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.

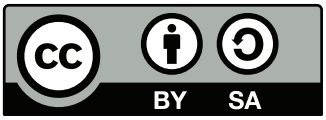
Andreas Okopenko

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 modelled distribution.

Contents

Notation

$\mathbb{P}[\Theta \in A \mid X = x]$

Random variables and their realizations will usually be denoted by upper and lower case letters, respectively (with occasional exceptions for Greek variable names). Sets are also named by uppercase letters.

$\mathbb{E}[X], \mathbb{V}_X[f(X, Y)]$

Expectation and variance; if necessary, the variable with respect to which the moment is taken is indicated as a subscript.

$\phi(x), f_Z(x)$

Density functions are named using letters commonly used for functions, with an optional subscript indicating the random variable they belong to. Densities always come with implied base measures depending on the type of the random variable.

$p(x, y \mid z)$

The usual abuse of notation with the letter "p" standing for any density indicated by the names of the variables given to it is used when no confusion arises (in this case, $f_{X,Y|Z}$ 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 explicitely specifying dependence or independence, where understood from context. Named distributions that are not themselves random variables are spelled out in upright script.

$Y \sim q(\cdot, X_{i-1})$

The same notation is used when a random variable is specified to be sampled from a given, possibly unnormalized,

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

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

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

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

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

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

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

$\mathrm{D}F(x, y) = (\Delta_1, \Delta_2) \mapsto \partial_1 F(x, y)\,\Delta_1 + \partial_2 F(x, y)\,\Delta_2$ — Derivatives are written using a capital D for the a total derivative operator, 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.

*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.

Many methods in the field of machine learning work on computation graphs of programs that represent mathematical expressions. One example are forms of automatic differentiation (AD) which derive an "adjoint" expression from a expression that usually represents a loss function, to calculate its gradient (Gebremedhin & Walther 2020; Griewank & Walther 2008). Another one are message passing algorithms (Minka 2005), which use the graph as the basic data structure for the operation they perform: passing values between nodes, representing random variables that depend on each other (in fact, message passing generalizes various other methods, including AD). But also in more or less unrelated fields, such as program analysis or program transformation (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 for the purpose of abstract interpretation.

There are several options how to provide the computation graph in question to an application, many of which are already established in the AD community (see Baydin et al. (2018) for a survey on AD methods). For one, graphs can be required to be written out explicitely by the user, by defining providing a library to build graphs "by hand" (e.g. Chewxy et al. (2020) and Jia et al. (2014) – these interfaces tend to be more low level) or through a higher-level API (e.g. PyTorch (Paszke et al. 2017) or TensorFlow (Abadi, Agarwal, et al. 2015)). Such APIs are called *operator overloading* in AD language, because they extend existing operations to additionally track the computation graph at runtime on so-called tapes or Wengert lists (Bartholomew-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. This notably happens for control statements, which can rarely be "overloaded". Furthermore, there are additional runtime costs due to the separate interpretation of derivatives stored on the tape.

Alternatively, an implementation can allow the user to write out computations as a "normal" program in an existing programming language (or possibly a restricted

subset of it), and use metaprogramming techniques to extract graphs from the input program. Such metaprograms, known under the name *source transformations*, can in turn operate on plain source code (cf. Tapenade (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 optimisations. Additionally, the user is not restricted to the domain specific language provided by a libray, and can use regular language constructs, data structures, and custom functions rather freely. But in this approach, usually, no records of the actual execution paths are constructed explicitly – purely static information is used only at compile time, and cannot be accessed for further analysis or transformation during execution.

In a variety of domains, though, the execution path of programs can drastically change at each run. Examples of this from machine learning are models with non-uniform data, such as parse trees (Socher et al. 2011) or molecular graphs (Bianucci et al. 2000), Bayesian nonparametric models (Hjort et al. 2010), or simply the occurrence of stochastic control flow in any probabilistic model. Such programs we call dynamic models. The lack of an explicit, unique graph structure makes it impossible, or at least diffult, to apply source transformation approaches on them. Operator overloading is the more direct way for supporting dynamic models, since it automatically records a new tape for each input. In fact, many of the already mentioned state-of-the-art machine learning libraries are based on dynamic graphs using operator overloading in some form.

However, relying on operator overloading makes it impossible to take advantage of the benefits of source transformations, such as utilizing information about the control flow, integrating with optimizations at compile time, or exploiting the source model structure. The source transformation approach based on intermediate compiler representations has recently gained popularity in machine learning; see 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 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 explicitely 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 optimised further. At run time, both data and control path are tracked alongside the original calculations, in the form of a nested data structure. This data structure contains all functions called during execution, enriched by recorded control flow decisions and possibly meta-information that can be used to analyse the execution. Thus, the system combines advantages of a source transformation with a tape-based runtime approach.

1.1 Related Work

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

Automatic differentiation has a long history, in which different styles becoming more or less fashionable depending on the dominating use-case and available languages and infrastructure. Traditionally, numerical code in Fortran or C was differentiated by whole-source transfomation systems like Tapenade (Tapenade developers 2019). In recent years, after phase of many library-based (or "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 lately. There are ongoing 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 (Innes 2018) has started to apply source transformation to the intermediate representation of the compiler, which enables differentiating through complex control flow, custom data types, and nested functions. A similar approach to Zygote is taken in Python with Tangent (van Merrienboer, Moldovan & Wiltschko 2018).

Generalizations of the kinds of analyses and transformations found in these systems can be found under multiple terms in the compiler literature: data- and control-flow analysis, information propagation, or abstract interpretation (Muchnick 1997; Singer 2018). These methods, most of which find fixed points in relations defined over a program, can in turn be seen as a form of message passing, under which not only a variety of learning algorithms can be summarized (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 (Gowda et al. 2019) or the detection of program parts 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 lan-

guages illustrate, there are often even multiple layers of intermediate representations used in the same system.

Recently, special intermediate representations for machine learning applications have been introduced. One of them is MLIR ("machine learning intermediate representation", Lattner et al. (2020)), with the purpose of forming a reusable mid-layer between programming languages and runtimes, featuring exchangeability between different machine learning frameworks and and "accelerators" (pieces of hardware), while taking advantage of modern compiler technology similar to LLVM. A second 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 numerical Python functions, that would otherwise be slow, to efficient platform code.

As for the trade-off between transformation-based and library-based implementations, several hybrid graph tracking approaches between source transformation and graph tracking exist. Among AD systems, recent TensorFlow versions have introduced AutoGraph[1], which rewrites regular Python functions to traced TensorFlow implementations by replacing control flow statements by TensorFlow combinators which. Such functions still need to be re-traced whenever a non-tensor input argument changes. Its predecessor TensorFlow Fold (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. The "dynamicity" problem can be approached in other ways as well: *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": multiple different traces are dynamically stored as alternative paralles paths in the execution trace, with possible interconnections.

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

2 Background

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

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

2.1 Bayesian Inference and MCMC methods

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

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

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

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

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

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

where x are the observed data, and θ are the unobserved parameters. The posterior represents the distribution of the unobserved variables as a combination of the prior belief updated by what has been observed (Congdon 2006). (In practice, not all of the unobserved variables have to be model parameters we are actually interested in; these can be integrated out).

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

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

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

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

in equation (2.1), often called the "evidence".

When the distributions involved form a sufficiently "nice" combination, e.g., a conjugate pair (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, the applicability of simple summation is limited by combinatorial explosion.

