Utrecht University

Graduate School of Natural Sciences

Conditional Independence in Functional Probabilistic Programming

Computing Science MSc Thesis

Supervisor:
Dr. Matthijs Vákár

Author:
Wink van Zon

Supervisor:
Prof. Dr. Gabriele Keller

September 2021

Abstract

Probabilistic programming languages are a relatively new field within computer science. As a result, not much research has been done on automatic improvements, such as static analyses and program transformations, that could increase the efficiency or usability of a probabilistic programming language. This is especially true for functional probabilistic programming languages, which are not widely used. We therefore set out to discover such improvements for probabilistic programming in a functional setting. Specifically, we look at methods to improve probabilistic programming that use information on the conditional dependencies between the variables of a probabilistic model.
In this thesis, we present a small but practical functional probabilistic programming language embedded in *Haskell*. We build a static analysis for this language that can automatically extract the conditional dependence information from a probabilistic model. Additionally, we implement a program transformation that uses the conditional dependence information that we have obtained to automatically perform complicated rewrites of probabilistic models, which previously had to be done by the user. Designing this language, the static analysis and the program transformation has given us many new insights on (functional) probabilistic programming language design, which will also be presented in the thesis.

Contents

Acknowledgements **4**

1 Introduction **5**
1.1 Research questions . . . 6
1.2 Contributions . . . 7
1.3 Structure . . . 7

2 Background **8**
2.1 Probabilistic Programming syntax and semantics . . . 8
2.2 Inference methods . . . 9
2.2.1 Markov Chain Monte Carlo . . . 9
2.2.2 Variational Inference . . . 11
2.3 Conditional independence rules . . . 12
2.3.1 Discrete parameters . . . 14
2.3.2 Other methods . . . 14
2.4 Modelling workflow . . . 14
2.5 Conclusion . . . 15

3 Language design **16**
3.1 Higher-Order Abstract Syntax . . . 16
3.1.1 Literals . . . 17
3.1.2 Parameters . . . 17
3.1.3 Functions . . . 18
3.1.4 Control flow . . . 20
3.1.5 Arrays . . . 20
3.1.6 Monads and sampling . . . 20
3.2 First-Order Abstract Syntax . . . 21
3.2.1 Conversion . . . 23
3.3 Semantics . . . 24
3.3.1 Literals and variables . . . 25
3.3.2 Functions . . . 25
3.3.3 Control flow . . . 28
3.3.4 Arrays . . . 28
3.3.5 Parameters . . . 28
3.3.6 Monads and sampling . . . 30
3.3.7 Using the evaluated model . . . 31
3.4 Syntactic sugar . . . 31
3.4.1 Literals . . . 32
3.4.2 Functions . . . 32
3.4.3 Arrays . . . 33
3.4.4 Parameters and constraints . . . 34

3.4.5 Monads and `Model` operators . . . 35
3.5 Example models . . . 36
3.5.1 Regression models . . . 36
3.5.2 Change point model . . . 37
3.5.3 Hidden Markov models . . . 38
3.6 Conclusion . . . 39

4 Conditional dependence analysis **40**
4.1 Effect system . . . 40
4.1.1 Typing judgements . . . 41
4.2 Type inference . . . 42
4.2.1 Algorithm . . . 45
4.3 Extending the effect system for arrays . . . 49
4.3.1 Type inference . . . 50
4.4 Conclusion . . . 54

5 Using conditional dependence information **56**
5.1 Variable elimination . . . 56
5.2 Program transformation . . . 58
5.2.1 Implementation . . . 59
5.2.2 Invariants . . . 61
5.2.3 Parameters . . . 61
5.2.4 Sampling . . . 62
5.2.5 Let statements . . . 62
5.2.6 Functions . . . 63
5.2.7 Control Flow . . . 65
5.2.8 Monads . . . 66
5.2.9 Limitations . . . 67
5.3 Support for discrete parameters . . . 69
5.3.1 Examples . . . 70
5.3.2 Conclusion . . . 73
5.4 Further usages . . . 73

6 Conclusion **75**
6.1 Discussion . . . 76
6.1.1 Language design . . . 76
6.1.2 Conditional dependence analysis . . . 79
6.1.3 Program transformation . . . 79

Bibliography **80**

Acknowledgements

I would like to thank my supervisors, dr. Matthijs Vákár and prof. dr. Gabriele Keller for their helpful advice, useful feedback and our countless digital meetings. Even though the circumstances were probably not ideal – the coronavirus pandemic prevented us from ever meeting in person – you have provided fantastic guidance throughout the entire project, which I deeply appreciate. I would also like to thank my girlfriend for her constant support and for listening to my long-winded explanations of the project, as well as thanking her, my friends and my family for their genuine interest in my thesis.

Chapter 1

Introduction

Probabilistic programming is a fairly new programming paradigm in computer science. The central idea behind the paradigm is that it attempts to simplify the usage of probabilistic models. Probabilistic programming languages provide syntax that allows users to specify their probabilistic models so that the inference of these models can then be handled automatically. Decoupling the inference of probabilistic models from their specification saves users time and makes using models more accessible, as the implementation of a model no longer requires the user to implement an entire inference algorithm as well.

With the rise of probabilistic programming have come many different probabilistic programming languages. Most of these languages have different areas of focus, with each language supporting different sets of expressible models, using different inference algorithms or implementing different language features. Some languages attempt to support a large number of different sets of models [1, 2], while others limit the expression of models to use more specialist inference methods [3, 4] that can be faster or more accurate. There is even a new set of probabilistic languages that implement so-called deep probabilistic programming [5–7] where variational inference methods are used to incorporate a set of deep learning models into probabilistic programming.

What most popular probabilistic programming languages have in common, however, is that they are all imperative languages. Some widely used languages have functional aspects, as their host languages provide them with the ability to write in a functional style [6, 8], but there are few popular probabilistic programming languages that fully adhere to the functional programming paradigm. Functional probabilistic programming languages [9, 10] do exist, but tend to be unpopular compared to their imperative counterparts as they are very experimental, not fast enough for practical use and do not make full use of their functional properties: for example, the simple type system of *Hakaru* is a lot less elaborate than the extensive type systems of traditional *ML*-style functional languages such as *ML*, *OCaml* or *Haskell*. We believe that there might be potential for functional probabilistic programming languages to use their functional setting to unlock improvements that might not be possible within their imperative alternatives. To give an example: pure functional languages have referential transparency and can easily isolate side-effects resulting from probabilistic programming, which might both simplify program transformations. This thesis will focus on finding these improvements and investigating if we can develop a functional probabilistic programming language that can implement these improvements and thus provide a usable and practical edge.

There are of course many directions in which to seek such improvements, for example by developing a new inference method that is very fast, or one that supports many types of models. Our goal, however, is to use the language as the base tool for improvement. We want to design our language, together with possible static analyses and transformations, in such a way that we can extract useful information from implemented models and use this information to either improve inference with existing algorithms, or increase the separation of modelling and inference. It is clear why the former is a practical improvement, but this is also true for the latter: the goal of probabilistic programming

languages is to provide automatic inference for models defined by the user, but many languages still force the user to rewrite certain models in order to be able to infer them more quickly, or sometimes to infer them at all [11].
Although not many currently-used compilers for probabilistic programming languages implement improvements specific to probabilistic programming, such as specialised static analyses or program transformations, there has been some earlier research into these types of improvements. Specifically, this research has given promising results through automatically obtaining the conditional dependence information of implemented models [11]. This information is given by the different conditional (in)dependence relationships between the probabilistic variables in the model. In our review of the relevant literature we discuss these promising results, along with other possible use-cases that we have found for this conditional dependence information. Because of these many possibilities for improvements and the fact that we believe that the inference of conditional dependence information might benefit from a functional programming environment, our research focuses on improvements that can be made using this conditional dependence information.
In this thesis, we present a functional probabilistic programming language that is able to automatically capture this conditional dependence information from probabilistic models. To do this, we present an accompanying static analysis that is built using a custom effect system. Furthermore, we also present a use-case for the conditional dependence information within the language. We show how we have developed a program transformation that allows us to perform a useful rewrite of certain models, that normally has to be tediously done by hand before these models can be used, automatically.

1.1 Research questions

The main goal of this thesis is thus to develop a prototype functional probabilistic programming language that uses the functional programming paradigm to improve the usability of the language itself and the efficiency of the inference that the language performs on its models. Our literature review in chapter 2 will show that the knowledge of conditional independence relationships within the models can lead to some promising optimisations. This is why the focus of the thesis will be on improvements that can be derived from these conditional independence relationships. This leads to the first research question:

Research question 1. *How can a functional probabilistic programming language be designed so that it can efficiently extract conditional independence relationships between variables from implemented models?*

The next step is then to use these relationships as a way to improve the usefulness of our language. There are two quality criteria of probabilistic programming languages that we would like to put the focus on: we want to attempt to make improvements on both the modelling and the inference aspects of probabilistic programming. Implementing models should be easy for the user and inference of these models should be fast and accurate to make our language practical. This leads to the second research question:

Research question 2. *How can knowledge on the conditional dependencies between variables within a model in our functional language be used to simplify the implementation of models and improve the efficiency of the inference in this language?*

The modelling aspect can be made easier for the user through greater separation of modelling and inference within the language, for example by reducing the cases in which a user has to rewrite a model so proper inference is possible. The efficiency of inference on the other hand can be increased by using the knowledge to increase the speed of model inference, for example by rewriting models so that they can be inferred faster with certain algorithms, and using it to increase the accuracy of the inference, for example by calculating some parts of the model exactly.

1.2 Contributions

The following concrete contributions are presented within this thesis:

- A small purely functional probabilistic programming language *ProbProg*[1], deeply embedded in *Haskell*, with several basic features.
- A static analysis on this probabilistic programming language that uses an effect system to obtain the conditional dependence information of a model.
- A program transformation that uses this information to automatically marginalise discrete parameters out of certain models, allowing for easier implementations of these models in the *ProbProg* language.
- Several insights and conclusions on obtaining and using the conditional dependence information of models in (functional) probabilistic programming languages.

1.3 Structure

Our thesis is structured in the following order of chapters:

- In chapter 2, we give relevant background information on probabilistic programming and discuss the earlier work on conditional dependence analysis in probabilistic programming languages.
- In chapter 3 we describe our functional probabilistic programming language *ProbProg*, that we have designed for this thesis. We describe its syntax, semantics and show a few example implementations of probabilistic models in the language.
- In chapter 4 we introduce an effect system for the *ProbProg* language. This effect system provides us with a static analysis that is able to obtain conditional dependence information from probabilistic models written in *ProbProg*.
- Then, in chapter 5 we discuss how we can use this conditional dependence that we have obtained through this analysis in order to improve aspects of our language.
- Finally, in chapter 6 we conclude our thesis, answer our research questions using our findings and discuss ideas for further research and any other remarks on our work.

[1]The implementation of *ProbProg*, the static analysis and the program transformation can be found at https://github.com/desparito/probprog-thesis.

Chapter 2

Background

There has been a lot of previous research into different ways of improving probabilistic programming languages. This research can in turn help us determine how we should design our specific improvements. In this chapter we will look at both general research into probabilistic programming as well as specific fields of improvement that can assist us in the development of our language.

2.1 Probabilistic Programming syntax and semantics

Programs built in probabilistic programming languages represent statistical models. Users can use these programs to describe their complex models in a compact way. Staton [12] explains the idea of probabilistic programming as follows: according to Bayes' law, we have that

$$\text{Posterior} \propto \text{Likelihood} \times \text{Prior}. \tag{2.1}$$

A probabilistic program then represents the measure of the product of the likelihood and the prior. Probabilistic programming languages provide tools for the user to define a model which represents this measure. We now want to extract useful information from this model: we can use Bayesian inference on a program to find or approximate the posterior distribution; in other words, finding the constant that normalises the measure that the program represents. Generally, inference of a model can be defined as either calculating the normalised posterior distribution exactly or calculating an approximation of it, from the joint probability density that the model, a program in a probabilistic language, represents. Probabilistic programming languages come with algorithms to infer their expressible models, allowing the users to automatically obtain useful results without needing knowledge of difficult inference methods.
We can now discuss how probabilistic programming languages allow users to define these models. Staton describes that a probabilistic programming language is nothing more than a general purpose programming language with three extra constructs:

- $Draw(D(\boldsymbol{\theta}))$, which represents a sample drawn from some given prior distribution D with parameters $\boldsymbol{\theta}$. Not all probabilistic programming languages implement a *Draw* statement (or an equivalent statement), leading to what some call declarative probabilistic programming, as opposed to generative probabilistic programming which does implement it [4, 6, 7].

- *Observe* x *from* $D(\boldsymbol{\theta})$, which records the likelihood that some data point x was observed from a distribution D, again with parameters $\boldsymbol{\theta}$. Languages often store this likelihood as a real weight value for each program trace: take for example the *target* variable that represents the log-density of a program in Stan [4]. The fundamental operation behind *Observe* in most languages is a more primitive $Score(E)$ or $Factor(E)$ statement, which multiplies the likelihood (or the weight of the program trace) by some real factor E [13]: In Stan this results in *Target*

$+= E$, for example. Using *Factor*, the statement *Observe x from* $D(\boldsymbol{\theta})$ would be equivalent to *Factor*$(D(x|\boldsymbol{\theta}))$. Similar to how there are declarative probabilistic languages that lack a *Sample* statement, there are also generative probabilistic programming languages that lack the *Observe* (or *Factor/Score*) statement [1]. In Stan, the *Observe x from* $D(\boldsymbol{\theta})$ statement is written as $x \sim D(\boldsymbol{\theta})$, which is the notation that we will adhere to from now on in this chapter.

- *Normalise*(S), which takes the model described in the statement S and computes the posterior distribution. The implementation of such a statement in practice is usually some function in the host language taking a finished model as a parameter, which has several hyper-parameter settings for the inference algorithm [4, 6, 8, 14]. *Normalise*(S) is therefore not often called within a model itself, which is why many authors leave this construct out of their syntax [13]. Generally, we can then assume that a model is always wrapped in a *Normalise* statement.

The exact implementation of these statements varies heavily with each language, depending on what type of inference method is used to calculate the posterior, for example. However, research done on the semantics of these statements does allow us to draw some conclusions independent of this implementation. For example, Staton [12] introduces general denotational semantics for probabilistic programming languages and uses this to prove that probabilistic programming languages are commutative. This means that only the flow of data within a program is relevant, as opposed to the control flow of a program. Besides the insight that this can give us for designing data and control flow in our language, this also allows us to perform a large set of program transformations as the order of statements within a program is irrelevant, allowing us to order a program in any way we would like without changing its semantics.
In addition to this single example, there are many more possible transformations: programs can be transformed in order to apply general simplifications, using algebra [15] or slicing [16], but there is also a wide variety of theoretical transformations that can help improve the performance of inference algorithms. We will discuss some of those transformations as we look at their relevant inference algorithms in section 2.2. We can also look beyond program transformations and examine static analysis of probabilistic programming as a whole. Work by Bernstein [17] shows that while there is limited use of specific static analyses in the current state of probabilistic programming, there is a large potential for static analysis with many possible techniques that could be implemented. We particularly highlight that his work finds that there are many static analyses that can be specifically designed with probabilistic programming in mind, as opposed to standard static analysis techniques used on general purpose programming languages that extend to probabilistic programming languages.

2.2 Inference methods

This section will describe several inference methods that are used in the probabilistic programming field. We will give a brief description of each relevant method, as well as discuss research on the benefits and optimal use-cases of each method, with the focus on finding ways to exploit these benefits in our own prototype language.

2.2.1 Markov Chain Monte Carlo

Markov Chain Monte Carlo, or MCMC-methods are a class of inference methods based on the formation of Markov chains. The general idea is that the posterior distribution with an unknown normalisation constant can be sampled by looking at states taken from such a chain, when the chain $\{X_i\}$ is constructed so that its equilibrium is equal to the posterior distribution. This means that, under the right conditions, the chain should eventually converge so that the samples are distributed as if taken from the posterior distribution. Different MCMC methods vary in the way that the Markov chain is constructed. The rise of MCMC-methods started with the Metropolis-Hastings algorithm [18], but there are many MCMC variants today that make up a large share of the inference algorithms that are used in probabilistic programming. We will discuss the most relevant methods.

Gibbs Sampling

Gibbs Sampling [19] is a MCMC-method that is a special case of the Metropolis-Hastings algorithm. The idea of the algorithm is as follows: If we were to draw parameters $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_n)$ from our posterior distribution $p(\theta_1, \ldots, \theta_n)$, one would partition the parameters in r different "blocks", so that $\boldsymbol{\theta} = (\boldsymbol{\theta_1}, \ldots, \boldsymbol{\theta_r})$. The iteration from step t in the Markov chain, where $\boldsymbol{\theta^{(t)}} = (\boldsymbol{\theta_1^{(t)}}, \ldots, \boldsymbol{\theta_r^{(t)}})$, to step $t+1$ is then given by:

$$
\begin{aligned}
&\text{Draw } \boldsymbol{\theta_1^{(t+1)}} \text{ from } p(\boldsymbol{\theta_1^{(t)}} \mid \boldsymbol{\theta_2^{(t)}}, \ldots, \boldsymbol{\theta_r^{(t)}}) \\
&\text{Draw } \boldsymbol{\theta_2^{(t+1)}} \text{ from } p(\boldsymbol{\theta_2^{(t)}} \mid \boldsymbol{\theta_1^{(t)}}, \boldsymbol{\theta_3^{(t)}}, \ldots, \boldsymbol{\theta_r^{(t)}}) \\
&\qquad\qquad\qquad\vdots \\
&\text{Draw } \boldsymbol{\theta_r^{(t+1)}} \text{ from } p(\boldsymbol{\theta_r^{(t)}} \mid \boldsymbol{\theta_1^{(t)}}, \ldots, \boldsymbol{\theta_{r-1}^{(t)}}),
\end{aligned}
\tag{2.2}
$$

as described by Gelfand [20]. The division of blocks can be chosen as desired. Often, the blocks consist of just one parameter, so that $\boldsymbol{\theta_i} = \{\theta_i\}$ for each block i [21]. Whenever this is not the case and the blocks consist of larger sets of parameters, the Gibbs Sampler is also called a Blocked Gibbs Sampler.

There are several interesting ways to improve the quality of inference with the Gibbs Sampling algorithm. One area where improvements can be made is by choosing the decomposition of blocks in different ways. For example, Riggelsen [22] shows a way to use information on the Markov blanket of each variable in a directed Bayesian network model to divide the network into different dependence components with which one can create a clever decomposition of variables into blocks. Additionally, work by Rivasseau [23] shows how undirected graphical models can be partitioned into so-called factor trees, where each tree can be seen as a block. This partitioning is done in such a way that the dependencies between the parameters in each tree allow for more optimal Gibbs Sampling. What these two methods have in common is that they make heavy use of dependence relationships between variables in the models, albeit sometimes indirectly. Later, in section 2.3, we will examine how these relationships can be inferred from a model, which might allow us to implement these improvements or even expand them to wider sets of models.

Another useful property of Gibbs sampling is that it can be used to combine different inference algorithms. If we exploit the fact that we can draw each block using different inference algorithms, we can combine them using Gibbs Sampling. The individual blocks and inference algorithms can then be chosen in such a way that the inference algorithm chosen to draw some $\boldsymbol{\theta_i^{(t+1)}}$ is highly effective at drawing $p(\boldsymbol{\theta_i^{(t+1)}} \mid \boldsymbol{\theta_1^{(t+1)}}, \ldots, \boldsymbol{\theta_{i-1}^{(t+1)}}, \boldsymbol{\theta_{i+1}^{(t+1)}}, \ldots, \boldsymbol{\theta_r^{(t+1)}})$. Later in this section we will see that some inference algorithms have greater performance on models that have a certain structure. If we are able to identify these types of structures, put them in their separate blocks and use the most suited inference algorithms to sample these blocks, using Blocked Gibbs Sampling to combine these inference algorithms could prove to be a useful tool.

