (usually convergence in probability is enough).

Examples of such methods are rejection sampling (Devroye 1986, chapter II.3; Vihola 2020, section 4), importance sampling (Vihola 2020, section 4), and particle filters (Dahlin & Schön 2015). 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:

$$I^{(k)}(f) = \frac{1}{k} \sum_{i=1}^{k} f(Y^{(i)}) \to \mathbb{E}[f(\Theta) \mid X = x] \tag{2.6}$$

If we can sample $Y^{(k)} \sim \pi(\cdot \mid x)$ 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 (Robert & Casella 1999; Vihola 2020), which, instead of sampling exactly from the density,

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

Algorithm 2.1: General scheme for the Metropolis-Hastings algorithm.

define $Y^{(k)}$ via a (time-homogeneous) Markov chain:

$$\begin{aligned} &\mathbb{P}[Y^{(k+1)} \in \mathrm{d}y \mid Y^{(k)} = y^{(k)}, \ldots, Y^{(1)} = y^{(1)}] \\ &\quad = \mathbb{P}[Y^{(k+1)} \in \mathrm{d}y \mid Y^{(k)} = y^{(k)}] \\ &\quad = K(y^{(k)}, \mathrm{d}y) \end{aligned} \tag{2.7}$$

for all $k \geq 1$. By constructing the parameterized measure K, the *transition kernel*, in the right way, the resulting has target density $\pi(\cdot \mid x)$ as the unique stationary distribution – that means for all $\pi(\cdot \mid x)$-measurable sets A,

$$\int \pi(\theta \mid x) K(\theta, A) \, \mathrm{d}\mu(\theta) = \int_A \pi(\theta \mid x) \, \mathrm{d}\mu(\theta) = \mathbb{P}[\Theta \in A \mid X = x], \tag{2.8}$$

and the LLN for Markov chains holds. (For discrete spaces, this relation is more familiarly written as a left eigen-problem on a stochastic matrix: $\pi K = \pi$ for x fixed, and π and K considered a vector and matrix.) 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 Vihola (2020, chapter 6), Robert & Casella (1999), and Murphy (2012, chapters 24 and following) as introductions to MCMC theory and practice.

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 transition kernel $(y^{(k-1)}, y^{(k)}) \mapsto q(y^{(k-1)}, y^{(k)})$ which is easy to sample from, and an acceptance rate function α. Subsequent samples are then produced by proposing values from the conditional distribution $q(y^{(k-1)}, \cdot)$ dependent on previous chain element $y^{(k-1)}$, and incorporating them into the chain with a probability given through α (see algorithm 2.1). 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 (Green 1995), to gradient-informed methods like Metropolis Adjusted Langevin and Hamiltonian Monte Carlo (HMC) (Betancourt 2018; Girolami & Calderhead 2011).

For multi-component structures, where the latent variables have a blocked form $\Theta = [\Theta_1, \ldots, \Theta_M]$, a good proposal distribution can be hard to find, though. One way to break down the problem is to use a family of block-wise updates, given by

conditional kernels q_i operating only on the single block Θ_i, with the rest, Θ_{-i}, fixed. Then we can use the following modified proposal $\hat{Y}$ in algorithm 2.1,

$$\begin{aligned} \hat{Y}_{-i}^{(k)} &= Y_{-i}^{(k-1)}, \\ \hat{Y}_{i}^{(k)} &\sim q_i(Y_i^{(k-1)}, \cdot \mid Y_{-i}^{(k-1)}) \end{aligned} \tag{2.9}$$

(negative indices denote removal of the indicated entry). The blocks can be scalar or multivariate, the kernels may themselves be any valid transition kernel, and the sampling order of the blocks (i) can be chosen in different random or determinstic ways under some technical conditions (Vihola 2020, chapter 6.6). 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 (S. Geman & D. Geman 1984), using as a simple set of transition kernels the conditional distributions of the parameters Θ:

$$q_i(y_i^{(k-1)}, \mathrm{d}y_i^{(k)} \mid y_{-i}^{(k-1)}) = \mathbb{P}[\Theta_i^{(k)} \in \mathrm{d}y_i^{(k)} \mid \Theta_{-i}^{(k-1)} = y_{-i}^{(k-1)}, X = x] \tag{2.10}$$

(notably independent of the previous $y_i^{(k-1)}$). They can directly be used as component proposals for a within-Gibbs sampler, leading to a canceling 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. Hence, the method is a popular starting point for general probabilistic programming systems, most prominently BUGS (Lunn, Spiegelhalter, et al. 2009; Lunn, Thomas, et al. 2000) and JAGS (Plummer 2003; Plummer 2017).

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) (see Plummer 2003).

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

$$\begin{aligned} \mu_k &\overset{\text{iid}}{\sim} \text{Normal}(m, s) \quad \text{for } 1 \le k \le K, \\ Z_n &\overset{\text{iid}}{\sim} \text{Categorical}(K) \quad \text{for } 1 \le n \le N, \\ X_n &\overset{\text{iid}}{\sim} \text{Normal}(\mu_{Z_n}, \sigma) \quad \text{for } 1 \le n \le N. \end{aligned} \tag{2.11}$$

(Categorical(K) is short for a uniform categorical distribution with probabilities $1/K, \ldots, 1/K$).) To derive the conditional distribution of Z_n given the remaining variables, we start by writing down the factorization of the joint density:

$$p(z_{1:N}, \mu_{1:K}, x_{1:N}) = \prod_k p(\mu_k) \prod_n p(z_n) \prod_n p(x_n | \mu_{z_n}) \tag{2.12}$$

(with $v_{1:M}$ denoting the combined vector of all v_i). From this, we can derive an unnormalized density proportional to the conditional by removing all factors not including the target variable (which become part of the normalization constant):

$$p(z_n \mid z_{-n}, \mu_{1:K}, x_{1:N}) \propto p(z_n)p(x_n|\mu_{z_n}) \tag{2.13}$$

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 Z by summation:

$$\begin{aligned} p(z_n \mid z_{-n}, \mu_{1:K}, x_{1:N}) &= p(z_n)p(x_n|\mu_{z_n})/Z \\ &= \frac{\text{Categorical}(z_n \mid K)\,\text{Normal}(x_n \mid \mu_{z_n}, \sigma)}{\sum_{k \in \text{supp}(Z_n)} \text{Categorical}(k \mid K)\,\text{Normal}(x_n \mid \mu_k, \sigma)}, \end{aligned} \tag{2.14}$$

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

$$\begin{aligned} &p(\mu_k \mid z_{1:N}, \mu_{-k}, x_{1:N}) \\ &\quad\propto p(\mu_k) \prod_n p(x_n|\mu_k)^{\mathbb{1}(z_n = k)} \\ &\quad= \prod_n \left(\text{Normal}(\mu_k \mid m, s)\,\text{Normal}(x_n \mid \mu_k, \sigma)\right)^{\mathbb{1}(z_n = k)} \end{aligned} \tag{2.15}$$

which we recognize as a product of conjugate pairs of normal distributions ($\mathbb{1}$ being the indicator function). More examples are extensively covered in Murphy (2012, chapter 24.2).

2.2 Probabilistic Programming

Probabilistic programming is a structured way for 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 van de Meent et al. (2018):

> 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.

```
@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
```

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.

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 (2.11) from above: to perform inference on it in `Turing.jl` (Ge, Xu & Ghahramani 2018), the probabilistic programming language used in this thesis, its mathematical description might be translated into the Julia program given in listing 2.1.

We can then sample from the model in several ways using Julia:

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

A model instance `m` is created first. The value of `forward` will then be a 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 targeting the posterior, conditionally on `x_observations`, created using the MH algorithm. If we were to write out code for these two functionalities manually, in idiomatic Julia, we would end up with at least two separate functions needed for the sampler:

```
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

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 of models written in a dedicated library such as`Turing.jl`: no general interface for sampling algorithms to automatically detect all latent and observed variables; no possibility for other, nonstandard execution forms as are needed for variational inference or gradient computation for HMC; no automatic name extraction and dataframe building for chains. All these points highlight the advantages of dedicated probabilistic programming languages (PPLs) over hand-written model code. (Additionally, there is of course a benefit of reducing errors introduced by the sampling function not matching the likelihood function, or errors involving log-probabilities.)

Many PPLs are implemented as external domain-specific languages (DSLs), like Stan (Carpenter, Gelman, et al. 2017), JAGS (Plummer 2003), and BUGS (Lunn, Spiegelhalter, et al. 2009; Lunn, Thomas, et al. 2000). Others are specified in the "meta-syntax" of Lisp S-expressions, as Church (Goodman, Mansinghka, et al. 2012), Anglican (Wood, van de Meent & Mansinghka 2015), or Venture (Mansinghka, Selsam & Perov 2014). A third group is embedded into host programming languages with sufficient syntactic flexibility, for example `Gen.jl` (Cusumano-Towner et al. 2019; Cusumano-Towner 2020) and `Soss.jl` (Scherrer 2019) in Julia (besides the already named `Turing.jl`), or Pyro (Bingham et al. 2018) and PyMC3 (Salvatier, Wiecki & Fonnesbeck 2016) 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 "low-level" form, e.g., as a graph or likelihood function. In a good probabilistic programming DSL, models should be expressible very concisely and intuitively, without much bookkeeping (e.g., as close to textbook model specifications as possible). At the same time, structures should exist to allow complex behaviour, such as to

- define recursive relationships,
- write models using imperative constructs, such as loops, or mutable intermediate computations for efficiency,
- optimize details of the 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 solver, or
- integrate calls to very complex external systems, e.g. simulators or renderers.

Depending how many of these features should be supported, several possibilities for the implementation of such a DSL exist. All are based on some form of abstract interpretation. A rough distinction can be made between *compilation-based methods*, which statically translate the model code to a graph or density function, and *evaluation-based methods*, which dynamically or implicitly build such a structure at run-time, by allowing an inference algorithm to interleave the execution. The latter makes it easier to include host-language control constructs. See van de Meent et al. (2018) for a general introduction into some common implementation approaches for PPLs, and Goodman & Stuhlmüller (2014) for a detailed overview of the internals of one specific, continuation-based implementation called WebPPL (using a Lisp-based syntax).

Models in `Turing.jl` are written in `DynamicPPL.jl` syntax (Tarek et al. 2020), which transforms valid Julia function definitions into a reusable representation (`@model` is a Julia macro; see section 2.3 for more explanation). The result is a new function which produces instances of a structure of type `Model`, which in turn will contain the provided data, 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 hyper-parameters. 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.

A special distinction is made for the tilde functions of variables that are based on the model's arguments. `DynamicPPL.jl` distinguishes between *assumptions*, i.e., latent variables that should be recovered through posterior inference, and *observations*, that need to be provided when instantiating the model and are conditioned upon. The latter by default 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 behavior can be different. For this purpose, the tilde functions for the variables `x[i]` in listing 2.2 are differentiated in a conditional statement.

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, and the joint log-likelihood incremented, as happens for most MCMC samplers.

```
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
```

Listing 2.2: Slightly simplified macro-expanded code of the model in listing 2.1. The inner code is put into an `evaluator` closure, and every tilde statement is replaced by a `tilde_*` function, to which additional data and state information are passed.

In this case, one call to the `evaluator` corresponds to one 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 algorithms (which might additionally use automatic differentiation on the density evaluation procedure). All shared information for external usage is thereby conventionally 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.

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.jl`, 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. Structure-based optimizations such as partial specialization of a model to save calculations, automatic conjugacy detection (Hoffman, Johnson & Tran 2018), or model transformations such as Rao-Blackwellization (Murray et al. 2017) cannot be performed in this representation.

2.3 Compilation and Metaprogramming in Julia

To better explain the inner workings of `Turing.jl` models and the program transformations introduced later, we will now turn to an overview of Julia's evaluation, compilation, and metaprogramming techniques.

Julia (Bezanson, Edelman, et al. 2017) is a programming language with a strong, dynamic type system with nominal, parametric subtyping and elaborate multiple dispatch. It uses LLVM (LLVM Project 2019) for JIT-compilation and while it is dynamically typed, a combination of method specialization and type inference allows it to produce very optimized, fast machine code (Bezanson, Chen, et al. 2018). The language syntactically draws on a certain resemblance to Matlab, Python, or Ruby. Contrary to them, it is designed to utilize a compiler, and not primarily rely on libraries calling foreign functions (e.g., Numpy), to achieve C-like speed. Although Julia does use, e.g., BLAS and LAPACK for numerical algebra, there is nothing that fundamentally prevents implementing their functions: true array types, fast loops, and various optimizations are available, as opposed to languages like Python, which are fundamentally limited by to their dynamic interpretation. This advantage carries over to domains outside of numeric computation, of course.

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

On top of that, the language is built on a very open compilation model. Underlying the surface syntax is an abstract syntax tree (AST), that is used in the initial stages of compilation, but also exposed to the programmer through macros, which allow to transform pieces of code at compile time. Julia macros are proper hygienic, LISP-style code transformations (cf. Hoyte 2008), not simple text-substitutions as C preprocessor macros. As an example, look at the following method[3] that sums up the `sin` values of a list of numbers:

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

The invocation of the standard library macro `@show` will be treated by the compiler, during parsing, as a function call receiving as input the following data structure, representing `y += sin(x[i])` in S-expression-like form:

```
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 converted to a string used to print the value of the expression, labeled by its form in the code:

```
macro show(ex)
    blk = Expr(:block)
    unquoted = sprint(Base.show_unquoted, ex) * " = "
    assignment = Expr(:call, :repr, Expr(:(=), :value, esc(ex)))
    push!(blk.args, Expr(:call, :println, unquoted, assignment))
    push!(blk.args, :value)
    return blk
end
```

The result is then spliced back into the AST, which is compiled further as if it were written as

```
function foo(x)
    y = zero(eltype(x))
    for i = eachindex(x)
        begin
            println("y += sin(x[i]) = ", repr(var"#1#value" = (y += sin(x[i]))))
            var"#1#value"
        end
    end
    return y
end
```

(Note the automatic conversion of the symbol `:value` to a generated name `#1#value`, in order to not possibly shadow any variables from the calling scope.)

[3]The terminology of Julia uses *function* for a callable object, which can have multiple *methods* for different combinations of argument types. This is what allows multiple dispatch: when a function is applied, the types of the arguments are determined, and the most specific matching methods selected and called. For example, the + function has many methods for adding integers, floats, arrays, etc.

```
1: (%1::Core.Compiler.Const(foo, false), %2::Array{Float64,1})
  %3 = eltype(%2)::Compiler.Const(Float64, false)
  %4 = zero(%3)::Float64
  %5 = eachindex(%2)::Base.OneTo{Int64}
  %6 = iterate(%5):Union{Nothing, Tuple{Int64,Int64}}
  %7 = (%6 === nothing)::Bool
  %8 = not_int(%7)::Bool
  br §3 (%4) unless %8
  br §2 (%6, %4)
2: (%9, %10)
  %11 = getfield(%9, 1)::Int64
  %12 = getfield(%9, 2)::Int64
  %13 = getindex(%2, %11)::Float64
  %14 = sin(%13)::Float64
  %15 = (%10 + %14)::Float64
  %16 = repr(%15)::String
  %17 = println("y += sin(x[i]) = ", %16)
  %18 = iterate(%5, %12)::Union{Nothing, Tuple{Int64,Int64}}
  %19 = (%18 === nothing)::Bool
  %20 = not_int(%19)::Bool
  br §3 (%15) unless %20
  br §2 (%18, %15)
3: (%21)
  return %21
```

Listing 2.3: SSA-form of the lowered form of the method `foo(::Vector{Float64})` as defined defined above, annotated with inferred types (as through `@code_warntype`).

After macro expansion, the code of the method is *lowered* into an intermediate representation consisting of only function calls and branches. This comprises of several transformations: for one, certain syntactic constructs are "desugared" into primitive function calls. For example, array accesses, `x[i]`, are replaced by calls to the library function `getindex(x, i)`. The for loop in the example is converted into a while loop using the `iterate` library function:

```
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
```

Consequently, all nested expressions are split apart, so that only simple, unnested calls remain, and any subsequent assignments to variables are linearized to a series of definitions, with newly introduced names of the form `%i`. The remaining control flow statements (e.g., while loops and conditionals) are represented through a sequence of labeled *basic blocks*, with (possibly conditional) branches between them (with return statements forming a special kind of branch without a block target). The sequence of assignments is further processed into *single static assignment (SSA) form* (Rosen, Wegman & Zadeck 1988; Singer 2018), the characteristic property of which is that every variable is assigned exactly once, by giving a unique, position-independent name to each intermediate value of an expression. By introducing this

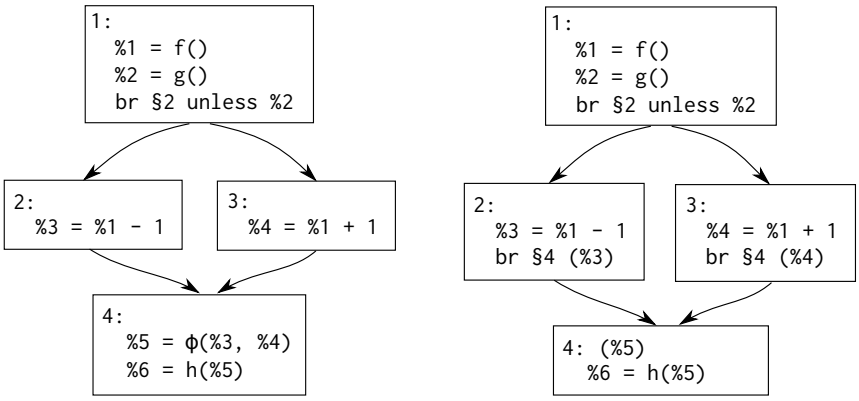


Figure 2.1: Two control flow graphs of the same function, illustrating the correspondence between SSA representations using ϕ-functions and block arguments. The SSA variables `%3` and `%4` correspond to the values of `y` in the two branches, which are merged in `%5`.

immutability guarantee and flattened structure, the resulting code is, in a certain sense, referentially transparent, which facilitates program flow analyses, and makes many transformations easier (Muchnick 1997). Accordingly, SSA form is widely used in intermediate forms of compiler systems (such as LLVM (LLVM Project 2019), or Swift (Apple 2020)), simplifying transformations and optimizations. The result of the translation of our example into three basic blocks can be found in listing 2.3.

explain listing

There is one notable complication regarding conversion to SSA form: we need to be able to distinguish between assignments of variables arising from "joined" control flow paths. Consider the assignment of `y` in the following code example:

```
x = f()
if !g()
    y = x - 1
else
    y = x + 1
end
h(y)
```

Here, the value of `h(y)` depends on two possible locations of `y` – hence, we cannot simply rename every variable in a naive way. Instead, in the variant of SSA form used in this text and most of Julia, values of variables that are assigned in multiple parent blocks are passed on as *block arguments*, as in figure 2.1 on the right, and subsequently in this work. This makes basic blocks resemble local functions, and cleanly resolves the problem of joins just like functions handle variable inputs. The traditional, functionally equivalent alternative is to introduce *ϕ-functions* (Rosen, Wegman & Zadeck 1988), which are defined ad-hoc to distinguish between several values depending on the control path taken before. This form is shown in the same figure on the left.

In the next step, type inference is applied. Until now, the operations involved were purely syntactic in nature, and could be performed by solely transforming the

code of the function `foo`, without taking into account semantic information. As soon as `foo` is called on a concrete type during evaluation, though, a specific overload, called a *method*, must be chosen. To apply type inference on the body of a given function, the most specific method fitting to the argument types of each call will be selecte. If we go with the example and and consider `foo([1.0])`, with `Vector{Int}` as the sole argument type, the types as annotated in listing 2.3 will be inferred.

The last step of compilation happening within Julia consists of inlining and optimizing the typed intermediate code, resulting in the form shown in listing 2.4. explain listing more There, several method calls have been inlined, and concrete argument types to invoke been inferred. This is in true, traditional SSA form, with all variable slots eliminated, and block arguments converted to the mentioned ϕ-functions. Finally, this representation will be translated and sent to LLVM, where further optimization can happen, and machine code will be generated and executed, as well as stored for later usage as part of the just-in-time compilation mechanism.

