Philipp Gabler, BSc

Automatic Graph Tracking in Dynamic Probabilistic Programs via Source Transformations

Master's Thesis

to achieve the university degree of
Master of Science

submitted to
Graz University of Technology

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

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

Institute of Signal Processing and Speech Communication

Faculty of Electrical and Information Engineering

Graz, XXXX 2020

Affidavit

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

Date Signature

This work is licensed under a
Creative Commons Attribution-ShareAlike 4.0 International License.

All code samples, unless otherwise noted or cited from other sources,
are also available under an MIT license:

```
The MIT License (MIT)

Copyright (c) 2020 Philipp Gabler

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject
to the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGE-
MENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF
CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

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

[*] pgabler@student.tugraz.at

Abstract

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

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

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

Contents

Notation

$\mathbb{P}[\Theta \in A \mid X = x]$	Random variables and their realizations will usually be denoted by upper and lower case letters, respectively (with occasional exceptions for Greek variable names). Sets are also named by uppercase letters.
$\mathbb{E}[X], \mathbb{V}_X[f(X, Y)]$	Expectation and variance; if necessary, the variable with respect to which the moment is taken is indicated as a subscript.
$\phi(x), f_Z(x)$	Density functions are named using letters commonly used for functions, with an optional subscript indicating the random variable they belong to. Densities always come with implied base measures depending on the type of the random variable.
$p(x, y \mid z)$	The usual abuse of notation with the letter “p” standing for any density indicated by the names of the variables given to it is used when no confusion arises (in this case, $f_{X,Y \mid Z}$ is implied). A q may be used as well, mostly for proposal distributions or unnormalized densities.
$\mathbb{P}[X \in A] = P_X(A) = \int_A p_X(x) \, \mathrm{d}\mu(x)$	A capital P with subscript is used for the probability measure associated with a random variable.
$\mathbb{P}[X \in \mathrm{d}x] = P_X(\mathrm{d}x) = p_X(x) \, \mathrm{d}\mu(x)$	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.
$\Theta^{(k)} = [\Theta_1^{(k)}, \dots, \Theta_N^{(k)}]$	Superscript indices in parentheses are used for series or sequences of variables, and subscript indices for components of multivariate variables.
$z_{-i} = [z_1, \dots, z_{i-1}, z_{i+1}, \dots, z_N]$	Negative indices denote all components of a variable without the negated one.
$f.(x, 1) = [f(x_1, 1), \dots, f(x_N, 1)]$	Function application with a period indicates vectorized application, as in Julia code[*]: the function is applied over all elements of the input arrays individually, whereby arrays of lower rank or scalars are "broadcasted" along dimensions as necessary.
`f(x) = rand(x)`	Julia code (including identifiers mention in the text) is always typeset in typewriter font.

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

1 Introduction

This chapter gives an overview over the scope of the thesis and existing approaches in the literature. 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 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 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 languages 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 similar role is played by Swift’s intermediate representation, SIL, in the Swift for TensorFlow project. XLA (“accelerated linear algebra”, TensorFlow Developers (2020)) plays a similar role for expression-graph based machine learning systems, compiling those graphs directly to optimized code; this system forms the basis of JAX (Bradbury et al. 2018), which allows to JIT-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 observations 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 xi 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:

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

for all $k \geq 1$. By constructing the parametrize 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 $\Theta = [\Theta_1, \dots, \Theta_N]$, 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:

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

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:

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

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

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

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 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

```julia
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:

```julia
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:

```julia
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:

```julia
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)
```