Hamiltonian Monte Carlo and NUTS

Hamiltonian or Hybrid Monte Carlo, also known as HMC, is an instance of the Metropolis-Hastings algorithm where an evolution of Hamiltonian dynamics is simulated. Random walk Metropolis-Hastings methods tend to converge slowly as they explore the parameter space using random walks that tend to be inefficient [24]. HMC tries to correct this behaviour by assigning values taken from our sampling problem to variables within a Hamiltonian system in such a way that the posterior distribution can be sampled by simulating this system [25]. The general idea of HMC is that the parameter vector $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_n)$ represents the position of a particle in a Hamiltonian system. The negative potential energy in the system is set to the logarithm of the distribution that we wish to sample from. To then generate a sample from our distribution, the particle is given a random Gaussian momentum (drawn from $\mathcal{N}(\mathbf{0}, \boldsymbol{M})$, where $\boldsymbol{M}$ is the mass matrix) and the Hamiltonian dynamics of

the system are simulated for a selected number of time steps of a chosen length: the new position of the particle is our sample. This sample then becomes the input position of the particle in the next iteration, which generates the usual chain of samples that we find in all MCMC methods.
HMC has three rather unfortunate requirements. Firstly, the Hamiltonian Dynamics are usually simulated using a numerical integrator, often the Störmer-Verlet or "leapfrog" integrator [25, 26]. A requirement for the usage of this integrator is that the gradient of the logarithm of the probability density can be calculated. Doing this automatically is generally the least efficient step, as well as the most complicated step in HMC, as finding efficient automatic differentiation techniques is still an ongoing research effort. Secondly, the usage of HMC requires that every parameter of the posterior distribution is continuous, so models inferred using HMC cannot contain any discrete parameters. Later in section 2.3, we will examine some clever ways to work around this restriction. The third and final requirement is that the user has to choose several hyper-parameters of the HMC algorithm, specifically the step size of the integrator and the number of steps taken per iteration. This usually requires the user to perform several initial runs of the algorithm, as well as interpret the results of these runs in order to find the correct hyper-parameters. If these hyper-parameters are chosen poorly, the performance of the algorithm will decrease drastically [26].
An extended version of HMC that tackles this third issue is NUTS, or the No U-Turn Sampler [26]. It provides an algorithm that is very similar to HMC and has performance that is at least as efficient, but removes the need to tune any hyper-parameters by equipping the algorithm with several heuristics that determine useful values for the step size and the amount of steps automatically. This means that NUTS provides us with an HMC algorithm that can be used without hand-tuning. This lack of a need for hand-tuning makes NUTS a good candidate for an inference algorithm in a probabilistic programming language, as it helps hide the inference from the user and therefore decouples the modelling of a program from its inference. This is why NUTS is used as an inference algorithm within many state-of-the-art probabilistic programming languages, such as Stan [4], Pyro [6] and PyMC3 [27].

2.2.2 Variational Inference

A second widely used class of inference algorithms that we will examine are the so-called variational methods. There are both benefits and downsides to using variational methods instead of the MCMC methods discussed in the previous section: generally, variational methods tend to be much more efficient than MCMC methods, but this comes at the cost of statistical accuracy as the methods could introduce bias and their approximations may lie far from the true posterior distribution [26, 28, 29]. Nevertheless, (deep) probabilistic programming languages that are built on optimisation frameworks such as TensorFlow [30] make use of variational inference as it transforms the approximation of the posterior distribution into an optimisation problem rather than a sampling problem [6, 31].
The general idea of variational methods is to approximate our posterior distribution p using some approximation q [28, 32]. If we rewrite our posterior function as $p(\boldsymbol{\theta} \mid O)$, where $\boldsymbol{\theta}$ and O are the sets of latent and observed variables respectively, we introduce a family of functions $q(\boldsymbol{\theta} \mid O, \boldsymbol{\lambda})$ with a set variational parameters $\boldsymbol{\lambda}$. Our goal is now to find the optimal function in this family by minimising the so-called Kullback-Leibler (KL) divergence, $KL(q \mid\mid p)$, with respect to the $\boldsymbol{\lambda}$ parameters:

$$\boldsymbol{\lambda}^* = \arg\min_{\boldsymbol{\lambda}} KL(p(\boldsymbol{\theta} \mid O) \mid\mid q(\boldsymbol{\theta} \mid O, \boldsymbol{\lambda})), \tag{2.3}$$

where the KL-divergence is given as

$$KL(q \mid\mid p) = \sum_{\{S\}} q(S) \ln \frac{q(S)}{p(S)}. \tag{2.4}$$

The resulting function $q(\boldsymbol{\theta}|O, \boldsymbol{\lambda}^*)$ is then returned as the approximation of the posterior. The choice for the KL-divergence is deliberate: an alternative formulation of the minimisation problem in variational inference methods is that $q(\boldsymbol{\theta}|O, \boldsymbol{\lambda}^*)$ is chosen so that it is the function with the best lower

bound on the likelihood $p(O)$. Jordan et al. [32] show, using Jensen's inequality, that the logarithm of the likelihood can be bounded as follows:

$$\begin{aligned} \ln p(O) &= \ln \sum_{\{\boldsymbol{\theta}\}} p(\boldsymbol{\theta} \mid O) \\ &= \ln \sum_{\{\boldsymbol{\theta}\}} q(\boldsymbol{\theta} \mid O) \cdot \frac{p(\boldsymbol{\theta} \mid O)}{q(\boldsymbol{\theta} \mid O)} \\ &\geq \sum_{\{\boldsymbol{\theta}\}} q(\boldsymbol{\theta} \mid O) \cdot \ln \frac{p(\boldsymbol{\theta} \mid O)}{q(\boldsymbol{\theta} \mid O)}. \end{aligned} \tag{2.5}$$

Because the difference between the two sides of this equation is equal to $KL(q \mid\mid p)$ and $KL(q \mid\mid p) \geq 0$, we find that the right hand side is a lower bound on the likelihood. Finding $\boldsymbol{\lambda^*}$ as in equation 2.3 returns the tightest possible bound.
Different variational inference methods differ in the techniques that they use to find the approximation function $q(\boldsymbol{\theta}|O, \boldsymbol{\lambda^*})$. To give an example: earlier in this section we mentioned that probabilistic programming languages that are built upon optimisation libraries such as TensorFlow [30] benefit from variational methods. The language Pyro [6] is built on an adaptation of TensorFlow. To optimise the fact that TensorFlow is able to provide fast gradient-descent, Pyro uses a variational method that finds $\boldsymbol{\lambda^*}$ through clever gradient-descent based optimisation [31].

Variational Message Passing

One variational method that is of particular interest is the Variational Message Passing (VMP) algorithm [28]. It is an algorithm that works on models that can be represented as a directed acyclic graph or a tree-shaped factor graph. This idea of VMP is to factorise the functions $q(\boldsymbol{\theta} \mid O, \boldsymbol{\lambda})$ in such a way that each factor corresponds to a single node $\boldsymbol{\theta_j}$ in the graph. A formula is then given to calculate each factor: The evaluation of this formula is only dependent on the conditional $p(\boldsymbol{\theta_j} \mid pa_j, O)$, where pa_j represents the parents of the node in the graph, together with the set of conditionals $p(\boldsymbol{\theta_i} \mid pa_i, O)$, where $i \in ch_j$, with ch_j representing the set of children of the node. Each factor is only dependent on these terms given by its parents and its children; these terms can be seen as "messages" that are passed throughout the graph. This is the origin of the name Variational Message Passing.
The VMP framework works very well in some cases: for example, the Infer.NET language [3] limits its range of expressible models to those that can be handled by the VMP algorithm – models that can be described by directed acyclic graphs where the conditional distributions at each node are members of the exponential family of distributions [28] – and can therefore provide efficient inference using VMP. This immediately shows the shortcomings of VMP as well: as most languages wish to support a wide range of different models, VMP is generally not an option for an inference method. Nevertheless, there are ways in which VMP could prove useful in our case. We could detect certain sections of models that could be inferred using VMP by first checking their compatibility, finding their directed acyclic graph and then using Gibbs Sampling as mentioned in the previous section to combine different inference methods with VMP. Another possibility might be finding a way to somehow extract dependence information from incompatible models that could allow us to either construct a way to still apply the VMP algorithm or rewrite these models so that VMP becomes a viable inference option. Both implementations would require a lot of knowledge on the dependence structure of the model, which is something we have seen before when looking at improvements for MCMC algorithms.

2.3 Conditional independence rules

In the previous section we have seen that many possible improvements of the discussed inference methods tend to rely on information about the dependencies between variables within a model,

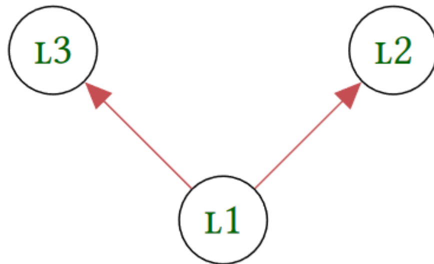

Figure 2.1: The information flow between the three levels in the lattice defined by Gorinova et al. [11].

either by using knowledge of conditional independence rules or of the Markov blanket of variables in the model. This shows that this information can be very valuable if it can be extracted from our models. In this section we will examine research on the automatic detection of such conditional independence relationships, which will both reveal additional benefits of knowing these relationships as well as show different ways to implement this automatic detection.
A very recent contribution is the research done by Gorinova et al. [11], where the conditional independencies within a program are obtained using type inference. The idea is that along with their regular type, expressions and statements in a probabilistic programming language are also typed at one of three levels. These three levels are set up as a lower semi-lattice $(\{L_1, L_2, L_3\}, \leq\}$, where $L_1 \leq L_2$ and $L_1 \leq L_2$. The goal is to have well-typed programs induce relations of the form $l_2 \perp\!\!\!\perp l_3 \mid l_1$, where each l_i is of level Li. Notice that the inequalities of the lattice correspond to the dependencies between levels, as seen in figure 2.1. The typing rules for the levels of expressions and statements are then chosen in such a way that they respect the flow of information in the lattice and therefore must adhere to the conditional independence rules. For example, typing the level of an observe statement $y_1 \sim D(y_2, y_3, \ldots, y_n)$ would give a level $\sqcup_{i=1}^{n} l_i$, where l_i is the level of y_i. In other words: the level of this statement is the least upper bound of the levels of its contained expressions. This upper bound cannot exist if there is no adherence to the aforementioned information flow. An example of this is that the upper bound does not exist if this statement contains both an L_2 and L_3 level variable, as there is no level l so that $L2 \leq l$ and $L3 \leq l$, which means that such an assignment of levels would never be inferred. This is to be expected, as that assignment would signal a dependency between these variables that contradicts the independence rules that we are trying to find.
These typing rules can then be used to type a full program. Using type inference, the typing rules will induce some set of possible level assignments to the variables within the program. Using these combinations and the fact that the type system finds relations of the form $l_2 \perp\!\!\!\perp l_3 \mid l_1$, we can infer the conditional independencies. This does reveal one of the downsides of this method: as all variables must be typed at some level, it is only possible to find conditional independence rules that contain every variable within the program. Ideally, one would like to find all possible conditional independence rules instead. While not every one of these conditional independence relationships is needed for most applications, detecting more of them does increase the number of applications that can be implemented. Not only do we want to detect all the conditional independence relationships because they could be of use in the improvements on inference methods that we have seen earlier, but also because it could prevent us from storing redundant information in large models. As an example, take a model with n parameters; if we can express a relationship as $x_1 \perp\!\!\!\perp x_2 \mid x_3$, we would rather have this compact representation instead of the relationship $x_1 \perp\!\!\!\perp x_2 \mid x_3, \ldots, x_n$.

2.3.1 Discrete parameters

The main practical result of this paper is another useful application of the usage of knowledge on dependence relationships. As mentioned in section 2.2, the HMC and NUTS inference methods can only be used if all variables in a model are continuous. The Stan language [4] therefore has no support for discrete parameters, but the user can work around this limitation by marginalising these discrete parameters out of their model by hand in the code for their models. Gorinova et al. mention that this is quite a breach of the idea of probabilistic programming that states that the user should be able to implement models without having to take the inference into account. Instead, they argue that it should be possible to marginalise these discrete parameters out of a program automatically.
They do this by creating a program transformation that performs the marginalisation of discrete parameters. If one were to implement such a transformation naively, it would not be very efficient: marginalising each discrete parameter out of every other parameter within a model would result in a transformation that has a time-complexity that is exponential in the amount of discrete parameters in that model. Instead, they use the knowledge on conditional independencies that they have obtained in order to perform the marginalisation so that only the variables that the discrete parameter directly depends on or the variables that directly depend on the discrete parameter are taken into account. This removes any unnecessary computation time spent by conditioning on irrelevant continuous variables in the automatic marginalisation, as each elimination of a discrete parameter only involves the minimal number of required variables.

2.3.2 Other methods

This research was directly focused on automatically finding the dependencies in a model, but there is also research in which different methods are given that extract this information indirectly. Bernstein et al. [33] provide static analyses on probabilistic programming languages that can obtain the factor graph of models automatically, as well as transform it into a directed acyclic graph. Additionally, Saad et al. [34] define a probabilistic programming language from which a sum-product representation of a model can easily be extracted. All of these structures induce the dependence relationships within a model and might be a good way to store these relationships. In practice, it is important to think about the ideal way to store dependence information as each method has advantages and disadvantages with regards to storage size, attainability and how well it suits the usage of the independence information. Transformations between different storage methods is possible, but may be quite inefficient [33].

2.4 Modelling workflow

Making accurate statistical models requires more than just the ability to construct models and infer them. Instead, performing good modelling is an iterative process where users constantly wish to check the quality of their assumptions, their models and their inference results [35]. This is why finding improvements in performing these checks can be as useful as finding improvements in model construction or inference. Some examples of such checks are:

Prior predictive checks, which examine the effect of the chosen priors by taking simulated draws from a model and checking if these simulations are in line with what can actually be observed from the data [36]. This helps users check if their prior assumptions within a model are correct, even before the model has been inferred.

Posterior predictive checks, which instead check the quality of the posterior distribution. The draws in a simulation are now taken from the posterior distribution that results from inference. One can then perform several checks that compare the results of the simulation with actual data, which allows the user to assess the quality of the posterior distribution [35].

Simulation-based calibration, which is a check that can help validate the quality of the inference algorithm on a particular model. For several iterations, the user does the following [35]: they

draw the parameters for a model from the prior distribution and simulate data conditional on the drawn parameter values. The model is then fitted on this data using the inference algorithm, producing a posterior distribution. The user can then check this posterior distribution against the original simulated parameter values: if a model is inferred using parameters drawn from the prior distribution, the posterior distribution should match this prior distribution.

Because these checks are necessary for a good modelling workflow, providing tools in a probabilistic programming language that makes the checks easier for a user to implement greatly increases the usability of the language. As seen in the examples, the checks rely on the ability to simulate data from the posterior or from the prior distributions. This might require the user to rewrite their models: the Stan user guide provides a section in which the user is shown how to rewrite their model to generate draws from the prior using random number generators [37] for example, which is known as the forward or ancestral sampling mode of a model. A possible improvement that would streamline the modelling workflow for the users of a language would be to provide methods to perform such program transformations automatically.

2.5 Conclusion

Next to providing relevant background context for our subject, the literature review done in this chapter shows the relevance of our research questions introduced in section 1.1. Answering these questions will show if the functional paradigm can provide some practical benefits over classical imperative probabilistic programming, which is something that has not been investigated with the approach of finding conditional independencies yet. In addition to this, not many analyses to obtain the conditional dependence information of models and improvements that could be implemented using this information have been implemented in practice. This means that our answers to the research questions could extend beyond the realm of functional probabilistic programming, as it might result in insights that can help infer and use these conditional dependence relationships within any probabilistic programming language. As we saw, many probabilistic programming languages solely implement traditional static analyses and compiler optimisations known from general purpose programming languages. This means that there is also a significant chance that any improvements that we might find could be used to improve the existing state-of-the-art probabilistic programming frameworks.

Chapter 3

Language design

In this chapter we will introduce the *ProbProg* domain specific language and describe its syntax and semantics. *ProbProg* is a purely functional probabilistic programming language that is deeply embedded in *Haskell*. The language largely resembles a traditional higher-order functional language, but there are a few differences which ensure that probabilistic programming in *ProbProg* is possible. The most important feature that is added for probabilistic programming is a special monad that captures the side-effects that occur when building a probabilistic model, such as introducing latent and observed parameters and maintaining or changing the target density of a model. However, there are other additions, such as support for arrays, built-in probability density functions and special syntactic sugar to streamline the construction of models.
In addition to explaining the syntax and semantics of the *ProgProg* language, we will also show how the correct samples for a model in this language can be inferred, as well as how we can present the sampling results to the user. We then conclude this chapter by presenting the implementation of several commonly used probabilistic models in the *ProbProg* language, in order to show how the language can be used in practice.

3.1 Higher-Order Abstract Syntax

Our *ProbProg* language is a deeply embedded domain specific language. We therefore use the syntax of our host language, *Haskell*, to build expressions within our language. However, as opposed to the expressions of a shallow embedding, the expressions in our language do not use the semantics of the host language. Instead, the expressions of our language construct an abstract syntax tree (AST). This abstract syntax tree can then be compiled or interpreted at a later stage using our own defined semantics.
We first present the higher-order abstract syntax for our language. The higher-order abstract syntax is the most human-readable abstract syntax and is therefore the syntax that is used internally to form the concrete syntax of the *ProbProg* language. We will later see that this higher-order representation, while very useful for human readers, is not well-suited for the internal representation of the abstract syntax. We will elaborate on this in section 3.2, where we convert the higher-order representation to an internal first-order representation.
This higher-order abstract syntax is implemented in *Haskell* using generalised algebraic datatypes (GADTs). GADTs allow us to make great use of the type checker provided by *Haskell*: it checks if our syntax conforms to type constraints that would not be possible to enforce using a regular ADT, for example, so that our programs are more likely to be correct. The GADT implementation also provides us with a way to convert our higher-order abstract syntax to a first-order abstract syntax as we will see in section 3.2.
Listing 3.1 shows how expressions are constructed in this higher-order abstract syntax tree GADT. In the rest of this section we will explain the different constructors of the higher-order abstract syntax

GADT for the *ProbProg* langauge.

```haskell
data Expr a where
    Parameter :: Expr String -> Maybe Constraint -> Expr (ModelResult Double)
    -- The literal below is used as a proxy argument to determine the input type.
    Input      :: Typeable a => Proxy (Literal a) -> Expr (ModelResult a)
    Con        :: Literal a -> Expr a
    Nil        :: Expr ()
    Pair       :: Expr a -> Expr b -> Expr (a, b)
    CArray     :: Vector n (Expr a) -> Array n a
    Generate   :: KnownNat n => (Finite n -> Expr a) -> Array n a
    PDF        :: PDF a -> Expr (a -> Double)
    If         :: Expr Bool -> Expr a -> Expr a -> Expr a
    Let        :: Typeable a => Expr a -> (Expr a -> Expr b) -> Expr b
    Func       :: (Typeable a, Typeable b) => (Expr a -> Expr b) -> Expr (a -> b)
    PrimFunc   :: Op a b -> Expr (a -> b)
    Apply      :: Expr (a -> b) -> Expr a -> Expr b
    Sample     :: Expr a -> Expr (a -> Double) -> Expr Model
    Return     :: Expr a -> Expr (ModelResult a)
    Bind       :: Expr (ModelResult a) -> Expr (a -> ModelResult b)
                                       -> Expr (ModelResult b)
    -- This tag is used for conversion to a 1st order AST,
    -- it represents environment size at defining occurrence.
    Tag        :: Typeable t => Int -> Expr t
```

Listing 3.1: The higher-order abstract syntax GADT for Expressions in *ProbProg*.

3.1.1 Literals

Constants can be introduced into programs using the `Con` constructor. This constructor requires an argument of type `Literal`: the syntax for these arguments is given in listing 3.2. The GADT `Literal a` takes a constant expression of type `a` from the host language *Haskell* and lifts it to our language. These literals are limited to the types `Int`, `Double`, `Bool` and `String`. As we are defining a prototype language, we only want to support these essential *Haskell* literal types.
In addition to `Con`, there is one more constructor that defines a constant: the `Nil` constructor, which defines the unit expression.

```haskell
data Literal a where
    LInt    :: Int     -> Literal Int
    LReal   :: Double  -> Literal Double
    LBool   :: Bool    -> Literal Bool
    LString :: String  -> Literal String
```

Listing 3.2: The abstract syntax GADT for literals in *ProbProg*.