A KEY PRINCIPLE in Julia's compilation model is type specialization (Bezanson, Chen, et al. 2018). As we have seen, whenever a function call is reached during evaluation, the concrete types of the arguments are first determined, and then the most specific method selected. This automatically gives the language dynamic semantics: a Julia implementation can theoretically perform type-based dynamic dispatch at every call. In reality, the Julia compiler at this point combines multiple dispatch and JIT compilation into one of the main principles of optimization. Instead of dynamically evaluating the code of a function at every call, methods are JIT-compiled the first time they are used. Their compiled code is then cached in a method table, which is used for lookup at subsequent calls. Note that method compilation does not happen recursively at once: only when the body of a compiled method is actually executed with concrete arguments, the same process is performed again, for each invoked method.

So, in a sense, JIT compilation can be seen as a function that returns compiled code, given a function and a tuple of types. Similar to macros, which transform original code, given an expression, the process of generating compiled methods from argument types is customizable in Julia: via so-called *generated functions*. These are a form of staged programming (Bolewski 2015; Rompf & Odersky 2010), a paradigm in which code generation is controlled via regular types and functions, instead of specially priviledged ones as macros are. Such generated functions, when called, are not directly translated into machine code: instead, they emit new code to the compiler based on the types of their arguments. The new code is then JIT-compiled. For example, when we have two methods of a function `f`:

```
f(x::Int) = println("Int")
f(x::String) = println("String")
```

we could replace them with the following generated function:

```
@generated function f_generated(x)
    if x == Int
```

```
1 ── %1  = arraysize(x, 1)::Int64
│    %2  = slt_int(%1, 0)::Bool
│    %3  = ifelse(%2, 0, %1)::Int64
│    %4  = 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 = 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 = arrayref(true, x, %14)::Float64
│    %17 = invoke sin(%16::Float64)::Float64
│    %18 = add_float(%13, %17)::Float64
│    %19 = sle_int(1, 1)::Bool
└───       goto §7 if not %19
6 ── %21 = 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 getindex(()::Tuple, 1::Int64)::Union{}
└───       $(Expr(:unreachable))::Union{}
10 ─       goto §11
11 ─       goto §12
12 ─       goto §13
13 ─       goto §14
14 ─ %32 = invoke :(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 println("y += sin(x[i]) = "::String, %32::String)::Any
│    %37 = (%15 === %3)::Bool
└───       goto §19 if not %37
18 ─       goto §20
19 ─ %40 = 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 = not_int(%44)::Bool
└───       goto §22 if not %45
21 ─       goto §5
22 ─ %48 = ϕ (§20 => %18, §4 => 0.0)::Float64
└───       return %48
```

Listing 2.4: Typed and optimized code of the call `foo([1.0])` in SSA form, as obtained through `@code_typed` (the extra bars are due to the formatting of `CodeInfo`).

```
        return :(println("Int"))
    elseif x == String
        return :(println("String"))
    else
        error("Method error")
    end
end
```

Calling `f_generated(1)` will then determine the argument type (`typeof(1) == Int`), and pass it to the function body of `f_generated`. There, the conditional will select the first branch, and the expression `:(println("Int"))` will be returned. This is now passed back to the compiler, which will lower the code and compile the method for `Int` arguments, and store the result in the method table. The stored code can then be executed – on the arguments that were used to determine the type tuple the generated function has been called with! The next time `f_generated` is executed, the function body is *not* executed anymore, but the generated code of the function defined through the expression `:(println("Int"))` directly looked up[4]. Of course, simply replacing dispatch, as with this example, is not what generated functions are used for in practice. Most applications concern parametric types with statically known shape arguments, such as tuples, named tuples, or array ranks. They can also be used for type-level computations on values that become known only at run-time, through singleton types such as `Val`.

The direct generation of code, given argument types, is however not the furthest we can go. For one, generated functions are not only allowed to return `Expr` objects – the internal representation of the surface AST – but also `CodeInfo` objects, which are the internal representation of lowered code in (almost) SSA form. This, on its own, would not be of much use most times, but there is a second, more interesting feature: it is possible to retrieve the lowered representation of a method programmatically, given a function and an argument type tuple. Combining these two, we now have all the tools to implement IR-level code transformations as follows:

1. Define a generated function, taking as arguments another function and its arguments.
2. Within this function, obtain the IR of the method of the passed-in function for the remaining arguments.
3. Transform this IR however necessary.
4. Return the IR, which will now be compiled and called on the actual arguments.

Importantly, unlike macros, such transformations can be performed *recursively*: one simply inserts the same generated function to inner function calls during the transformation in step 3. Since the transformation operates not during parsing, the function to be transformed needs not be known beforehand, and not be present literally in the code – the generated function can be called on every available callable object, at any time during run-time. This makes it possible to transform even

[4] A caveat: technically, the compiler is still free to call the generating code multiple times – which is the reason generated functions must never involve side effects or depend on external state.

functions from other libraries, internally calling yet other functions. One particular example of this principle is source-to-source automatic differentiation, as shown in the next section: a call to a function `gradient(f, x, y)` can obtain the IR of the method for `f` on `typeof(x)` and `typeof(y)`, produce differentiated code, and call the result on `x` and `y`. Naturally, differentiating `f` involves recursively differentiating the other, unknown functions within it, too (down to "primitive" functions, whose derivative is known), and combining the results using the chain rule.

This metaprogramming pattern is extremely powerful, and becoming more and more popular. It allows to change evaluation semantics in more profound ways than multiple dispatch can: by rewriting the code of the called function, it is possible to change what invoking a method within its body means. Through this, several abstract interpretation algorithms can be realized, by extending the existing data path with additional metadata (such as automatic differentiation, or other forms of information propagation analysis (Singer 2018, part II)), or non-standard execution be implemented (e.g., continuation-passing style transformations). There exist already two Julia packages with the goal of simplifying this kind of transformations: `Cassette.jl`[5], which provides overloadable function application by a so-called "overdubbing" mechanism, abstracting out some common patterns; and `IRTools.jl`[6], which has a more user-friendly alternative to `CodeInfo`, and a macro similar to `@generated` that makes writing recursive IR-transformation easier by directly using this data structure. The latter is what the work of this thesis builds on.

2.4 Automatic Differentiation and Computation Graphs

This section will explain some of the interrelations between automatic differentiation, computation graphs, and IR transformations, to be able to understand how SSA-form representation is a natural structure for extracting and analyzing computation graphs, and how the necessary transformations arise in practice. To appreciate how the form of the computation graphs interacts with the mathematics, some foundations need to be introduced first.

Many algorithms in machine learning and other domains can be expressed as optimization problems over a multivariate function with scalar output – typically a loss function over a parameter space, which measures the performance of a model for a specific task. The parameters minimizing the loss function then define the optimal model. When the loss function is (sub-)differentiable, there exist a variety of gradient-based optimization methods to minimize the loss (at least in terms of a practically sufficient local minimum).

[5] `https://github.com/jrevels/Cassette.jl`
[6] `https://github.com/FluxML/IRTools.jl`

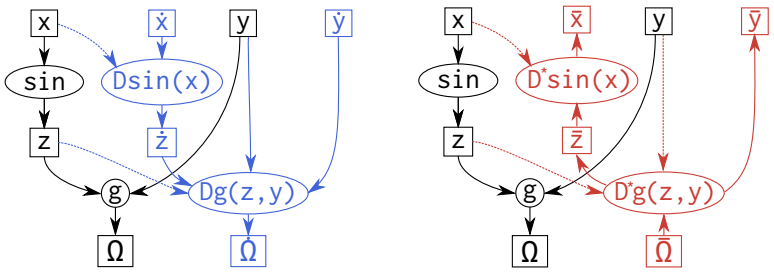
While in some cases the loss function is simple enough to find the gradient by hand, in general, the model, and therefore the loss function, may be specified in terms of rather complicated programs, for which hand-writing derivatives is difficult to infeasible. For this reason, computerized methods for differentiation have been developed. These can be categorized into three classes:

- Finite differences
- Symbolic differentiation
- Automatic differentiation

In finite differences, the idea is to discretize the definition of derivatives, and numerically evaluate the function within an environment. This is simple to implement, but does not scale well with the dimension of the involved space, and can become numerically unstable in various ways (Press et al. 2007, section 5.7). Symbolic differentiation works through representing the functions in question as symbolic algebraic objects, and applying the differentiation rules as one would manually. This does not lose precision or introduce divergence, but can suffer from blow-up of the size of the generated expressions; additionally, it requires the functions to be expressed in a custom representation, different from normal functions or programs (Baydin et al. 2018).

Automatic differentiation or AD, the third category, is perhaps unfortunately named – it does not signify much at first sight. The relevant idea is to not start from functions as black-box or symbolic objects, but from programs. Then the perturbation that makes up the value of the derivative at a point is propagated through the steps of the program. For this to work, there needs to exist an explicit representation of the computation graph at the evaluation at a point, which is what makes the topic relevant for this thesis. In contrast to the former two methods, AD relies on numeric, not symbolic evaluation, but is (up to the floating point errors already present in the input function) exact – no discretization error, as in finite differences, is introduced. For a more detailed treatment, I refer to Griewank & Walther (2008), the standard work on the topic, and the survey by Baydin et al. (2018), which includes a comparison of state-of-the-art implementations.

The appendix B contains a more formal introduction of the underlying concepts. In short, we treat a program as the composition of functions, and derivatives as equally-shaped compositions of the linear operators they represent. Since composition is associative, there are many possible orders for evaluation of these composed derivatives. The two main fashions of AD, *forward mode* and *backward mode*, correspond to evaluation aligning with the original function evalution order, and its reverse. These composition orders naturally can be described through the computation graphs of functions, and transformations on them. This is illustrated in figure 2.2, where we see the computation graph of a simple expression `g(sin(x), y)`, with input variables `x` and `y` and output Ω. There, the graphs of the corresponding forward-mode (left) and backward-mode (right) calculations are related to the computation graphs. The dotted ($\dot{x}$) and barred ($\bar{x}$) values are the perturbation values



(a) Forward mode. (b) Backward mode.

Figure 2.2: Computation graph and intermediate expressions of the expression `g(sin(x), y)`, together with the derivative graphs in forward- and backward mode. Dashed arrows indicate re-use of forward values in the derivative graph.

propagated through the derivative graphs, corresponding to intermediate variables in the original.

In this setting, forward-mode AD is simply an efficient way to calculate the *Jacobian-vector product* $J(x)\Delta$, or equivalently the total derivative at x for a fixed perturbation Δ, avoiding full matrix multiplication. Appling this to all all basis vectors, we get back the gradient. Backward mode, on the other hand, calculates the product of the Jacobian with a dual vector. This, in fact, is nothing else than a *vector-Jacobian product* with the transposed Jacobian. Recovering the (transposed) gradient of a loss function of type $\mathbb{R}^N \to \mathbb{R}$ then reduces to evaluating it at a constant scalar output perturbation of 1. Notably, this in involves only one pass over the graph, while in forward mode, N passes are required. For this reason, backward mode enjoys a much more prominent role in gradient-based machine learning.

The practical implementation of AD in programming languages opens up another set of possible choices. One way is to use an external, compiler-based system that transforms a complete program in a subset of a standard programming language (e.g., Tapenade, which transpiles Fortran and C code (Tapenade developers 2019)) or in a custom specification, as is done in Stan (Carpenter, Hoffman, et al. 2015). But both of these examples are really applied in niche cases: large numeric simulations, and log-densities in a probabilistic model. Moreover, these systems lack flexibility in programming, especially concerning abstractions and interaction with other libraries, and require external tooling besides a main programming language. Recently, the Swift for TensorFlow project (Hong & Lattner 2018; TensorFlow Developers 2018) introduced a modern variant of this by extending the compiler of the Swift programming language with facilities to perform automatic differentiation internally, and some features to simplify graph operations required by TensorFlow.

The second possibility is *operator overloading*. Forward mode can rather easily be added to any existing programming language which has a sufficiently extensible system for overloading mathematical operators by implementing dual numbers (see

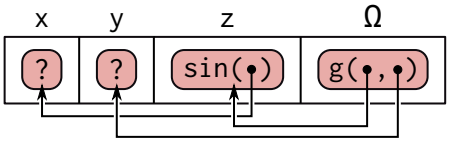


Figure 2.3: Wengert list of the example function `g(sin(x), y)` introduced above. Every intermediate variable becomes an element, linked through pointers. The gradient can be calculated by backward traversal and accumulating the adjoint values as metadata in the list elements.

section B.1). This can be done using ad-hoc polymorphism with traits (Amin 2016) or type classes (Wadler & Blott 1989), or by dynamic dispatch, which is what is used in Python and Julia. The latter is especially versatile in this respect, since every function can be extended to a new type of dual numbers by simply adding a method; unlike in Python, where only certain operators are open to extension – a fundamental limitation of its single dispatch, object oriented approach.

Backward-mode AD can be implemented using operator overloading as well, but this requires more effort. Since adjoint values cannot be simply threaded through in parallel to forward evaluation, one needs to build up a data structure during the forward pass, which can at the end be traced back in reverse order. One possibility of doing this is to use closures (function objects capturing an environment), but the usage of many higher order functions might lead to unwanted heap allocation and makes understanding harder.

The alternative is to use a tape structure, or *Wengert list* (Baydin et al. 2018, section 3). On such a list, the computation graph is stored as by pointers between elements, as shown in figure 2.3. The Wengert list can also be constructed through an operator overloading approach, which is exactly what graph-based machine learning frameworks do: PyTorch (Paszke et al. 2017), TensorFlow (Abadi, Agarwal, et al. 2015) in eager mode, DyNet (Neubig et al. 2017), and Chainer (Tokui et al. 2015). In these, the programmer interacts with a library mirroring the usual numerical functions, but operating on a special "variable" or "tensor" type. These operations are overloaded so that function calls, in addition to performing the primal calculations, are stored either explicitly on a global Wengert list structure, or implicitly in the constructed expression objects. Then, one can start a backward pass from any leaf to propagate back derivatives to the roots of the computation graph, by following the edges and summing up adjoint values in parent nodes' metadata. JAX (Bradbury et al. 2018) carries the idea further and allows general composable source transformations to implement not only differentiation, but also vectorization, parallelization, and other syntactic abstractions over functional programs written in Python, over a unified intermediate representation that is recovered from an original function by tracing.

This style of implementation has limitations, though: it requires building up many objects at run-time, and is completely oblivious of control structures. Additionally, the code expressing differentiable functions has to be written entirely in the DSL, in a library-aware fashion, preventing the usage of third-party functions

and language features, and forcing the user to adhere to certain semantic constraints that cannot be verified statically by the host language. TensorFlow in graph mode addresses some of these points. It builds up a complete expression graph, which is differentiated symbolically, and is therefore somewhat in the middle between operator overloading (since the graph is still a run-time data structure) and a static transformation (the resulting graph is not interpreted in the host language, but converted to run on an "accelerator", which can one of several kinds of processing device – CPU, GPU, TPU,…). It still requires to stick to the provided expression types and library functions, though.

Efforts to overcome these limitations lead to the third kind of approach: language-internal *source transformation*. Recent work in Julia (Innes 2018) has shown that the available metaprogramming mechanisms (described in section 2.3) allow to systematically derive "adjoint programs" for arbitrary user-provided Julia programs, given only a set of *primitive adjoints* (such as derivatives of built-in functions). This approach works purely structurally on the Julia IR, employing generated functions to analyze functions' code and transform them completely, including third-party functions and data types, and control flow. The key insight here is that SSA-form IR already resembles the structure of Wengert lists, extended by branches. As in building up reverse computation graphs, the adjoint code will therefore invert the control flow of the basic blocks in the primal function, taking into account that data flow may involve dynamic dependencies. Differentiation through data types and closures is supported via a unified treatment of them in a tuple-like form, with constructors and accessors (inspired by cons-cells in Lisp).

An implementation of this principle has been released as the `Zygote.jl` package[7]. In similar spirit, there is also work on directly differentiating the LLVM intermediate representation, by extending the compiler pipeline with a differentiation pass that comes after all language-specific and high-level optimizations (Moses & Churavy 2020), released as the Enzyme project[8]. Furthermore, there are applications that use the same techniques for other purposes, like sparsity detection (Gowda et al. 2019) or concolic execution (Churavy 2019) (cf. discussion in section 3.3).

Internal source-based methods can therefore be composable, extensible, and more user-friendly, since no special treatment of programs to be differentiated is required: primal functions can be implemented as any other regular function in the host language. A source-transformation approach also completely avoids the obscure issue of "perturbation confusion", which leads to hard-to-find errors when using nested differentiation with dual numbers (Baydin et al. 2018; Manzyuk et al. 2019).

As a concluding note, all these graph operations reveal that automatic differentiation is really only a special case of message passing algorithms in computation graphs (Minka 2019). Other learning methods that can be described as message passing are optimization algorithms (Dauwels, Korl & Loeliger 2005; Ruozzi 2011)

[7] `https://github.com/FluxML/Zygote.jl`
[8] `https://enzyme.mit.edu/`

and a variety of variational approximations (Minka 2005; Winn & Bishop 2005). Hence, it is no surprise that computation graphs play a large role as the foundation of other learning algorithms for probabilistic models, such as described below.

3 Implementation of Dynamic Graph Tracking in Julia

As described previously, there is a trade-off between source-transformation methods and library-based (operator overloading) approaches for tracking computation graphs. Since the ultimate goal of this work is to analyze dynamic probabilistic models written in `Turing.jl`, properties of both are desireable. An operator overloading approach has been considered, since it would have allowed to potentially reuse AD implementations, but was thought insufficient, because the structure of control flow and recursion are lost. Inspired by the work of Innes (2018), it seemed most promising to start from a source-transformation based approach implemented over the IR, especially from a usability point of view. The advantages of using a transformation of the IR over the surface AST are the same: there is less overhead from handling multiple syntactic forms, and naming is already referentially transparent. Additionally, there are existing Julia packages to simplify handling the IR data structures and set up the transformations.

However, the dynamicity of the trace structure of general probabilistic programs needs to be preserved and exposed to the user, for each function evaluation – which is different from the AD usage, where the adjoint function is already the ultimate goal, and does not change with the arguments. Hence, a method for a hybrid version was developed: through an IR transformation, the original code of a function to be tracked should be extended by additional statements to record a trace of the executed statements and control flow operations at run-time. The algorithm and structures on which this approach is based have already been shortly described in Gabler et al. (2019), and will be explained more extensively below. An open source implementation is available online[1].

As we have seen above, in section 2.3, generated functions allow the inspection and transformation of the IR of given functions. This technique can be applied to recursively traverse the implementation of a given function, annotating each operation with necessary tracking statements, and changing the inputs and outputs accordingly to extract this information from outside. To ensure sufficient generality, we requite the following properties of the tracking system:

1. Storage of all intermediate values during execution.

[1] `https://github.com/TuringLang/IRTracker.jl`

2. Symbolic capture of intermediate expressions and branches in an analyzable, graphical form.
3. Preservation of the relation of each part of the structure to the corresponding original IR.
4. Proper nesting of this information for nested function calls, making relations between arguments and function inputs recoverable.
5. Correct handling of constants and primitive functions in the IR.
6. Extensibility of the tracking functions, to allow multiple possible ways to analyze code (e.g., by different definitions of what should be recorded).
7. A way to add custom metadata to the recorded structure during tracking.

This kind of operation will be similar to the (explicit) construction of Wengert lists in backwards-mode AD (see section 2.4); but contrary to there, the nested call structure and control flow shall be preserved as well. Hence, we call this structure *extended Wengert list*.

3.1 Extended Wengert Lists

The extended Wengert list structure is implemented in Julia through nested objects of an abstract supertype `AbstractNode`, with several concrete subtypes for the different kinds of nodes. Additionally, there are special types for the tape- and block references, and an expression type `TapeExpression`, mimicking the built-in `Expr`, but adding more semantic distinctions (such as between references and constants, and between primitive and non-primitive function calls). On top of this, an API to query the graph structure is provided, allowing, for example, to find all children or parents of a tape reference up to a certain depth, or extract data from nodes, such as referenced variables, arguments, or metadata.