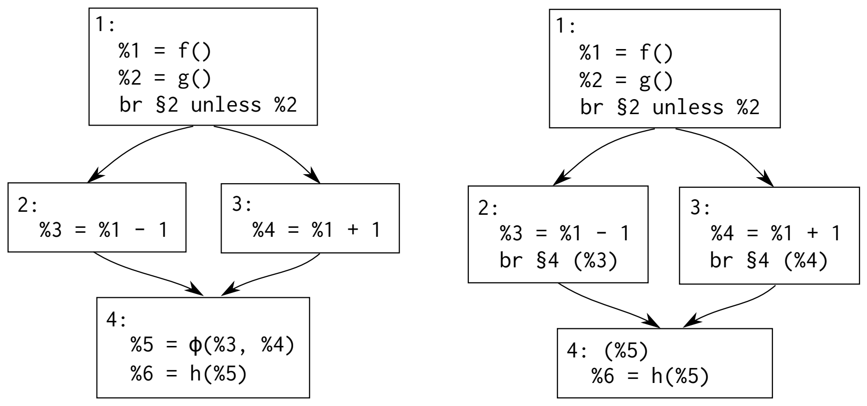
Here, the value of `h(y)` depends on two possible locations of `y` – hence, we cannot simply rename every varaible in a naive way. Instead, in the variant of SSA form used in this text and most of Julia, values of variables that are assigned in multiple parent blocks are passed on as *block arguments*, 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`).



```
1:
  %1 = f()
  %2 = g()
  br §2 unless %2

2:
  %3 = %1 - 1

3:
  %4 = %1 + 1

4:
  %5 = ϕ(%3, %4)
  %6 = h(%5)
```

```
1:
  %1 = f()
  %2 = g()
  br §2 unless %2

2:
  %3 = %1 - 1
  br §4 (%3)

3:
  %4 = %1 + 1
  br §4 (%4)

4: (%5)
  %6 = h(%5)
```

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 `:(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 onegenerated 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 transformation can be performed *recursively*: by inserting the same generated function to the inner function calls in step 3. Since the transformation operates not during parsing, the function to be transformed

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

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 AD, or concolic execution), 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 Computation Graphs and Automatic Differentiation

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, 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 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).

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

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

- Finite differences
- Symbolic differentiation
- Automatic differentiation

In finite differences, the idea is to discretize the definition of derivatives, and numerically evaluate the function within an environment. This is simple to implement, increases in complexity with the dimension of the involved spaced and can become numerically unstable in various ways. 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.

Automatic differentiation, the third category, is perhaps unfortunately named. The idea is to not start from functions as black-box or symbolic objects, but from programs, represented throught expression graphs. Then the perturbation that makes up the value of the derivative at a point is propagated through the graph.

To understand how that works, let us 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.

Instead of starting with a limit of tangents, more insight is provided by viewing it as a best-approximating linear operator. One of the most generaly useful definitions is provided through the *Fréchet derivative* (Bronstein, Semendjajew 1995), essentially a generalization of the *total differential*. Let X and Y be normed spaces. A function $f : U \subseteq X \longrightarrow Y$ is Fréchet differentiable at a point $x \in U$ if there exists a bounded linear operator $A : X \longrightarrow Y$ such that

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

or in other words,

$$f(x+h) = f(x) + A(h) + o(\|h\|). \tag{2.15}$$

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

The important fact here is that $\mathrm{D}f(x)$ is still a function: a linear function approximating how f reacts to a perturbation, Δ, around x. This fact allows one to propagate differential values through composed functions, by the chain rule:

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

In the one-dimensional case, we simply have

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

so

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

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 $\mathrm{D}\phi(\psi(x))$.

If $f : U \subseteq \mathbb{R}^N \longrightarrow \mathbb{R}$, as is the case for loss functions of parametric models, the partial derivatives can be recovered individually by applying the Fréchet derivative to basis vectors e^i:

$$\mathrm{D}f(x)(e^i) = \partial_i f(x) \tag{2.19}$$

This corresponds to the fact that the partial derivatives are the sensitivities to perturbation in coordinate directions – a special case of directional derivatives (which can be recovered the same way by application of the differential to any vector with unit norm.)