3.1.2 Parameters

Stochastic parameters can be introduced into a model via the `Parameter` constructor, which defines the latent parameters within a model and the `Input` constructor, which defines the observed parameters.
The latent parameters require a unique string identifier, so that a user can name them. This allows us to relay the identifiers back to the user when executing a model, so that when we generate samples of the latent parameters, the user can still easily identify them. This identifier is explicitly a string within our language and not just a *Haskell* string, as it is necessary in some situations to generate parameter identifiers within the language: for example, when the user defines an array of latent parameters with identifier s using our sugared syntax, we can automatically generate unique identifiers of the form $s[1], \dots, s[n]$ for the individual parameters.
In addition to the identifier, the `Parameter` constructor requires an optional `Constraint`. The contents of such a constraint are specified in listing 3.3. A constraint represents a transformation of a

latent parameter from a so-called constrained representation to an unconstrained representation. To use sampling methods such as HMC or NUTS, a model in our language should represent a function $f : \mathbb{R}^k \to \mathbb{R}$, where k is the number of latent parameters. This means that the input of this function can take on any value in $\mathbb{R}^k$. However, in practice, users want to further constrain this domain of parameters: examples of such constraints are parameters with a lower or an upper bound or arrays of parameters that should represent a simplex.
To facilitate such constraints, we use a method similar to what the Stan probabilistic programming language employs [37]. Our constraints contain a transformation $t : S \to \mathbb{R}$ for a single parameter from the constrained parameter space to $\mathbb{R}$ and its inverse transformation $t^{-1} : \mathbb{R} \to S$. These constraints are predefined for commonly used limitations on parameters, such as simplex arrays or upper/lower bounds. Within a model, this latent parameter X is then replaced by $t^{-1}(X)$, so that the unconstrained sample X is transformed to the constrained parameter space. Because such changes of variables affect the target density of the model, the constraint also stores the specific factor that needs to be added to the density for its transformation. This density factor is also a function, as it depends on the value of the latent parameter. Section 3.3 explains how these transformations are handled in the semantics of our language.

```
1 data Constraint = Constraint {
2                         transform     :: Expr (Double -> Double),
3                         invert        :: Expr (Double -> Double),
4                         densityFactor :: Expr (Double -> Double)
5                   }
```

Listing 3.3: The higher-order abstract syntax for constraints in *ProbProg*.

The `Input` constructor also requires an extra argument. A proxy type witness argument containing a literal is given to explicitly encode the type of the observed parameter. This is needed when evaluating these parameters, as explained in section 3.3.5. The user will never notice this argument in practice, as our syntactic sugaring will often fill in the correct proxy literal. We see this in detail in section 3.4.
The expressions which are constructed using the `Parameter` and `Input` constructors are wrapped in the `ModelResult` monad. We explain the purpose of this monad in section 3.1.6, where we discuss the monadic aspects of the syntax.

3.1.3 Functions

Several constructors in the higher-order abstract syntax tree can be used to construct function expressions. These functions are represented in the canonical higher-order fashion: there are no variable identifiers encoded in the expressions, whereas the first-order representation has explicit variables. The `Tag` constructor is used for the conversion to this first-order representation and it is never directly accessed by the user, as will be explained further in section 3.2.
Functions can be constructed from host-language functions in *Haskell* which take and return expressions, using the `Func` constructor. The constraint that the inner types of this constructor must be of the `Typeable` type class will later be used to convert the higher-order syntax to a first-order syntax. The same holds for the `Typeable` constraints on other constructors: these are generally a result of this conversion requirement. As all literal types are members of the `Typeable` type class, enforcing this constraint is not an issue.
In addition to this, there are some standard primitive operators built into the *ProbProg* language that can be converted to functions using the `PrimFunc` constructor. These primitive operators of type `Op a b`, where a represents the operand and b represents the resulting type, are shown in listing 3.4. Many of these operators are frequently used standard operators that exist in any (functional) programming language.
Operators that take multiple operands store these operands in an uncurried representation. This means that they store them as a single operand in a (nested) pair or tuple expression. These types of expressions can be constructed using the `Pair` constructor and deconstructed using the `Fst` and

`Snd` primitive operators. This representation makes for a more simple evaluation of the primitive operators, which is shown in section 3.3.2. Section 3.4.2 shows that our sugared syntax hides this uncurried representation from the user, allowing them to use similar curried functions.

```
data Op a b where
    -- Pair operators
    Fst          :: Op (a, b) a
    Snd          :: Op (a, b) b
    -- Real and Int operators
    Int2Real     :: Op Int Double
    Negate       :: Num a       => Op a a
    Abs          :: Num a       => Op a a
    Signum       :: Num a       => Op a a
    Add          :: Num a       => Op (a, a) a
    Sub          :: Num a       => Op (a, a) a
    Mul          :: Num a       => Op (a, a) a
    Div          :: Floating a => Op (a, a) a
    Log          :: Floating a => Op a a
    Exp          :: Floating a => Op a a
    -- Bool operators
    Not          :: Op Bool Bool
    And          :: Op (Bool, Bool) Bool
    Eq           :: Eq a        => Op (a, a) Bool
    Leq          :: Ord a       => Op (a, a) Bool
    -- String operators
    SAppend      :: Op (String, String) String
    Show         :: Show a => Op a String
    -- Array operators
    Head         :: Op (Vector (1 + n) a) a
    Tail         :: Op (Vector (1 + n) a) (Vector n a)
    Reverse      :: Op (Vector n a) (Vector n a)
    Map          :: Op (a -> b, Vector n a) (Vector n b)
    Foldr        :: Op (a -> b -> b, (b, Vector n a)) b
    ZipWith      :: Op (a -> b -> c, (Vector n a, Vector n b)) (Vector n c)
    Backpermute :: Op (Vector n a, Vector m Int) (Vector m a)
    Sequence     :: Monad m => Op (Vector n (m a)) (m (Vector n a))
    Replicate    :: KnownNat n => Op a (Vector n a)
    UnfoldrNM    :: (KnownNat n, Monad m) => Op (b -> m (a, b), b) (m (Vector n a))
```

Listing 3.4: The abstract syntax GADT for primitive operators in *ProbProg*.

There is one other constructor that can also be used to define function expressions in our language: the `PDF` constructor. This constructor is a natural result of the fact that we are building a probabilistic programming language. Just as how the primitive operators can be constructed using the `PrimFunc` constructor, the `PDF` constructor allows our language to incorporate several predefined probability density functions within its models. The syntax for these predefined probability density functions is shown in listing 3.5. Notice that the constructors in this listing take the parameters for their respective probability distributions and return a `PDF a`, where the type a represents the type of the domain of the resulting probability density function. This is also made clear in the `PDF` constructor, where this `PDF a` produces that function, which then has type `Expr (a → Double)`.

```
data PDF a where
    Exponential :: Expr Double -> PDF Double
    Normal       :: Expr Double -> Expr Double -> PDF Double
    Poisson      :: Expr Double -> PDF Int
    Dirichlet    :: Array (2 + n) Double -> PDF (Vector (2 + n) Double)
    UniformD     :: Array (1 + n) a -> PDF a -- Discrete uniform distribution
    Categorical :: Array (1 + n) Double -> PDF Int
```

Listing 3.5: The higher-order abstract syntax GADT for probability density functions (PDFs) in *ProbProg*.

Finally, the `Apply` constructor encodes the application of a function on its operand. The `Let` constructor works similarly: it binds its first argument to the operand of the second argument.

3.1.4 Control flow

There is one single constructor that handles control-flow in our language: the `If` constructor, which represents an if-then-else expression. The first argument of this constructor stores the conditional expression and the second and third argument store the then and else branches, respectively.

3.1.5 Arrays

Support for array expressions is very important for probabilistic programming languages, as many widely-used probabilistic models use large sets of parameters structured within arrays. Arrays in our higher-order syntax are represented by the following type alias:

```
type Array n a = Expr (Vector n a)
```

Arrays in our language are therefore simply expressions that contain a vector of other expressions. Array literals are constructed using the `CArray` constructor, which takes a vector of expressions as its argument. We use vectors that explicitly store their size in a type-level natural from the existing **vector-sized** *Haskell* package as our internal vector representation [38]. This means that our arrays are also size-tagged.

Alternatively, arrays can be constructed via `Generate`, which mimics the existing `generate` *Haskell* function for sized vectors. This function produces a vector with the use of a generative function that maps expressions of type `Finite n` to expressions that can fill the vector. This `Finite n` represents an expression inhibited by exactly n values, where n is a type-level natural index [39]. This index allows `generate` to produce a vector with the type-level size n. The `Generate` constructor is then adapted for our array representation, as it takes a function that generates expressions in our syntax as its argument. The `CArray` and `Generate` constructors are not the only ways to build arrays: listing 3.4 shows that there are a few primitive operators that can also construct array expressions. Notice that array constructors or operators might require that their type-level size is known at compile time: this is enforced by the `KnownNat` type class.

Listing 3.4 also shows several primitive operators that work on array operands (and therefore have one or more vector types within their operand types). These operators will allow us to perform simple array operations that are commonplace in functional languages such as mappings and folds, for example. Listing 3.5 shows us that arrays can also be parameters of probability density functions, or elements of their domain.

3.1.6 Monads and sampling

Finally, `Return` and `Bind` provide us with the two most common procedures that define monads in functional languages: `Return` wraps expressions within our language into a monad and `Bind` composites of functions that return monadic values. They are analogous to the `return` and (>>=) operations in *Haskell*, respectively.

The only monad that can be used within our language is the `ModelResult` monad. As we have seen in section 3.1.2, this monad is already wrapped around the result of the `Input` and `Parameter` constructors. As we will later see in section 3.3, we use this monad to store the side-effects introduced by interpreting programs in our language as probabilistic models. It is therefore wrapped around constructors that introduce latent or observed parameters, as the input of these parameters is a side-effect.

Finally, `Sample` represents the *Observe* sampling operation, as explained in section 2.1, that is the core of probabilistic programming languages. It takes a member of the domain of a probability density function x, together with that probability density function f and then multiplies the target density of the model with $f(x)$. The result of the `Sample` constructor is an expression of the type `Model`. This type is given by the following type alias:

```
type Model = ModelResult ()
```

This resulting empty expression obtained from the `Sample` constructor is again wrapped in the `ModelResult` monad. This is the case because `Sample` is also a constructor that clearly introduces side-effects: it changes the target density that our model should keep track of and the `ModelResult` also tracks this density. We will again see this when discussing the semantics of the language in section 3.3. Because the `ModelResult` tracks the target density of a model, the type alias for an empty expression wrapped in the `ModelResult` monad has the name `Model`: for our probabilistic models, we are solely interested in the resulting density that is tracked in this monad and not in any other values obtained throughout the execution of the program, hence why the contents of the monad are empty.

3.2 First-Order Abstract Syntax

Having a higher-order representation of our abstract syntax is useful for human readers. However, we need to perform static analysis as well as program transformations on our language. These operations are easier to execute on a first-order abstract syntax representation. We therefore also define a first-order abstract syntax (FOAS) for our language, together with a conversion from the higher-order abstract syntax to this internal first-order representation. In this section we give an explanation of this first-order abstract syntax by highlighting the differences from the higher-order representation and show what conversion algorithm we use to transform our programs to the first-order syntax.
The first-order representation that we use is a nameless representation with typed DeBruijn indices, in the style of Altenkirch and Reus [40] and the *Accelerate* language [41], another deeply embedded domain specific language implemented in *Haskell*. McDonell et al. convert their higher-order syntax for *Accelerate* into a first-order representation in order to perform program transformations more easily, which is what we want to accomplish for our language as well. We therefore use a similar first-order representation, as well as the same conversion from the higher-order representation.
As in *Accelarate*, we introduce a value environment that stores values for variables as a heterogeneous list, where the types of the variables are stored within nested pairs in an environment `env`. This is done as follows:

```
1 data Val env where
2   Empty :: Val ()
3   Push  :: Val env -> t -> Val (env, t)
```

The two constructors are straightforward: we can introduce an empty environment, or push a variable onto an existing environment. Next, we introduce the nameless DeBruijn indices from the *Accelerate* environment:

```
1 data Idx env t where
2   ZeroIdx ::                Idx (env, t) t
3   SuccIdx :: Idx env t -> Idx (env, s) t
```

At the type level, each DeBruijn index distinguishes some type `t` from an environment `env`. At the value level, this same index stores the position at which this type is stored in the environment `env`. The two constructors for our DeBruijn indices are used to specify this position of the type; the type can either be on top of the environment stack or it can be one level deeper than some other DeBruijn index.
Using a nameless DeBruijn index, we can now retrieve the variable which the index refers to from our environment:

```
1 prj :: Idx env t -> Val env -> t
2 prj ZeroIdx       (Push _   v) = v
3 prj (SuccIdx idx) (Push val _) = prj idx val
```

We call this the projection of that index onto the environment. Using this environment, we can now define the terms in our first-order abstract syntax. This new abstract syntax data type is shown in listing 3.6. We introduce the concept of an open expression: an expression that contains free variables. Such an expression of type `OpenExpr env a` is then not only identified by the type `a` of its

contents, but also by the types of its free variables stored in its typing environment `env`. A closed expression is then an expression with no free variables, so that

```haskell
type Expr = OpenExpr ()
```

```haskell
data OpenExpr env a where
    Var       :: Idx env a    -> OpenExpr env a
    Parameter :: OpenExpr env String -> Maybe (Constraint env)
              -> OpenExpr env (ModelResult Double)
    --The literal below is used as a proxy argument to determine the input type.
    Input     :: Typeable a => Proxy (Literal a) -> OpenExpr env (ModelResult a)
    Con       :: Literal a -> OpenExpr env a
    Nil       :: OpenExpr env ()
    Pair      :: OpenExpr env a -> OpenExpr env b -> OpenExpr env (a, b)
    CArray    :: Vector n (OpenExpr env a) -> Array env n a
    Generate  :: KnownNat n => (Finite n -> OpenExpr env a) -> Array env n a
    PDF       :: PDF env a -> OpenExpr env (a -> Double)
    If        :: OpenExpr env Bool -> OpenExpr env a -> OpenExpr env a
              -> OpenExpr env a
    Let       :: OpenExpr env a -> OpenExpr (env, a) b -> OpenExpr env b
    Func      :: OpenExpr (env, a) b -> OpenExpr env (a -> b)
    PrimFunc  :: Op a b -> OpenExpr env (a -> b)
    Apply     :: OpenExpr env (a -> b) -> OpenExpr env a -> OpenExpr env b
    Sample    :: OpenExpr env a -> OpenExpr env (a -> Double)
              -> OpenExpr env Model
    Return    :: OpenExpr env a -> OpenExpr env (ModelResult a)
    Bind      :: OpenExpr env (ModelResult a) -> OpenExpr env (a -> ModelResult b)
              -> OpenExpr env (ModelResult b)
```

Listing 3.6: The first-order abstract syntax tree GADT for Expressions in *ProbProg*.

We now also need to define new first-order syntax for constraints (given in listing 3.7) and for probability density functions (given in listing 3.8), as these constructs might both contain free variables. The syntax for literals and primitive operators stays the same as in listings 3.2 and 3.4, as these cannot contain free variables.

```haskell
data Constraint = Constraint {
                      transform     :: OpenExpr env (Double -> Double),
                      invert        :: OpenExpr env (Double -> Double),
                      densityFactor :: OpenExpr env (Double -> Double)
                  }
```

Listing 3.7: The first-order abstract syntax for constraints in *ProbProg*.

```haskell
data PDF env a where
    Exponential :: OpenExpr env Double -> PDF env Double
    Normal      :: OpenExpr env Double -> OpenExpr env Double -> PDF env Double
    Poisson     :: OpenExpr env Double -> PDF env Int
    Dirichlet   :: Array env (2 + n) Double -> PDF env (Vector (2 + n) Double)
    UniformD    :: Array env (1 + n) a -> PDF env a -- Discrete uniform distribution
    Categorical :: Array env (1 + n) Double -> PDF env Int
```

Listing 3.8: The first-order abstract syntax GADT for probability density functions (PDFs) in *ProbProg*.

There are several changes when we compare our new syntax to the higher-order syntax. First of all, we now have explicit variables, stored using the `Var` constructor. This is of course what classifies our new syntax as a first-order syntax. These variables are identified by their DeBruijn indices.
As a result of this, functions defined by the `Func` constructor are now simply constructed from open expressions, where the type of the top free variable is matched to the type of the function argument. The same holds for the new `Let` constructor, where the first expression is now bound to the top free variable in the environment of the body of the let expression.
The rest of the syntax is almost identical to our higher-order syntax, with the exception that most expressions are now possibly open expressions, as they could contain free variables. Defining them

explicitly as such is useful, because just like in the *Accelerate* language, the definition of our typed DeBruijn indices and our environment ensures that the types of the expressions in the environment are checked at compile-time.

3.2.1 Conversion

Now that we have defined our first-order syntax, we convert our higher-order syntax to this new representation using a slight variation of the algorithm devised by McDonell et al. [41] for *Accelerate*. In addition to converting the syntax representation, their algorithm also takes sharing recovery into account. We do not implement this sharing recovery into our algorithm, but mention that our algorithm could therefore easily be extended to incorporate this.
For the conversion, we introduce so-called layouts of environments to hold both the environment as well as the DeBruijn index corresponding to each element of the environment:

```
data Layout env env' where
  EmptyLayout :: Layout env ()
  PushLayout  :: Typeable t
              => Layout env env' -> Idx env t -> Layout env (env', t)
```

When we then add a new entry to this layout, we have to add the next DeBruijn index and also increment all the DeBruijn indices that are already inside of the layout:

```
inc :: Layout env env' -> Layout (env, t) env'
inc EmptyLayout          = EmptyLayout
inc (PushLayout lyt ix) = PushLayout (inc lyt) (SuccIdx ix)
```

We can then project the correct DeBruijn index out of the layout, given the integer representation of that index.

```
prjIdx :: Typeable t => Int -> Layout env env' -> Idx env t
prjIdx _ EmptyLayout         = error -- We cannot get anything from an empty layout
prjIdx 0 (PushLayout _ idx) = fromJust (gcast idx)
prjIdx n (PushLayout l _)   = prjIdx (n - 1) l
```

Notice that this is where the `Typeable` type class constraint that we have on many types within our abstract syntax trees comes into play: it is required to cast the index to the correct type. We can now use these layouts to convert a higher-order term, which is always closed, to a closed first-order term:

```
convert :: HOAST.Expr a -> AST.OpenExpr () a
convert = cvt EmptyLayout
```

The exact conversion algorithm `cvt` is given in listing 3.9, with the conversion of constraints given in listing 3.10 and the conversion of probability density functions given in listing 3.11. Terms from the higher-order abstract syntax tree are marked with the HOAST prefix, while terms from the first-order abstract syntax tree are marked with the AST prefix.

```
cvt :: Layout env env -> HOAST.Expr a -> AST.OpenExpr env a
cvt lyt (HOAST.Tag sz)                  = AST.Var (prjIdx (size lyt - sz - 1) lyt)
cvt lyt (HOAST.Parameter p (Just c)) = AST.Parameter (cvt lyt p)
                                                     (Just (cvtConstr lyt c))
cvt lyt (HOAST.Parameter p Nothing)  = AST.Parameter (cvt lyt p) Nothing
cvt _   (HOAST.Input i)              = AST.Input i
cvt lyt (HOAST.Con v)                = AST.Con v
cvt _    HOAST.Nil                   = AST.Nil
cvt lyt (HOAST.Pair e1 e2)           = AST.Pair (cvt lyt e1) (cvt lyt e2)
cvt lyt (HOAST.CArray a)             = AST.CArray (V.map (cvt lyt) a)
cvt lyt (HOAST.Generate f)           = AST.Generate ((cvt lyt) . f)
cvt lyt (HOAST.PDF p)                = AST.PDF (cvtPDF lyt p)
cvt lyt (HOAST.If c e1 e2)           = AST.If (cvt lyt c) (cvt lyt e1) (cvt lyt e2)
cvt lyt (HOAST.Let e1 e2)            = AST.Let (cvt lyt e1) (cvt lyt' (e2 tag))
    where
    tag  = HOAST.Tag (size lyt)
```

```haskell
    lyt' = inc lyt `PushLayout` ZeroIdx
cvt lyt (HOAST.PrimFunc f)            = AST.PrimFunc f
cvt lyt (HOAST.Func f)                = AST.Func (cvt lyt' (f tag))
    where
    tag  = HOAST.Tag (size lyt)
    lyt' = inc lyt `PushLayout` ZeroIdx
cvt lyt (HOAST.Apply f a)             = AST.Apply (cvt lyt f) (cvt lyt a)
cvt lyt (HOAST.Sample x f)            = AST.Sample (cvt lyt x) (cvt lyt f)
cvt lyt (HOAST.Bind a f)              = AST.Bind (cvt lyt a) (cvt lyt f)
cvt lyt (HOAST.Return a)              = AST.Return (cvt lyt a)
```

Listing 3.9: Conversion of higher-order expressions to first-order expressions.

```haskell
cvtConstr :: Layout env env -> HOAST.Constraint -> AST.Constraint env
cvtConstr lyt c = AST.Constraint { AST.transform     = cvt lyt (HOAST.transform c)
                                 , AST.invert        = cvt lyt (HOAST.invert c)
                                 , AST.densityFactor = cvt lyt (HOAST.densityFactor c)
                                 }
```

Listing 3.10: Conversion of higher-order constraint terms to first-order constraint terms.

```haskell
cvtPDF :: Layout env env -> HOAST.PDF a -> AST.PDF env a
cvtPDF lyt (HOAST.Exponential λ) = AST.Exponential (cvt lyt λ)
cvtPDF lyt (HOAST.Normal μ σ     = AST.Normal (cvt lyt μ) (cvt lyt σ)
cvtPDF lyt (HOAST.Poisson λ)     = AST.Poisson (cvt lyt λ)
cvtPDF lyt (HOAST.Dirichlet α)   = AST.Dirichlet (cvt lyt α)
cvtPDF lyt (HOAST.UniformD a)    = AST.UniformD (cvt lyt a)
cvtPDF lyt (HOAST.Categorical p) = AST.Categorical (cvt lyt p)
```

Listing 3.11: Conversion of higher-order probability density function terms to first-order probability density function terms.