Figure 3.1 illustrates the resulting extended Wengert list for one run of a short stochastic function:

```
geom(n, beta) = rand() < beta ? n : geom(n + 1, beta)
```

(for readability, the result is displayed to only three levels of nesting). The function draws a sample from the geometric distribution with parameter `beta`, starting to count at value `n`. On the left, we have its IR in textual form, consisting of two blocks. The central part is the graph of nested nodes. There, values and jumps from the top-level call are recorded in their encountered order, as nodes with "tape references" `@1` to `@9`. SSA variables (`%i`) occurring in expressions of SSA definitions are also replaced in the nodes by the respective tape references. Each node is linked to the original IR statement it records, as indicated by the red arrows.

In the lower middle part, we see the node corresponding to the statement `%7 = geom(%6, %3)`. It is recorded at reference `@8` with expression `geom(@7, @3)` and value `4` (the notation ⟨`geom`⟩ indicates that `geom` is a constant, and `()...` stands for the absence of optional arguments). The values of the arguments of this call can be inspected by looking up the respective references. Since `geom` is not a primitive

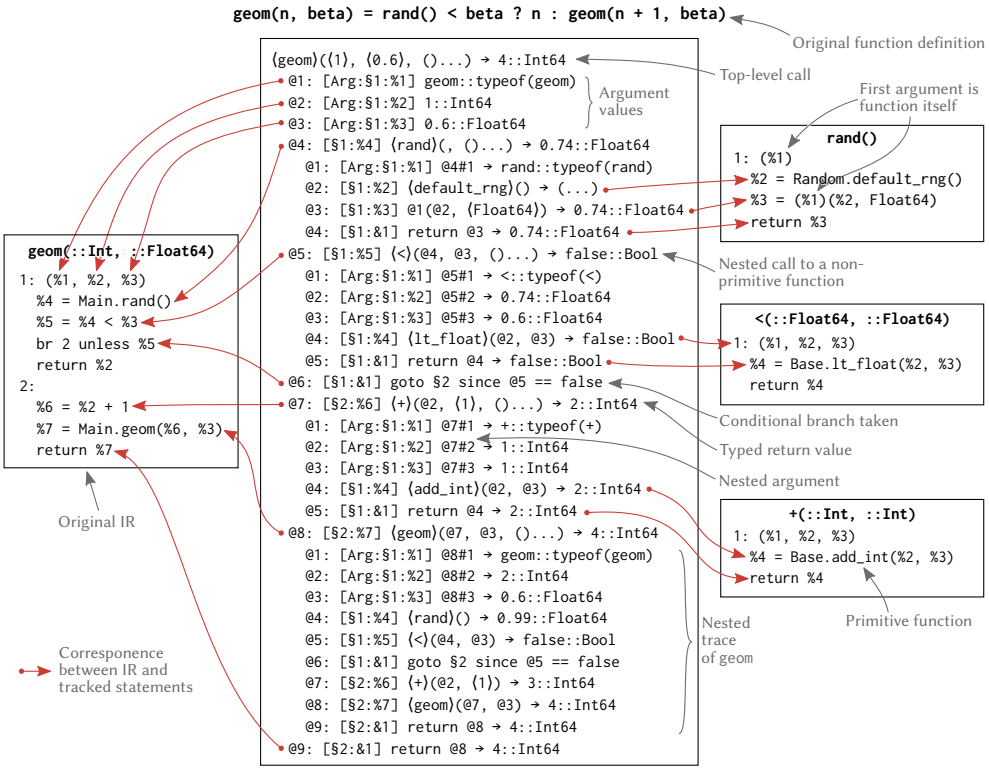


Figure 3.1: Extended Wengert list for one run of the stochastic function geom (only three levels shown). The central box is the tracked graph of the call `geom(1, 0.6)`. The other boxes show the original IR of the called non-primitive functions, to which the nodes are linked. Angle brackets indicate constant values.

function, the node holds tape of child nodes as well. In this case, it is equivalent to the top level, due to the recursivity of `geom`. We can see the three arguments `@1`, `@2`, and `@3`, corresponding to the block arguments `%1`, `%2`, and `%3`, with the value of `@2` being now 2 instead of 1. Further we can see function calls of `rand` and `<` as well as a conditional jump, corresponding to the branch the original IR, followed by calls of `+` and `geom`. Following back the tape references from the result value `@9`, the data path of the trace can be extracted. It can be used for reverse-mode AD, and only these nodes would be recorded in a conventional Wengert list. In this work, however, the system also records the nodes on the control path, consisting of `@6` and the nodes it depends on.

3.2 Automatic Graph Tracking

Recording an extended Wengert list requires to capture all block arguments, SSA definitions, and taken branches, with their actual values and metadata. This is achieved by extending the IR with new statements creating nodes and recording

them on the extended Wengert list structure described in the previous section. Care needs to be taken to properly record function calls, since we need to ensure that non-primitive functions are recursively tracked.

The recording functionality is implemented as a transformation using a generated function operating on the IR, using the `IRTools.jl` package, as described in section 2.3. The resulting IR consists of about three to five times as many statements as the original. This is a small constant overhead per statement, but should not affect complexity using proper implementation techniques, such as a good choice of collection structures and paying attention to type stability. The basic blocks and control structure are preserved, except for the redirection of return statements to the one block at the end. Due to JIT compilation, the transformation is performed at most a constant number of times per method, and then stored as compiled code. However, the tracking – the recording of all statements in the extended Wengert list structure – happens at every execution during run-time, which is an important feature to allow the application to dynamic models.. Furthermore, the extended code is available to all standard optimizations performed in the following passes of type inference and lowering.

The transformed code of the example function `geom`, whose untracked IR is given in figure 3.1 above at the sides, is displayed in figure 3.2. First, a "graph recorder" object is passed into the function via the extra argument `%5`. In this, the original IR is stored for later access. Subsequently, every original SSA statement is replaced by a call to one of the `trackedX` functions, to which both the function and its arguments, wrapped into `TapeExpressions` directly (for constants) or indirectly (through `trackedvariable` and `trackedargument`, which preserve the symbolic mapping to SSA variables). The `record!` function takes care of constructing the child node of the possibly nested call, and storing them on the recorder object.

make sure figure is on same spread as explanation

To get a more detailed understanding, consider the SSA statement `%6 = %2 + 1` in the second block of `geom`, which describes the application of the function `+` to an SSA variable and a constant. The IR of the block is shown again in 3.2 on the box on the lower right, with corresponding pieces of IR highlighted in matching colors (`%6` being transformed into `%33` to `%37` on the left).

1. First, a constant node `%33` for the function is set up.
2. Then, the variable argument in is tracked in `%34`. There, `trackedvariable` has the purpose to correctly relate the node in the trace (`@2`) to the original SSA variable `%6`. This is necessary since a block could be visited multiple times during tracking – for example, if it belongs to a loop body – which requires to give multiple, unique names to references to the same original variable. Additionally, `trackedvariable` copies values, since the tracked information would otherwise no preserve intermediate values correctly in the case of mutations.
3. Next, both of the function arguments are packed into the tuple `%35`; the second argument, which was a literal value `1` in the original IR, is preserved as a literal as well (wrapped into a `QuoteNode` object for technical reasons).
4. Finally, the function and arguments are passed to the function `trackedcall`,

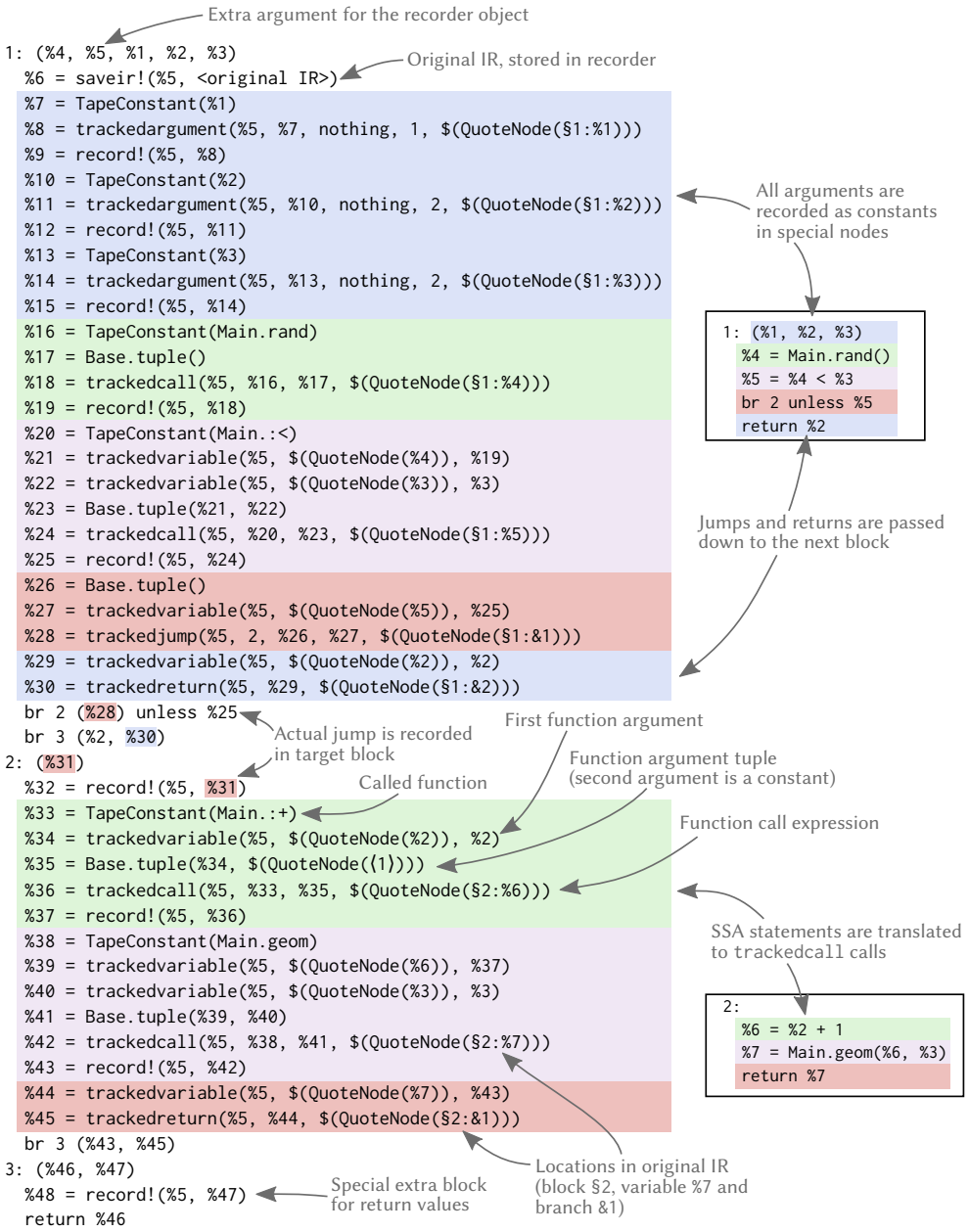


Figure 3.2: Tracked IR of the method `geom(::Int, ::Float64)`. Corresponding parts in original and transformed IR are highlighted in matching colors. (The original IR consists of two blocks, shown separately on the right.)

```
struct DepthLimitContext <: AbstractTrackingContext
    level::Int
    maxlevel::Int
end

DepthLimitContext(maxlevel) = DepthLimitContext(1, maxlevel)
canrecur(ctx::DepthLimitContext, f, args...) = ctx.level < ctx.maxlevel

function trackednested(ctx::DepthLimitContext, f_repr::TapeValue,
                       args_repr::ArgumentTuple{TapeValue}, info::NodeInfo)
    new_ctx = DepthLimitContext(ctx.level + 1, ctx.maxlevel)
    return recordnestedcall(new_ctx, f_repr, args_repr, info)
end
```

Listing 3.1: Implementation of a tracking context to limit the nesting depth to a maximum (which is part of the implemented package).

which takes care of actually calling the original function. Doing so, it will, if the called function is not considered primitive, recursively track it as well, and pack the resulting child nodes into a new nested node, together with the return value. Otherwise, the result will simply be stored in a special primitive-call node.

5. The resulting node is then stored on the recorder; this operation at the same time returns the value, `%37`, which is needed in subsequent calculations (in this case, in `%39`, as the argument of the recursive call to `geom`).

Branches, tracked with `trackejump` and `trackedreturn`, cannot be stored on the recorder object before the respective jumps are taken. The solution is to first construct the respective nodes of all possible branches of a block (`%28`), and adding them as an extra argument to the old branches. Then, in each target branch, the jump node from which the branch originated is recorded immediately (in this case, in statement `%32`). As a special case, all return branches are converted to unconditional jumps to one new block at the end, which contains a single unified return branch. (For illustration, `%28` and `%30`, the branch variables of block 1, are highlighted in colors matching to the original branches in figure 3.2.) This way, return branches can be treated in the same way as internal branches. An more formal description in pseudo-code is given in algorithm 3.1.

Lastly, some special dispatch is used for the transformation to work correctly on certain special kinds of function calls, such as built-in functions, type application, and `ccall` primitives, which require more careful handling. These functions are detected and replaced by `trackedspecial` tracking statements appropriately.

To provide some modularity and extensibility to the system, it also affords customization of some behavior by *tracing contexts*. All of the `trackedX` functions explained above, used directly in the transformed code, are really special methods that work directly on the recorder object. Their behavior – namely, performing the actual method calls and constructing the nodes – is defined in another method of the same function, which dispatches on a context object stored in the recorder object, and can be overloaded by the user for a custom context.

Input: `original_ir`
Output: New IR with tracking statements inserted

Initialize empty IR object `new_ir`

for `old_block` in `blocks(original_ir)` **do**
    Add an empty block `new_block` to `new_ir`
    if this is the first block **then**
        Add set up for `%recorder`
    end if

    ▷ *Handle arguments*
    Copy all arguments from `old_block` to `new_block`
    Add tracking and recording for each argument

    ▷ *Take care of branch recording in target blocks*
    if there exist branches to `old_block` **then**
        Add new argument `%branch_node` to `new_block`
        Add recording for `%branch_node`
    end if

    ▷ *Transform all statements*
    for `stmt` in `statements(old_block)` **do**
        Add tracking and recording for `stmt` to `new_block`
    end for

    ▷ *Transform all branches*
    for `branch` in `branches(old_block)` **do**
        if `branch` is a return branch **then**
            Add tracking for a return node corresponding to `branch`
            Add a branch replacing the original return
            Pass the original return value and the return node as branch arguments
        else
            Add tracking statement for a branch node corresponding to `branch`
            Copy the original `branch`
            Pass the branch node as extra argument to the branch
        end if
    end for
end for

▷ *Set up return block*
Add new block to `new_ir`, with arguments `%return_value` and `%return_node`
Add recording of `%return_node`
Add return branch, returning `%return_value` and `%recorder`

Algorithm 3.1: Overview of the IR transformation to record an extended Wengert list. This transformation happens inside a generated function called by `trackcall`, which assembles the resulting value and IR into a new node with the correct metadata. The details of statement tracking and branch transformation are explained in the text; the description of metadata recording, and the mechanisms to correctly rename SSA variables during the transformation and tape references at run-time were left out for simplicity.

This allows, for one, to overload the notion of what constitutes a *primitive function*. In the default context, primitive functions are only those that do not have IR on their own (such as intrinsics and functions defined in C), which leads to very large recursive traces. To circumvent this, we can introduce a new `DepthLimitContext`, as shown in listing 3.1. There, the function `canrecur` is overloaded to stop at depth `maxlevel`; this method will be called to determine whether a tracked function is considered primitive. Besides this, we also have to redefine the behavior of `trackednested` to specify that for non-primitive functions, i.e., nested calls, the level remembered in the context object should be updated. `recordnestedcall` is a built-in function of the library that simply performs the recursive tracking, then.

From this we see that `trackedcall` is only a thin wrapper around a conditional statement over `canrecur`, `trackednested`, and its sibling `trackedprimitive`. Beyond this, context dispatch allows a user to change any other of the tracking functions as well. This can be used to store custom metadata, calculate information during tracking, or even change return values or nesting dynamically. In addition to those methods, also `trackedargument`, `trackedreturn`, and `trackedjump` can be customized, which we have seen in the example; furthermore, there are `trackedspecial`, `trackedconstant`, and `trackederror`. `trackedvariable` is more primitive and cannot be overloaded, since this would change the relation between references of tracked nodes. More information is available on the package's public repository.

3.3 Evaluation

The extended Wengert list created by tracking a function can be used for many purposes in which computation graphs are required. All algorithms that can be formulated as message-passing can directly work on this, as well as all methods that operate on run-time dependency graphs, from simple debugging to concolic execution (see discussion below).

As a proof of concept, a small backward-mode AD system was implemented in the form of a context. This simply required storing the derivative operators for all intermediate values during the forward pass, and writing a backward pass as graph traversal on the resulting computation graph.[2] The implementation has been tested on some simple composed functions, but is not intended for serious application. Due to the very abstract nature of the implementation, not more individual evaluations of it are performed, except unit tests to ensure basic correctness of the interface. The approach discussed in chapter 4 illustrates a more realistic use-case of the proposed system, and serves thus a larger integration test.

More potential use cases arise when the tracked model is actually static – in this case, the complete structure can be recovered from one graph tracking pass. This graph can then be analyzed and used in various ways; even more so, when more semantic knowledge about the model program exists, such as meanings of certain domain-specific functions. Examples of this are conjugacy detection as described by

[2] `https://github.com/TuringLang/IRTracker.jl/blob/master/test/test_backward_ad.jl`

Hoffman, Johnson & Tran (2018), and automatic Rao-Blackwellization as in Murray et al. (2017).

The implementation is limited in two respects. First, in practical terms, there are some trade-offs to be made regarding the storage of intermediate results and functions in nodes. In the current design, nodes are parameterized by the types of their contents, which leads to very large types, and potential slow-down during type inference. Not doing this would prevent type stability of the transformed code, since all of the intermediate values that are passed directly in the original code are wrapped in node structures and unwrapped again. There are even still some cases in which the parametrization does not eliminate type instability. Alternatively, original values could be passed unwrapped into the `trackedcall` functions, besides the node arguments; this would lead to more complicated handling of values, though.

The other, more fundamental restriction is inherent to dynamic tracing: alternative paths, that were not taken due to run-time control flow, are not recorded. Compared to a traditional operator overloading system, `IRTracker.jl` does preserve the information about which branches were taken, and for which reason in the case of conditional branches, but this is not enough for complete static analysis in all cases.

One possible direction for extension is concolic execution (Zeller et al. 2019), in which the function is traced multiple times with different arguments, whose exact values are determined by constraint solving so that all possible execution paths are covered. This is potentially slow, and goes against the spirit of the idea of tracking once, in parallel to the normal forward execution. Also, it is not applicable to general user-defined types, but constrained to whatever theories the used SMT solver supports. Alternatively, a method to merge control paths in the transformed function could be conceived. This might, however, suffer from exponential blow-up in several cases, is difficult to get right in the presence of mutation, and has complicated theoretical properties (e.g., termination of the resulting code might be undecidable).

As another future direction, it is conceivable that a composable context system could be designed, such that, for example, one could perform automatic differentiation and dependency graph tracking within one tracking pass. However, this would require more careful design, since it is unclear how to deal with potential non-commutativity or non-associativity of the effects of contexts in different orders (e.g., which one gains priority in the decision about nested or primitive tracking).

4 Graph Tracking in Probabilistic Models

The system described in chapter 3, implemented in a Julia package `IRTracker.jl`, can now be utilized for the analysis of probabilistic models written in `DynamicPPL.jl`, and for posterior inference in `Turing.jl`. This part of the work is realized in another package, `AutoGibbs.jl`, which is available as open-source code[1]. There are two applications provided, built on top of the graph tracking functionality: first, dependency analysis of random variables in a model can be performed. This results in the complete graphical model for static models, and a slice of it for dynamic models. The resulting graph can be plotted for visualization. Second, given the dependency graph, the conditional likelihoods of unobserved variables in static models can be extracted. With these, analytic Gibbs conditionals can be derived and used in `Turing.jl`'s within-Gibbs sampler.

4.1 Dependency Analysis in Dynamic Models

In order to use `IRTracker.jl` to extract the dependencies in a probabilistic model written in `DynamicPPL.jl`, we need to remember the structure of such models, which was introduced in section 2.2: there is one evaluator function, into which the original code is transformed, and which evaluates the model in different modes. This function has the same structure as the original code, but adds some more complicated bookkeeping logic to it, and transforms the tilde statements into function calls with some additional metadata. Furthermore, when calling the model as a callable object, there are several layers of dispatch (about five layers of nesting, depending on the arguments), until the real evaluator function is actually hit. On the other hand, there is no further nesting involved beyond the evaluator function – `Turing.jl` simply does not support nested models, for technical reasons.