Different techniques for posterior approximation are available: among them are distribution-based approaches for general graphical models, such as variational inference (Murphy 2012, chapter 21 and 22) and other methods generalized under the

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

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 a specified density of $\Theta \sim \pi$, a sampling procedure with a consistent estimator for expectations:

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

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

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

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

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

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

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

$$\int \pi(\theta)K(A \mid \theta)\,\mathrm{d}\mu(\theta) = \int_A \pi(\theta)\,\mathrm{d}\mu(\theta) = \mathbb{P}[\Theta \in A], \tag{2.7}$$

and the LLN for Markov chains holds. (For discrete spaces, this relation is more familiarly written as a left eigenvalue equation on a stochastic matrix: $\pi K = \pi$.) The advantage of MCMC methods is that they apply equally well to many structurally complex models, and treat densities in a uniform way, without requiring special knowledge about the specific distribution in question. I refer to Vihola (2020, chapter 6), Robert & Casella (1999), and Murphy (2012, chapters 24 and following) as introductions to MCMC theory and practice.

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.

Frequently, MCMC methods are variations of the *Metropolis-Hastings algorithm* (MH), which splits the general definition of the transition kernel into two parts: a proposal distribution, given by a conditional density q that needs to be easy to sample from, and an acceptance rate α. Subsequent samples are then produced by proposing values from q given the previous element of the chain, and incorporating them into the chain with a probability given through α (see algorithm 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, of the form $\Theta = [\Theta_1, \ldots, \Theta_N]$, a good proposal distribution can be hard to find, though. One way to break down the problem is to use a family of componentwise updates, given by conditional distributions q_i operating on only one component of Θ, with the others fixed:

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

The components can be scalar or multivariate blocks, and the kernel may itself be any valid transition kernel (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): often, the simplest available set of transition kernels is given by the conditional densities $\theta_i \mapsto p(\theta_i \mid \theta_{-i}, x)$. They can directly be used as component proposals for a within-Gibbs sampler, leading to a cancelling acceptance rate of $\alpha \equiv 1$. This approach has the advantage of being very algorithmic, which makes it rather easy to apply, even by hand, to many models, and simply to express algorithmically. Hence, the method is a popular starting point for general probabilistic programming systems, most prominently BUGS (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:

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

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

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

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

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

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

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

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

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

which we recognize as a product of conjugate pairs of normal distributions. More examples are extensively covered in Murphy (2012, chapter 24.2).

```
@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.

2.2 Probabilistic Programming

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

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

The value of `forward` will be an dataframe-like object containing 10 values for each variable sampled from the forward (i.e., joint) distribution, matching the size of `x_observations`. Similarly, `chain` will contain a length 1000 sample from a Markov chain targetting the posterior, conditially on `x_observations`, created using the MH algorithm. If we were to write out 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 that models written in `Turing.jl`: no general interface for sampling algorithms to automatically detect all latent and observed variables; no possility for other, nonstandard execution forms as are needed for Variational Inference or gradient computation for HMC; no automatic 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, J. W. 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 (Cusumano-Towner 2020) and Soss (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 "mathematical" form, like as a graph or likelihood function. In a good probabilistic programming DSL, models will read as close to textbook model specifications as possible, while allowing to use the host language to:

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

Depending on the choice of 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 implicitely build such a structure at runtime, by allowing an inference algorithm to interleave the execution. The latter make it easier to include host-language control constructs. See J.-W. 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 hyperparameters. Subsequently, the `sample` function can be applied to this instance with different values for the sampling algorithm, which in turn will use the `evaluator` function of the instance to run the model with chosen

```
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.

`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 behaviour 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. 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 specializtion 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

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

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

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 is syntactically designed to bear a certain resemblance to Matlab, Python, or Ruby, but contrary to them, it is its own compiler, and not primarily the reliance on libraries calling foreign functions (e.g., Numpy), which is intrinsically enabling C-like speed. Although Julia does rely on, e.g., BLAS and LAPACK for numerical algebra, there is nothing that fundamentally prevents implementing their functions: true array types, fast loops, and various optimiations 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.

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 internally to the compiler, but also exposed to the programmer through macros, which allow to transform pieces of code at compile time. These macros resemble 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 libray 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, labelled 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
```

[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{Int})` as defined defined above, annotated with inferred types (as through `@code_warntype`).

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.)

After macro expansion, the code of the method is *lowered* into an intermediate representation consisting of only function calls and branches. This comprises of sevaral 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 a represented through sequence of labelled *basic blocks*, with (possibly conditional) jumps between them. The sequence of assignments is further processed into *single static assignment (SSA) form* (Singer 2018), the characteristic property of which is that every variable is assigned exactly once, thus giving it a unique, position-independent name to each intermediate value. By introducing this immutability guarantee, the resulting code is, in a certain sense, referentially transparent, which facilitates data-flow analysis, and makes many transformations easier. Accordingly, SSA form is widely used in intermediate forms of compiler systems, simplifying transformations and optimizations. The result of the translation of our example into into three basic blocks can be found in listing 2.3.

There is one noteable complication regarding conversion to SSA form: we need to be able to distinguish between assignments of variables arising from "joined" control flow. 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 varaible 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 function, in a way, 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.

Note that until now, the operations involved were purely syntactic in nature, and could be perfomed by solely taking into account the code of the function `foo`. As soon as `foo` is called on a concrete type during evaluation, though, the most specific method fitting to the argument types will be selected, and type inference on its body be applied. If we go on and call `foo([1.0])`, with `Vector{Int}` as the sole argument type, the types as annotated in the same listing will be inferred.

The last step of compilation within Julia consists of inlining and optimizing the typed intermediate code, resulting in the form shown in listing 2.4. There, several called methods have been inlined, and concrete argument types to invoke been

```
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`).

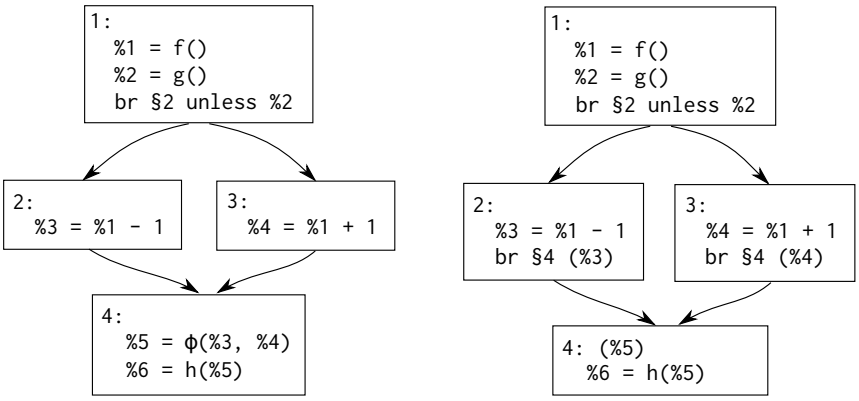


Figure 2.1: Two control flow graphs of the same function, illustrating the correspondence between SSA representations using ϕ-functions and block arguments. The SSA variables `%3` and `%4` correspond to the values of `y` in the two branches, which are merged in `%5`.