We will look at the most interesting conversions: these are the conversions where the first-order representation varies most from the higher-order representation. As we have seen from the two syntax representations, this is the case for the `Let` and `Func` constructors. We can see that when we convert these constructors, we add a new index to our layout representing the newly introduced variable and we apply the body of the expression to the `Tag` constructor. As we mentioned in section 3.1, this constructor is used purely for this conversion. In this `Tag` constructor, we store the size of the environment at that point. This `Tag` expression then ends up at the location of the newly introduced variable in the body of the `Func` or `Let` expression.
When we encounter this `Tag` constructor later in the conversion, we can obtain the correct variable from the layout: we take the current size of the layout and subtract the size of the layout at the point that the tag was placed, which is the number stored in the `Tag` constructor. We then subtract 1 as the indexing is zero-based and obtain the correct number for the index for the variable. We can then simply project the DeBruijn index for this variable out of our layout by using `prjIdx` and with that index we can form our first-order variable expression using the `Var` constructor.
The rest of the conversion is fairly straightforward: the terms in the abstract syntax trees are not dissimilar, so we simply have to apply the conversion recursively throughout the other constructors. This concludes the conversion from the higher-order to the first-order representation, which means that we can now use our first-order syntax for any further internal analysis and transformations.

3.3 Semantics

Now that we have established the syntax of our *ProbProg* language, we specify the semantics of the language in this section by showing how models in this embedded language are evaluated in our host language *Haskell*.
The goal is to translate a model that has been written in *ProbProg* to some function $f(x_1, \ldots, x_n)$, that maps the latent parameters $x_1, \ldots, x_n$ to a real number. This way, we can use MCMC sampling

methods such as NUTS in order to obtain values for these latent parameters. We will start by explaining how we translate a model in our language to this representation.
Remember that models in our language are expressions of type `ModelResult ()`. As mentioned in section 3.1.6, this `ModelResult` monad is a *Haskell* monad that tracks three side-effects: the latent parameters, the observed parameters and the target density of a model. This means that if we can obtain the *Haskell* expression for a model in our language, we can also obtain the target density from this `MonadResult` monad. We therefore define an evaluation function of type

```
1    eval :: Val env -> M.Map Int Double -> M.Map Int ModelInput -> OpenExpr env a -> a
```

that translates expressions in our language to *Haskell* expressions of the same type. It takes an additional three arguments: the variable environment as specified in section 3.2, a map of integers to latent parameters and a map of integers to observed parameters.
We can now create the desired function that maps latent parameters $x_1, \ldots, x_n$ to the target density by calling this evaluation function on our model. As arguments we pass in the empty environment (as a model is a closed expression), a map consisting of the pairs $(i-1, x_i)$ of the latent parameters and their zero-based indices, a map consisting of the pairs $(i-1, y_i)$ of our observed parameters and their zero-based indices and the model expression. This model expression is then converted to the `ModelResult ()` *Haskell* type, from which we can obtain the target density. This gives us a function mapping the latent parameters to this target density.
Now that the goal of this evaluation function is clear, we can look at how this evaluation function translates different *ProbProg* expressions to *Haskell*, why the maps of latent and observed parameters are formed in this exact way and what data the `ModelResult ()` monad contains exactly. We will explain this by showing how the evaluation function is applied to all the constructors given in the first-order abstract syntax tree.

3.3.1 Literals and variables

We start by showing how the evaluation function translates a literal expression to a *Haskell* literal:

```
1 eval _   _  _  (Con l)      = evalLit l
2 eval _   _  _   Nil         = ()
3 eval env ps is (Pair e1 e2) = (eval env ps is e1, eval env ps is e2)
4
5 evalLit :: Literal a -> a
6 evalLit (LReal r)    = r
7 evalLit (LInt  i)    = i
8 evalLit (LBool b)    = b
9 evalLit (LString s)  = s
```

Listing 3.12: Evaluation of literal and `Pair` *ProbProg* expressions.

This evaluation is straightforward: `Nil` simply translates to the unit type in *Haskell*. Other literals have their respective *Haskell* literals stored in their constructors, from which we can obtain them again in the evaluation. Finally, we simply evaluate pairs of expressions by evaluating both individual expressions and placing these in a *Haskell* tuple.
Another straightforward evaluation is that of variables:

```
1 eval env ps is (Var ix) = prj ix env
```

Listing 3.13: Evaluation of `Var` *ProbProg* expressions.

We simply use the `prj` function as defined in section 3.2 to project the correct variable from the environment using the DeBruijn index stored in the `Var` constructor.

3.3.2 Functions

Next, we show how functions and function application are evaluated in our language:

```haskell
eval env ps is (Func f)     = \x -> eval (push x env) ps is f
eval env ps is (PrimFunc f) = evalOp f
eval env ps is (Apply f e)  = eval env ps is f (eval env ps is e)
```

Listing 3.14: Evaluation of `Func`, `PrimFunc` and `Apply` *ProbProg* expressions.

As we can see, evaluating a `Func` expression is done by creating a *Haskell* function that maps its input to the evaluation of the body of the `Func` expression, to which the input of the *Haskell* function is added by pushing it onto the environment.
Primitive functions created by `PrimFunc` can simply be evaluated by evaluating their internal primitive operator. The evaluations of the different primitive operators are shown in listing 3.15. Most of these primitive operators evaluate to their (uncurried) *Haskell* equivalent, with the exception of some array operators: operators that have the V prefix are taken from the sized vector library [38] that has been mentioned earlier in section 3.1. The `unfoldrNM` operator is the only operator that is neither standard in *Haskell* or the sized vector library. This operator mimics the behaviour of the `unfoldrM` monadic unfolding operation that is defined within *Haskell*, but it has been adapted so that it is compatible with the sized vectors that we use.
The evaluation of function application, contained in `Apply` expressions, is then very simple: both the function and its operand are evaluated and the resulting *Haskell* function is applied to the resulting *Haskell* operand.

```haskell
evalOp :: Op a b -> (a -> b)
-- Pair operators
evalOp Fst         = fst
evalOp Snd         = snd
-- Real and Int operators
evalOp Int2Real    = int2Double
evalOp Negate      = negate
evalOp Abs         = abs
evalOp Signum      = signum
evalOp Add         = uncurry (+)
evalOp Sub         = uncurry (-)
evalOp Mul         = uncurry (*)
evalOp Div         = uncurry (/)
evalOp Log         = log
evalOp Exp         = exp
-- Boolean operators
evalOp Not         = not
evalOp And         = uncurry (&&)
evalOp Eq          = uncurry (==)
evalOp Leq         = uncurry (<=)
-- String operators
evalOp SAppend     = uncurry (++)
evalOp Show        = show
-- Array operators
evalOp Head        = V.head
evalOp Tail        = V.tail
evalOp Reverse     = V.reverse
evalOp Map         = uncurry V.map
evalOp Foldr       = \(f,(b, a)) -> V.foldr f b a
evalOp ZipWith     = \(f,(a, b)) -> V.zipWith f a b
evalOp Backpermute = uncurry V.backpermute
evalOp Sequence    = V.sequence
evalOp Replicate   = V.replicate
evalOp UnfoldrNM   = uncurry unfoldrNM
```

Listing 3.15: Evaluation of primitive operator *ProbProg* expressions.

The evaluation of `Let` expressions combines techniques seen in the evaluation of `Func` and `Apply` expressions:

```haskell
eval env ps is (Let e1 e2) = let v1   = eval env ps is e1
                                 env' = push v1 env
```

```
                              in  eval env' ps is e2
```

Listing 3.16: Evaluation of `Let` *ProbProg* expressions.

First, we evaluate the first operand of the `Let` constructor. We push it on top of the current environment and use this new environment to evaluate the body of the `Let` expression.
Finally, we have our probability density functions stored in the `PDF` expressions:

```
eval env ps is (PDF p) = evalPDF env ps is p
```

Listing 3.17: Evaluation of `PDF` *ProbProg* expressions.

Their evaluation amounts to the evaluation of their internal `PDF` datatype. The evaluation of these probability density functions is given in listing 3.18. This listing shows that evaluating the different probability density functions starts with evaluating all the individual parameters. We then check if these parameters conform to the correct domain as given by that specific distribution and throw an error if this is not the case. Finally, we return the logarithm of the corresponding probability density function with the evaluated parameters.
The logarithm is returned instead of just the normal probability density function as we also store the logarithm of the target density of a model (which we will see in section 3.3.5). This allows us to directly add logarithms of probability density functions to the target density. We use the *Haskell* statistics package to obtain most (logarithmic) probability density functions [42]. An exception is the logarithmic probability density function for the Dirichlet distribution, which we have implemented ourselves.
Notice that the discrete probability functions are evaluated, but discrete distributions do not have probability density functions. These probability functions therefore only work if the value that they are applied to is guaranteed to be one of a few discrete values. In chapter 2, we have seen that using discrete latent parameters in models that are sampled using MCMC sampling methods is not directly possible. In section 5.2.1, we will show how using discrete latent parameters can still be made possible in our language.

```
evalPDF env ps is (Exponential λ) = if λ' > 0
                                       then logDensity (exponential λ')
                                       else error
    where λ' = eval env ps is λ
evalPDF env ps is (Normal μ σ) = if \sigma' > 0
                                    then logDensity (normalDistr μ' σ')
                                    else error
    where μ' = eval env ps is μ
          σ' = eval env ps is σ
evalPDF env ps is (Poisson λ) = \i -> if
                                  | i < 0      -> error
                                  | λ' <= 0    -> error
                                  | otherwise -> logProbability (poisson λ') i
    where λ' = eval env ps is λ
evalPDF env ps is (Dirichlet α) = \v -> if
                                    | V.sum α' /= 1    -> error
                                    | V.any (<= 0) α' -> error
                                    | V.sum v /= 1     -> error
                                    | V.any (<= 0) v   -> error
                                    | otherwise -> β + V.sum (V.zipWith (\a x ->
                                                     log (x ** (a - 1.0))) α' v)
    where α' = eval env ps is α
          β  = logGamma (V.sum α') - V.sum (V.map logGamma α')
evalPDF env ps is (UniformD a) = \x -> if V.elem x a'
                                          then 1 / fromIntegral (length a')
                                          else error
    where a' = eval env ps is a
evalPDF env ps is (Categorical p) | V.sum p' /= 1    = error
                                  | V.any (<= 0) p' = error
                                  | otherwise        = \x -> if length p' <= x
                                                               then error
```

```
32                                                             else V.unsafeIndex p' x
33      -- We can use unsafeIndex here, as we check if the int x is within bounds.
34      where p' = eval env ps is p
```

Listing 3.18: Evaluation of probability density function *ProbProg* expressions.

3.3.3 Control flow

The `If` constructor is evaluated as follows:

```
1 eval env ps is (If c e1 e2) = if eval env ps is c
2                                then eval env ps is e1
3                                else eval env ps is e2
```

Listing 3.19: Evaluation of `If` *ProbProg* expressions.

We simply evaluate the Boolean conditional expression. If it evaluates to `True`, the then branch is evaluated and returned and otherwise the else branch is evaluated and returned instead.

3.3.4 Arrays

There are two constructors, `Generate` and `CArray`, associated with arrays in our language. These are evaluated as follows:

```
1 eval env ps is (CArray a)       = V.map (eval env ps is) a
2 eval env ps is (Generate f)     = V.generate ((eval env ps is) . f)
```

Listing 3.20: Evaluation of `CArray` and `Generate` *ProbProg* expressions.

As before, the V prefix is used to indicate that a function is imported from the sized vector library [38]. The evaluation of the `CArray` expression simply evaluates each expression in the internal vector stored in the constructor. The evaluation of the `Generate` expression is slightly more complicated: we create a function that evaluates the resulting expressions created by the generator function stored in the constructor and this function is then used as a generator function for the resulting vector, using the `generate` function imported from the sized vector library.
As with other primitive operators, the evaluation of the different operators that work on arrays is shown in listing 3.15. This evaluation often simply amounts to using the (uncurried) equivalent operator from the sized vector library, with the exception of our custom function `unfoldrNM`, as explained in section 3.3.2.

3.3.5 Parameters

To understand how the `Input` and the `Parameter` constructors are evaluated, we need to know how the `ModelResult` monad is defined, as these constructors will evaluate to a *Haskell* expression that is wrapped in this monad. The `ModelResult` monad is defined as

```
1 type ModelResult a = State ModelState a
2
3 data ModelState = ModelState { target     :: Double,
4                                parameters :: [ModelParameter],
5                                inputs     :: Int }
```

As we can see, it is simply a state monad that keeps track of the three aforementioned side-effects: introducing either latent or observed parameters and modifying the target density. The target density is stored in the `target` record. It is simply stored as a double. This double represents the logarithm of the target density. Benefits of storing the logarithm of the target density are that MCMC sampling methods commonly use the logarithm of the density in their algorithms and that logarithms of factors can be added to the logarithmic target density, instead of multiplying the target density with them. This last benefit is also the reason that we return the logarithm of probability density functions when

evaluating PDF expressions. Because most theoretical approaches to the target density work with the regular density and not the logarithmic density, we will not explicitly mention this logarithm in later chapters of this thesis, but we do note that our concrete implementation always uses the logarithmic representation of the density and added factors.
Information on the observed parameters is stored in an integer under the inputs record. This integer keeps track of the number of observed parameters that have been added into a model up to that point. This means that if we want to introduce a new observed parameter, we can obtain a unique identifier for that parameter by taking the current value of the inputs record. We will see that this will be useful in the evaluation of the Input constructor.
Information on the latent parameters is stored in the parameters record. This record contains a list of ModelParameter data, which is defined as follows:

```
type ModelParameter = (Id, Maybe (Double -> Double))
```

It stores a parameter identifier for each parameter. We will see that this identifier stores the string name that the user has provided for the parameter. Storing these names in an evaluated model allows us to relay the sampled parameters back to the user together with their unique names. Next to this, it is able to store an optional function. As mentioned in section 3.1.2, we often apply transformations on our parameters in the form of constraints. After a model has been evaluated and the parameter values have been sampled, we need to transform our sampled parameter values back to their original constrained space in order to present the sampled values back to the user in a comprehensible way. That is why evaluated models can store these extra transformations for each parameter in this optional function.
We can now show the evaluation of Input expressions.

```
eval env ps is (Input i) = do s <- get
                              modify addInputs
                              let idx = inputs s
                              in  return $ case M.lookup idx is of
                                             Just x  -> case x of
                                                          (IReal r) -> checkType r idx
                                                          (IInt  i) -> checkType i idx
                                             Nothing -> error
where
checkType x idx = case cast x of
                    Nothing -> error
                    Just x' -> x'
```

Listing 3.21: Evaluation of Input *ProbProg* expressions.

We explain each step of this evaluation in order: first, we get the current value of the state in the ModelResult monad. Next, we use addInputs. This function increments the inputs record of the ModelState, indicating that we want to add a new input to the model. We then retrieve this incremented value from the ModelState and use it as an identifier for our observed parameter. Remember that our eval function takes a mapping from integers to ModelInput values as an argument (denoted as is in the listing). We use our identifier to obtain the correct ModelInput from this mapping. If this ModelInput is not in the mapping, we throw an error, as this indicates that the amount of values for the observed parameters that the user has given does not match with the amount of observed parameters defined in this model. The ModelInput datatype is defined as

```
data ModelInput where
      IInt  :: Int    -> ModelInput
      IReal :: Double -> ModelInput
```

This allows us to store observed parameters of different types in the same datatype and thus in the same mapping that is given as an argument to the evaluation function. The final part of the evaluation of Input expressions shows how we use the checkType function to obtain the correct concrete value from this ModelInput datatype. Naturally, we throw an error if the user has given values for the observed parameters that do not match the types that the user has specified for the observed parameters in the model.

We split the evaluation of `Parameter` expressions into two different cases depending on if the `Parameter` constructor contains a constraint that has been added onto the parameter or not. We start by examining the case where the parameter does not have any constraints.

```
eval env ps is (Parameter p Nothing)
= do s <- get
     modify (addParameter (eval env ps is p) Nothing)
     return $ case M.lookup (length (parameters s)) ps of
                Just x   -> x
                Nothing -> error
```

Listing 3.22: Evaluation of `Parameter` `ProbProg` expressions for unconstrained parameters.

As with the evaluation of `Input` expressions, we start by obtaining the current value of the `ModelState`. Next, we modify the state using the `addParameter` function. This function creates a `ModelParameter` using a string and an optional transformation function and adds it to the list in the `parameters` record of the `ModelState`. (It also checks if the parameter name is not already in use and throws an error when a duplicate name is given.) As the parameter is unconstrained, we do not add a transformation function. Finally, we return the result of a lookup in the mapping of integers to latent parameters that is provided to the evaluation function, denoted as `ps` in this listing. Similar to how we indexed the mapping of integers to `ModelInput`, we use the amount of latent parameters as our indexing in the `ps` mapping. If we find the correct parameter value we can simply return it, otherwise we throw an error.
The evalaution of `Parameter` *ProbProg* expressions that do have a constraint is slightly more complicated.

```
eval env ps is (Parameter p (Just c)) =
    do s <- get
       modify (addParameter (eval env ps is p) (Just (eval env ps is (invert c))))
       case M.lookup (length (parameters s)) ps of
        Just x  -> let df = eval env ps is (densityFactor c) x
                   in  do modify (addTarget df)
                          return $ eval env ps is (invert c) x
        Nothing -> error
```

Listing 3.23: Evaluation of `Parameter` expressions for constrained parameters.

We start again by obtaining the current `ModelState`. Next, we also update the `parameters` record using the `addParameter` function. This time we do not only add the string identifier to the `ModelParameter`, but we also add the evaluated inverse of the constraint transformation. Remember that this is the transformation that allows us to transform the resulting samples of our parameter back to the constrained space before we show them to the user. We then again look up the correct parameter value from the provided mapping `ps`. We have to perform two extra steps that did not occur when evaluating the unconstrained parameter: first, we have to add the evaluated density factor of the constraint to the target value stored in the `ModelState`. We do this by using the `addTarget` function: this function simply adds its argument to the target density in the `ModelState`. We can use addition instead of multiplication as we store the logarithm of the density factor in the constraints. Finally, we return the parameter, but we apply the inverted transformation on the parameter before returning it. This is because the mapping `ps` given to the evaluation function contains the unconstrained values for parameters. By applying the inverted transformation we make sure that the parameter that is used within the model is constrained.