Therefore, we at first need to introduce an `IRTracker.jl` context that will record all the internal function calls down to the evaluator function, and stop there. Similar to the `DepthLimitContext` demonstrated on page 36, the main task here is to overload the `canrecur` method to stop at the right call. This can easily be done by introducing a helper predicate function `ismodelcall` that dispatches on the involved types.

[1] `https://github.com/phipsgabler/AutoGibbs.jl`

Next, we notice that the resulting computation graph consists of a nested and quite unusable structure, due to the initial levels of nesting. To work with the model code, we need to strip the outer layers off the inner node containing the trace of the evaluator function. Thirdly, many of the statements in the trace of the evaluator function do not have relevance for dependency analysis – like those that stem from internal calculations done by the model, or statements that were written by the user but to not lie on the dependency graph, such as debugging statements or the lowered code of for loops, in some cases. These we can strip off in advance, so as to clean the raw dependency trace. These three preparation steps are put together in one method:

```
function slicedependencies(model::Model{F}, args...) where {F}
    trace = trackmodel(model, args...)
    strip = strip_model_layers(F, trace)
    slice = strip_dependencies(strip)
    return slice
end
```

Here, `trackmodel` extracts the computation graph with the context for models tracking, `strip_model_layers` removes the outer method calls, and `strip_dependencies` removes all SSA code that is not on the dependency graph spanned by the sampling statements.

The final and most intricate step is to add all the remaining SSA statements to a new graph structure, that describes a more domain-specific representation. In this `Graph` type, only assumption, observation, call, and constant nodes remain, containing relevant metadata such as their values, variable names, and distribution objects. In addition, the object stores intermediate information used during its construction, such as the mapping between newly generated and original references. The graph construction is implemented in a function `makegraph`, and we finally have one exported function

```
function trackdependencies(model, args...)
    slice = slicedependencies(model, args...)
    return makegraph(slice)
end
```

There are two complications regarding `makegraph`. For one, model arguments are handled specially by `DynamicPPL.jl` – there are some internal arguments added, and the original arguments are inspected to allow to run the same model in generative or posterior mode. This part needs to be sorted out, so that the passed argument values are correctly set up as constants in the dependency graph, but since all information is present, the task is resolved by correctly identifying the arguments and restructuring their contents into the right form.

The other problem is the handling of mutation, and tracking of modified array elements. For example, a hidden Markov model might contain code like this:

```
s = zeros(Int, N)
s[1] ~ Categorical(K)
for i = 2:N
    s[i] ~ Categorical(T[s[i-1]])
```

```
end
```

In order to express the dependency between successive elements of `s`, an empty array is first set up, and then subsequently populated by the results of the tilde statements describing the Markov process. In this form, only the individual variables `s[i]` are recognized by the model language. Internally, the tilde statements are translated to array assignments of the form `s[i] = tilde_assume(...)`, but with additional lowering of the involved arguments, after which the corresponding IR will look approximately like this:

```
%9 = %i - 1                                    # i - 1
%10 = getindex(%s, %9)                         # s[i - 1]
%11 = getindex(%T, %10)                        # T[s[i - 1]]
%12 = VarInfo{:s}(((%i,),))
%13 = Categorical(%11)
%14 = tilde_assume(..., %13, ..., %12, ...)  # %14 ~ %13
%15 = setindex!(%s, %14)                       # s[i] = %14
```

(to be understood symbolically, not as real SSA – several statements have been collapsed). We see that the direct association between the variable `s` is not preserved in the line of the tilde method, but spread over multiple statements. Even worse, since all statements for the different `s[i]` result in mutations of `%s`, the immediate dependency between `s[i]` and `s[i-1]` is not available structurally, but must be recovered dynamically.

The `makegraph` implementation solves this by successively identifying mutated arrays representing random variables by inspecting the indexing calls around tilde statements, and storing the association between the assumption and the array elements. This part of the procedure is the most intricate one, and not complete; there may exist cases where mutation is able to "circumvent" the dependency analysis. Additionally, the matching between indexing arguments involves some careful treatment of variable names; the existing `DynamicPPL.jl` API for this functionality is not very comprehensive. Due to this, the current implementation of `AutoGibbs.jl` currently only supports "simple" indexing by one tuple of integers. Other, more general indexing styles allowed in Julia could be added in future extensions. Furthermore, broadcasting tilde statements, that are supported in `DynamicPPL.jl`, are not supported by `AutoGibbs.jl` either.

As an example for the resulting graphs, take the two simple models in listing 4.1. The pretty-printed dependency `Graph` objects extracted from them are shown in listing 4.2 below. We can see that the model arguments for observations occur as constant values, and all of the intermediate transformation visible in the original model definitions are observed. From this structure, `AutoGibbs.jl` can construct output in the Dot graph format and visualized using GraphViz (Gansner & North 2000). The visual outputs of the example models is shown in figure 4.1.

```
@model function bernoulli_mixture(x)
    w ~ Dirichlet(2, 1/2)
    π ~ DiscreteNonParametric([0.3, 0.7], w)
    x ~ Bernoulli(π)
end

@model function hierarchical_gaussian(x)
    λ ~ Gamma(2.0, inv(3.0))
    m ~ Normal(0, sqrt(1 / λ))
    x ~ Normal(m, sqrt(1 / λ))
end
```

Listing 4.1: Two simple example models: a mixture of two Bernoulli random variables with fixed probabilities, and a Gaussian model with conjugate prior. Both models are defined over one single observation.

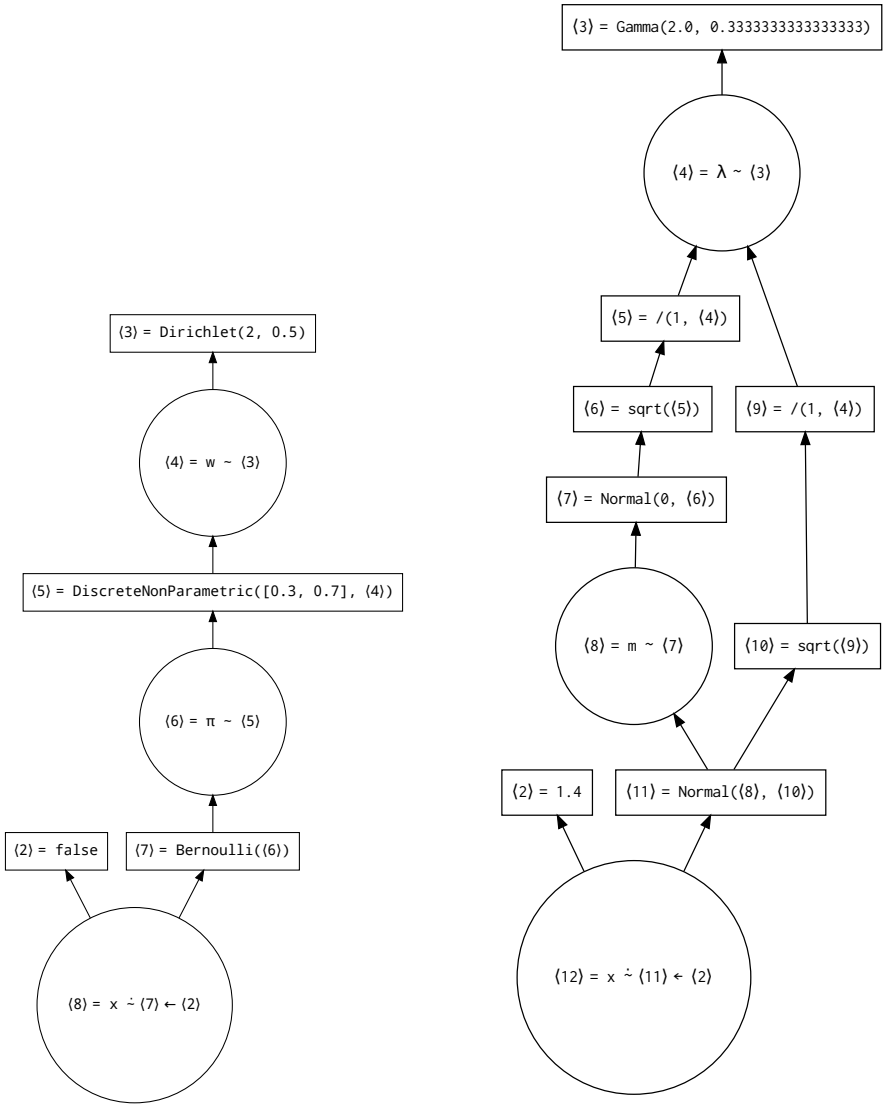
```
⟨2⟩ = false
⟨3⟩ = Dirichlet(2, 0.5) → Dirichlet{Float64}(alpha=[0.5, 0.5])
⟨4⟩ = w ~ ⟨3⟩ → [0.826304431175434, 0.17369556882456608]
⟨5⟩ = DiscreteNonParametric([0.3, 0.7], ⟨4⟩) → DiscreteNonParametric{...}(
        support=[0.3, 0.7], p=[0.826304431175434, 0.17369556882456608])
⟨6⟩ = π ~ ⟨5⟩ → 0.3
⟨7⟩ = Bernoulli(⟨6⟩) → Bernoulli{Float64}(p=0.3)
⟨8⟩ = x ~ ⟨7⟩ ← ⟨2⟩
```

(a) Trace of `bernoulli_mixture(false)` (some type parameters not shown).

```
⟨2⟩ = 1.4
⟨3⟩ = Gamma(2.0, 0.3333333333333333) → Gamma{Float64}(
        α=2.0, θ=0.3333333333333333)
⟨4⟩ = λ ~ ⟨3⟩ → 0.9257859525673857
⟨5⟩ = /(1, ⟨4⟩) → 1.0801632896100921
⟨6⟩ = sqrt(⟨5⟩) → 1.0393090443222806
⟨7⟩ = Normal(0, ⟨6⟩) → Normal{Float64}(μ=0.0, σ=1.0393090443222806)
⟨8⟩ = m ~ ⟨7⟩ → 1.8505166567138398
⟨9⟩ = /(1, ⟨4⟩) → 1.0801632896100921
⟨10⟩ = sqrt(⟨9⟩) → 1.0393090443222806
⟨11⟩ = Normal(⟨8⟩, ⟨10⟩) → Normal{Float64}(
         μ=1.8505166567138398, σ=1.0393090443222806)
⟨12⟩ = x ~ ⟨11⟩ ← ⟨2⟩
```

(b) Trace of `hierarchical_gaussian(1.4)`.

Listing 4.2: Traced structure of the two example models introduced above. Values in ⟨angle brackets⟩ denote intermediate values (similar to SSA variables), and right arrows denote the resulting values of function calls. The left arrow indicates the source of the observed value.



(a) `bernoulli_mixture(false)` (b) `hierarchical_gaussian(1.4)`

Figure 4.1: Dependency graphs of the models in listing 4.1, generated by `AutoGibbs.jl` and rendered by GraphViz. More information, such as node values, is stored in the real model graph, but not printed for better readability. Circular nodes denote tilde statements, while deterministic intermediate values, corresponding to normal SSA statements, are written in rectangles.

4.2 Automatic Calculation of Gibbs Conditionals

The ultimate contribution of this work is to utilize the dependency extraction system to extend `Turing.jl` with JAGS-style automatic calculation of Gibbs Conditionals. In JAGS (and its sibling, BUGS) conditional extraction works over a wide range of variable types (Plummer 2003) by symbolic analysis and recognition of several patterns (e.g., conjugate distributions from exponential families, log-concave or compactly supported distributions; see Lunn, Thomas, et al. (2000).), which is possible since the class of models is constrained by the modeling language, and available in completely structured form.

In `Turing.jl`, models are much less restricted, and the symbolic form has to recovered from outside, as we have seen. To focus on the principal ideas and not to extend the scope too much, the implementation described in this section was restricted to finite, discrete conditionals, which are trivial to sample from, given the respective log-density. Since the construction of conditional log-densities is independent from the normalization step, though, this can serve as a starting point for further, more general conditional samplers, as those in JAGS and BUGS. Additionally, and this is a more fundamental limitation, the models to which the extraction algorithm can be applied must be static in a specific sense: the whole Markov blanket of the variable in question must be unique, and reachable within one run of model tracking. A large fraction of the models used in practice do fulfill this condition, though. As this problem is difficult so solve in general, the same constraint applies to JAGS and BUGS, which makes `AutoGibbs.jl` not more limited than these.

The implementation of the conditional extraction system involves three main steps:

1. Extracting the symbolic form of the conditional likelihood of Markov blankets in a given dependency graph.
2. Constructing closures calculating the normalized discrete conditionals from these likelihoods.
3. Providing a Gibbs-component sampler for `Turing.jl`, that can utilize the resulting conditional distributions.

The third step turned out to be the easiest, since the sampling system of `Turing.jl` is designed to be extensible. Ideally, a Gibbs-conditional sampler would have first been added to `Turing.jl` and then simply been reused for `AutoGibbs.jl`; in practice, it worked out the other way round, and the `AutoGibbs.jl` sampler has, in generalized form, been added to `Turing.jl` afterwards (without the automatic extraction, only supporting user-provided conditional distributions).

Step 1, the symbolic extraction of likelihood functions, is implemented by first converting the full trace into a symbolic joint log-density. Therefor the expression of each node in the dependency graph is associated with a corresponding symbolic representation of a function of the "trace dictionary" θ, which holds the values of the random variables by name (this is to view the probabilistic model as a joint distribution over trace dictionaries). This is done in the following simple fashion:

```
⟨2⟩ = 1.4                    ↝ 1.4
⟨3⟩ = Dirichlet(2, 0.5)      ↝ Dirichlet(2, 0.5)
⟨4⟩ = w ~ ⟨3⟩                ↝ logpdf(Dirichlet(2, 0.5), θ[w])
⟨5⟩ = DNP([0.3, 0.7], ⟨4⟩)  ↝ DNP([0.3, 0.7], θ[w])
⟨6⟩ = π ~ ⟨5⟩                ↝ logpdf(DNP([0.3, 0.7], θ[w]), θ[π])
⟨7⟩ = Bernoulli(⟨6⟩)         ↝ Bernoulli(θ[p])
⟨8⟩ = x ~ ⟨7⟩ ← ⟨2⟩           ↝ logpdf(Bernoulli(θ[p]), θ[x])
```

(a) `bernoulli_mixture(false)`

```
⟨2⟩ = 1.4                    ↝ 1.4
⟨3⟩ = Gamma(2.0, 1/3)        ↝ Gamma(2.0, 1/3)
⟨4⟩ = λ ~ ⟨3⟩                ↝ logpdf(Gamma(2.0, 1/3), θ[λ])
⟨5⟩ = /(1, ⟨4⟩)              ↝ /(1, θ[λ])
⟨6⟩ = sqrt(⟨5⟩)              ↝ sqrt(/(1, θ[λ]))
⟨7⟩ = Normal(0, ⟨6⟩)         ↝ Normal(0, sqrt(/(1, θ[λ])))
⟨8⟩ = m ~ ⟨7⟩                ↝ logpdf(Normal(0, sqrt(/(1, θ[λ]))), θ[m])
⟨9⟩ = /(1, ⟨4⟩)              ↝ /(1, θ[λ])
⟨10⟩ = sqrt(⟨9⟩)             ↝ sqrt(/(1, θ[λ]))
⟨11⟩ = Normal(⟨8⟩, ⟨10⟩)     ↝ Normal(θ[m], sqrt(/(1, θ[λ])))
⟨12⟩ = x ~ ⟨11⟩ ← ⟨2⟩        ↝ logpdf(Normal(θ[m], sqrt(/(1, θ[λ]))), θ[x])
```

(b) `hierarchical_gaussian(1.4)`

Figure 4.2: Association of the dependency graph of the example models from listing 4.1 with intermediate symbolic functions. The expressions on the right are implicit functions of θ. (`DNP` is used instead of `DiscreteNonParametric` to avoid breaking lines.)

- References to call nodes or constant nodes (⟨i⟩ = x) are inlined.
- References to tilde nodes (⟨j⟩ = v ~ D) are converted to dictionary lookups: θ[v].
- Call nodes are converted to functions from the trace dictionary to a function call on the converted references: f(⟨i⟩, ⟨j⟩) ↝ f(x, θ[v]).
- Tilde nodes are converted to log-density evaluations of their values given the corresponding distribution: ⟨j⟩ = v ~ D ↝ logpdf(D, θ[v])

All resulting expressions are thereby to be understood as implicit functions of θ. These new expression function objects can then be numerically evaluated as log-densities for given values of all random variables. For illustration, the joint densities of the `bernoulli_mixture` and `hierarchical_gaussian` models introduced above in listing 4.1, are associated with corresponding symbolic functions as shown in figure 4.2. By adding the log-likelihoods for each tilde statement, we get the symbolic log-joint density as, for example,

fix caption alignment

```
logpdf(Gamma(2.0, 0.333333), θ[λ]) +
  logpdf(Normal(0, sqrt(/(1, θ[λ]))), θ[m]) +
  logpdf(Normal(θ[m], sqrt(/(1, θ[λ]))), θ[x]),
```

corresponding to the density over λ, m, and x, factorized as

$$p(\lambda, m, x) = p(\lambda)\, p(m \mid \lambda)\, p(x \mid m, \lambda). \tag{4.1}$$

From this we can then derive conditionals in the usual way of normalizing the proportional conditional, which can be obtained by removing all terms of the joint factorization that do not depend on the conditioned variable:

$$\begin{aligned} p(m \mid \lambda, x) &\propto p(m \mid \lambda)\, p(x \mid m, \lambda), \\ p(\lambda \mid m, x) &\propto p(\lambda)\, p(m \mid \lambda)\, p(x \mid m, \lambda), \end{aligned} \tag{4.2}$$

which in more technical terms are given through the *Markov blanket* of m and λ (Murphy 2012, section 24.2; Koller & Friedman 2009, section 4.5).

The crucial problem here is, of course, to find the normalization factor, as always in Monte Carlo methods. Normalization could, for example, be implemented by analysing the structure of the resulting expression and detecting conjugacies, such as the normal/normal-gamma relationship between m, λ, and x above. The simplest possible case, however, occurs when when a conditioned variable has finite support; and as mentioned above, this is what has been implemented out in this work. For example, p in the `bernoulli_mixture` model is such a finitely supported variable – we get

$$p(\pi \mid w, x) = \frac{p(w)\, p(\pi \mid w)\, p(x \mid \pi)}{\sum_{\varpi \in \{0.3, 0.7\}} p(w)\, p(\varpi \mid w)\, p(x \mid \varpi)} \tag{4.3}$$

Since the distribution of every variables is preserved in the dependency graph, we now can do the same thing programmatically, and turn the symbolic log-density into a distribution object by simply tabulating the values of the denominator through evaluating of the expression over the whole support of π, the set $\{0.3, 0.7\}$, and summing it up to get the normalization factor. (This uses the interface of distribution objects from the `Distributions.jl` package, which have a `support` method whose result is an iterable.)

Concretely, the construction works as follows, finalizing step 2 of the above scheme, exemplified by `bernoulli_mixture`:

1. Find the likelihood expressions that match a given conditioned variable (this includes indexed variables subsumed by a parent, like `v[i]` and `v`), and their distribution:

   $$\ell_1 = \texttt{logpdf}(\texttt{DNP}([0.3, 0.7], \theta[\texttt{w}]), \theta[\pi]).$$
   $$\mathcal{D} = \texttt{DNP}([0.3, 0.7], \theta[\texttt{w}]).$$

2. For each of these (sub-)variables, collect the likelihoods of their children variables, thus completing the Markov blanket:

   $$\ell_2 = \texttt{logpdf}(\texttt{Bernoulli}(\theta[\pi]), \theta[\texttt{x}])$$

   The complicated part of this and the previous step is the correct matching of indexed variables in the trace dictionary: forms like θ`[v][1]` and θ`[v[1]]` need to be resolved correctly to the same value.