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 for compilation, 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 evaluated, the types of the arguments are first determined, and then the most specific method selected and called. This automatically gives the language dynamic semantics: an implementation can perform evaluation at every call. In practice, however, at this point multiple dispatch and JIT compilation combine into one of the main principles of optimization. Instead of evaluating the same code over and over again, methods are JIT-compiled the first time they are called. The compiled code is then cached in the method table. Method compilation does not happen recursively at once, though. Only when the body of a compiled method is then 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, this process of generating compiled methods from types is customizable in Julia: via so-called *generated functions*, there exists a process to dynamically generate code, given argument types – a form of stated programming (Bolewski 2015; Rompf & Odersky 2010). 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
        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"))` 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 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 practise. 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 known at runtime, 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 query the `CodeInfo` of a method by reflection, 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

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

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 runtime. This makes it possible to transform even 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 chapter: a call to a function `gradient(f, x, y)` will 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 simplify working with this kind of transformation: `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 functiona IR-transformation using this data structure easier. The latter is what the work of this thesis builds on.

2.4 Automatic Differentiation and Computation Graphs

This section may appear as an outlier from the original topic; in fact, the mathematics developed here are not even used later. However, it is important to explain the interrelations between automatic differentiation (AD), 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 practise. 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 an optimization problem over a multivatiate function with scalar output – typically a loss function over a parameter space, which measures how far a model prediction is form the true target values. The optimal model is then just that one

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

for which the parameters minimize the loss function. When the loss function is (sub-)differentiable, there exist a variety of gradient-based optimization methods to find this optimum (or, in the non-convex case, at least a practically sufficient local minimum).

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, 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 ov 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. There are many works on the formalization of AD in programming language theory; see, for example, Abadi & Plotkin (2020).

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. 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

$$\lim_{\|\Delta\|_X \to 0} \frac{\|f(x+\Delta) - f(x) - A(\Delta)\|_Y}{\|\Delta\|_X} = 0. \tag{2.14}$$

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.[7]

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 other words:

$$f(x+\Delta) = f(x) + \mathrm{D}f(x)(\Delta) + o\left(\|\Delta\|\right). \tag{2.15}$$

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

$$\mathrm{D}(\phi \circ \psi)(x) = \mathrm{D}\phi(\psi(x)) \circ \mathrm{D}\psi(x). \tag{2.16}$$

In the one-dimensional case, we simply have

$$\mathrm{D}\phi(x) = \Delta \mapsto \partial_1\phi(x)\,\Delta, \tag{2.17}$$

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

$$\begin{aligned} \mathrm{D}(\phi \circ \psi)(x)(\Delta) &= \left(\mathrm{D}\phi(\psi(x)) \circ \mathrm{D}\psi(x)\right)(\Delta) \\ &= \left(\partial_1\phi(\psi(x))\,\partial_1\psi(x)\right)\,\Delta, \end{aligned} \tag{2.18}$$

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.

[7] 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).

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 $\mathrm{D}g(x)(\Delta_1, \Delta_2) = \Delta_1 - \Delta_2$. By composition, we have:

$$\begin{aligned} \mathrm{D}f(x, y) &= \mathrm{D}(g \circ (\sin \otimes \mathrm{id}))(x, y) \\ &= \mathrm{D}g\,((\sin \otimes \mathrm{id})(x, y)) \circ \mathrm{D}(\sin \otimes \mathrm{id})(x, y) \\ &= (\Delta_1, \Delta_2) \mapsto \cos(x)\,\Delta_1 - \Delta_2. \end{aligned} \tag{2.19}$$

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:

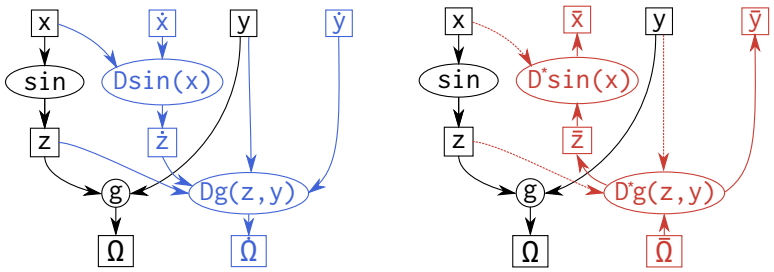
$$\begin{aligned} x &= ?, \\ y &= ?, \\ z &= \sin(x), \\ \Omega &= g(z, y). \end{aligned} \tag{2.20}$$

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 figure 2.2(a). Following the graph, or equivalently, following the equations in (2.20), 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*:

$$\begin{aligned} \dot{x} &= \Delta_1, \\ \dot{y} &= \Delta_2, \\ \dot{z} &= \mathrm{D}\sin(x)(\dot{x}) \\ &= \cos(x)\,\Delta_1, \\ \dot{\Omega} &= \mathrm{D}g(z, y)(\dot{z}, \dot{y}) \\ &= \cos(x)\,\Delta_1 - \Delta_2. \end{aligned} \tag{2.21}$$

The tangent values of input variables x and y become the input perturbations Δ_i. 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, which will be explained in more detail below.

Recovering the full gradient of a 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



(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 primal values in the derivative graph.

coordinate directions, for each of the input variables:

$$\nabla\phi(x) = \begin{pmatrix} \mathrm{D}\phi(x)(1,0,\ldots,0) \\ \vdots \\ \mathrm{D}\phi(x)(0,\ldots,0,1) \end{pmatrix} = \begin{pmatrix} \partial_1\phi(x) \\ \vdots \\ \partial_N\phi(x) \end{pmatrix}, \tag{2.22}$$

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 in "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:

$$\mathrm{D}^*\phi(x)(\mathfrak{d}) = \Delta \mapsto \mathfrak{d}(\mathrm{D}\phi(x)(\Delta)). \tag{2.23}$$

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:

$$\begin{aligned} \mathfrak{d}(\mathrm{D}\phi(x)(\Delta)) &= d^{\mathrm{T}} J_\phi(x)\Delta \\ &= \left(J_\phi(x)^{\mathrm{T}} d\right)^{\mathrm{T}} \Delta \\ &= \mathrm{D}^*\phi(x)(\mathfrak{d})(\Delta), \end{aligned} \tag{2.24}$$

where we assume $\mathfrak{d}$ to be represented by the covector 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 covector – and in fact, nothing else than a transformation of the transposed Jacobian. 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:

$$\begin{aligned}\mathrm{D}^*(\phi \circ \psi)(x)(\mathfrak{d}) &= d^{\mathrm{T}} J_\phi(\psi(x)) J_\psi(x) \\ &= \left(J_\psi(x)^{\mathrm{T}} J_\phi(\psi(x))^{\mathrm{T}} d\right)^{\mathrm{T}} \\ &= \left(\mathrm{D}^*\psi(x) \circ \mathrm{D}^*\phi(\psi(x))\right)(\mathfrak{d}).\end{aligned} \tag{2.25}$$

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:

$$\begin{aligned}\mathrm{D}^* f(x, y) &= \mathrm{D}^*(g \circ (\sin \otimes \mathrm{id}))(x, y) \\ &= \mathrm{D}^*(\sin \otimes \mathrm{id}) \circ \mathrm{D}^* g((\sin \otimes \mathrm{id})(x, y)) \\ &= \delta \mapsto [\cos(x)\delta, -\delta]^{\mathrm{T}}.\end{aligned} \tag{2.26}$$

In this form, starting with an outpt perturbation $\delta = 1$, we get back the gradient tuple through just one evaluation. Incidentially, this is nothing else than the backpropagation "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 (2.20), we get

$$\begin{aligned}\bar{x} &= \bar{z}_2 = -\delta, \\ \bar{y} &= \mathrm{D}^* \sin(x)\, \bar{z}_1 \\ &= \cos(x)\, \delta \\ \bar{z} &= \mathrm{D}^* g(x, y)(\bar{\Omega}) \\ &= [\delta, -\delta] \\ \bar{\Omega} &= \delta,\end{aligned} \tag{2.27}$$

which is displayed in the red graph in 2.2(b). Note that now, the backpropagated 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 in neural network. These have a shape of $\mathbb{R}^N \to \mathbb{R}^N$, but only involve a linear number of operations, so forward mode pays off[8]. 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).