3.3.6 Monads and sampling

The `Return` and `Bind` expressions are evaluated in the following way:

```
eval env ps is (Return a) = return (eval env ps is a)
eval env ps is (Bind a f) = eval env ps is a >>= eval env ps is f
```

Listing 3.24: Evaluation of `Return` and `Bind` *ProbProg* expressions.

Both constructors are simply evaluated to their *Haskell* equivalent that can then be applied to the evaluated arguments. This is possible because the `ModelResult` monad within our *ProbProg* language is also a *Haskell* monad.
The final constructor in the abstract syntax for which we have to provide an evaluation is the `Sample` constructor. As with the `Parameter` and `Input` constructors, the `Sample` constructor returns an expression wrapped in the `ModelResult` monad. This is because the `Sample` constructor changes the target density of a model. The evaluation of `Sample` expressions is done as follows:

```
1 eval env ps is (Sample x f) = do score (eval env ps is f (eval env ps is x))
2
3 score :: Double -> Model
4 score s = modify (addTarget s)
```

Listing 3.25: Evaluation of `Sample` `ProbProg` expressions.

We first evaluate both arguments of the `Sample` constructor and apply the evaluated function to the evaluated argument in order to obtain the factor that needs to be added to the target density. We use the function `score` to handle this addition: this function modifies the `ModelState` by adding its argument to the `target` record. The evaluation returns this modification of the target density.

3.3.7 Using the evaluated model

Now that we can evaluate a *ProbProg* expression to the equivalent *Haskell* expression, we describe how we fully infer a model and return the results to the user. The user provides us with a model m :: `Expr Model` and a set of n inputs $i_1, \ldots, i_n$ of type `ModelInput`; these inputs need to be given in the order of appearance in the model. As mentioned in the beginning of this section, we create a mapping `is` from integers to `ModelInput` so that `is` = `Map` $[(i_k, k) \mid 0 \leq k \leq n]$.
We can then create a function f that takes latent parameters $x_1, \ldots, x_m$. This function then returns `getTarget eval is ps m`, where `ps` = `Map` $[(x_k, k) \mid 0 \leq k \leq m]$. In other words, we now have a function f that takes the values for latent parameters as input and returns the resulting target density of model m. This function f can then be sampled using a MCMC sampling method such as NUTS.
This sampling process returns a set of N_s draws of the form $s_1, \ldots, s_m$, where each s_i is a value for latent parameter x_i. The number of samples N_s can be chosen by the user. We relay the results of these samples to the user by retrieving the `ModelParameter` data for each parameter from the `ModelState` of the evaluated model (where the values for x_i chosen for the evaluation do not matter when retrieving this data). This gives us m tuples containing a user-defined name id_i for each parameter x_i and an optional transformation τ_i that projects the ith parameter back to its unconstrained space if needed. We can then relay the sampling results by pairing each parameter value s_i with the name for that parameter and transforming the value s_i using τ_i if it exists. The draws that we return to the user are therefore sets of pairs $(id_1, \tau'_1(s_1)), \ldots, (id_m, \tau'_m(s_m))$, where $\tau'_i = \tau_i$ if τ_i exists and the identity function otherwise.
This concludes the section on the semantics of our language and how we can use these semantics in practice: we are now able to obtain samples from a model and relay the results of the sampling back to the user in a comprehensible format.

3.4 Syntactic sugar

Now that we are able to infer models that are constructed in the abstract syntax tree of the *ProbProg* language, we would like to build such probabilistic models. However, even building very simple probabilistic models becomes difficult when they can only be constructed using the different elements of the higher-order syntax. Models built in this way are both hard for a user to read and also to construct in the first place. Next to this, we will later see that we also would like to restrict the usage

some of the elements of the abstract syntax for the users of our language. (We can see that we would like to limit the use of the `Generate` constructor in section 4.3.1, for example.)
This is why we introduce a large set of *Haskell* functions, also known as smart constructors, that sugar the syntax of the *ProbProg* language. These functions build expressions using the constructors in the higher-order syntax, but provide a simpler way for the user to interact with the syntax of our language. We see in section 3.5 that this makes models significantly easier to read. In this section we will go over the most important syntax sugar functions and explain how these functions work.

3.4.1 Literals

Literals are currently introduced by providing a `Literal` type to the `Con` constructor. We simplify this process by introducing functions such as

```
1 real :: Double -> Expr Double
2 real f = Con (LReal f)
```

This function makes it easier to instantly create a *ProbProg* real number from a *Haskell* double. We define similar functions for different literal types, such as `string`, `int` and `bool`.

3.4.2 Functions

Using functions using just the higher-order syntax is not very user friendly. If we want to add two integer expressions a and b, for example, we would have to write `Apply (PrimFunc Add) (Pair a b)`, which can quickly become unreadable after adding a few more integers. To make this more intuitive, we implement the *Haskell* `Num` type class for integer expressions:

```
1 instance Num (Expr Int) where
2     a + b          = Apply (PrimFunc Add) (Pair a b)
3     a - b          = Apply (PrimFunc Sub) (Pair a b)
4     a * b          = Apply (PrimFunc Mul) (Pair a b)
5     negate         = Apply (PrimFunc Negate)
6     abs            = Apply (PrimFunc Abs)
7     signum         = Apply (PrimFunc Signum)
8     fromInteger i = Con (LInt (fromInteger i))
```

This allows us to use primitive integer operators within our language more easily. Using the same method, we implement functions from the `Num`, `Fractional` and `Floating` type classes for real expressions. This allows us to use operators such as `log`, `(/)` and `exp`. With this simpler notation for applying primitive operators, we can then define more complicated operators in our sugared syntax as well. For example, a logit function:

```
1 logit :: Expr Double -> Expr Double
2 logit x = log (x / (1 - x))
```

We also sugar the application of other primitive operators for different types of expressions such as string or Boolean expressions. For strings, we define:

```
1 (+++) :: Expr String -> Expr String -> Expr String
2 a +++ b = Apply (PrimFunc SAppend) (Pair a b)
3
4 shows :: Show a => Expr a -> Expr String
5 shows = Apply (PrimFunc Show)
```

Notice that these mimic the *Haskell* `(++)` and `show` functions for strings, respectively, but are named differently to avoid name clashing. We also implement the `IsString` GHC type class:

```
1 import GHC.Exts
2
3 instance IsString (Expr String) where
4     fromString = string
```

This type class, along with the *OverloadedStrings* extension for GHC, allows a user to construct string literals in the *ProbProg* language using the default notation for strings in *Haskell*. An example:

```haskell
s :: Expr String
s = "This is a string."
```

For booleans, we also sugar our primitive functions:

```haskell
not :: Expr Bool -> Expr Bool
not = Apply (PrimFunc Not)

and :: Expr Bool -> Expr Bool -> Expr Bool
and a b = Apply (PrimFunc And) (Pair a b)

(|==|) :: Eq a => Expr a -> Expr a -> Expr Bool
a |==| b = Apply (PrimFunc Eq) (Pair a b)

(|<=|) :: Ord a => Expr a -> Expr a -> Expr Bool
a |<=| b = Apply (PrimFunc Leq) (Pair a b)
```

We again use names for these sugared functions that do not clash with the *Haskell* prelude. In addition, we use the newly defined primitive functions to define more complicated Boolean relationships. We give two examples of this:

```haskell
or :: Expr Bool -> Expr Bool -> Expr Bool
or a b = not (not a `and` not b)

(|>=|) :: Ord a => Expr a -> Expr a -> Expr Bool
a |>=| b = (a |>| b) `or` (a |==| b)
```

Finally, we sugar the `Fst` and `Snd` primitive operators in the exact same way, producing the `fst` and `snd` functions. Of course, we also sugar the primitive operators for arrays and monads. We will discuss this sugaring in detail in sections 3.4.3 and 3.4.5.
Next to sugaring the primitive operators, we also want to make it easier to define probability density functions. If we only use the higher-order syntax constructors, creating a function expression from a probability density function requires us to define a `PDF` type and then pass it along to the `PDF` constructor. Similar to how we have sugared the construction of literals, we also sugar the construction of probability density functions. As an example, we give the `exponential` function:

```haskell
exponential :: Expr Double -> Expr (Double -> Double)
exponential λ = PDF (Exponential λ)
```

Every other possible probability density function within our language has been sugared in the same fashion.
Lastly, we introduce a few functions that can be used in place of the `Func` and `Let` constructors:

```haskell
(#=) :: Typeable a => Expr a -> (Expr a -> Expr b) -> Expr b
e1 #= e2 = Let e1 e2

lambda :: (Typeable a, Typeable b) => (Expr a -> Expr b) -> Expr (a -> b)
lambda = Func

lambda2 :: (Typeable a, Typeable b, Typeable c)
        => (Expr a -> Expr b -> Expr c) -> Expr (a -> b -> c)
lambda2 f = Func (Func . f)
```

These functions are not very complicated, but again, they contribute to making models in `ProbProg` more readable.

3.4.3 Arrays

As mentioned in the previous section, we also sugar the primitive operators that we have defined for arrays. As an example, we give our `mapA` function:

```haskell
mapA :: Expr (a -> b) -> Array n a -> Array n b
mapA f a = Apply (PrimFunc Map) (Pair f a)
```

The other array functions are, just like the primitive functions in section 3.4.2, sugared in the same fashion. We use the A suffix (which stands for "array") to avoid any name clashing with the equivalent functions in the *Haskell* prelude. Using our primitive operators, we can then also define more complicated operators, such as `sumA`:

```
sumA :: (Typeable a, Num (Expr a)) => Array (n + 1) a -> Expr a
sumA = foldrA (Func (\x -> Func (x +))) 0
```

This function is the *ProbProg* equivalent to the *Haskell* `sum` function.
Next to this, we implement the `IsList` type class found in GHC for our array expressions. This type class implements a conversion between the constant arrays in our language to *Haskell* lists. This is done as follows:

```
instance KnownNat n => IsList (Array n a) where
  type Item (Array n a) = Expr a
  fromList = CArray . fromJust . V.fromList
  toList (CArray a) = V.toList a
```

With this type class implemented, we can use the *OverloadedLists* language extension found in GHC in order to define array expressions in our language using the regular *Haskell* notation for constant lists. This means that if we write a list of *ProbProg* expressions in this notation, such as:

```
a :: Array 5 Int
a = [int 1, int 2, int 3, int 4, int 5]
```

this expression is now immediately interpreted as an array expression in the *ProbProg* language by the compiler.
Finally, it is very useful to define arrays of latent and observed parameters. We will discuss how this is done in the next section.

3.4.4 Parameters and constraints

We will start our explanation of the sugaring of the parameter syntax by looking at observed parameters. The `Input` constructor currently requires a `Proxy` argument that determines the type of the observed parameter. To make this more comprehensible for the user, we simply introduce different functions for each type of observed parameter. We illustrate this using real and integer input parameters as an example:

```
inputReal :: Expr (ModelResult Double)
inputReal = Input Proxy

inputInt :: Expr (ModelResult Int)
inputInt = Input Proxy
```

Next, we can look at latent parameters. The `Parameter` constructor requires an optional argument that can contain a `Constraint` on that parameter. As most parameters do not require constraints, we give two functions, `parameter` and `parameterC`, for unconstrained and constrained latent parameters, respectively.

```
parameter :: Expr String -> Expr (ModelResult Double)
parameter s = Parameter s Nothing

parameterC :: Expr String -> Constraint -> Expr (ModelResult Double)
parameterC s c = Parameter s (Just c)
```

The only other argument that the user has to specify is their custom name for the parameter.
We do not want the user to have to construct their own different `Constraint` data types. Instead, we provide different pre-built constraints in the sugared syntax that can be applied on the latent parameters. As mentioned before in section 3.1.2, the method of using constraints is similar to the method that the Stan probabilistic programming language uses [37]. Many of the constraint transformations, their inverse functions and their density factors are in fact given in the Stan user

guide, which we have used in order to implement our different constraints. As an example of an implementation, we give the `lower` constraint, which enforces a lower bound onto a parameter:

```
lower :: Expr Double -> Constraint
lower l = let t = Func $ \x -> log (x - l)
              i = Func $ \y -> exp y + l
              d = Func id -- == log (exp y)
          in  Constraint { transform = t, invert = i, densityFactor = d }
```

Notice that the logarithm of the density factor for this parameter is stored, which reduces this factor to the identity function. This is a fairly simple constraint, but the idea can be extended to much more complicated constraints. Later in this section we will see that we can create so-called simplex arrays, where each element of the array lies between 0 and 1 and the total sum of the elements is always equal to 1. These arrays can be constructed using different constraints for each element of the array, that are also dependent on the values for the earlier elements in the array. We leave the full explanation of the simplex constraints out of this thesis for the sake of brevity, but want to note that the ability to implement them shows that our way of formulating constraints is quite flexible.
As mentioned before in the previous section, we also wish to introduce a simple way to denote arrays of latent and observed parameters. We can now do this by combining our sugared syntax for arrays and the sugared syntax for parameters. We give the definition for arrays of real numbers as an example:

```
inputRealA :: KnownNat n => Expr (ModelResult (V.Vector n Double))
inputRealA = sequenceA (replicateA inputReal)

parameterA :: KnownNat n => Expr String -> Expr (ModelResult (V.Vector n Double))
parameterA s = let makeParam i = parameter (s
                                            +++ string "["
                                            +++ string (show (getFinite i))
                                            +++ string "]")
               in  sequenceA (generateA makeParam)
```

The implementation for an array of observed parameters, `inputRealA` is quite straightforward: we simply replicate the action of introducing a new real input and use our `sequenceA` function to correctly sequence the side-effects of these actions so that the `ModelResult` monad is wrapped outside of our array.
For the array of latent parameters, given by `parameterA`, we can largely follow the same procedure. However, we also wish to generate a unique name for each individual parameter in the array, based on the name for the array given by the user. As mentioned in section 3.1.2, we do this by appending the index of each parameter between brackets after the array name to give us the name for that specific parameter. This allows for an intuitive notation when the sampled parameter values are relayed back to the user. This requires us to also use our `generateA` function to generate these unique names for every individual parameter.
This technique, in combination with the constraints, also allows us to specify more complex arrays such as `simplex`. This function returns an array of latent parameters where each parameter has an upper bound of 1, a lower bound of 0 and the sum of the parameters is equal to 1. We also define several functions for nested parameter arrays, such as an array consisting of arrays of latent parameters (given by `parameterAA`) or an array consisting of multiple simplex arrays (given by `simplexA`), for example.

3.4.5 Monads and `Model` operators

We end this section on the sugared syntax by explaining the sugared syntax for sampling and other monadic operations. Firstly, we have sugared the `Sample` constructor to a slightly simpler notation:

```
(<~) :: Expr a -> Expr (a -> Double) -> Expr Model
x <~ f = Sample x f
```

This arrow notation was chosen to resemble the notation of sampling with an infix tilde operator that we have seen in chapter 2, which is common in Stan, for example.
Next, we show the `modelFor1` and `modelFor2` functions:

```
modelFor1 :: Typeable a => Expr (a -> Model) -> Array n a -> Expr Model
modelFor1 f xs = sequenceA_ (mapA f xs)

modelFor2 :: (Typeable a, Typeable b)
          => Expr (a -> b -> Model) -> Array n a -> Array n b -> Expr Model
modelFor2 f xs ys = sequenceA_ (zipWithA f xs ys)
```

As we can see, these functions apply a function element-wise onto an array, either by mapping or zipping it over that array. However, this function specifically returns a `Model` expression and the resulting array is sequenced with `sequenceA_`, which is a sugared function that performs monad sequencing and deletes the internal expression in the monad afterwards. This means that the result of applying `modelFor1` or `modelFor2` will also be a `Model` expression. The most common use-case for these functions is when a user would like to perform a similar sample operation element-wise over an array: we will see examples of this usage in section 3.5.
Next to this notation that makes performing sample operations over arrays more simple, we would also like to introduce additional notation for common monadic operations. We implement the following infix functions:

```
infixl 1 #>
(#>) :: (Typeable a, Typeable b)
     => Expr (ModelResult a) -> (Expr a -> Expr (ModelResult b))
     -> Expr (ModelResult b)
a #> f = a `Bind` Func f

infixl 1 #-
(#-) :: (Typeable a, Typeable b)
     => Expr (ModelResult a) -> Expr (ModelResult b) -> Expr (ModelResult b)
a #- b = a `Bind` Func (Prelude.const b)
```

In our example models in section 3.5, we will see that sugaring these common operations makes models a lot more human-readable and compact. Using these sugared infix functions also gives the code for models a more imperative look, similar to the sugared syntax of the do-notation for monads in *Haskell*.
Finally, we also sugar primitive monad operators such as `UnfoldrNM` in the same way that we have sugared the other primitive operators in earlier sections.

3.5 Example models

Using the sugared syntax, we can now write models in the *ProbProg* in a comprehensible and readable way. We will illustrate this in this section by showing the implementation of various commonly used probabilistic models using the sugared *ProbProg* syntax. In particular, we will look at some models that contain latent discrete parameters, as these models are interesting candidates for program transformations that we introduce in chapter 5.

3.5.1 Regression models

We will start with a few examples of regression models that do not require any discrete parameters in order to show what a probabilistic model constructed in the *ProbProg* language looks like.
We first describe a simple linear regression model: we have a set of n observations with predictors $\{x_i\}_{i=0}^{n-1}$ and outcome values $\{y_i\}_{i=0}^{n-1}$. For each of these observations we would like to model that $y_i = \alpha + \beta x_i + \epsilon_i$, for an intercept coefficient α, a slope coefficient β and a normally distributed error ϵ_i for each observation, so that $\epsilon_i \sim \mathcal{N}(0, \sigma)$. We can rewrite this model to $\forall i : y_i \sim \mathcal{N}(\alpha + \beta x_i, \sigma)$. If we take $n = 10$, this model can now be encoded in the *ProbProg* language as follows:

```
linearReg :: Expr Model
linearReg = (inputRealA :: Expr (ModelResult (Vector 10 Double))) #> \x ->
              inputRealA #> \y ->
              parameter "alpha" #> \α ->
              parameter "beta"  #> \β ->
              parameterC "sigma" (lower 0) #> \σ ->
              modelFor2 (lambda2 (\x' y' -> y' <~ normal (α + β * x') σ)) x y
```

In this example we can see many of the sugared syntax elements that have been introduced in the previous section come into play. We use `inputRealA` to introduce the two arrays containing $\{x_i\}$ and $\{y_i\}$ for the observed data. We then use `parameter` to introduce our latent parameters α and β, as well as `parameterC` to introduce the final latent parameter σ while attaching to it the constraint that $\sigma \geq 0$. Finally, we use `modelFor2` to loop over the two arrays containing the observations and perform the sampling operation $y_i \sim \mathcal{N}(\alpha + \beta x_i, \sigma)$ for each i. Notice that the sugared monad operator `(#>)` streamlines the binding of parameters to their variables in the lambda functions, which makes the model more readable. This was of course the goal of this operator, as mentioned in section 3.4.5.