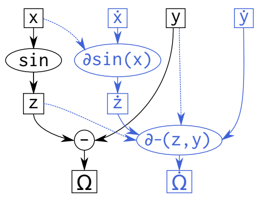
Consider the specific case of $f(x, y) = sin(x) - y$. Writing $g = (x, y) \mapsto x - y$ instead of the infix subtraction operator (with a derivative of $\mathrm{D}g(x)(\Delta_1, \Delta_2) = \Delta_1 - \Delta_2$), we have:

$$\begin{aligned} \mathrm{D}f(x, y) &= \mathrm{D}(g \circ (\sin \otimes \mathrm{id}))(x, y) \\ &= \mathrm{D}g\,((\sin \otimes \mathrm{id})(x, y)) \circ \mathrm{D}(\sin \otimes \mathrm{id})(x, y)) \\ &= ((\Delta_1, \Delta_2) \mapsto \Delta_1 - \Delta_2) \circ (\mathrm{D}\sin(x) \otimes \mathrm{D}\,\mathrm{id}(y)) \\ &= (\Delta_1, \Delta_2) \mapsto \partial_1 \sin(x)\,\Delta_1 - 1\Delta_2 \\ &= (\Delta_1, \Delta_2) \mapsto \cos(x)\,\Delta_1 - \Delta_2. \end{aligned} \tag{2.20}$$

To construct the full gradient from this, however, requires to evaluate $\mathrm{D}f(x)$ at all the N basis functions. In order to overcome this, we can rewrite the above equation as

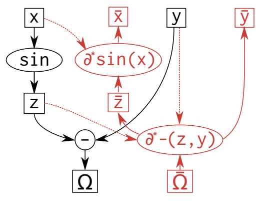
$$\begin{aligned} \partial_i f(x) &= \mathrm{D}f(x)(e^i) \\ &= \langle 1, \mathrm{D}f(x)(e^i)\rangle \\ &= \langle \mathrm{D}^* f(x)(1), e^i\rangle, \end{aligned} \tag{2.21}$$

where $\mathrm{D}^* f(x)$ is the *adjoint operator*.



```
x = ?
ẋ = Δx
y = ?
ẏ = Δy
z = sin(x)
ż = ∂sin(x)(ẋ)
  = cos(x) * ẋ
Ω = z - y
Ω̇ = ∂-(z, y)(ż, ẏ)
  = ż - ẏ
```

(a) Forward mode.



```
x = ?
x̄ = ∂*sin(x)(z̄)
  = cos(x) * z̄
y = ?
ȳ = ∂*₂-(z,y)(Ω̄) = -Ω̄
z = sin(x)
z̄ = ∂*₁-(z,y)(Ω̄) = Ω̄
Ω = z + y
Ω̄ = Δ
```

(b) Backward mode.

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

When we want to recover the gradient from the Jacobian, we can either multiply it with all the basis vectors, or simply transpose it:

$$\begin{aligned} \mathrm{D}f(x)(\mathrm{e}^i) &= \left\langle 1, [\cos(x), -1]^{\mathrm{T}} e^i \right\rangle \\ &= \left\langle [\cos(x), -1], e^i \right\rangle \end{aligned} \tag{2.22}$$

(Note that the adjoint operator inverses the order of composition)

$$\begin{aligned} \mathrm{D}^* f(x,y)(\delta) &= \mathrm{D}^*(g \circ \sin \otimes \mathrm{id})(x,y)(\delta) \\ &= (\mathrm{D}^* \sin \otimes \mathrm{D}^* \mathrm{id} \circ \mathrm{D}^* g)(x,y)(\delta) \\ &= ((\mathrm{D}^* \sin(x) \otimes \mathrm{D}^* \mathrm{id}(y)) \circ (\xi \mapsto [\xi, -\xi]))(\delta) \\ &= ((\partial \sin(x) \times \cdot) \otimes (1 \times \cdot))([\delta, -\delta]) \\ &= [\cos(x), -1] \cdot \delta \end{aligned} \tag{2.23}$$

3 Implementation of Dynamic Graph Tracking in Julia

It has been mentioned above that there is a trade-off between source-transformation methods and library-based 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. 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 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, I developed a method for a hybrid version: 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.
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