The practical implementation of automatic differentiation 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 exampes are really applied in niche cases: large numeric simulations, and log-densities in a probabilistic model. 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 be recast in mathematically equivalent form by using dual numbers (see Baydin et al. 2018, section 3.1.1). 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 nonstandard interpretation as truncated Taylor series:

$$\phi(x + \epsilon) = \phi(x) + \partial_1\phi(x)\epsilon + \underbrace{\frac{\partial_1^2\phi(x)}{2}\epsilon^2 + \dots}_{\epsilon^2(\dots)=0} \tag{2.28}$$

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:

$$(z, \dot{z}) = (\phi(x), \mathrm{D}\phi(x)(\dot{x})) \quad \Leftrightarrow \quad z + \dot{z}\epsilon = \phi(x + \dot{x}\epsilon). \tag{2.29}$$

(generalization to higher dimensions, as well as higher derivatives in form of hyper-dual numbers, follow equally naturally.)

Dual numbers can rather easily be added to an existing programming language that has a sufficiently extensibly system for overloading mathematical operators.

[8] 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`.

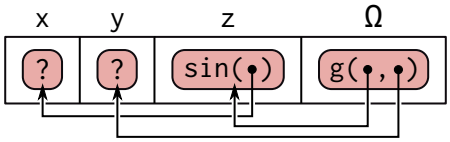


Figure 2.3: Wengert list of the example function `g(sin(x), y)` introduced above. Every intermediate variable becomes an element, linked through pointers. The gradient can be calculated by backward traversal and accumulating the adjoint values as metadata in the list elements.

This can be done using traits or type classes, like in Haskell, 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 adoint 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, but the usage of many higher order function 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 explicitely on a global Wengert list structure, or implicitly in the constructed expression objects. Then, one can start a backward pass from any leaf variable to propagate back derivatives to the roots of the computation graph, by following the edges and summing up adjoint values in parent nodes' metadata.

This style of implementation has limitations, though: it requires building up many objects at runtime, 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 runtime 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 unit – 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 through the available metaprogramming mechanisms (described in section 2.3) allow to systematically derive "adjoint programs" for arbitrary user-provided Julia functions, given only an extensible set of *primitive adjoints*. 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[9]. list some code_adjoint output?
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). Furthermore, there are applications that use the same techniques for other purposes, like sparsity detection (Gowda et al. 2019) or concolic execution (Churavy 2019).

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 implemeted 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) 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 in Bayesian methods, such as described below.

[9] `https://github.com/FluxML/Zygote.jl`

3 Implementation of Dynamic Graph Tracking in Julia

It has been described above that 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 was to analyze dynamic probabilistic models written in `Turing.jl`, properties of both were derired. Operator overloading has been considered as well, 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 the the work of Innes (2018), it seemed most promising to start from a source-transformation based approach implemented over the intermediate representation, 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 exteded by additional statements to record a trace of the executed statements and control flow operations at runtime. The algorithm and data structure on which this approach is based have already been shortly described in Gabler et al. (2019), and will be more extensively explained 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 intermediate representation passed-in functions. This technique can be applied to recursively traverse the implementation of a given function, annotating each operation with necessay 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 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 refererence 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`⟩`(@7, @3, ()...)` indicates that `geom` is a constant, and no variadic arguments are passed). The values of the arguments of this call can be inspected by looking up the respective references. Since `geom` is not a primitive function, the node holds tape of child nodes as well. In this case, it is equivalent to

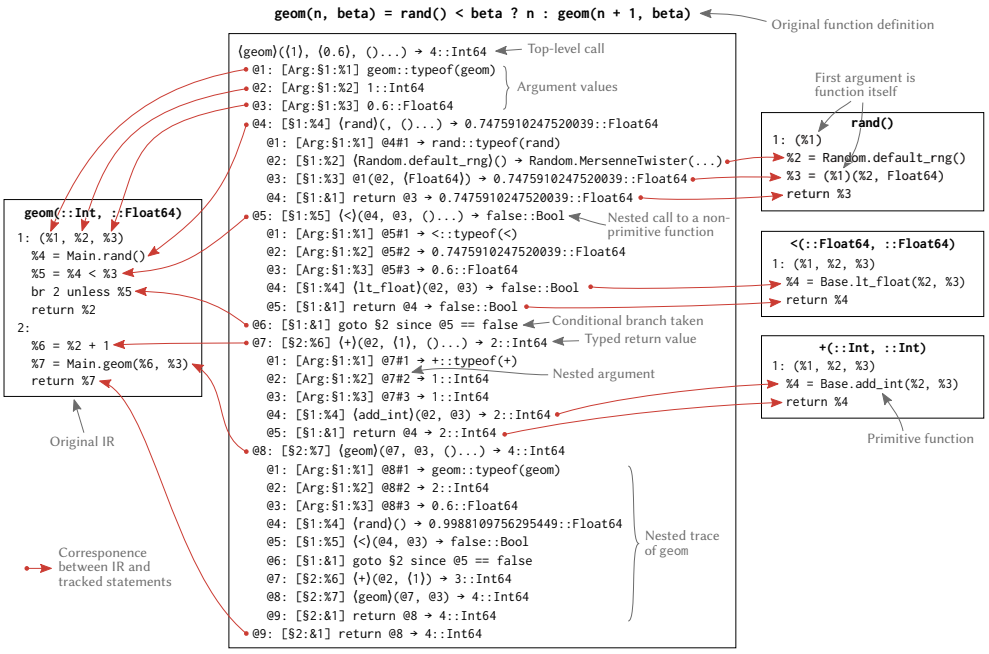


Figure 3.1: Extended Wengert list for one run of the stochastic function geom (only three levels shown). The central box is the tracked graph of the call `geom(1, 0.6)`. The other boxes show the original IR of the called non-primitive functions, to which the nodes are linked. Angle brackets indictate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

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 gener-

ated 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. 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 runtime. 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 IR is displayed in figure 3.1 above, 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, which describes the application of the function `+` to an SSA variable and a constant. The corresponding transformed IR is shown in the lower part of the figure, highlighted in green.

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`, which takes care of acually 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