3.5.2 Change point model

A change point model is a simple probabilistic model that contains a latent discrete parameter. We base our example model on a change point model shown in the Stan and PyMC User's guides [37, 43]. We want to create a Poisson model that models the occurrences of some event D_t at a known time-step t, where this time-step can range from 1 to T. There are two exponentially distributed rate parameters for this Poisson model: an early rate e and a late rate l. At some point in time s, the change point, the rate of the Poisson model for D_t changes from the early to the late rate. Using this point s as the latent discrete parameter, we have that

$$\begin{aligned} e &\sim \text{Exp}(r_e) \\ l &\sim \text{Exp}(r_l) \\ s &\sim \mathcal{U}\{1, T\} \\ D_t &\sim \text{Pois}(\textbf{if } t > s \textbf{ then } e \textbf{ else } l), \end{aligned} \tag{3.1}$$

which gives us the mathematical notation for our change point model. This mathematical notation of the model can then directly be translated into a *ProbProg* model:

```
changePoint :: Expr Model
changePoint = inputReal #> \re ->
              inputReal #> \rl ->
              inputReal #> \t  ->
              inputInt  #> \d  ->
              parameterC "e" (lower 0) #> \e ->
              parameterC "l" (lower 0) #> \l ->
              parameter "s"            #> \s ->
              e <~ exponential re #-
              l <~ exponential rl #-
              s <~ uniformD ([1, t] :: Array 2 Double) #-
              d <~ poisson (ifThenElse (t |<| s) e l)
```

As before, we first introduce the required observed and latent parameters with their respective constraints. We then perform the sample operations needed to encode the model, which are very similar to the mathematical notation of the model. As we are not using any arrays of parameters, we do not use the `modelFor` or `modelFor2` functions. Instead, we can use the infix monad function `(#-)` to cleanly connect our sequential sample operations. For the sake of simplicity, we have slightly changed the model: the range of s has been decreased from $s \in \{1, \dots, T\}$ to $s \in \{1, T\}$. (More information on this simplification can be found in section 6.1.1.)
Notice that, as has been mentioned before in section 3.3, the semantics for the uniform discrete distribution require that s is discrete, but we cannot enforce this at this point, meaning that we cannot evaluate this model in the way that it is currently written. Later in this thesis we will see

that we can apply an automatic program transformation in order to rewrite this change point model so that the discrete latent parameter s can be marginalised out of the model, enabling us to evaluate it.
We have chosen this specific example of a change point model as it is a widely used model that is often difficult to express in the direct mathematical notation in existing probabilistic programming languages. As we have seen in section 2.3.1 the user has to marginalise the latent discrete parameter s out of their model when writing their code in Stan, for example [37]. Later in chapter 5 we will show how we are able to transform this model so that this marginalisation is performed automatically, allowing the user to simply write this model in its current form.

3.5.3 Hidden Markov models

Another class of commonly used models that often contain latent discrete parameters are the so-called hidden Markov models or HMMs. We will give a quick explanation of a simple hidden Markov model: we have a set of N observed parameters $\{y_i\}_{i=0}^{n-1}$ that are each conditioned on a hidden parameter x_i. Each x_i can take on any value from a set $\{1, \ldots, K\}$ of K states. These parameters x_i form a Markov chain so that each x_i is independent of any other parameters if x_{i-1} is given. The transition probabilities of this Markov chain are given as K vectors $\boldsymbol{\theta}_k$, where the jth element of each $\boldsymbol{\theta}_i$ contains the probability that some parameter in the chain x_t has state j, given that x_{t-1} has state i. These vectors $\boldsymbol{\theta}_k$ are therefore simplex vectors: each element lies between 0 and 1 and the sum of the elements is always equal to 1.
We can then write the probability of transitioning from state x_{t-1} to state x_t as

$$x_t \sim \text{Cat}(\boldsymbol{\theta}_{x_{t-1}}). \tag{3.2}$$

These parameters x_t now each influence their respective observed variable y_t. Say each y_t can take on values in the set $\{1, \ldots, V\}$. We then use K simplex vectors $\boldsymbol{\phi}_k$ to model this output effect that each x_t has on each y_t: the jth element of $\boldsymbol{\phi}_i$ represents the probability that some y_t is equal to j, given that x_t has state i. For each of the observed parameters y_t, we can then say that it depends on x_t as follows:

$$y_t \sim \text{Cat}(\boldsymbol{\phi}_{x_t}). \tag{3.3}$$

Given these rules, we can use them to model a hidden Markov model in *ProbProg*:

```
1  hmm :: Expr Model
2  hmm = (inputIntA :: Expr (ModelResult (Vector 10 Int))) #> \y ->
3        inputIntA #> \x ->
4        (inputRealAA :: Expr (ModelResult (Vector 10 (Vector 10 Double)))) #> \α ->
5        (inputRealAA :: Expr (ModelResult (Vector 10 (Vector 10 Double)))) #> \β ->
6        simplexA "theta" #> \θ ->
7        simplexA "phi"   #> \ϕ ->
8        modelFor2 (lambda2 (\t a -> t <~ dirichlet a)) θ α #-
9        modelFor2 (lambda2 (\p b -> p <~ dirichlet b)) ϕ β #-
10       modelFor2 (lambda2 (\w c -> w <~ categorical c)) y (backpermuteA ϕ x) #-
11       modelFor2 (lambda2 (\c1 c2 -> c1 <~ categorical c2)) (tailA x)
12                          (backpermuteA theta ((reverseA . tailA . reverseA) x))
```

Before we explain the workings of this model, we would like to note two things: first, we explicitly assign priors to the different $\boldsymbol{\theta}_k$ and $\boldsymbol{\phi}_k$ vectors using Dirichlet distributions (parameterised by the elements of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$). Similar example models in Stan do this as well, but this is not necessary: if the priors are left out, the $\boldsymbol{\theta}_k$ and $\boldsymbol{\phi}_k$ vectors are distributed uniformly over all possible simplex vectors [37]. Secondly, we deviate from our earlier mathematical model by stating that the state parameters x_i are observed variables. This allows us to construct the model without having latent discrete parameters. We can simply rewrite line 3 to change the state parameters to latent parameters in order to obtain the original mathematical model. Notice that this does require us to eliminate the latent discrete parameters using an automatic program transformation before the model can be evaluated, just as was required for evaluating the change point model in the previous section.

Now that we know what our model represents exactly, we can examine how we have implemented it in *ProbProg*. As always, we start by defining all the parameters of the model, which can be seen in the first seven lines. Next, we use `modelFor2` in lines 8 and 9 in order to set the Dirichlet priors for each $\boldsymbol{\theta}_k$ and $\boldsymbol{\phi}_k$, using the accompanying $\boldsymbol{\alpha}_k$ and $\boldsymbol{\beta}_k$ values, respectively. The final two `modelFor2` expressions show how we achieve the sample statements modelled in equations 3.3 and 3.2. Notice that we use a lot of different array transformation functions on our parameter array in order to map over the correct elements with our sample statements.

As a final note, we would like to mention that this example hidden Markov model can easily be extended to a hidden Markov model with a more complicated structure. We give two examples of this: firstly, we explain how a factorial hidden Markov model, which is a hidden Markov model that has multiple Markov chains that influence the observed parameters y_i, can be implemented. Each new Markov chain has a set of transition probabilities (as those found in $\boldsymbol{\theta}$ before) and a set of output probabilities (as those found in $\boldsymbol{\phi}$ before). This requires two additions in the code: we have to add the new chains with their new transition probabilities, which can be done by initialising the Dirichlet priors for these probabilities (as in line 8) and encoding the dependencies for each new chain (as in line 11). Next to this, we have to initialise the Dirichlet priors for each new set of output probabilities (as in line 9) and we have to encode how each new chain influences the output (as in line 10). We can therefore easily add a new Markov chain to our hidden Markov model using the same `modelFor2` operations that we use in our normal hidden Markov models.
Secondly, we can implement a second order hidden Markov model, which is a hidden Markov model in which parameters in the Markov chain x_i do not only depend on the previous parameter x_{i-1}, but also on the parameter before that, x_{i-2}. This extension means that a transition vector $\boldsymbol{\theta}_t$ now becomes a matrix of transition vectors, where element j of $\boldsymbol{\theta}_{t,s}$ is the probability that some $x_i = j$, given that $x_{i-1} = t$ and $x_{i-2} = s$. Implementing this in *ProbProg* code comes down to rewriting the hidden Markov model so that we correctly initialise Dirichlet priors over all of these new simplex vectors (by changing line 8) and rewriting line 11 so that the back-permutation uses both the previous parameter and the parameter before that to obtain the correct probability vector from $\boldsymbol{\theta}$. Again, we see that we can adapt our hidden Markov models, this time by using higher-order Markov chains, without too many complicated changes.

3.6 Conclusion

Our description of the *ProbProg* language is now complete. We have seen the basic syntax of the language, how we are able to evaluate models encoded in the language and how we are able to build several commonly used probabilistic models using the sugared syntax. Now that we have a purely functional domain specific language that can perform the desired probabilistic inference, we can use it for our further analysis in the rest of this thesis. The following chapters will show how we can use the *ProbProg* language with the design decisions that we have made in order to analyse the conditional independencies of our implemented models and answer our research questions.

Chapter 4

Conditional dependence analysis

In chapter 2 we saw that Gorinova et al. [11] have constructed an extension of the type system of their SlicStan language that is able to infer conditional independence information from probabilistic models implemented within this language. In this thesis, we use a similar approach and extend the type system for *ProbProg* with effect annotations to capture dependence information. Type inference within this effect system amounts to a dependency analysis that determines on which parameters expressions depend. From this analysis, we can construct the factor graph representation of models in our language. Obtaining the factor graph differs from how Gorinova et al. infer conditional dependence information. We want to obtain it, because this allows us to store every conditional dependency of a model; even the conditional dependence rules that do not contain every parameter of that model.

4.1 Effect system

To describe this analysis, we construct our annotated types τ with their annotations ϕ in the form of a Levy-style call-by-name type and effect system [44]. The annotated types are defined as follows:

$$\begin{aligned} \tau_c &::= \mathit{Int} \mid \mathit{Real} \mid \mathit{Bool} \mid \mathit{String} \\ \tau &::= \tau_c^{\phi} \mid (\tau_1, \tau_2) \mid \tau_1 \rightarrow \tau_2 \mid m(\tau) \mid \alpha^{\phi} \\ \phi &::= \sigma \mid \beta \\ \sigma &::= \emptyset \mid \{\pi\} \cup \sigma. \end{aligned} \tag{4.1}$$

With this, we also introduce the following syntactical categories:

$$\begin{aligned} \alpha &\in \textbf{Type Variables} \\ \beta &\in \textbf{Annotation Variables} \\ \pi &\in \textbf{Parameter Variables} \end{aligned} \tag{4.2}$$

The **Type Variables** and **Annotation Variables** categories are two disjoint categories that contain unique identifiers for type and annotation variables respectively. They are often used in many type and effect systems [45].
The **Parameter Variables** category is a similar category that contains identifiers that uniquely represent latent and observed parameters within a probabilistic model. Our effect annotations therefore represent sets of these parameters. The effect that we are modelling using our effect system can be interpreted as the set containing the parameters that are used in the expression with this effect. We will refer to this as the set of parameters that an expression "depends on".
We extend this idea to tuples, where the left element of the tuple might depend on different parameters than the right element, and to functions, where the input and output expressions depend on

different parameters. The dependencies of the output expression can naturally be influenced by the dependencies of the input expression, as will be shown in the typing judgements in section 4.1.1.
The different sets of parameters σ are members of a set Σ which is the power-set of the parameters in a model: $\Sigma = \mathcal{P}(\textbf{Parameter Variables})$. Together with the subset relation, this set forms a lattice $(\Sigma, \subseteq)$. This lattice allows us to compare different parameter sets: this is useful, as we often wish to find the minimal parameter set that an expression depends on. We will later see that it also helps us introduce subeffecting into our analysis.
The translation from this effect to the factor graph of the model is straightforward. We know that there is only one syntactic element within our language that can adjust the target expression that our model represents: the sample operation. Each sample operation within a model introduces a new factor - or in cases where the sample operation is applied multiple times, multiple factors - to the target expression. Therefore, correctly inferring the smallest possible parameter set ϕ_i for every sample operation within a model provides us with the parameters contained within each factor in the target expression.
This information corresponds directly with the nodes and edges of the factor graph of our model: the factor nodes of the graph are given by our parameters $\{\pi_1, \dots, \pi_n\}$. The variable nodes are given by the sample operations $\{s_1, \dots, s_m\}$. We then create sets S_i for $i = 0, \dots n$, so that S_i contains the sample operations that depend on parameter π_i. We can form these sets using our dependency analysis. With these sets, we can form the edges of the factor graph: there is an edge between the factor node given by π_i and the variable node given by s_j if $s_j \in S_i$.

4.1.1 Typing judgements

To generate our desired effects using typing judgements, we introduce an environment Γ that stores the annotated types for variables:

$$\Gamma ::= \emptyset \mid \Gamma[x \mapsto \tau]. \tag{4.3}$$

Notice that Γ stores both the types and effects (which are contained in τ) for its variables. It is therefore a call by name environment. The actual implementation in our language uses an environment indexed by DeBruijn indices, but here we use explicit variable names for better readability.
Together with our environment, we also introduce an operation that can add effects to types. This operation is needed to define the typing judgements within a Levy-style call-by-name effect system [44]. This operation $\langle - \rangle$ maps effects onto our types in the following manner:

$$\begin{aligned} \langle \phi' \rangle \tau_c^{\phi} &= \tau_c^{\phi \cup \phi'} \qquad \text{where } \tau_c \text{ is a } \textit{Bool}, \textit{Int} \text{ or } \textit{Double}. \\ \langle \phi' \rangle \tau_1 \to \tau_2 &= \langle \phi' \rangle \tau_1 \to \langle \phi' \rangle \tau_2 \\ \langle \phi' \rangle (\tau_1, \tau_2) &= (\langle \phi' \rangle \tau_1, \langle \phi' \rangle \tau_2) \\ \langle \phi' \rangle m(\tau) &= m(\langle \phi' \rangle \tau). \end{aligned} \tag{4.4}$$

The typing judgements for our analysis that apply to all constructs that do not involve arrays within our language are given in table 4.1, with the typing rules for primitive operators given in table 4.2. The method with which we handle expressions that involve arrays will be given later in section 4.3. Most of the typing judgements are very straightforward. As our language is purely functional, its referential transparency ensures that determining the dependencies of an expression is often quite simple. For example, dependencies can never be introduced through side-effects.
The different typing judgements work together to produce the desired effects. *Param* and *Input* are the two judgements that introduce singleton effect sets. As one would expect, latent and observed parameters are dependent on themselves. Similarly, *Const* shows that constants do not depend on any parameters. The *PDF* and *PDF2* judgements show how dependencies spread through probability density functions by combining the dependencies of their parameters. Notice that probability density functions with different dependency structures might exist, but these are currently not implemented in our language. The typing judgements *Func*, *PrimFunc* and *Apply* show how we can propagate the dependencies through functions while still treating functions as first-class citizens.

Most primitive functions can be typed according to the *Unary* or *Binary* pattern, where they depend on all of their operands, as seen in table 4.2. Two exceptions are the functions `Fst` and `Snd` with their respective typing judgements *Fst* and *Snd*. This shows the importance of having a tuple effect of the form (τ_1, τ_2): we are interested in the smallest possible set of dependencies of an expression. This means that while a typing judgement such as

$$\frac{\Gamma \vdash e_1 : \langle \phi_1 \rangle \tau_1 \qquad \Gamma \vdash e_2 : \langle \phi_1 \rangle \tau_2}{\Gamma \vdash \texttt{Pair}\ e_1, e_2 : \langle \phi_1 \cup \phi_2 \rangle (\tau_1, \tau_2)} \tag{4.5}$$

where the individual dependencies of each element are discarded could still be useful, it forces us to include the dependencies of e_2 in those of `Apply Fst (Pair` e_1, e_2`)`, which is something that we would rather avoid.
One instance where we have to instead take a conservative estimate of our effect is within the context of the *If* judgement. The effect of an if-statement `If` e_c e_1 e_2 will always depend on the union of the dependencies of both branches, while its true dependencies will only be either the dependencies of e_1 or those of e_2, depending on the evaluation of e_c. However, we cannot determine the evaluation of e_c during a static analysis and therefore choose the conservative effect in this case.
Notice that the fact that our effect system uses a call-by-name environment helps to obtain a more accurate set of parameters in the effect annotation. For example, if we define a program `Let` $x = e_1$ `in` e_2, we do not want its effect to contain any parameters that only appear in e_1 and not in e_2, which is what would happen if the effect system was a call-by-value variant instead. This choice is independent of the evaluation of the program, because if the program is evaluated eagerly, the effect system should still use a call-by-name environment.
The judgements *Sample* and *Bind* are very similar to the *Apply* judgement, as those expressions spread dependencies throughout a model as if one was applying a function. In a similar fashion, the *Return* and *Bind* judgements help propagate the effects throughout the monadic structure of our models.
Finally, the *Sub* judgement allows us to use sub-effecting. This means that we can always add to the dependencies of an expression, whereas we can never remove an existing dependency. This can be necessary when applying several different typing judgements in sequence.

4.2 Type inference

Our goal is now to implement an inference algorithm for this theoretical type and effect system so that the unique minimal sets of dependencies of the sample operations in a model can be inferred in reasonable time. Our inference algorithm will consist of a custom version of algorithm $\mathcal{W}$. This algorithm is used for Hindley-Milner type inference [46] and is often adapted to infer effects [45]. Before we can describe the algorithm, we first introduce a new set of definitions:

$$\begin{aligned} C &::= \emptyset \mid \{c\} \cup C \\ c &::= \beta \supseteq \phi \\ \gamma &::= \rho \mid \forall \alpha : \gamma \mid \forall \beta : \gamma \\ \rho &::= \tau \mid C \to \tau \\ \theta &::= id \mid [\alpha \mapsto \tau] \circ \theta \mid [\beta \mapsto \phi] \circ \theta \end{aligned} \tag{4.6}$$

These definitions describe three new concepts that are necessary to ensure polyvariance: the ability for our types to be polymorphic in the annotations. Firstly, we introduce so-called constraints C, which are sets consisting of superset relations between annotation variables β and annotations ϕ. These constrains will be used to store intermediate superset relations that need to be maintained while inferring the effects. As any type could have annotation variables in its effect annotation, the

Rule	Judgement	Condition
Var	$\Gamma \vdash \texttt{Var}\ x : \tau$	if $\Gamma(x) = \tau$.
Const	$\Gamma \vdash c : \tau_c^{\emptyset}$	for constants c of type τ_c.
Param	$\Gamma \vdash \texttt{Parameter}\ s\ c : m(\tau_p^{\{\pi\}})$	with a fresh identifier π and unannotated parameter type τ_p.
Input	$\Gamma \vdash \texttt{Input}\ i : m(\tau_i^{\{\pi\}})$	with a fresh identifier π and unannotated input type τ_i.
Pair	$\dfrac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{Pair}\ e_1, e_2 : (\tau_1, \tau_2)}$	
PDF1	$\dfrac{\Gamma \vdash e_{pdf} : \tau}{\Gamma \vdash \texttt{PDF}\ e_{pdf} : \tau}$	(also see *PDF2*.)
PDF2	$\Gamma \vdash D : \tau_1^{\phi_1} \to \cdots \to \tau_i^{\phi_1 \cup \cdots \cup \phi_i}$	for probability density functions D with parameters of type $\tau_1 \ldots \tau_i$.
If	$\dfrac{\Gamma \vdash e_c : \mathit{Bool}^{\phi_c} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{If}\ e_c\ e_1\ e_2 : \langle \phi_c \rangle \tau}$	
Let	$\dfrac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{Let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$	
Func	$\dfrac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \texttt{Func}\ x \mapsto e : \tau_1 \to \tau_2}$	
PrimFunc	$\dfrac{\Gamma \vdash e_{op} : \tau}{\Gamma \vdash \texttt{PrimFunc}\ e_{op} : \tau}$	(see table 4.2.)
Apply	$\dfrac{\Gamma \vdash e_f : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_x : \tau_1}{\Gamma \vdash \texttt{Apply}\ e_f\ e_x : \tau_2}$	
Sample	$\dfrac{\Gamma \vdash e_f : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_x : \tau_1}{\Gamma \vdash \texttt{Sample}\ e_x\ e_f : m(\tau_2)}$	
Return	$\dfrac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{Return}\ e : m(\tau)}$	
Bind	$\dfrac{\Gamma \vdash e_f : \tau_1 \to m(\tau_2) \qquad \Gamma \vdash e_x : m(\tau_1)}{\Gamma \vdash \texttt{Bind}\ e_x\ e_f : m(\tau_2)}$	
Sub	$\dfrac{\Gamma \vdash e : \langle \phi \rangle \tau \qquad \phi \subseteq \phi'}{\Gamma \vdash e : \langle \phi' \rangle \tau}$	

Table 4.1: The main typing judgements for the dependency analysis of our language.