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

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

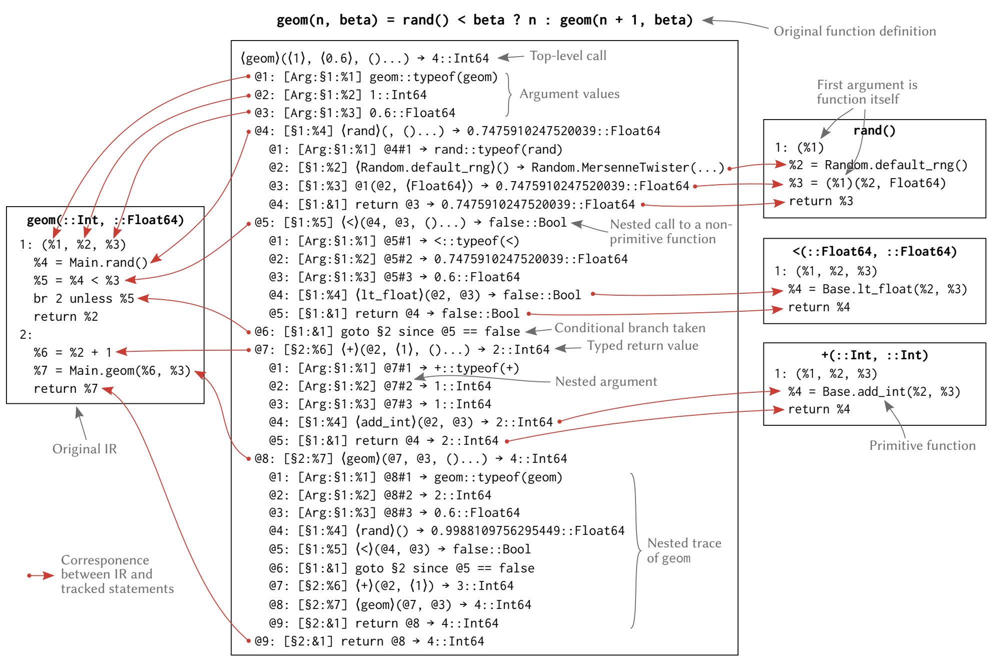
The extended Wengert list structure is implemented in Julia through nested objects of an abstract supertype `AbstractNode`, with several concrete subtypes for the different kinds of nodes. Additionally, there are special types for the tape- and block references, and an expression type `TapeExpression`, 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, recorded 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 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`



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

```
⟨geom⟩(⟨1⟩, ⟨0.6⟩, ()...) → 4::Int64
@1: [Arg:§1:%1] geom::typeof(geom)
@2: [Arg:§1:%2] 1::Int64
@3: [Arg:§1:%3] 0.6::Float64
@4: [§1:%4] ⟨rand⟩(, ()...) → 0.7475910247520039::Float64
  @1: [Arg:§1:%1] @4#1 → rand::typeof(rand)
  @2: [§1:%2] ⟨Random.default_rng⟩() → Random.MersenneTwister(...)
  @3: [§1:%3] @1(@2, ⟨Float64⟩) → 0.7475910247520039::Float64
  @4: [§1:&1] return @3 → 0.7475910247520039::Float64
@5: [§1:%5] ⟨<⟩(@4, @3, ()...) → false::Bool
  @1: [Arg:§1:%1] @5#1 → <::typeof(<)
  @2: [Arg:§1:%2] @5#2 → 0.7475910247520039::Float64
  @3: [Arg:§1:%3] @5#3 → 0.6::Float64
  @4: [§1:%4] ⟨lt_float⟩(@2, @3) → false::Bool
  @5: [§1:&1] return @4 → false::Bool
@6: [§1:&1] goto §2 since @5 == false
@7: [§2:%6] ⟨+⟩(@2, ⟨1⟩, ()...) → 2::Int64
  @1: [Arg:§1:%1] @7#1 → +::typeof(+)
  @2: [Arg:§1:%2] @7#2 → 1::Int64
  @3: [Arg:§1:%3] @7#3 → 1::Int64
  @4: [§1:%4] ⟨add_int⟩(@2, @3) → 2::Int64
  @5: [§1:&1] return @4 → 2::Int64
@8: [§2:%7] ⟨geom⟩(@7, @3, ()...) → 4::Int64
  @1: [Arg:§1:%1] @8#1 → geom::typeof(geom)
  @2: [Arg:§1:%2] @8#2 → 2::Int64
  @3: [Arg:§1:%3] @8#3 → 0.6::Float64
  @4: [§1:%4] ⟨rand⟩() → 0.9988109756295449::Float64
  @5: [§1:%5] ⟨<⟩(@4, @3) → false::Bool
  @6: [§1:&1] goto §2 since @5 == false
  @7: [§2:%6] ⟨+⟩(@2, ⟨1⟩) → 3::Int64
  @8: [§2:%7] ⟨geom⟩(@7, @3) → 4::Int64
  @9: [§2:&1] return @8 → 4::Int64
@9: [§2:&1] return @8 → 4::Int64
```

```
geom(::Int, ::Float64)
1: (%1, %2, %3)
  %4 = Main.rand()
  %5 = %4 < %3
  br 2 unless %5
  return %2
2:
  %6 = %2 + 1
  %7 = Main.geom(%6, %3)
  return %7
```

```
rand()
1: (%1)
  %2 = Random.default_rng()
  %3 = (%1)(%2, Float64)
  return %3
```

```
<(::Float64, ::Float64)
1: (%1, %2, %3)
  %4 = Base.lt_float(%2, %3)
  return %4
```

```
+(::Int, ::Int)
1: (%1, %2, %3)
  %4 = Base.add_int(%2, %3)
  return %4
```