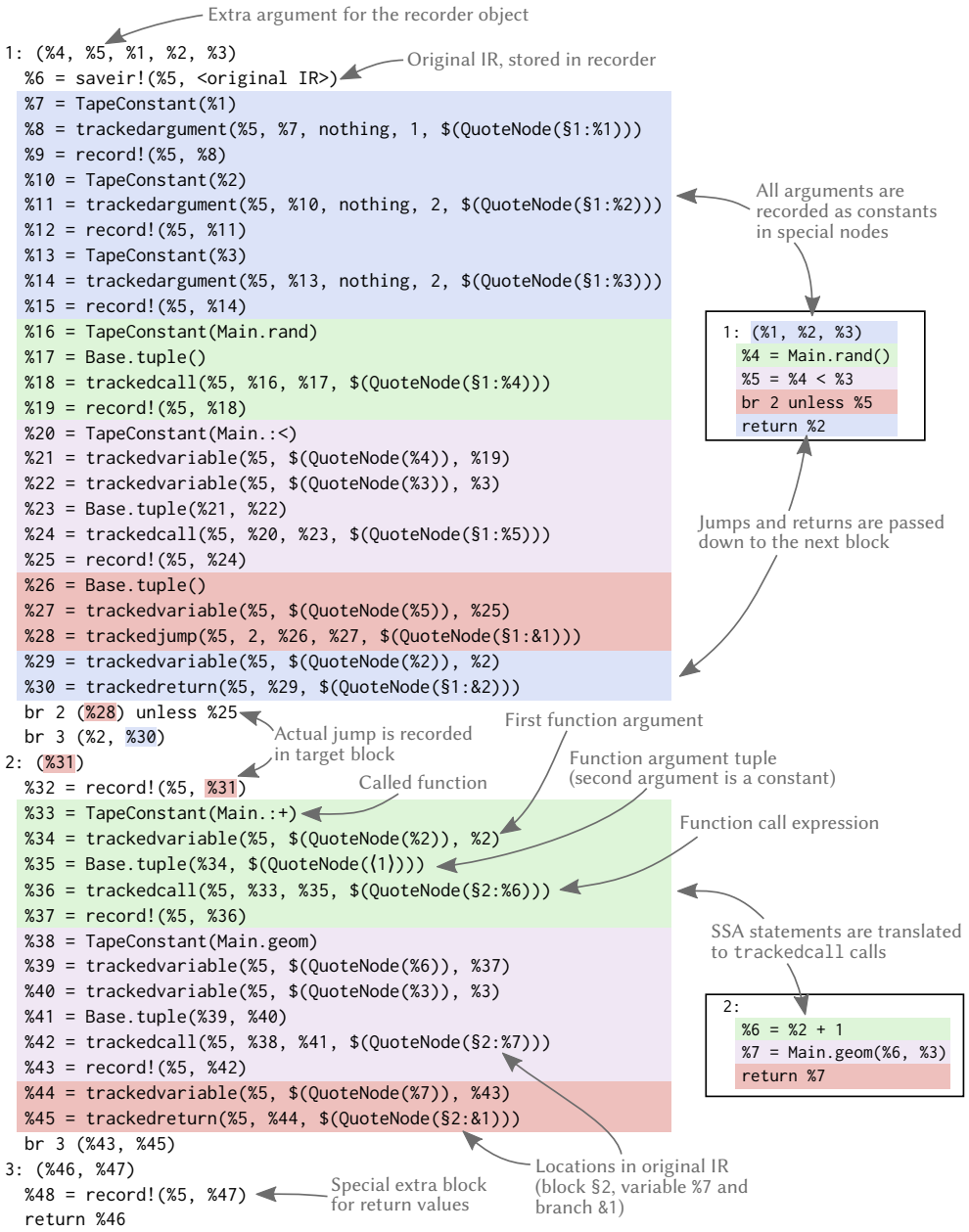


Figure 3.2: Tracked IR of the method `geom(::Int, ::Float64)`. Corresponding parts in original and transformed IR are highlighted in matching colors. (The original IR consists of two blocks, shown separately on the right.)

```
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).

recorder object before the respective jumps are taken. The solution is to first construct the respective nodes of all possible branches of a block (e.g., `%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 statement (`%30` and `%48` in our example; the branch variables of block 1 are highlighted in colors matching to the original branches). 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.

To provide some modularity and extensibility to the system, it also affords customization of some behaviour 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 behaviour – 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.

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 behaviour 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.

```
procedure TransformIR(original_ir)
    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
end procedure
```

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 runtime were left out for simplicity.

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 runtime 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 implemetation, not more individual evaluations of it are performed, except unit tests to ensure basic correctness of the interface. The system developed in chapter 4 provides a larger "integration test", though.

More potential use cases arise when the tracked model is actually static – in this case, the complete struture can be recovered from one graph tracking pass. This can then be analyzed and used in various ways; even more so, when more semantic knowledge about the model exists, such as meanings of certain domain-specific functions. Specifically, two show-cases for the system, applied to probabilistic programs written in `Turing.jl` were considered: conjugacy detection as described by Hoffman, Johnson & Tran (2018), and Rao-Blackwellization as in Murray et al. (2017). Due to the additional complexity required for them, only the automatic derivation of Gibbs conditionals, as shown in the next chapter, was finally executed, though.

The implementation is limited in two respects. First, in practical terms, there is are some trade-offs to be made regarding the storage of intermediate results and functions in nodes. In the current design, nodes are parametrized by the types of their contents, which leads to very large types, and potential slow done 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

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

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 one that is inherent to dynamic tracing: alternatives path, that were not taken due to runtime control flow, are not recorded. Compared to a traditional operator overloading system, `IRTracker.jl` does at least preserve the information about which branches were taken, and for which reason in the case of conditional branches; this is not enough for complete static analysis in all cases, though. The system is sufficient for code that does not change data flow paths depending on arguments or stochastic decisions, though.

One possible direction for extension that would extend the applicablity somewhat is concolic execution (Zeller et al. 2019), in which the function is traced multiple times with different arguments, whose exact values are determined by contraint 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, though. 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 for extension, 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 dependecy 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 dependecy 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 `Graphs` of 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 vizualized 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)
    p ~ DiscreteNonParametric([0.3, 0.7], w)
    x ~ Bernoulli(p)
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.

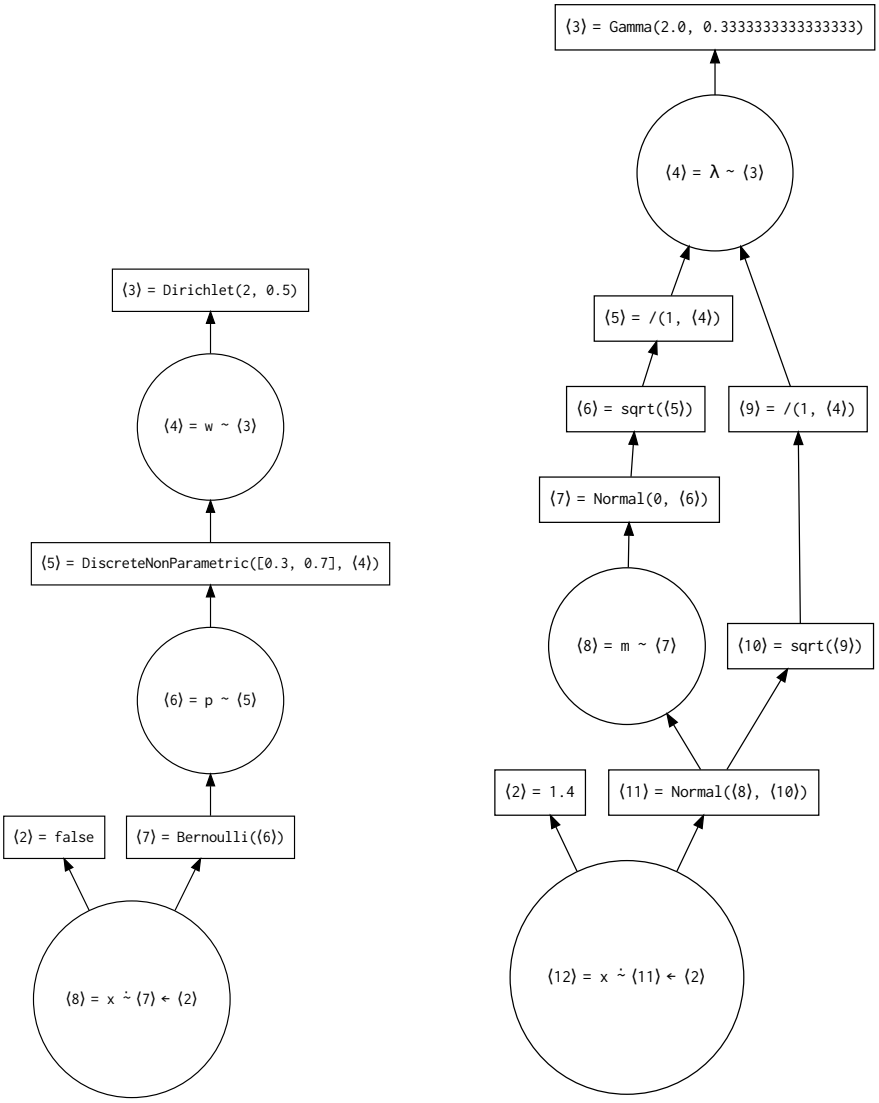
```
⟨2⟩ = false
⟨3⟩ = Dirichlet(2, 0.5) → Dirichlet{Float64}(alpha=[0.5, 0.5])
⟨4⟩ = w ~ ⟨3⟩ → [0.826304431175434, 0.17369556882456608]
⟨5⟩ = DiscreteNonParametric([0.3, 0.7], ⟨4⟩) → DiscreteNonParametric{...}(
        support=[0.3, 0.7], p=[0.826304431175434, 0.17369556882456608])
⟨6⟩ = p ~ ⟨5⟩ → 0.3
⟨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 modelling language, and available in completely structured form.