3. Construct for each a closure function that takes as an argument a fixed trace dictionary, tabulates the conditional log-likelihood over it with the conditioned variable fixed to all values of its support, and normalizes the result:

Better typesetting?

$$
\begin{aligned}
&\theta \mapsto \{ \\
&\quad \Omega = \texttt{support}(\mathcal{D}) \\
&\quad \texttt{table} = [\text{eval}(\ell_1, \theta[\pi \rightsquigarrow \varpi]) + \text{eval}(\ell_2, \theta[\pi \rightsquigarrow \varpi]) \mid \varpi \in \Omega] \\
&\quad \text{DNP}(\Omega, \text{softmax}(\texttt{table})) \\
&\}
\end{aligned}
$$

(whereby $\text{softmax}(x) = \exp.(x)/\sum_i \exp(x_i)$ is the normalization operation on log-probabilities).

The result of this process is a collection of closures that represent the conditional likelihoods as "kernels": functions from conditioned-on variables to distribution objects. These closures can then be used to construct a conditional sampler for usage in `Turing.jl`'s `Gibbs` sampler, in combination with other samplers for the continuous variables.

As for potential improvements, there is of course a wide range of possibilities, as the current implementation is a most primitive one. As has been mentioned before, further classes of random variables beyond those with finite support could be handled, using methods and heuristics as in BUGS or JAGS. This could even involve symbolic methods like those in AutoConj (Hoffman, Johnson & Tran 2018). More generally, variance-reducing transformation, as those in Murray et al. (2017), are applicable. For computational efficiency, a parallel Gibbs sampler (Gonzalez et al. 2011) could be constructed, improving scalability with increasing data size (i.e., numbers of observations). Or, as in `Gen.jl`, a variant of "argument diffs" could be devised to prevent unnecessary re-evaluation of model parts (see Cusumano-Towner 2020, section 1.2.3; Becker 2020).

Besides improvements via inference algorithms, it would be possible for models that are written in a vectorized or otherwise "trace constant" fashion, such that the structure of the conditionals does not change with the number of observations, to record the trace for a small model and reuse it for arbitrary larger ones, thus avoiding recomputation and recompilation. Finally, the evaluation of the conditional closures, which is currently performed by simple interpretation of expressions, could be sped up by compiling them to Julia methods, or even better by reusing the SSA-like structure to emit Julia IR directly.

Realistically, though, I consider it more worthwhile to follow a different approach, like the one outlined in section 5.1 below, and moving away from working on a trace-based reconstruction and to a domain-specific intermediate representation with generalized transformation capabilities. On such a representation, all of the named ideas still apply, but it would have the advantage of being more invariant to a specific inference system, and be able to handle more general probabilistic programs (foremost, not only static ones).

4.3 Evaluation

To begin with a qualitative assessment, it must be admitted that `AutoGibbs.jl` is, when run manually and only once, empirically so noticeably slow, that a user may be tempted to dismiss it outright (concretely speaking, extraction of one conditional for some not-so-large models takes 20 to 200 seconds on the author's laptop). This is a valid point, but two counter-arguments must be considered. For one, the implementation is a preliminary one, more conceptual than optimized. Much of the slow-down could be mitigated by just improving key parts of `AutoGibbs.jl` and `IRTracker.jl`.

In addition to that, one must realize where this apparent slowness comes from: namely, from compilation, and therein primarily type inference, of the functions handling all the strongly typed expression trees. Besides the possibility of just optimizing these further, the following fact is most important to realize: compilation takes place only once – as soon as a conditional is constructed, it can be reused in arbitrarily many sampling runs of the same model. The finished conditionals then do not take so much time anymore, quite the contrary: they are much faster than other within-Gibbs samplers, since they only involve evaluating a fixed expression, constructing a distribution, and sampling from it once (and even this could be sped up further). This makes it possible to sample much longer chains in the same time, which is an overall advantage.

Furthermore, due to the limitations `AutoGibbs.jl` puts on the structure of variable names, there are cases in which models cannot be formulated in a certain desirable way in `DynamicPPL.jl`, thus losing certain advantages such as block-wise treatment of collections of independent variables. This can lead to a disadvantage compared to other samplers. Again, the restrictions are mostly a detail of the current implementation, and there is no theoretical hindrance in overcoming than.

So, to conclude, while the implementation in current form is not applicable in practice for all possible cases, it is very much so in principle. Based on the lessons learned through this work, further contributions to Gibbs sampling in `Turing.jl` are already planned (although not necessarily based on `AutoGibbs.jl`).

For a more quantitative point of view, let us now turn to some empirical evaluations. Besides several unit tests for correctness of the derived dependencies and conditionals on a variety of small models chosen to test certain features and corner cases, an experimental comparison of `AutoGibbs.jl` and existing `Turing.jl` samplers has been conducted. Three off-the-shelf Bayesian models were chosen: a Gaussian mixture model (GMM) with known variances and priors over cluster centers, weights, and assignments (Marin & Robert 2007, section 6.2):

$$\begin{aligned} w &\sim \text{Dirichlet}(\text{K}) \\ z_n &\sim \text{Categorical}([1, \dots, K], w), \quad n = 1, \dots, N \\ \mu_k &\sim \text{Normal}(0, \sigma_1), \quad k = 1, \dots, K \\ x_n &\sim \text{Normal}(\mu_{z_n}, \sigma_1), \quad n = 1, \dots, N; \end{aligned} \tag{4.4}$$

a hidden Markov model (HMM) with known variances and priors over transition and emission probabilities (Marin & Robert 2007, section 7.3):

$$\begin{aligned} T_k &\sim \text{Dirichlet}(K), \quad k = 1, \ldots, K \\ m_k &\sim \text{Normal}(k, \sigma_1), \quad k = 1, \ldots, K \\ s_1 &\sim \text{Categorical}([1, \ldots, K], [1/K, \ldots, 1/K]) \\ s_k &\sim \text{Categorical}([1, \ldots, K], T_{s_{k-1}}), \quad k = 2, \ldots, N \\ x_k &\sim \text{Normal}(m_{s_k}, \sigma_2), \quad k = 1, \ldots, N; \end{aligned} \tag{4.5}$$

and an infinite mixture model (IMM) in stick-breaking construction, but otherwise of the same form as the GMM, to represent a nonparametric example (Hjort et al. 2010, section 2.2):

$$\begin{aligned} w &\sim \text{TruncatedStickBreakingProcess}(\alpha, \text{K}) \\ z_n &\sim \text{Categorical}([1, \ldots, K], w), \quad n = 1, \ldots, N \\ \mu_k &\sim \text{Normal}(0, \sigma_1), \quad k = 1, \ldots, K \\ y_n &\sim \text{Normal}(\mu_{z_n}, \sigma_2), \quad n = 1, \ldots, N. \end{aligned} \tag{4.6}$$

In the context of this work, the three interesting classes of metrics are

1. the "extraction time" of `AutoGibbs.jl`, i.e., the time it takes to extract and a conditional including compilation times,
2. the sampling speed when used as component of a within-Gibbs sampler, and
3. the quality of the resulting chains, in terms of convergence and variance diagnostics. (Although this really measures Gibbs sampling, not the implementation of `AutoGibbs.jl`, it is a relevant comparison for the practitioner.)

To estimate them, a series of experiments has been conducted. Each of the test models involves one discrete and two continuous parameter arrays. As a baseline for `AutoGibbs.jl`'s static conditional (AG), `Turing.jl`'s Particle Gibbs sampler (PG) was chosen, which is also suited to discrete parameters. PG was always used with 100 particles, since lower values did not lead to convergent chains. Continuous variables were all sampled using Hamiltonian Monte Carlo (HMC) with hand-tuned parameters (10 leapfrog steps with step size 0.05). The experiments have been set up to vary between AG and PG, and between 10, 25, and 50 observations, since their number determines the size of the trace, and thus influences both AG's compile times and the overall sampling time. All measurements were conducted using the following system configuration, as shown by `InteractiveUtils.versioninfo()`:

```
Julia Version 1.3.1
Commit 2d5741174c (2019-12-30 21:36 UTC)
Platform Info:
  OS: Linux (x86_64-pc-linux-gnu)
  CPU: Intel(R) Core(TM) i5-4690 CPU @ 3.50GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-6.0.1 (ORCJIT, haswell)
```

Everything was executed single-threaded, with exclusive resource access on the server, to preclude measurement noise as much as possible. Each of the three models was benchmarked in a separate Julia session. The last two chains of the HMM with 50 observations using Particle Gibbs could not be completed, due to the twelve hour time limit set by the job scheduler. The raw data can be found in Gabler (2020).

The `DynamicPPL.jl` implementations of the models can be found in listing 4.3. Note that for each, there exists a second, equivalent implementation used for PG, since particle samplers in `Turing.jl` require the usage of special data types due to their task copying mechanism.

It follows a detailed analysis of the results per model in graphical form, based on six plots each. First, PG and AG are compared in terms of sampling times by number of observations on the top left. Right of this, we have the dependency of the extraction time of AG given the number observations. The first of these points is almost always an outlier, since it involves additional compilation time. Through the rest, a quadratic function is fitted, which always matched the values very closely.

On the recto pages, plots for convergence analysis are visualized. Since these are estimated per parameter, and the test models involve larger arrays of parameters, the first elements of the continuous and the fifth element of the discrete parameters have always been chosen as representatives. On the top, the first chain of each experimental combination is shown as a qualitative representative (thinned for ease of plotting). Well-converging chains should look like white noise, not like a random walk. In the middle plot, the estimated autocorrelation functions of the same chains are given, which are a way to evaluate convergence speed by eye. The functions should vanish quickly (indicated by the gray significance bar around the abscissa); uncorrelated samples would have zero autocorrelation except for the first lag.

Lastly, two diagnostic values are plotted for each combination and chain. The "scale-reduction" $\widehat{\mathrm{R}}$, after Gelman & Rubin (1992), estimates the factor by which the scale of the current distribution might be reduced if the simulations were continued (see Gelman, Carlin, et al. 2020, p. 285). It should be close to one for indicating good convergence; as a rule of thumb, a value of more than 1.1 is suspicious. The effective sample size (ESS) arises in the estimation of the asymptotic variance of Markov chains (Vihola 2020, section 7.2), where it is formally analogous to the sample size in the variance estimator for i.i.d. variables. The ESS can also be interpreted as a one-point summary of the autocorrelation estimate, normalized by chain length. It should be close to the actual number of samples, or at least of the same order of magnitude.

```
@model function gmm(x, K)
    N = length(x)
    w ~ Dirichlet(K, 1/K)  # Cluster association prior
    z ~ filldist(Categorical(w), N)  # Cluster assignments
    μ ~ filldist(Normal(0.0, s1_gmm), K)  # Cluster centers

    for n = 1:N
        x[n] ~ Normal(μ[z[n]], s2_gmm)  # Observations
    end
end

@model function hmm(x, K, ::Type{T}=Float64) where {T<:Real}
    N = length(x)

    T = Vector{Vector{X}}(undef, K)
    for i = 1:K
        T[i] ~ Dirichlet(K, 1/K)  # Transition probabilities
    end

    s = zeros(Int, N)
    s[1] ~ Categorical(K)
    for i = 2:N
        s[i] ~ Categorical(T[s[i-1]])  # State sequence
    end

    m = Vector{T}(undef, K)
    for i = 1:K
        m[i] ~ Normal(i, s1_hmm)  # Emission probabilities
    end

    x[1] ~ Normal(m[s[1]], s2_hmm)
    for i = 2:N
        x[i] ~ Normal(m[s[i]], s2_hmm)  # Observations
    end
end

@model function imm_stick(y, α, K)
    N = length(y)
    crm = DirichletProcess(α)
    v ~ filldist(StickBreakingProcess(crm), K - 1)
    w = stickbreak(v)  # Cluster weights

    z = zeros(Int, N)
    for n = 1:N
        z[n] ~ Categorical(w)  # Cluster assignments
    end

    μ ~ filldist(Normal(0.0, s1_imm), K)  # Cluster centers

    for n = 1:N
        y[n] ~ Normal(μ[z[n]], s2_imm)  # Observations
    end
end
```

Listing 4.3: Gaussian mixture model, hidden Markov model, and infinite mixture model using a stick-breaking construction. The two-step calculation of `w` via `v` is a technicality due to `Turing.jl`'s handling of nonparametric models. The function `stickbreak` normalizes the stick-lengths `v` into a Dirichlet-like distribution. The `Categorical(p)` constructor automatically infers the support of the categorical distribution from the weight vector as `1:length(p)`.

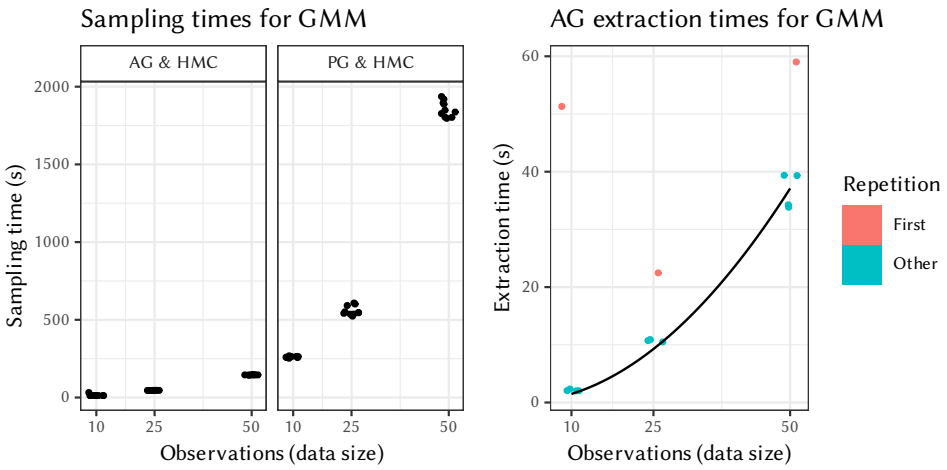


Figure 4.3: Sampling and extraction times for GMM, factored by algorithm and number of observations. Points are jittered horizontally to increase readability. A quadratic curve is fitted to the extraction times.

Gaussian Mixture Model

For the GMM, the AG sampling times lie consistenly below the minimum of the PG sampling times, even with the largest number of observations. Extraction time seems to grow quadratically, with exception of the first call of the conditional extraction, involving compilation and type inference.

The mixing behaviour of the chains shows a large variation. With 10 and 25 observations, neither of the altgorithms reaches consistently satisfactory results; the distribution of $\widehat{R}$ values is quite diffuse and suspiciously large (more so for PG), and especially the ESS numbers are way too low. A look at the exemplary autocorrelation plots seems to confirm bad convergence. The corresponding chains clearly show random-walk-like or "lumped" behaviour for some combinations.

For 50 observations, the result is different with AG. The w and μ parameters appear to converge well in most cases, with very low-variance chains and visibly large ESS values. But for unknown reasons, the z parameter seems to have gotten "stuck" in this particular example and not moved at all, which is the reason no autocorrelation function could be estimated. PG might have improved somewhat, looking at the lower $\widehat{R}$ distributions, but not enough to make a meaningful difference, as ESS and autocorrelation plots show no sufficiently good behaviour.

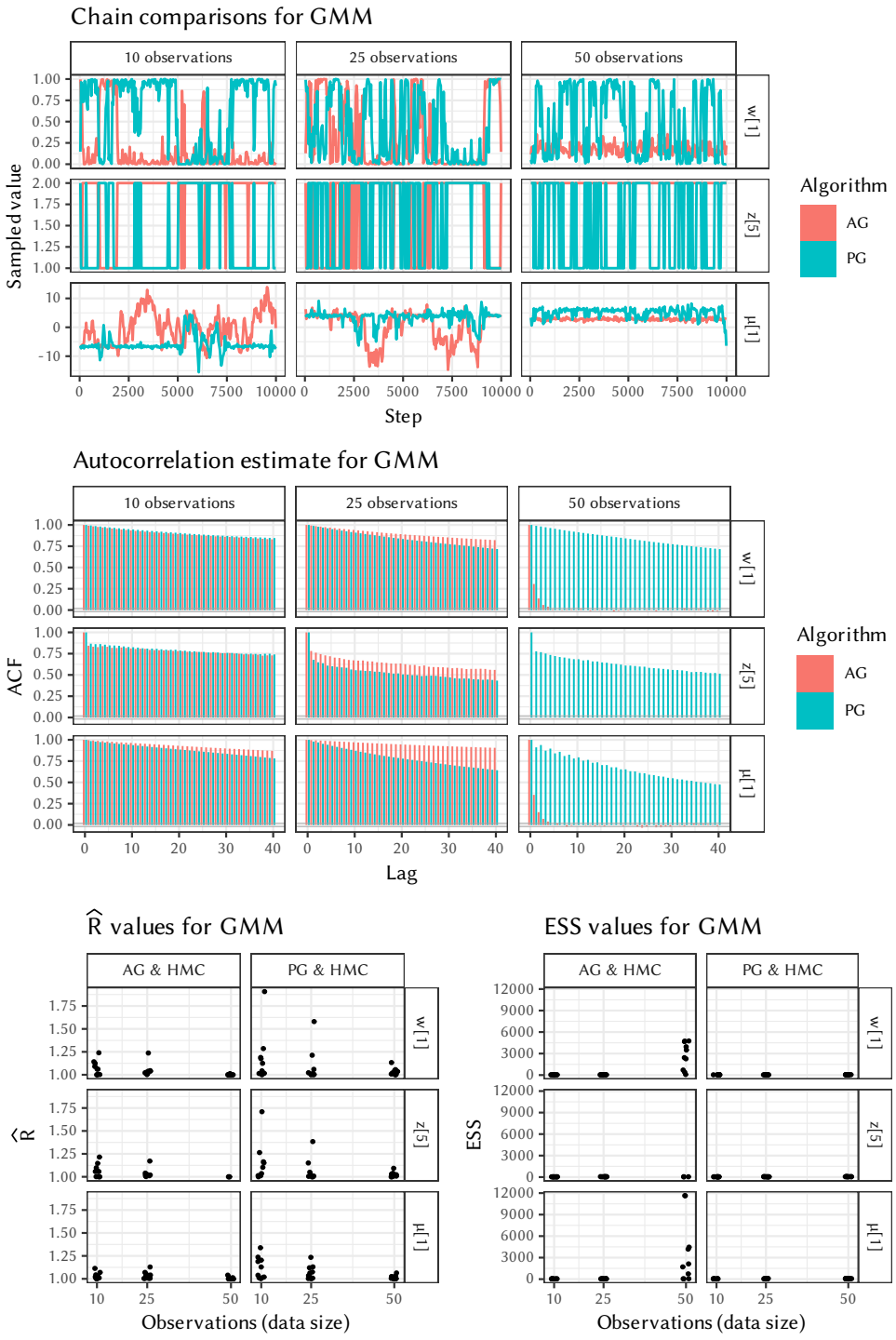


Figure 4.3: Diagnostics, factored by algorithm, number of observations, and a selection of model parameters. $\widehat{R}$ and ESS point estimates are jittered horizontally for better readability. For chain plots and autocorrelation, the first chain of the respective combination has been used.

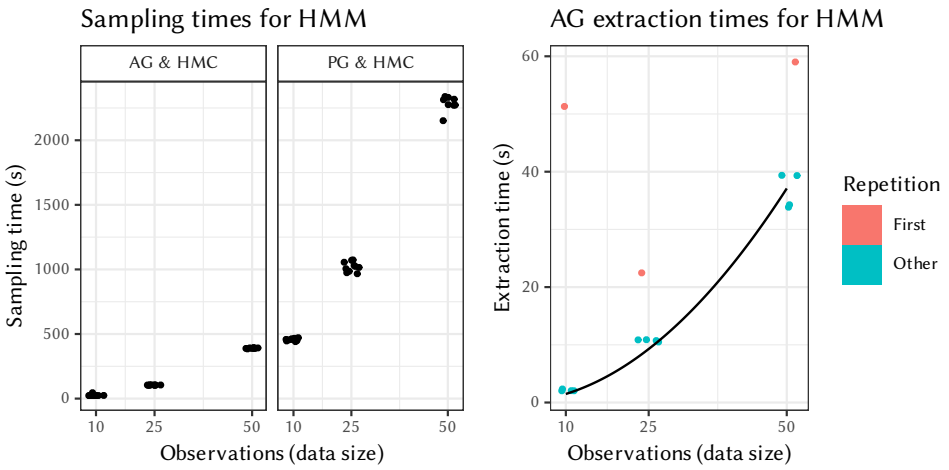


Figure 4.4: Sampling and extraction times for HMM, factored by algorithm and number of observations. Points are jittered horizontally to increase readability. A quadratic curve is fitted to the extraction times.