Original function definition
Top-level call
Argument values
First argument is function itself
Nested call to a non-primitive function
Conditional branch taken
Typed return value
Nested argument
Primitive function
Nested trace of geom
Original IR
Corresponence between IR and tracked statements

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.

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 our system, however, we also record 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 record 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 above. Care needs to be taken to properly record function calls, since we need to ensure that non-primitive functions are recursively tracked.

The transformed code of the example function geom, whose IR is displayed in figure 3.1 above, is displayed in listing 3.1. First, a "graph recorder" object is set up in the extra argument %5. In this, the original IR is stored. Subsequently, every make sure the figure in on same spread as explanation

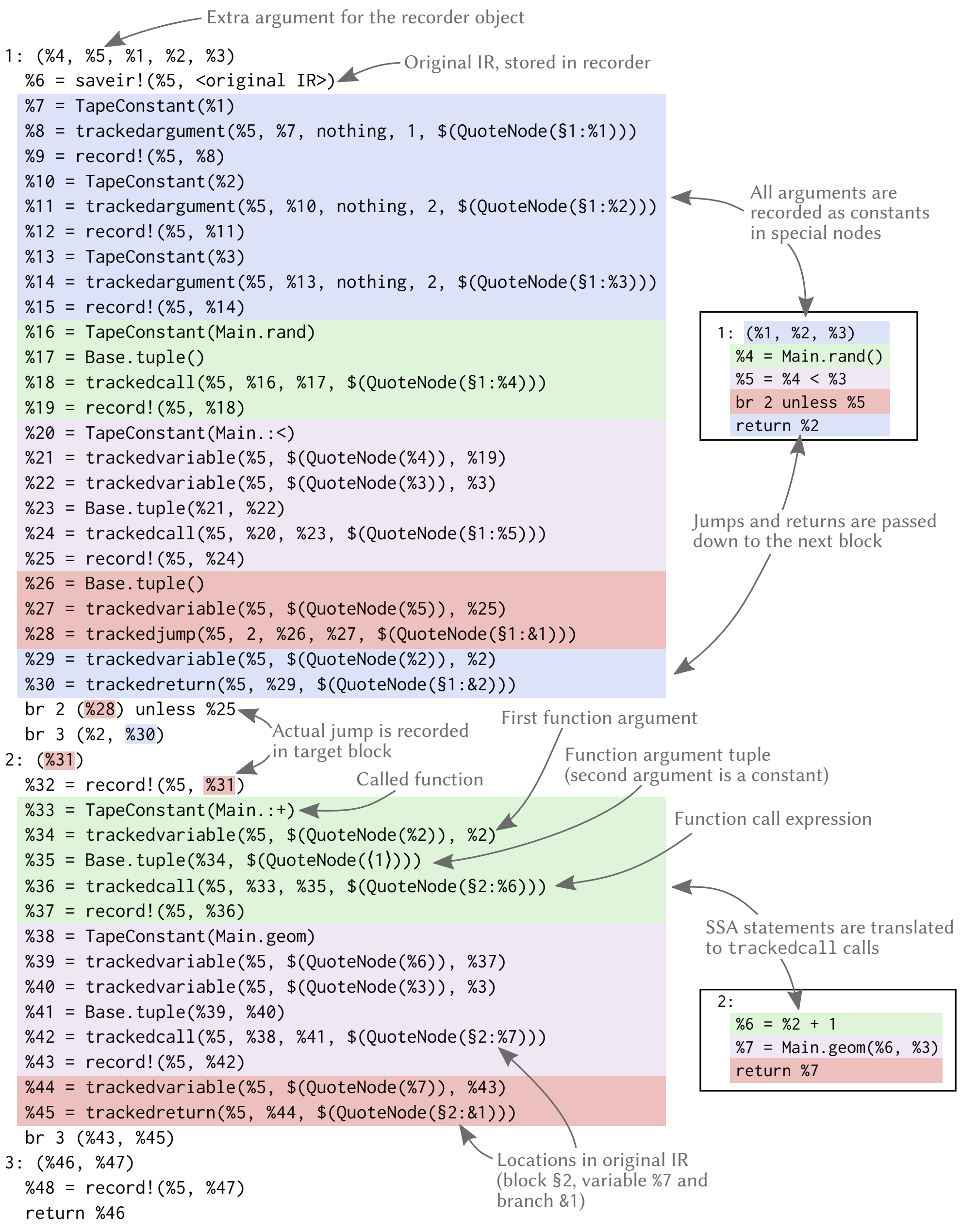
describe algorithm

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.

Branches, tracked with `trackejump` and `trackedreturn`, cannot be stored on the recorder object before they are taken, of course. The solution is to first construct the respective nodes of all possible branches of a block, and adding them as an extra argument to the branches taken. Then, in each target branch, the jump node from which the branch originated is recorded immediately. 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. This way, they can be treated in the same way as other branches.

The resulting IR consists of about three to five times as many statements as the original. The transformation, due to JIT compilation, is performed at most once 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.

Please see the appendix for a pseudo-code specification of the IR transformation in Algorithm 1, and the transformed code for the geom function in Figure 3.