fix parskips, see background.tex:445

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 reachedable within one run of model tracking. A large fraction of the models used in practise 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 distribution 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⟩ = p ~ ⟨5⟩                ⇝ logpdf(DNP([0.3, 0.7], θ[w]), θ[p])
⟨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` for better spacing.)

- 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 expression are thereby to be understood as an implicit function 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 likelihood functions for each tilde statement, we get the symbolic log-joint density as, for example,

```
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, factorizd 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:

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

reference

which in more technical terms are given through the *Markov blanket* of m and λ.

The complicated part of this step is the correction subsumption of indexed variables in the trace dictionary: making sure that both θ`[x][1]` and θ`[x[1]]` are resolved correctly to the same value.

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 just more conceptual than optimized. Much of the slow-down could be mitigated by improving key parts of `AutoGibbs.jl` and `IRTracker.jl`.

In addition to that, one must realize where this appearent 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.

So, to conclude, while the implementation in current form is not very applicable in practice for all 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 models chosen to test certain features and corner cases, an experimental comparison of `AutoGibbs.jl` and existing `Turing.jl` samplers has

been conducted. For this purpose, 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),

$$\begin{aligned} w &\sim \text{Dirichlet}(\text{K}) \\ z_n &\sim \text{Categorical}([1, \ldots, K], w), \quad n = 1, \ldots, N \\ \mu_k &\sim \text{Normal}(0, \sigma_1), \quad k = 1, \ldots, K \\ x_n &\sim \text{Normal}(\mu_{z_n}, \sigma_1), \quad n = 1, \ldots, N; \end{aligned} \tag{4.3}$$

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

$$\begin{aligned} T_k &\sim \text{Dirichlet}(K), \quad k = 1, \ldots, K \\ m_k &\sim \text{Normal}(k, \sigma_1), \quad k = 1, \ldots, K \\ s_1 &\sim \text{Categorical}([1, \ldots, K], [1/K, \ldots, 1/K]) \\ s_k &\sim \text{Categorical}([1, \ldots, K], T_{s_{k-1}}), \quad k = 2, \ldots, N \\ x_k &\sim \text{Normal}(m_{s_k}, \sigma_2), \quad k = 1, \ldots, N; \end{aligned} \tag{4.4}$$

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

$$\begin{aligned} w &\sim \text{TruncatedStickBreakingProcess}(\alpha, \text{K}) \\ z_n &\sim \text{Categorical}([1, \ldots, K], w), \quad n = 1, \ldots, N \\ \mu_k &\sim \text{Normal}(0, \sigma_1), \quad k = 1, \ldots, K \\ y_n &\sim \text{Normal}(\mu_{z_n}, \sigma_2), \quad n = 1, \ldots, N. \end{aligned} \tag{4.5}$$

The models are implemented in `DynamicPPL.jl` as shown in listing 4.3. The three interesting classes of metrics to evaluate are, in the case of this work,

1. the "compilation time" of `AutoGibbs.jl`, i.e., the time it takes to extract a conditional,
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.)

Each of the test models involves one discrete and two continuous parameters. As benchmark for `AutoGibbs.jl`'s static conditional (AG), it is compared to `Turing.jl`'s Particle Gibbs sampler (PG), which is also suited to discrete parameters. Continuous variables are all sampled using Hamiltonian Monte Carlo (HMC) with hand-tuned parameters per model. The experiments have been set up such that each model is evaluated for ten chains of length 10000, varying between AG and PG, and four different values for the number of observations (since these determine the size of

```
@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 weights as `1:length(p)`.

		GMM			HMM			IMM
	HMC Step size		0.05			0.05		0.05
	HMC Steps		10			10		10
AG + HMC	Observations	10	25	50	10	25	50	10
	Chains	30	30	30	30	30	30	30
	Compilations	3	3	3	3	3	3	3
PG + HMC,	Observations	10	25	50	10	25	50	10
	Chains	10	10	10	10	10	10	10

Table 4.1: Experimental conditions for evaluating AutoGibbs (AG) agains Particle Gibbs (PG). Chains were always of length 10000. A new static Gibbs conditional was extracted for each block of 10 chains that was run with the same parameters while Particle Gibbs was varied over the three particle sizes. Particle Gibbs with 50 particles was sometimes killed due to timeouts on the server.

the trace, and thus have an influence on both AG's compile times and the overall sampling time). PG was always used with 100 particles, since lower values did lead to convergent chains.

Concrete parameters in table 4.1.

more meaningful table

5 Discussion

The history of this project forms a large arc from a general problem in `Turing.jl`, over a digression into compiler technology and automatic differentiation, back the the implementation of a proof of concept in the form of a very specific inference method. As we have seen, two separate pieces of software have emerged from it: `IRTracker.jl` and `AutoGibbs.jl`. The underlying issue – that `Turing.jl` lacks a structural representation of models – is not at all resolved by them, unfortunately. This makes `AutoGibbs.jl` not completly satisfactory, since the recursion and branch tracking features of `IRTracker.jl` cannot be applied in a useful way.

`AutoGibbs.jl` isn's so bad: significantly faster and therefore longer chains are possible, parallelization is possible. Reuse between (parametrized) models.

write out

The real difficulty is that dynamic models cannot be satisfactorily handled through static "snapshots" in the form of traces. Systems trying to achieve this either become restrictive in their expressibility, or limited in their applicability. Furthermore, during implementation, 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 (e.g., when one has an array `x`, samples `x[1]`, writes it to `x` with `setindex!`, and then uses `getindex(x, i)` somewhere downstream). These cannot be really solved using external tooling, in the way that has been demonstrated. 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.

compare to autograd, venture, church

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, also the structure of `DynamicPPL.jl`'s model representations is not stable, and actually an implementation detail that should not be relied on from the outside – especially not by an important feature such as dependency extration. In a certain sense, the whole approach is misguided: why rely on external tracking for a framework that is really under ones control, using such heavy machinery as IR transformations? This is illuminated by following very telling quote about similar tendencies (`Cassette.jl` is a package very similar to `IRTools.jl`):

> Using Cassette on code you wrote is a bit like shooting youself with a

> experimental mind control weapon, to force your hands to move like you knew how to fly a helicopter. Even if it works, you still had to learn to fly the helicopter in order to program the mind-control weapon to force yourself to act like you knew how to fly a helicopter.[1]

In conclusion, the dynamic graph tracking system has turned out to be an interesting idea with some application possibilities. Unfortunatley, the analysis of dynamic probabilistic models could not be shown to be among them. In the course of development, though, 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 layed out in the following section.