Unary	$\Gamma \vdash e_{un} : \langle\phi\rangle\tau_1 \rightarrow \langle\phi\rangle\tau_2$	This holds for unary operators where the result depends upon the operand, such as `Negate`, `Abs`, `Signum`, `Log`, `Exp` and `Not`. Types τ_1 and τ_2 are inferred based on the unannotated type of the function.
Binary	$\Gamma \vdash e_{un} : \langle\phi_1\rangle\tau_1 \rightarrow \langle\phi_2\rangle\tau_2 \rightarrow \langle\phi_1 \cup \phi_2\rangle\tau_3$	This holds for binary operators where the result directly depends on both operands, such as `Add`, `Sub`, `Mul`, `Div`, `And`, `Eq` and `Leq`. Types τ_1, τ_2 and τ_3 are inferred based on the unannotated type of the function.
Fst	$\Gamma \vdash \texttt{Fst} : (\tau_1, \tau_2) \rightarrow \tau_1$	
Snd	$\Gamma \vdash \texttt{Snd} : (\tau_1, \tau_2) \rightarrow \tau_2$	

Table 4.2: The main primitive operator typing judgements for the dependency analysis of our language.

knowledge stored in constraints ensures that we can still resolve to a concrete effect when possible [45].
Secondly, we introduce type schemes γ. This allows us to store types that are universally quantified over the annotation or type variables in our environment. To do this, we redefine our environment as follows:

$$\hat{\Gamma} ::= \emptyset \mid \hat{\Gamma}[x \mapsto \gamma]. \tag{4.7}$$

Notice that these type schemes contain so-called qualified types ρ [47]. These qualified types can have a restrictive constraint attached to them. Again, this is done to ensure proper polyvariance: when universally quantifying over annotation variables, we must make sure that the constraints that contain these variables are also quantified over properly. The same holds for instantiating universally quantified variables: when instantiating a γ of the form $\forall\beta_1 \ldots \forall\beta_n : C \rightarrow \tau$, we must find values for each annotation variable β_i that make sure that the constraint C still holds.
Finally, we introduce substitutions θ. These are sets of mappings from type and annotation variables to concrete types and annotations, respectively. Substitutions can be combined: we denote the composition of two substitutions as $\theta_1 \circ \theta_2$. The empty substitution is denoted as *id*. These substitutions can be applied on types as follows:

$$\begin{aligned}
\theta\tau_c^{\phi} &= \tau_c^{\theta\phi} && \text{where } \tau_c \text{ is a } \textit{Bool}, \textit{Int} \text{ or } \textit{Double}. \\
\theta(\tau_1 \rightarrow \tau_2) &= \theta(\tau_1) \rightarrow \theta(\tau_2) \\
\theta(\tau_1, \tau_2) &= (\theta(\tau_1), \theta(\tau_2)) \\
\theta(m(\tau)) &= m(\theta(\tau)) \\
\theta\alpha &= \tau' && \text{if } \{\alpha \mapsto \tau'\} \in \theta \\
\theta\alpha &= \alpha && \text{otherwise.}
\end{aligned} \tag{4.8}$$

They work similarly on annotations:

$$\begin{aligned} \theta\phi &= \phi \\ \theta\beta &= \phi' \quad \text{if } \{\beta \mapsto \phi'\} \in \theta \\ \theta\beta &= \beta \qquad\quad \text{otherwise.} \end{aligned} \tag{4.9}$$

Finally, they can also be applied on constraints, type schemes, qualified types and environments in order to update the variables contained in those constructs. For constraints we define this as

$$\begin{aligned} \theta C &= \{\theta c \mid c \in C\} \\ \theta c &= \theta\beta \supseteq \theta\phi, \end{aligned} \tag{4.10}$$

for type schemes we have

$$\begin{aligned} \theta(\forall\alpha : \gamma) &= \forall\alpha : (\theta/\alpha)\gamma \\ \theta(\forall\beta : \gamma) &= \forall\beta : (\theta/\beta)\gamma \\ (\theta/x)x' &= x' \qquad\qquad \text{if } x = x' \\ (\theta/x)x' &= \theta x' \qquad\quad \text{otherwise,} \end{aligned} \tag{4.11}$$

for qualified types we get

$$\theta(C \to \tau) = \theta C \to \theta\tau, \tag{4.12}$$

and for type environments we have that

$$\begin{aligned} \theta(\Gamma[x \mapsto \gamma]) &= \theta\Gamma[x \mapsto \theta\gamma] \\ \theta\emptyset &= \emptyset. \end{aligned} \tag{4.13}$$

With these definitions, we can now introduce our algorithm $\mathcal{W}_{DEP}$. Given an expression e and an environment $\hat{\Gamma}$, our algorithm $\mathcal{W}_{DEP}(\hat{\Gamma}, e) = (\tau, \theta, C)$ returns the annotated type τ of our expression, together with the substitution θ generated by the inference and the constraint C that needs to be satisfied for the inference to be correct.
One thing we note, however, is that our host language *Haskell* can already handle the inference of the unannotated types. We make use of this in our inference algorithm and only infer the annotations and not the base types underneath. We therefore introduce the following notation in order to omit the unannotated types from our inference algorithm:

$$\begin{aligned} \langle\phi\rangle &::= \exists\tau : \langle\phi\rangle\tau \\ \langle\phi_1\rangle \to \langle\phi_2\rangle &::= \exists\tau_1, \tau_2 : \langle\phi_1\rangle\tau_1 \to \langle\phi_2\rangle\tau_2 \\ (\langle\phi_1\rangle, \langle\phi_2\rangle) &::= \exists\tau_1, \tau_2 : (\langle\phi_1\rangle\tau_1, \langle\phi_2\rangle\tau_2). \end{aligned} \tag{4.14}$$

As we only care for the inference of the correct annotations, we also leave any operations needed to infer the correct unannotated type out of our algorithm. Operations needed to ensure polymorphism are therefore left out, whereas operations that ensure proper polyvariance are still performed.

4.2.1 Algorithm

Tables 4.3 and 4.4 show the implementation of the typing judgements introduced in section 4.1.1 using algorithm $\mathcal{W}_{DEP}$. As is the case with the Hindley-Milner inference algorithm, this algorithm has three accompanying functions. Firstly, a unification algorithm $\mathcal{U}_{DEP}$, which is given in table 4.5. This algorithm returns a substitution θ that unifies two given annotations, if possible.
Secondly, the *Gen* function, defined as follows:

$$\begin{aligned} \mathit{Gen}_{\hat{\Gamma}}(C, \langle\phi\rangle) &= (C'', \forall\beta_1, \ldots, \beta_n : C' \to \langle\phi\rangle) \\ \text{where } C' &= \{c \in C \mid \exists i : \beta_i \in \mathit{freeAnnVars}(c)\} \\ C'' &= C - C' \\ \{\beta_i\}_1^n &= \mathit{freeAnnVars}(\phi) - \mathit{freeAnnVars}(\hat{\Gamma}). \end{aligned} \tag{4.15}$$

This function generalises annotations by universally quantifying over the free annotation variables within it that are not free in the environment. Additionally, it also quantifies correctly over the relevant constraints in the given constraint set by adding it as a qualified type. Together with the type scheme, it also returns the constraints that have not been added to the qualified type.
Finally, the *Inst* function, which does the opposite of the *Gen* function:

$$\begin{gathered} \mathit{Inst}_{\hat{\Gamma}}(\forall \beta_1, \ldots, \beta_n : \rho) = [\beta_1 \mapsto \beta_1'] \ldots [\beta_n \mapsto \beta_n']\rho \\ \text{where } \beta_1', \ldots, \beta_n' \text{ are fresh.} \end{gathered} \tag{4.16}$$

Instead, this function instantiates a type scheme into a qualified type by instantiating all the universally quantified annotation variables as fresh variables.
Our algorithm $\mathcal{W}_{DEP}$ encodes many of the properties of the typing judgements as constraints. Again, this is similar to the implementation of other effect systems [45]. In practice, this introduces another step in the dependency analysis: resolving these constraints. Say we find the effects on a sample statement e using the algorithm: $\mathcal{W}_{DEP}(\hat{\Gamma}, e) = (\beta, \theta, C)$, where β is some annotation variable. We will then have to resolve the superset constraints in C in order to find the minimal set of parameters ϕ that β represents (and thus the minimal set of parameters that e depends on). There are specialised subset solvers that can solve these constraints efficiently, but for our current implementation the following naive algorithm also suffices:

$$\begin{aligned} &\quad \text{let } S = \{\beta\} \\ &\text{while } \exists x \in S \text{ so that } x \text{ is an annotation variable} \\ &\quad \text{do } S = \bigcup_{\beta' \in S} \{\phi \mid \{\beta' \supseteq \phi\} \in C\} \\ &\text{return } \bigcup_{\phi \in S} \phi \end{aligned} \tag{4.17}$$

This gives us the required minimal set of parameters, which completes the dependency analysis.

$$\mathcal{W}_{DEP}(\hat{\Gamma}, c) = (\langle\emptyset\rangle, id, \emptyset)$$

For a constant c.

$$\mathcal{W}_{DEP}(\hat{\Gamma}, x) = \text{ let } C \to \langle\phi\rangle = \mathit{Inst}(\hat{\Gamma}(x)) \\ \text{in } (\langle\phi\rangle, id, C)$$

For a variable x.

$$\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Parameter}\ \ s\ c) = \text{ let } \pi \text{ be fresh} \\ \text{in } (\langle\{\pi\}\rangle, id, \emptyset)$$

$$\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Input}\ \ i) = \text{ let } \pi \text{ be fresh} \\ \text{in } (\langle\{\pi\}\rangle, id, \emptyset)$$

$$\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Pair}\ \ e_1\ e_2) = \text{ let } (\langle\phi_1\rangle, \theta_1, C_1) = \mathcal{W}_{DEP}(\hat{\Gamma}, e_1) \\ (\langle\phi_2\rangle, \theta_2, C_2) = \mathcal{W}_{DEP}(\theta_1\hat{\Gamma}, e_2) \\ \text{in } ((\theta_2\langle\phi_1\rangle, \langle\phi_2\rangle), \theta_2 \circ \theta_1, (\theta_1 C_2) \cup (\theta_2 C_1))$$

$$\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{PDF}\ \ D(e_1, \ldots, e_n)) = \text{ let } \beta_x, \beta \text{ be fresh} \\ (\langle\phi_1\rangle, \theta_1, C_1) = \mathcal{W}_{DEP}(\hat{\Gamma}, e_1) \\ (\langle\phi_2\rangle, \theta_2, C_2) = \mathcal{W}_{DEP}(\theta_1\hat{\Gamma}, e_2) \\ \vdots \\ (\langle\phi_n\rangle, \theta_n, C_n) = \mathcal{W}_{DEP}((\theta_n \circ \cdots \circ \theta_1)\hat{\Gamma}, e_n) \\ \text{in } (\langle\beta_x\rangle \to \langle\beta\rangle, \theta_n \circ \cdots \circ \theta_1, \\ (C_n \cup \cdots \cup (\theta_n \circ \cdots \circ \theta_2)C_1) \cup (\bigcup_i \{\beta \supseteq \phi_i\}) \cup (\beta \subseteq \beta_x))$$

For a distribution D with parameters $e_1 \ldots e_n$.

$$\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{If}\ \ e_c\ e_1\ e_2) = \text{ let } \beta \text{ be fresh} \\ (\langle\phi_c\rangle, \theta_c, C_c) = \mathcal{W}_{DEP}(\hat{\Gamma}, e_c) \\ (\langle\phi_1\rangle, \theta_1, C_1) = \mathcal{W}_{DEP}(\hat{\Gamma}, e_1) \\ (\langle\phi_2\rangle, \theta_2, C_2) = \mathcal{W}_{DEP}(\hat{\Gamma}, e_2) \\ \text{in } (\langle\beta\rangle, \theta_2 \circ \theta_1 \circ \theta_c, \\ ((\theta_2 \circ \theta_1)C_c) \cup (\theta_2 C_1) \cup C_2 \cup \{\beta \supseteq (\phi_1 \cup \phi_2 \cup \phi_c\}\})$$

$$\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Let}\ \ x = e_1\ \ \texttt{in}\ \ e_2) = \text{ let } (\langle\phi_1\rangle, \theta_1, C_1) = \mathcal{W}_{DEP}(\hat{\Gamma}, e_1) \\ (C_1', \langle\phi_1'\rangle) = \mathit{Gen}(C_1, \langle\phi_1\rangle) \\ (\langle\phi_2\rangle, \theta_2, C_2) = \mathcal{W}_{DEP}(\theta_1\hat{\Gamma}[x \mapsto \phi_1'], e_2) \\ \text{in } (\langle\phi_2\rangle, \theta_2 \circ \theta_1, C_2 \cup (\theta_2 C_1'))$$

$$\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Func}\ \ x \to f) = \text{ let } \beta_x \text{ be fresh} \\ (\langle\phi_f\rangle, \theta_f, C_f) = \mathcal{W}_{DEP}(\hat{\Gamma}[x \mapsto \beta_x], f) \\ \text{in } ((\theta_f\langle\beta_x\rangle) \to \langle\phi_f\rangle, \theta_f, C_f)$$

$$\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{PrimFunc}\ \ f) = \mathcal{W}_{DEP}(\hat{\Gamma}, f)$$

(See table 4.4.)

$$\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Apply}\ \ f\ x) = \text{ let } \beta \text{ be fresh} \\ (\langle\phi_f\rangle, \theta_f, C_f) = \mathcal{W}_{DEP}(\hat{\Gamma}, f) \\ (\langle\phi_x\rangle, \theta_x, C_x) = \mathcal{W}_{DEP}(\theta_f\hat{\Gamma}, x) \\ \theta_u = \mathcal{U}_{DEP}(\langle\phi_x\rangle \to \langle\beta\rangle, \theta_x\langle\phi_f\rangle) \\ \text{in } (\theta_u\langle\beta\rangle, \theta_u \circ \theta_x \circ \theta_f, ((\theta_u \circ \theta_x)C_f) \cup (\theta_u C_x))$$

$$\begin{aligned}
\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Sample}\ y\ f) = \ & \text{let } \beta \text{ be fresh} \\
& \quad (\langle\phi_f\rangle, \theta_f, C_f) = \mathcal{W}_{DEP}(\hat{\Gamma}, f) \\
& \quad (\langle\phi_y\rangle, \theta_y, C_y) = \mathcal{W}_{DEP}(\theta_f\hat{\Gamma}, y) \\
& \quad \theta_u = \mathcal{U}_{DEP}(\langle\phi_y\rangle \to \langle\beta\rangle, \theta_y\langle\phi_f\rangle) \\
& \text{in } (\theta_u\langle\beta\rangle, \theta_u \circ \theta_y \circ \theta_f, ((\theta_u \circ \theta_y)C_f) \cup (\theta_u C_y))
\end{aligned}$$

$$\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Return}\ e) = \mathcal{W}_{DEP}(\hat{\Gamma}, e)$$

$$\begin{aligned}
\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Bind}\ e\ f) = \ & \text{let } \beta \text{ be fresh} \\
& \quad (\langle\phi_f\rangle, \theta_f, C_f) = \mathcal{W}_{DEP}(\hat{\Gamma}, f) \\
& \quad (\langle\phi_e\rangle, \theta_e, C_e) = \mathcal{W}_{DEP}(\theta_f\hat{\Gamma}, e) \\
& \quad \theta_u = \mathcal{U}_{DEP}(\langle\phi_e\rangle \to \langle\beta\rangle, \theta_e\langle\phi_f\rangle) \\
& \text{in } (\theta_u\langle\beta\rangle, \theta_u \circ \theta_e \circ \theta_f, ((\theta_u \circ \theta_e)C_f) \cup (\theta_u C_e))
\end{aligned}$$

Table 4.3: Algorithm $\mathcal{W}_{DEP}$ for dependency analysis without arrays.

$\mathcal{W}_{DEP}(\hat{\Gamma}, f) = \text{let } \beta_1, \beta \text{ be fresh in } (\langle\beta_1\rangle \to \langle\beta\rangle, id, \{\beta \supseteq \beta_1\})$	For unary operators f where the result depends on the operand.
$\mathcal{W}_{DEP}(\hat{\Gamma}, f) = \text{let } \beta_1, \beta_2, \beta \text{ be fresh in } ((\langle\beta_1\rangle, \langle\beta_2\rangle) \to \langle\beta\rangle, id, \{\beta \supseteq \beta_1 \cup \beta_2\})$	For binary operators f where the result depends on both operands.
$\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Fst}) = \text{let } \beta_1, \beta_2, \beta \text{ be fresh in } ((\langle\beta_1\rangle, \langle\beta_2\rangle) \to \langle\beta\rangle, id, \{\beta \supseteq \beta_1\})$	
$\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Snd}) = \text{let } \beta_1, \beta_2, \beta \text{ be fresh in } ((\langle\beta_1\rangle, \langle\beta_2\rangle) \to \langle\beta\rangle, id, \{\beta \supseteq \beta_2\})$	

Table 4.4: Algorithm $\mathcal{W}_{DEP}$ for the standard primitive operators.

$$\mathcal{U}_{DEP}(\langle\{\pi_1, \ldots, \pi_n\}\rangle, \langle\{\pi_1, \ldots, \pi_n\}\rangle) = id$$

$$\begin{aligned}
\mathcal{U}_{DEP}((\langle\phi_1\rangle, \langle\phi_2\rangle), (\langle\phi_1'\rangle, \langle\phi_2'\rangle)) = \ & \text{let } \theta_1 = \mathcal{U}_{DEP}(\langle\phi_1\rangle, \langle\phi_1'\rangle) \\
& \quad \theta_2 = \mathcal{U}_{DEP}(\theta_1\langle\phi_2\rangle, \theta_1\langle\phi_1'\rangle) \\
& \text{in } \theta_2 \circ \theta_1 \\
\mathcal{U}_{DEP}(\langle\phi_1\rangle \to \langle\phi_2\rangle, \langle\phi_1'\rangle \to \langle\phi_2\rangle) = \ & \text{let } \theta_1 = \mathcal{U}_{DEP}(\langle\phi_1\rangle, \langle\phi_1'\rangle) \\
& \quad \theta_2 = \mathcal{U}_{DEP}(\theta_1\langle\phi_2\rangle, \theta_1\langle\phi_1'\rangle) \\
& \text{in } \theta_2 \circ \theta_1
\end{aligned}$$

$$\mathcal{U}_{DEP}(\langle\phi\rangle, \langle\beta\rangle) = \begin{cases} [\beta \mapsto \phi] & \text{if } \beta \text{ does not occur in } \phi \text{ or if } \beta \text{ equals } \phi \\ fail & \text{otherwise} \end{cases}$$

$$\mathcal{U}_{DEP}(\langle\beta\rangle, \langle\phi\rangle) = \mathcal{U}_{DEP}(\langle\phi\rangle, \langle\beta\rangle)$$

$$\mathcal{U}_{DEP}(\langle\phi_1\rangle, \langle\phi_2\rangle) = fail \text{ in all other cases}$$

Table 4.5: The unification algorithm $\mathcal{U}_{DEP}$ as used in table 4.3.

4.3 Extending the effect system for arrays

In this section we present an informal extension to the effect system that helps us handle the inference of conditional dependencies of models containing array expressions. We want to note that while this extension does provide useful insights on the dependence analysis of array expressions, it is not an essential part of the formal effect system and we will see in chapter 5 that we are not yet able to fully utilise the results of the extension.
Similarly to limitations that occur in the effect system designed by Gorinova et al. [11], we see that problems might arise when we use our effect system within the context of array operations. We can only express that some array expression e_A has one single set of dependencies ϕ_A, but different elements of the array might depend on different parameters.
This might cause problems when working with arrays: Take an array of expressions `CArray` $[e_1, \dots, e_n] = e_A$, so that for each element e_i we have $\Gamma \vdash e_i : \langle\{\pi_i\}\rangle\tau$. This array e_A could currently be typed as $\Gamma \vdash e_A : \langle\{\pi_1, \dots, \pi_n\}\rangle[\tau]$. For this example, we apply a head function: $e_B =$ `Apply Head` e_A. Ideally this new expression e_B would type so that $\Gamma \vdash e_B : \langle\{\pi_1\}\rangle\tau$, but there is no possibility to know that π_1 is the identifier that corresponds to the first element within the array e_A from its type.
Another problem stems from the fact that we often create arrays of parameters in our models by using the sugared `parameterA` function, as seen in section 3.4.4. This function is simply an application of `Generate` followed by `Sequence` on a single parameter constructor `Parameter`. Currently, our effect system would type this `Parameter` constructor to have effect $\{\pi\}$, using a unique identifier π. However, when we infer the effect of the array given by `parameterA`, it is important that its first element (obtained by applying `Head`) and the second element (obtained by applying `Tail` and then applying `Head`) have different effects. These expressions should not both have the effect $\{\pi\}$, because this would mean that we lose the element-wise dependency information. Ideally, we want to solve this problem without having to generate unique identifiers for each element, as this would require us to know the exact size of the array at compile time – which is known at that time in the *ProbProg* language, but is often not available at compile in other probabilistic programming languages – and would cause the speed of our inference algorithm to be dependent on the sizes of our parameter arrays.