```
1: (%4, %5, %1, %2, %3)
  %6 = saveir!(%5, <original IR>)
  %7 = TapeConstant(%1)
  %8 = trackedargument(%5, %7, nothing, 1, $(QuoteNode(§1:%1)))
  %9 = record!(%5, %8)
  %10 = TapeConstant(%2)
  %11 = trackedargument(%5, %10, nothing, 2, $(QuoteNode(§1:%2)))
  %12 = record!(%5, %11)
  %13 = TapeConstant(%3)
  %14 = trackedargument(%5, %13, nothing, 2, $(QuoteNode(§1:%3)))
  %15 = record!(%5, %14)
  %16 = TapeConstant(Main.rand)
  %17 = Base.tuple()
  %18 = trackedcall(%5, %16, %17, $(QuoteNode(§1:%4)))
  %19 = record!(%5, %18)
  %20 = TapeConstant(Main.:<)
  %21 = trackedvariable(%5, $(QuoteNode(%4)), %19)
  %22 = trackedvariable(%5, $(QuoteNode(%3)), %3)
  %23 = Base.tuple(%21, %22)
  %24 = trackedcall(%5, %20, %23, $(QuoteNode(§1:%5)))
  %25 = record!(%5, %24)
  %26 = Base.tuple()
  %27 = trackedvariable(%5, $(QuoteNode(%5)), %25)
  %28 = trackedjump(%5, 2, %26, %27, $(QuoteNode(§1:&1)))
  %29 = trackedvariable(%5, $(QuoteNode(%2)), %2)
  %30 = trackedreturn(%5, %29, $(QuoteNode(§1:&2)))
  br 2 (%28) unless %25
  br 3 (%2, %30)
2: (%31)
  %32 = record!(%5, %31)
  %33 = TapeConstant(Main.:+)
  %34 = trackedvariable(%5, $(QuoteNode(%2)), %2)
  %35 = Base.tuple(%34, $(QuoteNode(⟨1⟩)))
  %36 = trackedcall(%5, %33, %35, $(QuoteNode(§2:%6)))
  %37 = record!(%5, %36)
  %38 = TapeConstant(Main.geom)
  %39 = trackedvariable(%5, $(QuoteNode(%6)), %37)
  %40 = trackedvariable(%5, $(QuoteNode(%3)), %3)
  %41 = Base.tuple(%39, %40)
  %42 = trackedcall(%5, %38, %41, $(QuoteNode(§2:%7)))
  %43 = record!(%5, %42)
  %44 = trackedvariable(%5, $(QuoteNode(%7)), %43)
  %45 = trackedreturn(%5, %44, $(QuoteNode(§2:&1)))
  br 3 (%43, %45)
3: (%46, %47)
  %48 = record!(%5, %47)
  return %46
```

```
1: (%1, %2, %3)
  %4 = Main.rand()
  %5 = %4 < %3
  br 2 unless %5
  return %2
```

```
2:
  %6 = %2 + 1
  %7 = Main.geom(%6, %3)
  return %7
```