Hidden Markov Model

Also for HMM, the same trends in sampling and extraction times as with GMM are visible, with AG being consistenly faster. The extraction times seem to be quite the same as GMM, even in absolute terms, as are the outliers of the first function calls.

Mixing behaviour for this model is much better overall. The chains look less like random walks, especially for μ. Autocorrelation plots are sometimes quite good, especially for s, and in all cases better as those above for GMM. The $\widehat{R}$ values are all in better ranges (note the difference in the scale of the ordinate!), and ESS noticeably higher. Overall, AG seems to improve over PG on average, although neither of the results is stellar.

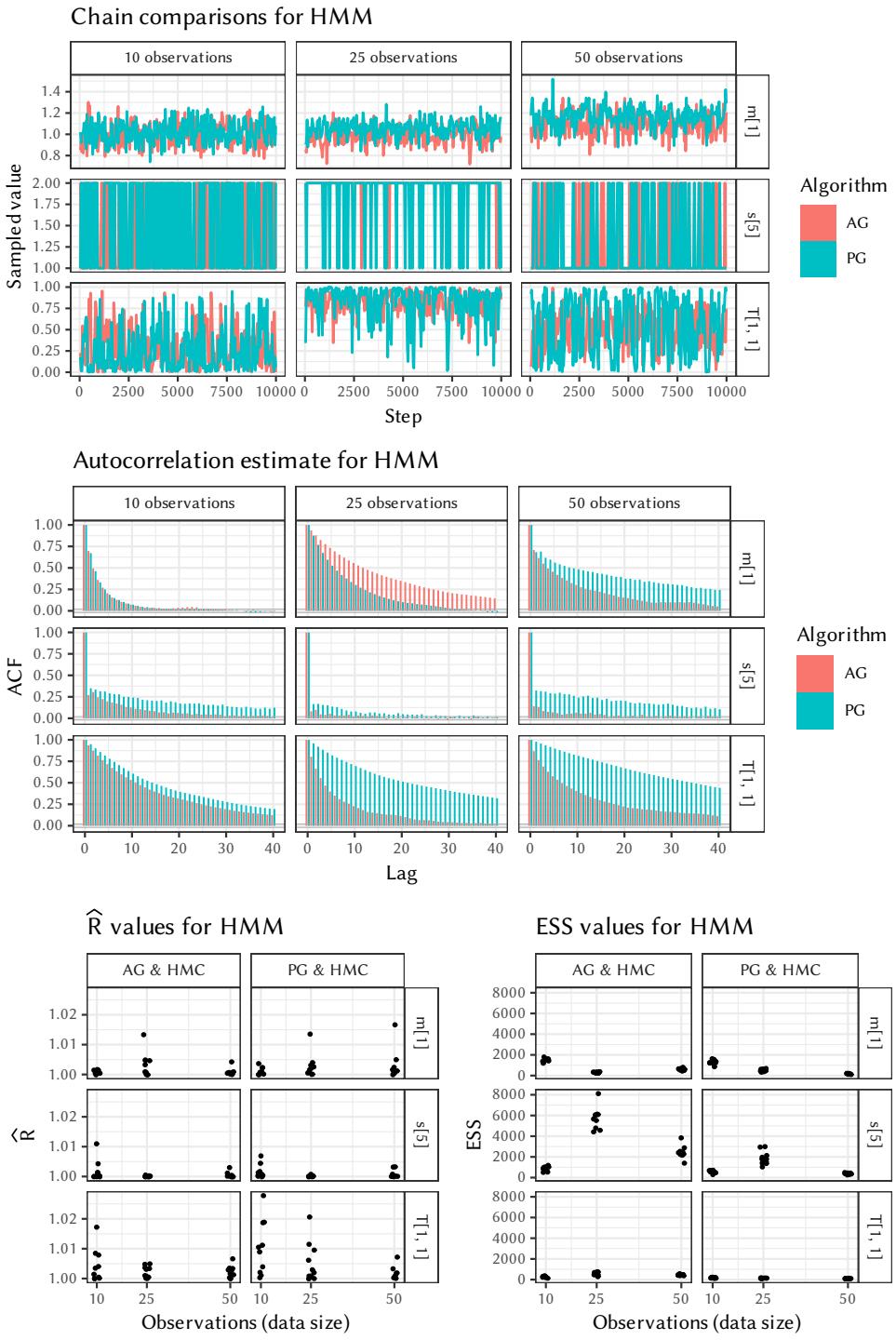


Figure 4.4: Diagnostics, factored by algorithm, number of observations, and a selection of model parameters. $\widehat{R}$ and ESS point estimates are jittered horizontally for better readability. For chain plots and autocorrelation, the first chain of the respective combination has been used.

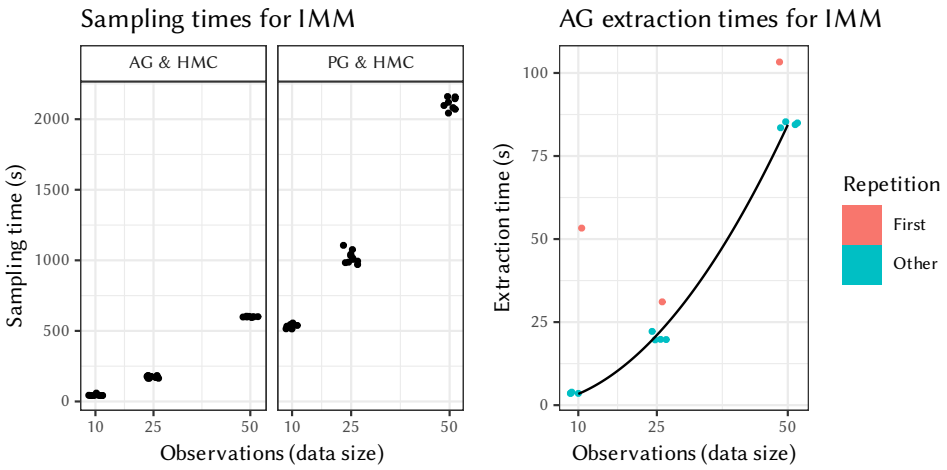


Figure 4.5: Sampling and extraction times for IMM, factored by algorithm and number of observations. Points are jittered horizontally to increase readability. A quadratic curve is fitted to the extraction times.

Infinite Mixture Model

Again, similar trends of sampling times and extraction times are noticeable. Here we can observe some larger involved factors, though; both curves grow faster, with PG on 10 observations even being faster than AG on 50 observations; although still on a significantly higher scale in general.

In this example, PG appears to work better on average. In the example chain plots, we can only see a noticeable difference for μ, while the autocorrelation graphs are almost all worse for AG (although both algorithms seem to do better than in the GMM test). ESS is only satisfactory for the z parameters, but PG here shows a much more consistent behaviour of the $\widehat{R}$ distribution.

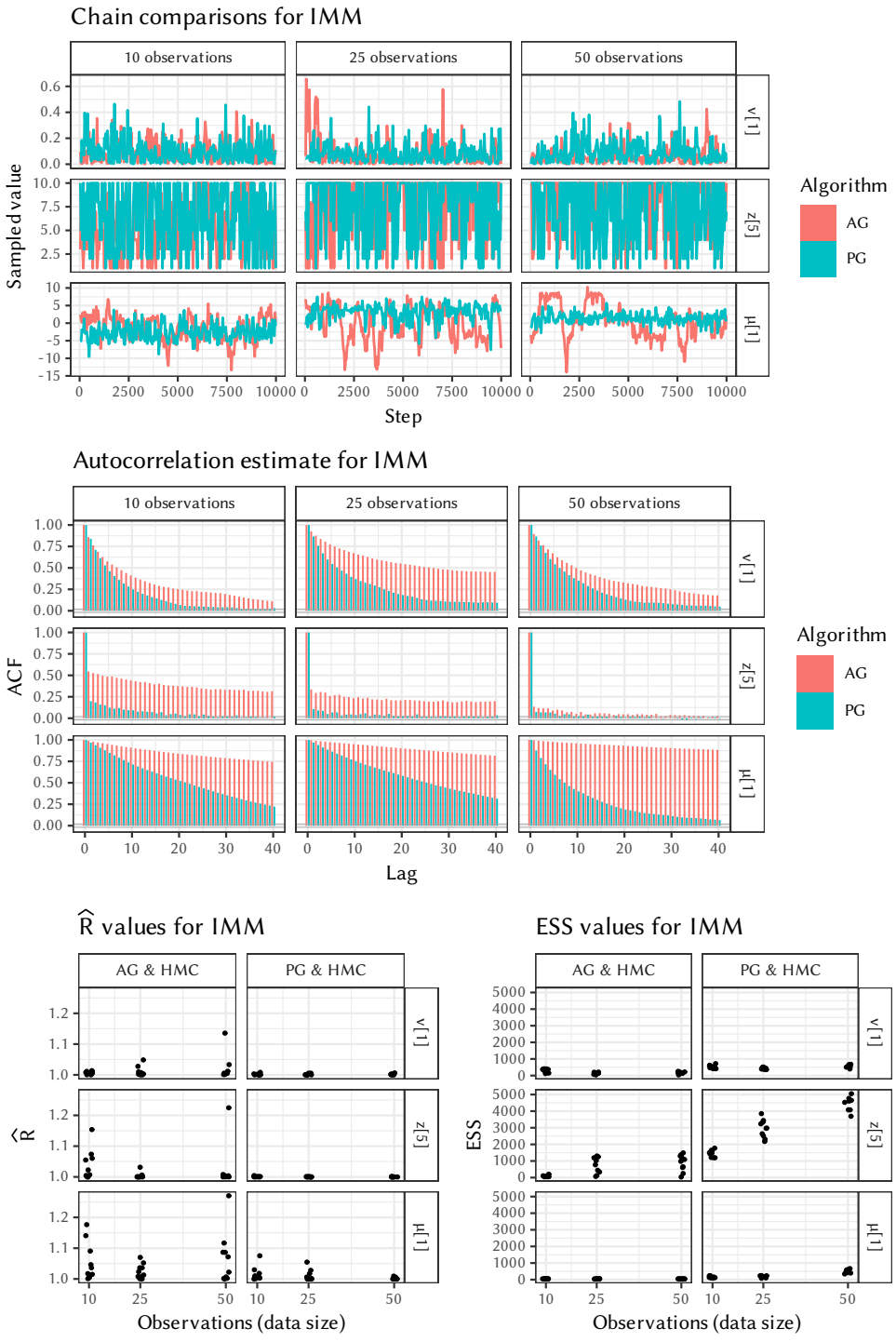


Figure 4.5: Diagnostics, factored by algorithm, number of observations, and a selection of model parameters. $\widehat{R}$ and ESS point estimates are jittered horizontally for better readability. For chain plots and autocorrelation, the first chain of the respective combination has been used.

Summary

Whereas the sampling times of Particle Gibbs always grows rather fast, depending on the number of observations, the rate of growth seems to be much lower for AutoGibbs. The behaviour of these curves appears to be superlinear, perhaps quadratic. For GMM and HMM, the maximal sampling time of AG is always below the minimal sampling time of PG. Even in the case of IMM, AG's sampling time with 50 observations is closest to PG's with only 10 particles, with the latter still obviously rising much faster.

With regard to the extraction times, we can note a pretty clear quadratic run time depending on the number of observations. The first run is always significantly above this trend, due to the impact of compilation and type inference. Additionally, the first invocation for the lowest number of observations might have involved additional compilation of library functions, explaining the larger residual compared to the first runs of the larger numbers.

In terms of convergence, AG and PG deliver quite comparable results, varying with some variation in quality depending on model and number of observations. In most cases, judging by eye through the exemplary autocorrelation plots, one or the other seems to slightly beat the other, which is buttressed by the distribution of the diagnostic values. IMM seems poses a particularly bad application for AG, but otherwise, no consistent "winner" is visible, and variations do not seem to follow a consistent pattern.

In conclusion, it can be said that for models where both are applicable, AutoGibbs provides a viable alternative to PG, delivering comparable results in less time. Care has to be taken to diagnose mixing behaviour, though, as always in MCMC simulations.

5 Discussion

The history of this project forms a large arc, starting from a general problem in `Turing.jl`, over a digression into compiler technology and automatic differentiation, back 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`. On its own, the latter is indeed an improvement over the previous situation: Gibbs conditional samplers can be significantly faster than particle-based samplers, the go-to instrument for discrete variables in `Turing.jl` so far, while delivering comparable inference results. Already the addition of a "manual" Gibbs conditional sampler in `Turing.jl` allows to directly implement many models from the literature, for which conditionals are often provided analytically. Automatic derivation allows to generalize this to a large class of models that have been found useful in other systems such as JAGS. However, the underlying issue – that `Turing.jl` lacks a structural representation of models – is not resolved by the implementation. This makes `AutoGibbs.jl` not completely satisfactory, since the recursion and branch tracking features of `IRTracker.jl` cannot be applied in a useful way.

The real difficulty is that dynamic models cannot be satisfactorily handled through snapshot-like slices in the form of traces. Systems trying to achieve this either become restrictive in their expressibility, or very complex in some aspects, up to practical limitation (see Mansinghka, Selsam & Perov (2014) and Goodman, Mansinghka, et al. (2012)). Furthermore, during implementation of the work, the two main practical difficulties were matching of variable names, e.g., subsuming `x[1:10]` under `x[1:3][2]`, and the correct handling of mutations that shadow actual data dependencies. The latter occurs in cases where one has, for example, an array `x`, samples a value `x[1]`, writes that to `x` with `setindex!`, and then uses `getindex(x, 1)` somewhere downstream. A more versatile dictionary structure for variable name keys could improve the situation for variable names, but wouldn't solve all of the underlying issues.

There is also a fragility problem: Julia IR, while being publicly documented, is a rather internal feature of the language, and may change between compiler versions. The `IRTools.jl` package provides a good mid-layer mitigating this, but still there's lot of reasons why a custom IR would be nicer. From the other side, the internal structure of `DynamicPPL.jl`'s model representations might change, and is an implementation detail that should not be relied on from the outside – especially

not by an important feature such as dependency extraction. In a certain sense, the whole approach is misguided: why rely on external tracking for a framework that is really under ones own control, using such heavy machinery as IR transformations? This is illuminated by following very telling comment about similar tendencies (`Cassette.jl` is a package very similar to `IRTools.jl`):

> Using Cassette on code you wrote is a bit like shooting yourself 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]

In conclusion, even though it has enabled the implementation of `AutoGibbs.jl`, the dynamic graph tracking system of `IRTracker.jl` does not solve the underlying problem of analysis of dynamic probabilistic models. In the course of development, various techniques have been tried or ruled out, challenges identified, and other alternatives explored. This knowledge has lead me to a better understanding of the domain and some more advanced ideas for the future, some of which are laid out in the following section.

5.1 Future Work

While `IRTracker.jl` is quite a satisfying and complete system, the approach that `AutoGibbs.jl` takes provides only an ad-hoc solution to a major shortcoming of `Turing.jl`: the lack a structural model representation that is open to analysis and transformations. This has made me consider alternatives, approaching the representation problem for probabilistic programming languages on a more fundamental level.[2].

Let us review the important features of a universal, flexible PPL as mentioned in section 2.2. Its DSL should allow general recursion and nesting, support for all language constructs and custom types and extensions, and be able to delegate to other samplers or complex programs. In addition, the internal representation should be such that multiple forms of analysis, optimization, non-standard execution, and transformation can be performed. Currently, `Turing.jl` is following a rather simple approach: one data structure (`VarInfo`) contains a map from variable names to values, the accumulated log-likelihood, and some other sampling metadata. `AutoGibbs.jl`' solution consists of retrofitting some more structure onto this representation – which is not ideal, and for proper analysis, it would be desirable to begin with a better representation from the start.

From difficulties described above, which became apparent during the implementation of the Gibbs conditional extraction, together with the knowledge about

[1] Lyndon White (2020), private communication on `https://julialang.slack.com`.

[2] The following ideas are based on a previous informal collection at `https://github.com/phipsgabler/probability-ir`

`DynamicPPL.jl`'s internals, I developed an understanding of what a more advanced representation of probabilistic models, with a focus on transformation and analysis, could be, from a metaprogramming and static analysis, and language design perspective. The idealized goal would be for variable names and dependency graphs in general probabilistic programs to behave more conveniently as abstract data structures, and to be part of a closed, elegant, high-level language. Many successful approaches to PPL design probably come from the perspective of efficient and general inference algorithm, putting the language design problem second to such a desire – but it should be possible to approach the field from a more "linguistic" perspective as well. A further goal would be to close the gap between practical inference systems and the mostly theoretical, functional-programming-based approaches of just formalizing probabilistic programs (such as probabilistic lambda calculi, or type-theoretic formulations; see Bhat et al. (2012), Heunen et al. (2017), Ramsey & Pfeffer (2002), and Ścibior, Ghahramani & Gordon (2015))

Universal PPLs have as their goal to let the user write down every model the language allows, and still be able to do inference on it. Of course, at the boundary of the space of "reasonable" programs, trade-offs need to be made to still be able to do this. It seems advantageous to split up this conjunction: by creating a format in which one can denote every possible model of a very large class, without a priori having to deal with the restrictions of inference. Then for each model, suitable transformations and analyses can be performed in a uniform representation, and specialized backends be chosen from a wide range, which understand precisely the fragment of the model language used.

What I propose is a "probabilistic intermediate representation", that turns around how things are currently construed in most of the approaches. Instead of starting from a model as a "sampling function", which is evaluated to extract graphs or other symbolic representations from it, one would begin from a representation that already is general, yet richly structured, and derive inference programs from it. Viewed from the opposite direction, in contrast to PPLs that are built on top of a DSL representation, such an representation should be backend-agnostic, and instead allow all kinds of models to be specified in a uniform syntax, without being constraint by the demands of a specific sampling algorithm or inference technique. Furthermore, it shouldn't matter whether the model is complicated, nonparametric, dynamic – the object that is worked with is always a fixed, full program in a specified syntax, with an intuitive denotation.

This separation between the a "specification abstraction" in form of a general representation and "evaluator abstractions" provided by interfaces to multiple sampler implementations seems novel. The closest correspondence would be the formalization attempts of probabilistic models through monads and type systems; but that is more semantic than syntactic. There exist some domain-specific "linguae frankae" like the syntax of Stan and JAGS, but they are, too, somewhat restricted, and not independently defined and maintained – the systems coming later just chose to take over the same kind of input format for their own implementation. `Gen.jl`

(Cusumano-Towner 2020) provides an extensibel interface for the class of models it supports, but this is still quite tightly bound to its inference system. All these approaches could rather be abstracted out into a model specification formalism in its own right, that has more general analysis capabilities, and can then be transformed abstractly, ultimately producing the form some concrete evaluator (i.e., sampling algorithm or PPL system) requires.

The advantage of such aseparation, besides making available solutions and techniques from programming language theory and compiler constrution, is that it provides a different kind of common abstraction for PPLs than is possible through a "one DSL per system" approach. Recently, developers in Julia 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 is necessary due to the lack of division of each system into an evaluator and a model specification part: they always go together. (`DynamicPPL.jl` is itself supposed to define an extensible model description language, but in practice is still quite strongly integrated with `Turing.jl`.)

I believe that starting from a common model specification language is in many cases preferable, and more general than just a common interface for evaluators. Such interfaces tend to assume much more about the internals, while the capabilities of universal probabilistic programs are essentially fixed: the notation of random variables used in model specification by hand, extended through the forms of an embedding programming language. Starting from this, we could consider the following a least upper bound of all the PPL modeling approaches:

- General code: covered by normal Julia IR with SSA statements, branches, and blocks.
- "Sampling statements": special assignment forms for tildes, or assumptions and observations in `Turing.jl` parlance, which relate names or values to distributions (or generally sub-models) in an declarative way.
- First-class variable names: these may be quite complex, containing for example indexing, fields, link functions, etc., which can be identified and analyzed in a structured way.