5.1 Future Work

While `IRTracker.jl` is a quite satisfying and complete system, the approach that `AutoGibbs.jl` takes provides only an ad-hoc solution to one shortcoming of `Turing.jl`: the lack a structural model represenation that is open to analysis and transformations. This has lead me to consider alternatives, approaching the representation problem for probabilistic programming languages on a more fundamental level. Most of the following ideas I have already informally described online[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. And the internal representation should be such that multiple forms of analysis, optimization, non-standard execution, and transformation can be performed.

Currently, `Turing.jl` is very primitive in this respect: one data structure (`VarInfo`) contains a map from variable names to values, the accumulated log-likelihood, and some other (stateful!) sampling metadata. `AutoGibbs.jl`' approach consists of retrofitting some more structure onto this representation – this is not ideal, and for proper analysis, it would be desireable to begin with a better representation from the start, at least for the reasons mentioned above.

From difficulties described above, that became appearent during the implementation of the Gibbs conditional extraction, together with the knowledge about `DynamicPPL.jl`'s internals, I developed an understanding of what an more advanced representation of probabilistic models, with a focus on transformation and analysis, could be; coming mostly from a metaprogramming and static analysis perspective, emplying ideas from functional programming language design. My abstract, idealized wish was to for variable names and dependencies to behave nicely as abstract data structures, and to operate primarily in a closed, elegant, high-level language. Many successful approaches to PPL design probably come from the perspective of

[1] Lyndon White (2020), private communication on `https://julialang.slack.com`.
[2] `https://github.com/phipsgabler/probability-ir`

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 linguistic perspective as well. A further goal of mine is the wish to close the gap between practical inference systems and the mostly theoretical, functional-programming-based approaches of just formalizing probabilistic programs.

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, tradeoffs need to be made to still be able to do this. It seem advantageous to split up this conjuction: 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 performd 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 in a way turns around how things are constructed right now in most of the approaches. Instead of starting from a "sampling function", which is evaluated to extract graphs or other symbolic representations from it, one should start from a model representation that already is general, yet richly structured, and derive evaluators from it. On the other hand, in contrast to PPLs that are built on top of a DLS representation, such an IR 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. And if furthermore such a representation takes the form of SSA-form IR, it doesn'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 are also very general "pure inference" libraries, examples of general evaluators, which take a function and do their work; but they are not really a PPL. There are some domain-specific "linguae frankae" like the syntax of Stan and JAGS, but they are, too, somewhat restricted, and not independently maintained – the ones coming later just chose to take over the same kind of input format for their own implemenation. All these approaches should rather be abstracted out into a model specification form in its own right, that has more general analysis capabilites, and can then be transformed town to whatever the evaluator requires.

The advantage of this kind of approach, besides solving "compiler domain" problems like the ones I mentioned above, is that it provides a different kind of common abstraction for PPLs. Recently, people have started writing "bridge code" to allow PPL interaction: there is invented a common interface that multiple PPL

systems can be fit under, and then models in each can be used from within the other at evaluation. This is necessary due to the lack of division of a system into an evalator and a model specification part: they always go together. I believe that starting from a common model specification language is in many cases more feasible and general than defining a common interface for evaluators. The latter tends to assume much more about the internals, while model syntax is essentially fixed: the notation of random variables used in model specification by hand, extended through general Julia syntax.

As far as I see it, a sort of least upper bound of all the PPL modelling approaches consists of

- General Julia code: this can most conveniently be represented like normal Julia IR with SSA statements, branches, and blocks.
- "Sampling statements": tildes, or assumptions and observations in `Turing.jl` parlance, which relate names or values to distributions in an immutable way.
- Variable names: which may be "complex", containing indexing, fields, link functions, etc., which can be identified and analyzed in a structured way.

So my idea was to just combine all that into an extended IR-like syntax. This amounts to writing out a directed graphical model with deterministic and stochastic nodes, named random variables, but generalized to programs – i.e., allowing dynamic structure due 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), factorized into primitive statements and blocks.

Bibliography

Abadi, M., A. Agarwal, et al. (2015). "TensorFlow: Large-scale machine learning on heterogeneous systems". Preliminary White Paper. Preliminary White Paper. URL: https://www.tensorflow.org/ (visited on 2020-07-29).

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. URL: https://dl.acm.org/doi/10.1145/3371106 (visited on 2020-11-20).

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

Apple (2020). *Swift Compiler*. Swift.org. 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*. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations 124.1, pp. 171–190. DOI: 10.1016/S0377-0427(00)00422-2. URL: http://www.sciencedirect.com/science/article/pii/S0377042700004222 (visited on 2019-04-22).

Baydin, A. G. et al. (2018). "Automatic differentiation in machine learning: a survey". In: *Journal of Machine Learning Research* 18.153, pp. 1–43. URL: http://jmlr.org/papers/v18/17-468.html.

Becker, M. R. (2020). "Dynamic specialization in trace-based probabilistic programming systems". In: ProbProg2020.

Betancourt, M. (2018). *A Conceptual Introduction to Hamiltonian Monte Carlo*. arXiv: 1701.02434 [stat]. URL: http://arxiv.org/abs/1701.02434 (visited on 2020-10-17).

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. URL: https://doi.org/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. URL: https://epubs.siam.org/doi/10.1137/141000671 (visited on 2019-10-02).

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*. arXiv: 1810.09538 [cs, stat]. URL: http://arxiv.org/abs/1810.09538 (visited on 2019-10-14).

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. Boston: JuliaCon 2015. URL: https://www.youtube.com/watch?v=KAN8zbM659o (visited on 2019-10-09).

Bradbury, J. et al. (2018). *JAX: composable transformations of Python+NumPy programs*. Version 0.1.55. URL: http://github.com/google/jax.

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. URL: https://www.jstatsoft.org/index.php/jss/article/view/v076i01 (visited on 2020-10-19).

Carpenter, B., M. D. Hoffman, et al. (2015). *The Stan Math Library: Reverse-Mode Automatic Differentiation in C++*. arXiv: 1509.07164 [cs]. URL: http://arxiv.org/abs/1509.07164 (visited on 2020-03-26).

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*. URL: https://github.com/vchuravy/ConcolicFuzzer.jl/tree/v0.0.1-alpha (visited on 2020-11-20).

Congdon, P. (2006). *Bayesian statistical modelling*. 2nd ed. Wiley Series in Probability and Statistics. Chichester, England ; Hoboken, NJ: John Wiley & Sons. 573 pp.

Cusumano-Towner, M. F. (2020). "Gen: A High-Level Programming Platform for Probabilistic Inference". PhD Thesis. Massachusetts: Massachusetts Institute of Technology.

Dauwels, J., S. Korl & H.-A. Loeliger (2005). "Steepest descent as message passing". In: IEEE Information Theory Workshop. Rotorua. DOI: 10.1109/ITW.2005.1531853. URL: http://ieeexplore.ieee.org/document/1531853/ (visited on 2019-10-03).

Elliott, C. (2018). *The simple essence of automatic differentiation*. arXiv: 1804.00746 [cs]. URL: http://arxiv.org/abs/1804.00746 (visited on 2019-03-19).