Extra argument for the recorder object
Original IR, stored in recorder
All arguments are recorded as constants in special nodes
Jumps and returns are passed down to the next block
Actual jump is recorded in target block
First function argument
Function argument tuple (second argument is a constant)
Called function
Function call expression
SSA statements are translated to trackedcall calls
Locations in original IR (block §2, variable %7 and branch &1)

Listing 3.1: Tracked IR of the method `geom(::Int, ::Float64)`. Parts corresponding to original IR are highlighted in matching colors.

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.

Missing from the description are the recording of metadata, the exact constructions of nodes, and the mechanisms to correctly rename SSA variables during the transformation and tape references at runtime.

```
function trackcall(ir)
    Initialize new_ir                                                  ▷ create empty IR object
    for old_block in blocks(ir) do
        Add an empty block new_block to new_ir
        if (arg) is the first block then
            Add variable %recorder to new_block
        end if
        for arg in arguments(old_block) do
            Add arg to new_block
        end for
        if there exist branches to old_block then
            Add argument %branch_node to new_block
            Add statement to new_block, recording %branch_node in %recorder
        end if
        for arg in arguments(old_block) do
            Add statement %node to new_block, creating a node for arg
            Add statement to new_block, recording %branch_node in %recorder
        end for
        for stmt in statements(old_block) do
            if stmt is a normal call then
                Add statement %call_node to new_block, calling trackcall on stmt
                Add statement to new_block, recording %call_node in %recorder
            else if stmt is a "special" call or constant then
                Add statement %node to new_block, creating a node for stmt
                Add statement to new_block, recording %node in %recorder
            end if
        end for
        for branch in branches(old_block) do
            if branch is a return branch then
                ▷ Substitute return by a branch to the "return block"
                Add statement %return_node to new_block, creating a return node corresponding to branch
                Add branch to new_block, targeting the return block, copying branch's argument, with %return_node as extra argument
            else
                Add statement %branch_node to new_block, creating a node for branch
                Add branch to new_block, copying branch, with %branch_node as extra argument
            end if
        end for
    end for

    ▷ Set up "return block"
    Add block return_block to new_ir
    Add arguments %return_value, %return_node to return_block
    Add statement to return_block, recording %return_node in %recorder
    Add statement %result to return_block, creating a tuple of %return_value and recorder
    Add return to return_block, returning %result
end function
```

Algorithm 3.1: IR transformation to record an extended Wengert list (simplified)

3.3 Evaluation

4 Graph Tracking in Probabilistic Models

4.1 Dependency Analysis in Dynamic Models

4.2 JAGS-Style Automatic Calculation of Gibbs Conditionals

4.3 Evaluation

5 Discussion

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

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

compare to autograd, venture, church

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

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

5.1 Future Work

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

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

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

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

[2]https://github.com/phipsgabler/probability-ir

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

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

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

List of Algorithms

Colophon

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

The document source has been written in Emacs with AUCTeX mode, using TeXworks as PDF viewer. Figures were created in Inkscape, and plotting done in R using ggplot2.