Given this, it seems feasible to define arbitrary probabilistic programs an IR-like syntax, similar to an extended SSA-form; the crucial point being that names and tildes are not separated from a host language. The idea amounts to writing out a directed graphical model with deterministic and stochastic nodes, named random variables, but generalized to programs – e.g., allowing dynamic structure with to branching and recursion. A model in this kind of format then defines an abstract and uninterpreted parametrized joint density function over its trace space (as given through the unified name set of all possible runs, see e.g. Lew et al. (2020)), factorized into primitive statements and blocks.

There is still much to be clarified and researched about the syntax and semantics of such a representation, but the underlying principle should be intuitively clear

by just matching the existing Julia IR to probabilistic semantics of models like in `Turing.jl`, `Soss.jl`, or `Gen.jl`. Consider, for example, a hierarchical Gaussian model, informally written as

```
n = 1
while n <= N
    {x[n]} ~ Normal({mu[z[n]]})
    n += 1
end
{y} ~ MvNormal({x}, {sigma})
```

We can imagine this to be represented directly in probabilistic IR by conceiving of a lowering mechanism that treats tilde statements just like assignments, and preserves variable names:

```
1:
  goto 2 (1)
2 (%1):
  %2 = {z[%1]}
  %3 = {mu[%2]}
  {x[%1]} ~ Normal(%3)
  %4 = %1 < N
  br 4 unless %4
  br 3
3:
  %5 = %1 + 1
 br 2 (%5)
4:
  {y} ~ MvNormal({x}, {sigma})
```

If we define the tilde statements to behave like stochastic function calls, with a side effect of somehow storing the intermediate stochastic values and their names as metadata, this is exactly how the evaluation semantics of `DynamicPPL.jl` in most cases work.

In contrast to `AutoGibbs.jl`'s data structures, this kind of model is not a slice, but preserves the complete information about a model specification though a first-class representations. The probabilistic part of it, as opposed to the code generated by `DynamicPPL.jl`, is referentially transparent. These properties make analysis and code transformations, similar to the ones possible with `IRTools.jl`, significantly easier and more general. On the formal representation we can then apply transformations such as specialization on a constant parameter or observation, resulting in a new model in IR form, exploitation of probabilistic knowledge, like collapsing or conjugacy exploitation, or "disaggretating" a non-parametric model into something sampleable (e.g., re-representing a Dirichlet process model with a CRP-based one), or other changes of the probabilistic structure. We can also apply static analysis or abstract interpretation techniques: e.g., extraction of Gibbs conditionals. Finally, the model can be converted into a form fit for evalation – the transformation of the IR into executable code, or data structures for other PPL systems.

All these usages can be represented through composition of several small structural functions: constructing a new model that is the Gibbs conditional for some

varible; turning a model into a plain Julia generative function; extracting the log-likelihood function of the model; specializing a model with some variable to given observations, and checking that the result is static; perform a causal intervention on some random variable; of converting a model with fixed data into a factor graph representation.

A similar principle of is currently developed with JAX (Bradbury et al. 2018), which is inteded for numeric functions, and in which there exists a unified representation of functional programs that undergo various transformations. On top of JAX, Oryx[3] should provide with the necessary infrastructure to apply this in the setting of probabilistic programs. JAX, though, is closer to lambda calculus in A-normal form than SSA-form IR; it assumes referential transparency and has no representation of control structures. In Julia, Soss (Scherrer 2019) takes a somewhat comparable approach by representing models written in a Julia DSL in completely symbolic expression form, from which inference code is generated. Also here, not the full generality of the host language is available, but only a pure subset of it; and again, control structure can only be realized through combinator functions, not at language level.

The approach most comparable to the presented ideas, although stemming from a complete different domain, would be tactic or elaborator systems in proof assistants (e.g., Brady 2013; Coq Development Team 2020). There, user-written programs are iteratively refined into other, more specialized forms through functions expressed in a metalanguage (the so-called tactics), interleaving automated transformations and manual interventions. A similar style of development could boost the usability and flexibility of Bayesian inference: after writing a down model syntactically, the user can interactively refine the model code in symbolic form, applying their knowledge and constraints, until they arrive at a form that can be passed to some inference mechanism.

[3] `https://www.tensorflow.org/probability/oryx`

Appendices

A Measure Theory in Probability Theory

Measure theory (Tao 2011; Bronstein & Semendjajew 1995, section 10.5) allows to treat both discrete and continuous probabilities under a common, generalized notation, primarily by the introduction of integrals over measures. The basic idea of a measure is to generalize the concept of a "volume function" on sets.

A *measure space* is a triple $(\Omega, \mathcal{A}, \mu)$, where $\mathcal{A} \subseteq 2^\Omega$ is a σ-algebra of *measurable sets* (i.e., closed under complement, countable union, and countable intersection), and $\mu : \mathcal{A} \to \mathbb{R} \cup \{\infty\}$ is a σ-additive function:

$$\mu\left(\bigcup_k A_k\right) = \sum_k \mu(A_k) \tag{A.1}$$

for all disjoint countable families (A_k) in $\mathcal{A}$. Additionally, we require that $\mu(\emptyset) = 0$. The necessity for $\mathcal{A}$ and μ being defined in such an elaborate way, instead of just taking it as 2^Ω, is that for uncountable Ω, it is not possible to consistently assign a measure for the complete powerset. The restriction to measurable subsets specifically filters out those pathological cases.

In probability theory (Kallenberg 2006), one always operates within a special measure space called *probability space*. In a probability space $(\Omega, \mathcal{A}, P)$, we additionally require that $P(\Omega) = 1$. Ω is then called the set of *events* – think of all possible outcomes of some experiment.

A function between measure spaces, or probability spaces in particular, is called *measurable* when every preimage of a measureable set is measurable. A *random variable* X is a measureable function $(\Omega, \mathcal{A}, P) \to (\Psi, \mathcal{B}, P_X)$ from a probability space to another one with a *pushforward* P_X, such that

$$P_X(B) = P(X^{-1}(B)) \tag{A.2}$$

for all $B \in \mathcal{B}$. The introduction of random variables allows to consistently convert set-theoretic operations on events into "numerical" ones: think of assigning to each outcome of a coin throw a number in $\{1, 2\}$, or to each measurement of some height a value in $\mathbb{R}^+$. In practice, this allows us to forget about the underlying event space and think solely in terms of the values in the domain of the random variable, with

notation like

$$\mathbb{P}[\alpha(X)] = P(\{\omega \in \Omega \mid \alpha(X(\omega))\}) \tag{A.3}$$

where α is an arbitrary predicate defining a set in the domain of X.

In such a setting, it might the case that there exist some *base measure* μ, such that probability evaluation can be expressed as integral over some *density* with respect to μ:

$$\mathbb{P}[X \in A] = \int_A p_X(x)\,\mathrm{d}\mu(x), \tag{A.4}$$

for all P_X-measureable sets A, or in differential notation

$$\mathbb{P}[X \in \mathrm{d}x] = p_X(x)\,\mathrm{d}\mu(x). \tag{A.5}$$

In this case P_X is said to be *absolutely continuous* with respect to μ, written $P_X \ll \mu$. This statement is equivalent to the existence of a *Radon-Nikodym derivative*

$$\frac{\mathrm{d}P_X}{\mathrm{d}\mu} = p_X. \tag{A.6}$$

For discrete values, a density always exists with respect to the counting measure, and the random variable is also called discrete. For finite-dimensional continuous values, when a density exists with respect the Lebesgue measure, we speak of a continuous random variable.

B Details of Automatic Differentiation

The standard reference for AD is Griewank & Walther (2008). Baydin et al. (2018) gives a comprehensive survey including a comparison of state-of-the-art implementations. There are many works on the formalization of AD; see, for example, Abadi & Plotkin (2020), Vytiniotis et al. (2019), Wang et al. (2019), Sajovic & Vuk (2016), or Elliott (2018).

To understand how AD works, let us first start with the mathematics. What is a derivative, really? When we talk about gradients, which is what we really need in a gradient algorithm, this is usually a rather informal term for "the vector of partial derivatives", which then points into an ascent direction. This is however not the most natural form to work with in a compositional approach. Instead of starting with a limit of tangent slopes, more insight is provided by viewing derivatives as best-approximating linear operators. One of the most general definitions is provided through the *Fréchet derivative* (Bronstein & Semendjajew 1995, p. 463), essentially a generalization of the total differential[1]. Let X and Y be normed spaces. A function $f : U \subseteq X \to Y$ is Fréchet differentiable at a point $x \in U$ if there exists a bounded linear operator $A : X \to Y$ such that

$$\lim_{\|\Delta\|_X \to 0} \frac{\|f(x+\Delta) - f(x) - A(\Delta)\|_Y}{\|\Delta\|_X} = 0. \tag{B.1}$$

When such an A does exist, it is unique, and we may call it *the* derivative of f at x, writing $\mathrm{D}f(x) = A$. When the derivative exists for all x, we can use D as a well-defined higher-order function on its own; we will assume this in the following.[2]

The important fact here is that $\mathrm{D}f(x)$ is still a function: specifically, a linear function approximating how f reacts to an input perturbation, Δ, around x. Or, in

[1] I prefer thinking in terms of the Fréchet derivative, since it makes explicit the fact that derivatives are operators, and provides enough flexibility while still being intuitive. Different abstractions are possible, though.

[2] In in practical cases, functions are often only piecewise differentiable due to branches, failing this definition on a countable set of points. Fortunately, the formalism of AD remains the same under weaker notions of differentiability. Additionally, such points usually behave well enough to admit a subdifferential, from which we can just choose an arbitrary subgradient; this does not necessary lead to a descent direction, but still allows minimization under reasonable conditions (see Pock 2017, section 6.1; Griewank & Walther 2008, chapter 14; Abadi & Plotkin 2020).

other words:

$$f(x + \Delta) = f(x) + \mathrm{D}f(x)(\Delta) + o\left(\|\Delta\|\right). \tag{B.2}$$

This fact allows one to propagate differential values through composed functions, by the chain rule, which we write in the following compositional form:

$$\mathrm{D}(\phi \circ \psi)(x) = \mathrm{D}\phi(\psi(x)) \circ \mathrm{D}\psi(x). \tag{B.3}$$

for differentiable functions ϕ and ψ. In the one-dimensional case, we simply have

$$\mathrm{D}\phi(x) = \Delta \mapsto \partial_1\phi(x)\,\Delta, \tag{B.4}$$

where $\partial_1\phi(x)$ denotes the standard "primitive" derivative, since linear maps are exactly multiplications by a scalar. Therefore, we can recover the chain rule

$$\begin{aligned} \mathrm{D}(\phi \circ \psi)(x)(\Delta) &= \left(\mathrm{D}\phi(\psi(x)) \circ \mathrm{D}\psi(x)\right)(\Delta) \\ &= \left(\partial_1\phi(\psi(x))\,\partial_1\psi(x)\right)\,\Delta, \end{aligned} \tag{B.5}$$

as we know it from calculus. Here, the product in the resulting expression arises from the fact that we propagated through $\partial_1\psi(x)\Delta$ as the input value of $\mathrm{D}\phi(\psi(x))$. It is, however, remarkable that this formula is not entirely compositional: to construct $\mathrm{D}(\phi \circ \psi)$, it is not only necessary to know $\mathrm{D}\phi$ and $\mathrm{D}\psi$, but also ψ (Elliott 2018). Still, this is not as bad as it may seem: as I will now explain, AD algorithms evaluate both $(\phi \circ \psi)(x)$ and $\mathrm{D}(\phi \circ \psi)(x)$ at once, in lockstep fashion, so that the intermediate values of the former can be reused in calculation of the latter.

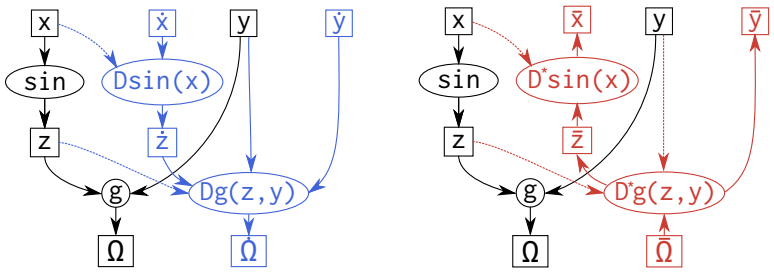
Consider the specific case of $f(x, y) = sin(x) - y$. For simpler notation, let $g = (x, y) \mapsto x - y$ replace the infix subtraction operator, with a derivative of $\mathrm{D}g(x)(\Delta_1, \Delta_2) = \Delta_1 - \Delta_2$, a linear function of two arguments. By composition, we have:

$$\begin{aligned} \mathrm{D}f(x, y) &= \mathrm{D}(g \circ (\sin \otimes \mathrm{id}))(x, y) \\ &= \mathrm{D}g\left((\sin \otimes \mathrm{id})(x, y)\right) \circ \mathrm{D}(\sin \otimes \mathrm{id})(x, y) \\ &= (\Delta_1, \Delta_2) \mapsto \cos(x)\,\Delta_1 - \Delta_2 \end{aligned} \tag{B.6}$$

(where id is the identity function, and $(\phi \otimes \psi)(\alpha, \beta) = (\phi(\alpha), \psi(\beta))$ defines product morphisms). In order to calculate this algorithmically, let us expand the computation of f into a sequence of intermediate, primitive calculations, as we would have in a programmatical representation:

$$\begin{aligned} x &= ?, \\ y &= ?, \\ z &= \sin(x), \\ \Omega &= g(z, y). \end{aligned} \tag{B.7}$$

We have given the final result the name Ω, and and introduced an intermediate value z. This is known as the *forward*, or *primal* function in AD terminology. The relations of these values can be expressed as the black computation graph in



(a) Forward mode. (b) Backward mode.

Figure B.1: Computation graph and intermediate expressions of the expression `g(sin(x), y)`, together with the derivative graphs in forward- and backward mode. Dashed arrows indicate re-use of primal values in the derivative graph.

figure B.1(a). Following the graph, or equivalently, following the equations in (B.7), the composition of the derivative operators can be built up incrementally, as shown in the blue part of that figure, by calculating the following *tangent values*:

$$\begin{aligned} \dot{x} &= \Delta_1, \\ \dot{y} &= \Delta_2, \\ \dot{z} &= \mathrm{D}\sin(x)(\dot{x}) \\ &= \cos(x)\,\Delta_1, \\ \dot{\Omega} &= \mathrm{D}g(z,y)(\dot{z},\dot{y}) \\ &= \cos(x)\,\Delta_1 - \Delta_2. \end{aligned} \tag{B.8}$$

The tangent values of input variables x and y become the input perturbations Δ_1, Δ_2. For every subsequent tangent value, we apply the derivative at the corresponding primal variable (depending on the primal parents) to the tangent values of the parents – this way, the composition of the derivative operators follows the chain rule. This algorithm, called *forward-mode AD*, can now be applied practically not only on symbolic functions, but on programs, by always jointly computing $(v, \dot{v})$ for every variable v, given its parents in the graph. This requires a form of non-standard execution.

Recovering the full gradient of a multivariate function $\phi : U \subseteq \mathbb{R}^N \to \mathbb{R}$ (which is generally the form of loss functions for parametric models) requires to evaluate $\mathrm{D}\phi(x)$ N times, however. This is because individual partial derivatives can only be extracted from $\mathrm{D}\phi(x)$ by calculating the sensitivities to unit input perturbations in coordinate directions, for each of the input variables:

$$\nabla\phi(x) = \begin{pmatrix} \mathrm{D}\phi(x)(1,0,\ldots,0) \\ \vdots \\ \mathrm{D}\phi(x)(0,\ldots,0,1) \end{pmatrix} = \begin{pmatrix} \partial_1\phi(x) \\ \vdots \\ \partial_N\phi(x) \end{pmatrix}, \tag{B.9}$$

which is really a special case of taking directional derivatives (which can be recovered generally by application of the differential to any vector with unit norm.)

In order to overcome the increase of complexity with the number of input dimensions, we can reformulate the compositional equation. Let us introduce $\mathrm{D}^*\phi(x)$, the *adjoint operator* of $\mathrm{D}\phi(x)$, whose defining property is that "inverts" the order of the perturbation application: instead of calculating a primal sensitivity with respect to an input perturbation (Δ), it maps a linear output perturbation ($\mathfrak{d}$) to an operator that applies this to the primal sensitivity:

$$\mathrm{D}^*\phi(x)(\mathfrak{d}) = \Delta \mapsto \mathfrak{d}(\mathrm{D}\phi(x)(\Delta)). \tag{B.10}$$

The adjoint differential is therefore an object of the double dual space. This becomes more readable when we fix a basis to represent the derivative. Doing so, in the finite-dimensional case, the derivative $\mathrm{D}\phi(x)$ is the Jacobian matrix at x, $J_\phi(x)$. In this setting, forward-mode AD is simply an efficient way to calculate the *Jacobian-vector product* $J_\phi(x)\Delta$, or equivalently the total derivative for a fixed perturbation, avoiding full matrix multiplication – which is the reason we have to apply it to the basis vectors to get back the gradient. Backward mode, on the other hand, calculates the product of the Jacobian with the operator that should be applied to the result, but does not yet apply it to the input perturbation – therefore, it returns a matrix:

$$\begin{aligned}\mathfrak{d}(\mathrm{D}\phi(x)(\Delta)) &= d^\mathrm{T} J_\phi(x)\Delta \\ &= \left(J_\phi(x)^\mathrm{T} d\right)^\mathrm{T} \Delta \\ &= \mathrm{D}^*\phi(x)(\mathfrak{d})(\Delta),\end{aligned} \tag{B.11}$$

where we assume $\mathfrak{d}$ to be represented by the co-vector d^T. Since the unapplied $\mathrm{D}^*\phi(x)(\mathfrak{d})$ is itself an object in the dual space, it is also represented as a co-vector – and in fact, nothing else than a transformation of the transposed Jacobian, or a *vector-Jacobian product*. Recovering the gradient of a loss function then reduces to evaluating it at a constant scalar output perturbation of 1, which is equivalent to the application of the primal differential to the matrix of basis vectors.

Note that due to this relation to the transpose, the adjoint operator inverses the order of composition in the chain rule:

$$\begin{aligned}\mathrm{D}^*(\phi \circ \psi)(x)(\mathfrak{d}) &= d^\mathrm{T} J_\phi(\psi(x)) J_\psi(x) \\ &= \left(J_\psi(x)^\mathrm{T} J_\phi(\psi(x))^\mathrm{T} d\right)^\mathrm{T} \\ &= \left(\mathrm{D}^*\psi(x) \circ \mathrm{D}^*\phi(\psi(x))\right)(\mathfrak{d}).\end{aligned} \tag{B.12}$$

For our example function f, this gives the same structural form of the result as the forward mode – only that now, the value is a vector:

$$\begin{aligned}\mathrm{D}^* f(x, y) &= \mathrm{D}^*(g \circ (\sin \otimes \mathrm{id}))(x, y) \\ &= \mathrm{D}^*(\sin \otimes \mathrm{id}) \circ \mathrm{D}^* g((\sin \otimes \mathrm{id})(x, y)) \\ &= \delta \mapsto [\cos(x)\delta, -\delta]^\mathrm{T}.\end{aligned} \tag{B.13}$$

In this form, starting with an output perturbation $\delta = 1$, we get back the gradient tuple through just one evaluation. Incidentally, this is nothing else than the back-propagation "trick" (Bishop 2006)! Furthermore, applying this result to $[\Delta_1, \Delta_2]$ gives back the linear combination of the forward mode result.

In programmatic terms, we can proceed similar to above, only this time introducing *adjoint* intermediate values $\bar{v}$. For the values in equation (B.7), we get

$$\begin{aligned} \bar{x} &= \bar{z}_2 = -\delta, \\ \bar{y} &= \mathrm{D}^* \sin(x)\, \bar{z}_1 \\ &= \cos(x)\, \delta \\ \bar{z} &= \mathrm{D}^* g(x, y)(\bar{\Omega}) \\ &= [\delta, -\delta] \\ \bar{\Omega} &= \delta, \end{aligned} \tag{B.14}$$

which is displayed in the red graph in B.1(b). Note that now, the back-propagated values can not be computed in parallel with forward evaluation; hence the equations are stated in reverse order. Instead, the intermediate primal values have to be remembered and reused in a second, backward pass.