Gabler, P. et al. (2019). "Graph Tracking in Dynamic Probabilistic Programs via Source Transformations". In: 2nd Symposium on Advances in Approximate Bayesian Inference. Vancouver. URL: https://openreview.net/forum?id=r1eAFknEKr (visited on 2020-07-07).

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*. International Conference on Artificial Intelligence and Statistics, pp. 1682–1690. URL: http://proceedings.mlr.press/v84/ge18b.html (visited on 2019-03-01).

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. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/widm.1334 (visited on 2020-07-29).

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. URL: https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-9868.2010.00765.x (visited on 2020-10-17).

Goodman, N. D., V. Mansinghka, et al. (2012). *Church: a language for generative models*. arXiv: 1206.3255 [cs]. URL: http://arxiv.org/abs/1206.3255 (visited on 2019-03-22).

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: URL: https://openreview.net/forum?id=rJlPdcY38B (visited on 2020-07-29).

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. URL: https://academic.oup.com/biomet/article/82/4/711/252058 (visited on 2020-10-17).

Griewank, A. & A. Walther (2008). *Evaluating derivatives: principles and techniques of algorithmic differentiation*. 2nd ed. Philadelphia: Society for Industrial and Applied Mathematics. 438 pp.

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*. arXiv: 1811.11926 [cs, stat]. URL: http://arxiv.org/abs/1811.11926 (visited on 2019-10-09).

Hong, M. & C. Lattner (2018). “Graph Program Extraction and Device Partitioning in Swift for TensorFLow”. 2018 LLVM Developers’ Meeting. URL: https://www.youtube.com/watch?v=HSneJdPkaKk (visited on 2019-05-01).

Hoyte, D. (2008). *Let over lambda*.

Innes, M. J. (2018). *Don't Unroll Adjoint: Differentiating SSA-Form Programs*. arXiv: 1810.07951 [cs]. URL: http://arxiv.org/abs/1810.07951 (visited on 2019-04-26).

Jia, Y. et al. (2014). *Caffe: Convolutional Architecture for Fast Feature Embedding*. arXiv: 1408.5093 [cs]. URL: http://arxiv.org/abs/1408.5093 (visited on 2020-10-26).

Lattner, C. et al. (2020). *MLIR: A Compiler Infrastructure for the End of Moore's Law*. arXiv: 2002.11054 [cs]. URL: http://arxiv.org/abs/2002.11054 (visited on 2020-10-26).

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*. arXiv: 1702.02181 [cs, stat]. URL: http://arxiv.org/abs/1702.02181 (visited on 2019-03-18).

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. URL: https://www.onlinelibrary.wiley.com/doi/abs/10.1002/sim.3680 (visited on 2020-10-19).

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.

Mansinghka, V., D. Selsam & Y. Perov (2014). *Venture: a higher-order probabilistic programming platform with programmable inference*. arXiv: 1404.0099 [cs, stat]. URL: http://arxiv.org/abs/1404.0099 (visited on 2019-09-09).

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. URL: https://www.cambridge.org/core/product/identifier/S095679681900008X/type/journal_article (visited on 2020-11-14).

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. URL: https://www.microsoft.com/en-us/research/publication/divergence-measures-and-message-passing/ (visited on 2019-10-09).

Minka, T. (2019). *From automatic differentiation to message passing*. Presentation. Advances and challenges in Machine Learning Languages workshop. URL: https://www.microsoft.com/en-us/research/video/from-automatic-differentiation-to-message-passing/ (visited on 2019-09-03).

Moses, W. S. & V. Churavy (2020). *Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients*. arXiv: 2010.01709 [cs]. URL: http://arxiv.org/abs/2010.01709 (visited on 2020-11-16).

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, MA: MIT Press. 1067 pp.

Murray, L. M. et al. (2017). *Delayed Sampling and Automatic Rao-Blackwellization of Probabilistic Programs*. arXiv: 1708.07787 [stat]. URL: http://arxiv.org/abs/1708.07787 (visited on 2019-10-09).

Naumann, U. (2007). "Optimal Jacobian accumulation is NP-complete". In: *Mathematical Programming* 112.2, pp. 427–441. DOI: 10.1007/s10107-006-0042-z. URL: http://link.springer.com/10.1007/s10107-006-0042-z (visited on 2020-11-10).

Neubig, G. et al. (2017). *DyNet: The Dynamic Neural Network Toolkit*. arXiv: 1701.03980 [cs, stat]. URL: http://arxiv.org/abs/1701.03980 (visited on 2019-03-08).

Paszke, A. et al. (2017). "Automatic differentiation in PyTorch". In: NIPS 2017 Workshop Autodiff.

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 (DSC 2003)*. Vienna.

——— (2017). "JAGS Version 4.3.0 user manual". URL: https://sourceforge.net/projects/mcmc-jags/files/Manuals/4.x/jags_user_manual.pdf/download (visited on 2020-07-17).

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, NY, USA: Cambridge University Press.

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: *ACM 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: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '88*. The 15th ACM SIGPLAN-SIGACT Symposium. San Diego, California, United States: ACM Press, pp. 12–27. DOI: 10.1145/73560.73562. URL: http://portal.acm.org/citation.cfm?doid=73560.73562 (visited on 2020-11-07).

Ruozzi, N. R. (2011). “Message Passing Algorithms for Optimization”. Dissertation. Yale University.

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. URL: https://peerj.com/articles/cs-55 (visited on 2020-10-19).

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

Singer, J. (2018). *Static Single Assignment Book*. URL: http://ssabook.gforge.inria.fr/latest/book-full.pdf (visited on 2020-07-30).

Socher, R. et al. (2011). “Parsing Natural Scenes and Natural Language with Recursive Neural Networks”. In: *Proceedings of the 28th International Conference on International Conference on Machine Learning*. ICML’11. USA: Omnipress, pp. 129–136. URL: http://dl.acm.org/citation.cfm?id=3104482.3104499.

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*. Version 1. arXiv: 2002.02702 [cs, stat]. URL: http://arxiv.org/abs/2002.02702 (visited on 2020-10-18).

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

——— (2020). *XLA: Optimizing Compiler for Machine Learning*. TensorFlow. 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: *Proceedings of Workshop on Machine Learning Systems (LearningSys) in the Twenty-Ninth Annual Conference on Neural Information Processing Systems (NIPS)*. URL: http://learningsys.org/papers/LearningSys_2015_paper_33.pdf.

Van de Meent, J.-W. et al. (2018). *An Introduction to Probabilistic Programming*. arXiv: 1809.10756 [cs, stat]. URL: http://arxiv.org/abs/1809.10756 (visited on 2019-03-08).

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*. Ed. by Bengio, S. et al. Curran Associates, Inc., pp. 6256–6265. URL: http://papers.nips.cc/paper/7863-tangent-automatic-differentiation-using-source-code-transformation-for-dynamically-typed-array-programming.pdf.

Vihola, M. (2020). *Lectures on stochastic simulation*. University of Jyväskylä. URL: http://users.jyu.fi/~mvihola/stochsim/.

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

Wood, F., J. W. van de Meent & V. Mansinghka (2015). *A New Approach to Probabilistic Programming Inference*. arXiv: 1507.00996 [cs, stat]. URL: http://arxiv.org/abs/1507.00996 (visited on 2019-10-15).

Zeller, A. et al. (2019). "Concolic fuzzing". In: *The Fuzzing Book*. Saarland University. 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, and plotting done in R using ggplot2.