Finally, it has to be noted that the two described modes of automatic differentiation are only two extremes of a spectrum. Forward and backward calculations can really be interleaved in arbitrary order, just as it is possible to multiply Jacobians and their transposes in different order. One frequent use case of this *mixed-mode AD* is when loss functions, differentiated using backward mode, contain broadcasting functions; for example, nonlinearities within neural networks. These have a type of $\mathbb{R}^N \to \mathbb{R}^N$, but only involve a linear number of operations, so forward mode pays off[3]. Similar properties hold for second order derivatives: the calculation of Hessians is often fastest by using forward-over-reverse composed differentiation. In general, unfortunately, determining the optimal order of derivative evaluation is hard – this so-called *optimal Jacobian accumulation* problem is known to be NP-complete (Naumann 2007).

B.1 Dual Numbers

Forward mode can be recast in mathematically equivalent form by using dual numbers (see Baydin et al. 2018, section 3.1.1)deakin1966functions. These consist of two parts, similar to complex numbers: $z = x + y\epsilon$. However, contrary to the imaginary unit, the infinitesimal unit ϵ vanishes under multiplication with itself: $\epsilon^2 = 0$. The consequence of this is that functions can naturally be extended to dual numbers by

[3] As a rule of thumb in Julia, for $f : \mathbb{R}^M \to \mathbb{R}^N$, forward mode typically performs better when $M \ll N$ or as long as $M \lessapprox 100$. This folklore should always be confirmed by benchmarking, though. See `https://github.com/JuliaDiff/ReverseDiff.jl#should-i-use-reversediff-or-forwarddiff`.

nonstandard interpretation as truncated Taylor series:

$$\phi(x+\epsilon) = \phi(x) + \partial_1\phi(x)\epsilon + \underbrace{\frac{\partial_1^2\phi(x)}{2}\epsilon^2 + \ldots}_{\epsilon^2(\ldots)=0} \tag{B.15}$$

Since the higher order terms vanish, this is exactly the tuple of primal and tangent value that is calculated during the lockstep evaluation in forward mode:

$$(z, \dot{z}) = (\phi(x), \mathrm{D}\phi(x)(\dot{x})) \quad \Leftrightarrow \quad z + \dot{z}\epsilon = \phi(x + \dot{x}\epsilon). \tag{B.16}$$

The generalization to higher dimensions, as well as higher derivatives in form of hyper-dual numbers (Fike & Alonso 2012), follow naturally.

Bibliography

Abadi, M., A. Agarwal, et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Preliminary White Paper.

Abadi, M. & G. D. Plotkin (2020). "A Simple Differentiable Programming Language". In: *Proceedings of the ACM on Programming Languages* 4 (POPL), pp. 1–28. DOI: 10.1145/3371106.

Aho, A., R. Sethi & J. Ullman (1986). *Compilers: Principles, Techniques and Tools*. Massachusetts: Addison-Wesley.

Amin, N. (2016). "Dependent Object Types". PhD Thesis. EPFL. DOI: 10.5075/epfl-thesis-7156.

Apple (2020). *Swift Compiler*. URL: https://swift.org/swift-compiler/ (visited on 2020-11-01).

Bartholomew-Biggs, M. et al. (2000). "Automatic Differentiation of Algorithms". In: *Journal of Computational and Applied Mathematics* 124.1, pp. 171–190. DOI: 10.1016/S0377-0427(00)00422-2.

Baydin, A. G. et al. (2018). "Automatic Differentiation in Machine Learning: A Survey". In: *Journal of Machine Learning Research* 18.153, pp. 1–43.

Becker, M. R. (2020). "Dynamic Specialization in Trace-Based Probabilistic Programming Systems". In: ProbProg 2020.

Betancourt, M. (2018). "A Conceptual Introduction to Hamiltonian Monte Carlo". In: arXiv: 1701.02434 [stat].

Bezanson, J., J. Chen, et al. (2018). "Julia: Dynamism and Performance Reconciled by Design". In: *Proc. ACM Program. Lang.* 2 (OOPSLA). DOI: 10.1145/3276490.

Bezanson, J., A. Edelman, et al. (2017). "Julia: A Fresh Approach to Numerical Computing". In: *SIAM Review* 59.1, pp. 65–98. DOI: 10.1137/141000671.

Bhat, S. et al. (2012). "A Type Theory for Probability Density Functions". In: *SIGPLAN Notices* 47.1, pp. 545–556. DOI: 10.1145/2103621.2103721.

Bianucci, A. M. et al. (2000). "Application of Cascade Correlation Networks for Structures to Chemistry". In: *Applied Intelligence* 12.1, pp. 117–147. DOI: 10.1023/A:1008368105614.

Bingham, E. et al. (2018). "Pyro: Deep Universal Probabilistic Programming". In: arXiv: 1810.09538 [cs, stat].

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Information Science and Statistics. New York: Springer. 738 pp.

Bolewski, J. (2015). “Staged Programming in Julia”. Presentation. JuliaCon 2015 (Boston).
Bolstad, W. M. (2004). *Introduction to Bayesian Statistics*. New York: Wiley.
Bradbury, J. et al. (2018). *JAX: Composable Transformations of Python+NumPy Programs*. Version 0.1.55.
Brady, E. (2013). “Idris, a General-Purpose Dependently Typed Programming Language: Design and Implementation”. In: *Journal of Functional Programming* 23.05, pp. 552–593. DOI: 10.1017/S095679681300018X.
Bronstein, I. N. & K. A. Semendjajew (1995). *Taschenbuch der Mathematik: Ergänzende Kapitel*. 7th ed. Leipzig: Teubner.
Carpenter, B., A. Gelman, et al. (2017). “Stan: A Probabilistic Programming Language”. In: *Journal of Statistical Software* 76.1 (1), pp. 1–32. DOI: 10.18637/jss.v076.i01.
Carpenter, B., M. D. Hoffman, et al. (2015). “The Stan Math Library: Reverse-Mode Automatic Differentiation in C++”. In: arXiv: 1509.07164 [cs].
Chewxy et al. (2020). *Gorgonia/Gorgonia: Bugfix Release: Vectors Were Not Properly Broadcasted*. Version 0.9.15. Zenodo. DOI: 10.5281/zenodo.4054193.
Churavy, V. (2019). *Vchuravy/ConcolicFuzzer.Jl*.
Congdon, P. (2006). *Bayesian Statistical Modelling*. 2nd ed. Wiley Series in Probability and Statistics. Chichester: Wiley. 573 pp.
Coq Development Team (2020). *Coq 8.12.1 Documentation*. URL: https://coq.inria.fr/distrib/current/refman/ (visited on 2020-12-08).
Cox, M., T. van de Laar & B. de Vries (2018). “ForneyLab. Jl: Fast and Flexible Automated Inference through Message Passing in Julia”. In: ProbProg.
Cusumano-Towner, M. F. et al. (2019). “Gen: A General-Purpose Probabilistic Programming System with Programmable Inference”. In: SIGPLAN 2019. PLDI, pp. 221–236. DOI: 10.1145/3314221.3314642.
Cusumano-Towner, M. F. (2020). “Gen: A High-Level Programming Platform for Probabilistic Inference”. PhD Thesis. Massachusetts: Massachusetts Institute of Technology.
Dahlin, J. & T. B. Schön (2015). “Getting Started with Particle Metropolis-Hastings for Inference in Nonlinear Dynamical Models”. In: arXiv: 1511.01707 [q-fin, stat].
Dauwels, J., S. Korl & H.-A. Loeliger (2005). “Steepest Descent as Message Passing”. In: IEEE Information Theory Workshop. DOI: 10.1109/ITW.2005.1531853.
Devroye, L. (1986). *Non-Uniform Random Variate Generation*. New York: Springer. 843 pp.
Elliott, C. (2018). “The Simple Essence of Automatic Differentiation”. In: arXiv: 1804.00746 [cs].
Fike, J. A. & J. J. Alonso (2012). “Automatic Differentiation through the Use of Hyper-Dual Numbers for Second Derivatives”. In: *Recent Advances in Algorithmic Differentiation*. Springer, pp. 163–173. DOI: 10.1007/978-3-642-30023-3_15.

Gabler, P. (2020). *MCMC Simulation Data for {AutoGibbs.Jl Benchmarks*. DOI: 10.5281/zenodo.4307916.

Gabler, P. et al. (2019). "Graph Tracking in Dynamic Probabilistic Programs via Source Transformations". In: 2nd Symposium on Advances in Approximate Bayesian Inference.

Gansner, E. R. & S. C. North (2000). "An Open Graph Visualization System and Its Applications to Software Engineering". In: *Software: Practice and Experience* 30.11, pp. 1203–1233.

Ge, H., K. Xu & Z. Ghahramani (2018). "Turing: A Language for Flexible Probabilistic Inference". In: International Conference on Artificial Intelligence and Statistics, pp. 1682–1690.

Gebremedhin, A. H. & A. Walther (2020). "An Introduction to Algorithmic Differentiation". In: *WIREs Data Mining and Knowledge Discovery* 10.1, e1334. DOI: 10.1002/widm.1334.

Gelman, A., J. B. Carlin, et al. (2020). *Bayesian Data Analysis*. 3rd.

Gelman, A. & D. B. Rubin (1992). "Inference from Iterative Simulation Using Multiple Sequences". In: *Statistical Science* 7.4, pp. 457–472. JSTOR: 2246093.

Geman, S. & D. Geman (1984). "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images". In: *IEEE Transactions on pattern analysis and machine intelligence* 6, pp. 721–741.

Girolami, M. & B. Calderhead (2011). "Riemann Manifold Langevin and Hamiltonian Monte Carlo Methods". In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 73.2, pp. 123–214. DOI: 10.1111/j.1467-9868.2010.00765.x.

Gonzalez, J. et al. (2011). "Parallel Gibbs Sampling: From Colored Fields to Thin Junction Trees". In: *Proceedings of Machine Learning Research*. JMLR. Vol. 15. Fort Lauderdale, pp. 324–332.

Goodman, N. D., V. Mansinghka, et al. (2012). "Church: A Language for Generative Models". In: arXiv: 1206.3255 [cs].

Goodman, N. D. & A. Stuhlmüller (2014). *The Design and Implementation of Probabilistic Programming Languages*. URL: http://dippl.org (visited on 2019-10-15).

Gowda, S. et al. (2019). "Sparsity Programming: Automated Sparsity-Aware Optimizations in Differentiable Programming". In: *Program Transformations for Machine Learning Workshop*. NeurIPS 2019.

Green, P. J. (1995). "Reversible Jump Markov Chain Monte Carlo Computation and Bayesian Model Determination". In: *Biometrika* 82.4, pp. 711–732. DOI: 10.1093/biomet/82.4.711.

Griewank, A. & A. Walther (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. 2nd ed. Philadelphia: Society for Industrial and Applied Mathematics. 438 pp.

Heunen, C. et al. (2017). "A Convenient Category for Higher-Order Probability Theory". In: arXiv: 1701.02547 [cs, math].

Hjort, N. L. et al. (2010). *Bayesian Nonparametrics*. Cambridge Series in Statistical and Probabilistic Mathematics 28. Cambridge: Cambridge University Press.
Hoffman, M. D., M. J. Johnson & D. Tran (2018). "Autoconj: Recognizing and Exploiting Conjugacy Without a Domain-Specific Language". In: arXiv: 1811.11926 [cs, stat].
Hong, M. & C. Lattner (2018). "Graph Program Extraction and Device Partitioning in Swift for TensorFLow". 2018 LLVM Developers' Meeting.
Hoyte, D. (2008). *Let over Lambda*.
Innes, M. J. (2018). "Don't Unroll Adjoint: Differentiating SSA-Form Programs". In: arXiv: 1810.07951 [cs].
Jia, Y. et al. (2014). "Caffe: Convolutional Architecture for Fast Feature Embedding". In: arXiv: 1408.5093 [cs].
Kallenberg, O. (2006). *Foundations of Modern Probability*. Springer Science & Business Media.
Koller, D. & N. Friedman (2009). *Probabilistic Graphical Models: Principles and Techniques*. Adaptive Computation and Machine Learning. Cambridge: MIT Press. 1231 pp.
Lattner, C. et al. (2020). "MLIR: A Compiler Infrastructure for the End of Moore's Law". In: arXiv: 2002.11054 [cs].
Lew, A. K. et al. (2020). "Trace Types and Denotational Semantics for Sound Programmable Inference in Probabilistic Languages". In: *Proceedings of the ACM on Programming Languages*. POPL 2020. Vol. 4, pp. 1–32. DOI: 10.1145/3371087.
LLVM Project (2019). *LLVM Language Reference Manual*. URL: https://llvm.org/docs/LangRef.html.
Looks, M. et al. (2017). "Deep Learning with Dynamic Computation Graphs". In: arXiv: 1702.02181 [cs, stat].
Lunn, D. J., D. Spiegelhalter, et al. (2009). "The BUGS Project: Evolution, Critique and Future Directions". In: *Statistics in Medicine* 28.25, pp. 3049–3067. DOI: 10.1002/sim.3680.
Lunn, D. J., A. Thomas, et al. (2000). "WinBUGS - A Bayesian Modelling Framework: Concepts, Structure, and Extensibility". In: *Statistics and Computing* 10, pp. 325–337. DOI: 10.1023/A:1008929526011.
Maclaurin, D., D. Duvenaud & R. P. Adams (2015). "Autograd: Effortless Gradients in Numpy". In: *AutoML Workshop*. ICML. Vol. 238, p. 5.
Mansinghka, V., D. Selsam & Y. Perov (2014). "Venture: A Higher-Order Probabilistic Programming Platform with Programmable Inference". In: arXiv: 1404.0099 [cs, stat].
Manzyuk, O. et al. (2019). "Perturbation Confusion in Forward Automatic Differentiation of Higher-Order Functions". In: *Journal of Functional Programming* 29, e12. DOI: 10.1017/S095679681900008X.
Marin, J.-M. & C. P. Robert (2007). *Bayesian Core: A Practical Approach to Computational Bayesian Statistics*. Springer Texts in Statistics. New York: Springer. 255 pp.

Minka, T. (2005). *Divergence Measures and Message Passing*. Technical Report MSR-TR-2005-173. Microsoft Research.
Minka, T. (2019). "From Automatic Differentiation to Message Passing". Presentation. Advances and Challenges in Machine Learning Languages Workshop.
Moses, W. S. & V. Churavy (2020). "Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients". In: arXiv: 2010.01709 [cs].
Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. San Francisco: Morgan Kaufmann. 856 pp.
Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. Adaptive Computation and Machine Learning Series. Cambridge: MIT Press. 1067 pp.
Murray, L. M. et al. (2017). "Delayed Sampling and Automatic Rao-Blackwellization of Probabilistic Programs". In: arXiv: 1708.07787 [stat].
Naumann, U. (2007). "Optimal Jacobian Accumulation Is NP-Complete". In: *Mathematical Programming* 112.2, pp. 427–441. DOI: 10.1007/s10107-006-0042-z.
Neubig, G. et al. (2017). "DyNet: The Dynamic Neural Network Toolkit". In: arXiv: 1701.03980 [cs, stat].
Paszke, A. et al. (2017). "Automatic Differentiation in PyTorch". In: *Autodiff Workshop*. NIPS 2017.
Plummer, M. (2003). "JAGS: A Program for Analysis of Bayesian Graphical Models Using Gibbs Sampling". In: *Proceedings of the 3rd International Workshop on Distributed Statistical Computing*. Vienna.
——— (2017). "JAGS Version 4.3.0 User Manual".
Pock, T. (2017). *Convex Optimization*. Lecture Notes. Graz: Graz University of Technology.
Press, W. H. et al. (2007). *Numerical Recipes: The Art of Scientific Computing*. 3rd ed. New York: Cambridge University Press.
Ramsey, N. & A. Pfeffer (2002). "Stochastic Lambda Calculus and Monads of Probability Distributions". In: *SIGPLAN Notices* 37.1, pp. 154–165. DOI: 10.1145/565816.503288.
Robert, C. P. & G. Casella (1999). *Monte Carlo Statistical Methods*. New York: Springer.
Rompf, T. & M. Odersky (2010). "Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs". In: *SIGPLAN Notices* 46. DOI: 10.1145/1868294.1868314.
Rosen, B. K., M. N. Wegman & F. K. Zadeck (1988). "Global Value Numbers and Redundant Computations". In: *Symposium on Principles of Programming Languages*. SIGPLAN-SIGACT 1988. San Diego: ACM Press, pp. 12–27. DOI: 10.1145/73560.73562.
Ruozzi, N. R. (2011). "Message Passing Algorithms for Optimization". Dissertation. Yale University.
Sajovic, Ž. & M. Vuk (2016). "Operational Calculus on Programming Spaces".

Salvatier, J., T. V. Wiecki & C. Fonnesbeck (2016). "Probabilistic Programming in Python Using PyMC3". In: *PeerJ Computer Science* 2, e55. DOI: 10.7717/peerj-cs.55.

Scherrer, C. (2019). *Soss.Jl*. URL: https://github.com/cscherrer/Soss.jl.

Ścibior, A., Z. Ghahramani & A. D. Gordon (2015). "Practical Probabilistic Programming with Monads". In: *Symposium on Haskell*. SIGPLAN 2015. Vancouver: ACM, pp. 165–176. DOI: 10.1145/2804302.2804317.

Sen, K., D. Marinov & G. Agha (2005). "CUTE: A Concolic Unit Testing Engine for C". In: *SIGSOFT Software Engineering Notes* 30.5, pp. 263–272. DOI: 10.1145/1095430.1081750.

Singer, J. (2018). *Static Single Assignment Book*.

Socher, R. et al. (2011). "Parsing Natural Scenes and Natural Language with Recursive Neural Networks". In: ICML'11. Madison: Omnipress, pp. 129–136. DOI: 10.5555/3104482.3104499.

Tao, T. (2011). *An Introduction to Measure Theory*. Graduate Studies in Mathematics 126. American Mathematical Society.

Tapenade developers (2019). *The Tapenade A.D. Engine*. URL: https://www-sop.inria.fr/tropics/tapenade.html (visited on 2019-10-09).

Tarek, M. et al. (2020). "DynamicPPL: Stan-like Speed for Dynamic Probabilistic Models". In: arXiv: 2002.02702 [cs, stat].

TensorFlow Developers (2018). *Swift for TensorFlow*. URL: https://github.com/tensorflow/swift (visited on 2020-11-01).

——— (2020). *XLA: Optimizing Compiler for Machine Learning*. URL: https://www.tensorflow.org/xla (visited on 2020-10-27).

Tokui, S. et al. (2015). "Chainer: A Next-Generation Open Source Framework for Deep Learning". In: *Workshop on Machine Learning Systems*. NIPS 2015.

Van de Meent, J.-W. et al. (2018). "An Introduction to Probabilistic Programming". In: arXiv: 1809.10756 [cs, stat].

Van Merrienboer, B., D. Moldovan & A. Wiltschko (2018). "Tangent: Automatic Differentiation Using Source-Code Transformation for Dynamically Typed Array Programming". In: *Advances in Neural Information Processing Systems 31*. NeurIPS. Ed. by Bengio, S. et al. Curran Associates, Inc., pp. 6256–6265.

Vihola, M. (2020). *Lectures on Stochastic Simulation*. Lecture Notes. University of Jyväskylä.

Vytiniotis, D. et al. (2019). "The Differentiable Curry". In: *Program Transformations for ML Workshop*. NeurIPS. Vancouver.

Wadler, P. & S. Blott (1989). "How to Make Ad-Hoc Polymorphism Less Ad Hoc". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, pp. 60–76.

Wang, F. et al. (2019). "Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator". In: arXiv: 1803.10228 [cs, stat].

Winn, J. & C. M. Bishop (2005). "Variational Message Passing". In: *Journal of Machine Learning Research* 6, pp. 661–694.

Winn, J., C. M. Bishop, et al. (2019). *Model-Based Machine Learning*.

Wood, F., J.-W. van de Meent & V. Mansinghka (2015). “A New Approach to Probabilistic Programming Inference”. In: arXiv: 1507.00996 [cs, stat].
Zeller, A. et al. (2019). *Concolic Fuzzing*. The fuzzing book. URL: https://www.fuzzingbook.org/html/ConcolicFuzzer.html.

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. Figures were created in Inkscape, plots in R using ggplot2.