In order to overcome these limitations, we present an informal extension of our effect system, together with an extension for the type inference algorithm $\mathcal{W}_{DEP}$. We adjust our types τ to introduce a new possible array type:

$$\tau ::= \tau_c^\phi \mid (\tau_1, \tau_2) \mid \tau_1 \to \tau_2 \mid m(\tau) \mid \alpha^\phi \mid [\tau]. \tag{4.18}$$

We now want to construct an indexing operation on these arrays, that specifically indexes their effect

annotations. For example, say we have an expression with the array type $[\langle\phi\rangle\tau]$. If we then index this type at index i, we interpret this indexing $[\langle\phi\rangle\tau](i)$ as an expression of type τ, where the effect annotation of τ shows that the expression depends on the ith element of each parameter array $\pi \in \phi$. We will denote such an effect as **index** π i, so that the expression has type $\langle\{\textbf{index}\ \pi\ i\}\rangle\tau$. This means that we extend our possible effect annotations:

$$\begin{aligned} \sigma &::= \emptyset \mid \{\pi'\} \cup \sigma \\ \pi' &::= \pi \mid \textbf{index}\ \pi'\ i \end{aligned} \tag{4.19}$$

As a result of this, we can also apply our indexing operation on type and effect parameters, which gives us $\alpha(i)$ or $\beta(i)$, for example.
In order to properly store how different array operations in *ProbProg* propagate the effects contained in the effect annotations, the index i that we use to index the array types can not just be any natural number. Instead, we give a specific definition for possible values of i:

$$i ::= \begin{cases} \iota & \text{for a symbolic value } \iota \\ c & \text{where } c \in \mathcal{N} \\ i + c & \text{where } c \in \mathcal{N} \\ \textbf{reverse}(i) & \\ \textbf{backpermute}(\phi, i) & \end{cases} \tag{4.20}$$

This is where our description of the extension of the effect system takes on an informal algorithmic form. As we can see, these indices can take on a variety of symbolic values that encode different index transformations. We do this, for example, because we should assume that we do not know the sizes N of arrays at compile time. Therefore, when we reverse some array, we cannot store that the new index of element i is now $N - i$ after reversing the array. Instead, we only store that a reversing operation has been performed at that point. Similarly, we do not know the exact index values that are contained within an array of indices when we perform a backpermute operation using that array, but we can still store that a backpermute has been performed on the indices of the other array.
Finally, there is the symbolic index value ι. We will see in section 4.3.1 that our extended inference algorithm heavily relies on this symbolic value. We use this ι to encode special array constraints during our type inference. For example, we can now encode the constraint $\beta_1(\iota) \supseteq \beta_2(\iota)$, using our new indexing operation. We interpret this constraint as follows: it means that for every possible integer index value j, we have that each $\beta_1(j) \supseteq \beta_2(j)$. Interpreting the ι value in this way also allows us to create more complex constraints together with our other possible index values. Say for example we have the constraint $\beta_1(\textbf{reverse}(\iota)) \supseteq \beta_2(\iota + 5)$. We can then interpret this constraint as $\forall j : \beta_1(N - j) \subseteq \beta_2(j + 5)$. Of course, as we do not know the size of arrays, we do not know N or every possible value for j at compile time. However, because of our symbolic storage of the information, we do not need to know these values at this time; we are still able to check if possible conditional independencies hold without knowing the exact size of an array.

4.3.1 Type inference

With this informal extension to our effect system we can give an updated version of the type inference algorithm $\mathcal{W}_{DEP}$, that is also able to infer conditional dependencies of models with arrays. Tables 4.6 and 4.7 show the extension to algorithm $\mathcal{W}_{DEP}$ for the array constructors and the array primitive operators respectively. Additionally, we extend the accompanying unification algorithm $\mathcal{U}_{DEP}$ with the cases for array types in table 4.8. We walk through these new cases in order to explain the extended algorithm.
Table 4.6 shows how we can infer the effects of the two constructors in the *ProbProg* syntax that construct arrays. The inference of the `CArray` constructor shows how we solve the problem described in the previous section: for a constant array of expressions, we want to be able to define different

effect annotations for each element. In the algorithm $\mathcal{W}_{DEP}$, we use our extended constraints to encode that each individual element of the newly constructed array has the correct effect annotation. When we instead construct an array using the `Generate` constructor, we assume that every element of the resulting array can be identified with the same effect annotation. This is a restriction that we enforce on the `Generate` constructor. This restriction holds in our sugared syntax, so we simply ensure that the user can only use the `Generate` constructor through usage of the sugared syntax. When this restriction holds, we can simply type the resulting array as in table 4.6. This solves the second problem introduced in the previous section: when we now use `parameterA`, the entire array is typed as $[\langle\{\pi\}\rangle]$ for some identifier π. Indexing this type at index i gives us the effect annotation $\{\textbf{index}\ \pi\ i\}$, which is exactly what we can then use as the unique effect annotation of an individual element of this array.

Table 4.7 shows the type inference of the different array primitive operators in *ProbProg*. Here we can see that the `Head` primitive operator indeed has the desired effect: our constraint encodes that the result of the `Head` operation is (at least) the first element of the input array. Similarly, we have other operators that introduce a transformation on the indices of an array, such as `Tail` or `Reverse`. In these cases we see that we use the index values that correspond to these operations ($i = \iota + 1$ for the `Tail` operator and $i = \textbf{reverse}(\iota)$ for the `Reverse` operator) in our constraints to correctly propagate the effect annotations through these array operations.
If we look at the cases for `Map` and `ZipWith`, we see how the effect constraints on arrays can propagate through functions when performing mapping operations. Notice that this shows that we generally should not interpret intermediate constraints by quantifying over the symbolic index ι: when we see constraints of the type $\beta_\alpha \supseteq \beta_A(\iota)$, it does not mean that for all possible indices j we have $\beta_\alpha \supseteq \beta_A(\iota)$ so that β_α would become the union set $\cup_j \beta_A(j)$. Instead, we use the symbolic index value of ι to propagate the effects through the function that is mapped over the array by representing each element as $\beta_A(\iota)$. Take the following example: the partially applied array operation $e =$ `Apply Map Add`. Our algorithm then returns $\mathcal{W}_{DEP}(\emptyset, e) = (\langle\beta_1\rangle \to \langle\beta_2\rangle, \theta, C)$, using a combination of the case for `Map` and the cases for `Add` and `Apply` in tables 4.4 and 4.3. Solving the constraints in C will then result in a set $\{\beta_2(\iota) \supseteq \beta_1(\iota),\ \ldots\}$. As we can see, the effect of each element within the resulting array contains at least the effect of the element of the input array at the same index. Instead of having to store this relationship as the set of constraints $\{\beta_2(1) \supseteq \beta_1(1), \ldots, \beta_2(N) \supseteq \beta_2(N)\}$, we store it efficiently in a single constraint.
The algorithm uses the same technique on the `Foldr` primitive operator: we use the constraints to propagate the effect of the function on the result of the folding operation. One difference from what we have seen with the `Map` and `ZipWith` operators is that the case for the `Foldr` operator introduces a constraint of the form $\beta_\alpha \supseteq \beta_A$, where β_A represents an array. This is a super-set relation where β_α is a super-set of the entire array that β_A represents; this is how we encode that this β_α is a super-set of every individual effect set contained in the array β_A.
Type inference on the `Backpermute` operator shows how we use the special symbolic **backpermute** index value. We use this value to store that some array has been indexed by a different array. As we mentioned in the beginning of this section, we cannot simply resolve this to a concrete index at compile time.
The inference for the `Sequence` operator is fairly simple, as this operator has no influence on the effect annotations contained within the array that it operators on.
The `Replicate` primitive operator represents another way to construct an array. As we can see, the inference for this operator is very similar to the inference on the `Generate` constructor, as this operator constructs arrays in the same fashion. Finally, we can also construct arrays using the `UnfoldrNM` operator. The type inference for this operator again shows how we can propagate the array effect annotations through a function. This is also slightly different from what we have seen before, as in this case we have a function that performs an unfolding operation instead of a folding or a mapping operation. We can see that this alters the way in which the effects flow through the function: in this case, our constraint enforces that each element of the resulting array depends on the result of the

function.

As before, once we have completed the type inference on a model using our algorithm $\mathcal{W}_{DEP}$, we must solve the constraints in the resulting constraint set C. However, resolving the constraints with the new possibilities for indexed constraints becomes a bit more difficult than the naive algorithm we have seen in equation 4.17. Say we have some set of constraints $\{\beta_1(\iota) \supseteq \{\pi\}, \beta_2(\iota+1) \supseteq \beta_1(\iota+1)\}$ and we want to know the dependencies of β_2. This means that would like to be able to infer that $\beta_2(\iota) \supseteq \{\pi\}$ as well. We therefore add an additional step to the naive algorithm, where we substitute the array indices if needed:

$$\begin{aligned} &\text{let } S = \{\beta\} \\ &\text{while } \exists x \in S \text{ so that } x \text{ is an annotation variable} \\ &\quad \text{do } S = \bigcup_{\beta' \in S} \{\phi \mid \{\beta' \supseteq \phi\} \in C\} \\ &\qquad \cup \bigcup_{\beta'(i_1) \in S} \{[f/\iota]\phi \mid \{\beta'(i_2) \supseteq \phi\} \in C, i_1 = [f/\iota]i_2\} \\ &\text{return } \bigcup_{\phi \in S} \phi \end{aligned} \tag{4.21}$$

This algorithm then gives us the correct solved set of dependencies of some effect variable β.

$$
\begin{aligned}
\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Carray}\ [e_1, \ldots, e_n]) = \ & \text{let } \beta \text{ be fresh} \\
& \quad (\langle\phi_1\rangle, \theta_1, C_1) = \mathcal{W}_{DEP}(\hat{\Gamma}, e_1) \\
& \quad \vdots \\
& \quad (\langle\phi_n\rangle, \theta_n, C_n) = \mathcal{W}_{DEP}(\hat{\Gamma}, e_n) \\
& \text{in } ([\langle\beta\rangle], \theta_n \circ \cdots \circ \theta_1, \\
& \quad ((\theta_{n-1} \circ \cdots \circ \theta_1)\{[\langle\beta\rangle](n) \supseteq \phi_n\}) \cup \ldots \\
& \quad \cup \{[\langle\beta\rangle](1) \supseteq \phi_1\})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Generate}\ f) = \ & \text{let } (\langle\phi\rangle, \theta, C) = \mathcal{W}_{DEP}(\hat{\Gamma}, f(\mathit{Finite}\ 0)) \\
& \text{in } ([\langle\phi\rangle], \theta, C)
\end{aligned}
$$

With the constraint that $\forall i$: $\mathcal{W}_{DEP}(\hat{\Gamma}, f(\mathit{Finite}\ i)) = (\langle\phi\rangle, \theta, C)$.

Table 4.6: The extension of algorithm $\mathcal{W}_{DEP}$ for dependency analysis with arrays.

$$
\begin{aligned}
\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Head}) = \ & \text{let } \beta_A, \beta \text{ be fresh} \\
& \text{in } (\beta_A \to \beta, id, \{\beta \supseteq \beta_A(0)\})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Tail}) = \ & \text{let } \beta_A, \beta \text{ be fresh} \\
& \text{in } (\beta_A \to \beta, id, \{\beta(\iota) \supseteq \beta_A(\iota + 1)\})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Reverse}) = \ & \text{let } \beta_A, \beta \text{ be fresh} \\
& \text{in } (\beta_A \to \beta, id, \{\beta(\iota) \supseteq \beta_A(\mathbf{reverse}(\iota))\})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Map}) = \ & \text{let } \beta_A, \beta_a, \beta_b, \beta \text{ be fresh} \\
& \text{in } (((\langle\beta_a\rangle \to \langle\beta_b\rangle, \beta_A) \to \beta, id, \{\beta_a \supseteq \beta_A(\iota), \beta(\iota) \supseteq \beta_b\})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Foldr}) = \ & \text{let } \beta_A, \beta_a, \beta_{b1}, \beta_{b2}, \beta_b, \beta \text{ be fresh} \\
& \text{in } (((\langle\beta_a\rangle \to \langle\beta_{b1}\rangle \to \langle\beta_{b2}\rangle, (\langle\beta_b\rangle, \beta_A)) \to \beta, id, \\
& \quad \{\beta_a \supseteq \beta_A, \beta_{b1} \supseteq \beta_b, \beta \supseteq \beta_{b2}\})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Zipwith}) = \ & \text{let } \beta_A, \beta_B \beta_a, \beta_b, \beta_c, \beta \text{ be fresh} \\
& \text{in } (((\langle\beta_a\rangle \to \langle\beta_b\rangle \to \langle\beta_c\rangle, (\beta_A, \beta_B)) \to \beta, id, \\
& \quad \{\beta_a \supseteq \beta_A(\iota), \beta_b \supseteq \beta_B(\iota), \beta(\iota) \supseteq \beta_c\})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Backpermute}) = \ & \text{let } \beta_A, \beta_B, \beta \text{ be fresh} \\
& \text{in } ((\beta_A, \beta_B) \to \beta, id, \{\beta(\iota) \supseteq \beta_A(\mathbf{backpermute}(\beta_B, \iota)))\})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Sequence}) = \ & \text{let } \beta_A, \beta \text{ be fresh} \\
& \text{in } (\beta_A \to \beta, id, \{\beta \supseteq \beta_A\})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{Replicate}) = \ & \text{let } \beta \text{ be fresh} \\
& \text{in } (\langle\beta\rangle \to [\langle\beta\rangle], id, \emptyset)
\end{aligned}
$$

$$\mathcal{W}_{DEP}(\hat{\Gamma}, \texttt{UnfoldrNM}) = \text{ let } \beta_1, \beta_2, \beta_3, \beta \text{ be fresh} \\ \text{in } (((\langle\beta_1\rangle \to (\beta, \beta_2), \langle\beta_3\rangle) \to \beta, id, \{\beta_1 \supseteq \beta_3, \beta(i) \supseteq \beta_2\})$$

Table 4.7: The extension of algorithm $\mathcal{W}_{DEP}$ for dependency analysis on primitive array operators.

$$\mathcal{U}_{DEP}([\langle\phi_1\rangle], [\langle\phi_2\rangle]) = \mathcal{U}_{DEP}(\langle\phi_1\rangle, \langle\phi_2\rangle)$$

$$\mathcal{U}_{DEP}(\beta_1(i_1), \beta_2(i_2)) = \text{ let } \theta_1 = \mathcal{U}_{DEP}(\beta_1, \beta_2) \\ \theta_2 = \mathcal{U}'_{DEP}(i_1, i_2) \\ \text{in } \theta_2 \circ \theta_1$$

$$\mathcal{U}'_{DEP}(i_1 + c_1, i_2 + c_2) = \begin{cases} \mathcal{U}'_{DEP}(i_1, i_2) & \text{if } c_1 \text{ is equal to } c_2 \\ fail & \text{otherwise} \end{cases}$$

$$\mathcal{U}'_{DEP}(\textbf{reverse}(i_1), \textbf{reverse}(i_2)) = \mathcal{U}'_{DEP}(i_1, i_2)$$

$$\mathcal{U}'_{DEP}(\textbf{backpermute}(i_1, \phi_1), \textbf{backpermute}(i_2, \phi_2)) = \text{ let } \theta_1 = \mathcal{U}_{DEP}(\phi_1, \phi_2) \\ \theta_2 = \mathcal{U}'_{DEP}(i_1, i_2) \\ \text{in } \theta_2 \circ \theta_1$$

$$\mathcal{U}'_{DEP}(c, c) = id$$

$$\mathcal{U}'_{DEP}(\iota, \iota) = id.$$

Table 4.8: The extended cases for the unification algorithm $\mathcal{U}_{DEP}$ for use with arrays.

4.4 Conclusion

We have now built an effect system that can generate a (representation of the) factor graph, and thus the conditional dependencies of a model. An effect system that can infer conditional dependencies of a model was already introduced before by Gorinova et al. [11], but our functional language helps us extend this system further. Our constructed factor graph contains the full dependence information, whereas the solution to this problem presented by Gorinova et al. can not infer conditional dependence rules that do not contain every parameter of the model. We are able to build the factor graph through our effects which store sets of dependencies: as we have seen in the typing judgements, the referential transparency of our functional language makes obtaining these dependencies very simple. Because of the many side-effects that might occur, inferring the factor graph with an effect space such as ours would lead to a more complicated static analysis in an imperative language.

Moreover, we are able to infer the dependencies of array elements in an efficient way, which is useful to extend the applications where the conditional dependencies are used that we will see in chapter 5 to models that contain array expressions. The efficient inference and storage of these dependencies is possible again largely due to our functional setting. As mentioned in section 4.2, the fact that array transformations in functional languages are performed by applying several functions that have a clear pattern through which dependence information spreads from the input to the output helps us track this information using our effect system. In contrast, most imperative languages would perform these same array operations using `for` or `while` loops containing different operations, which do not always have these clear patterns. The lack of these patterns heavily complicates the dependency analysis on arrays and might require the unrolling of these loops during the analysis to obtain the dependencies, which becomes inefficient for larger models.
As discussed before, our solution does cause the user to lose a small amount of expressivity when declaring their models, for example due to the fact that we can not let the user freely use the `Generate` constructor. However, this loss of expressivity generally does not prevent the user from implementing most commonly used models. We therefore believe that the benefit of knowing the conditional dependencies generally outweighs this loss.
One downside to our method to obtain conditional dependencies of models that contain arrays is that the extension to our effect system is still an informal extension. We have been able to describe and implement the extended $\mathcal{W}_{DEP}$ algorithm, but we were not able to capture this extension to the effect system in formal typing judgements. We were unable to do this due to the many symbolical values contained in the indices and the informal interpretation of the indexed array constraints in the algorithm. Further research could be done to mathematically formalise these informal parts of our algorithm and complete the effect system with correct typing judgements for the different array constructors and operators.

Chapter 5

Using conditional dependence information

Now that we have described our effect system that can reliably extract the conditional dependencies from a model in the *ProbProg* language, we want to examine the possible ways in which we can use the information on these conditional dependencies to improve the probabilistic programming workflow in *ProbProg*.
In this chapter we present a variable elimination program transformation on models in the *ProbProg* language that is able to eliminate (discrete) parameters from a model. Using this transformation, we will automatically and efficiently marginalise discrete parameters from any models that contain them, which allows us to support a wide range of useful models (as seen in section 3.5) while still supporting HMC and NUTS as inference methods. As mentioned in section 2.3, allowing users to use discrete parameters in their models without having to marginalise them out of their models by hand is a widely requested feature that is not yet available in most probabilistic programming languages and greatly improves the usability of a probabilistic programming language. Our method of applying an automatic program transformation is similar to how Gorinova et al. extend their *SlicStan* probabilistic programming language with support for discrete variables [11]. We both use a variable elimination algorithm implemented through a program transformation. However, the actual implementation of their transformation does differ from ours.
Finally, in this chapter we will also examine the limitations of our current approach for discrete parameter support and present ideas on how these limitations might be overcome in the future. Additionally, we will discuss some other possible applications that could be implemented in further research, in which the conditional dependency structure that is obtained through our effect system could also prove very useful.

5.1 Variable elimination

Variable elimination is an exact inference algorithm that is used for graphical models [48], which are models for which the conditional dependence structure is captured in a graph. In our case, we obtain this conditional dependence structure of a model using our effect system. We then want to use the variable elimination algorithm to efficiently compute the marginal distribution over the set of continuous variables from the target probability density of this model. That is, say we have our target density p, with continuous parameters $c_1, \ldots, c_n$ and discrete parameters $d_1, \ldots, d_m$, we would like to obtain $p(c_1, \ldots, c_n) = \sum_{d_1} \cdots \sum_{d_m} p(c_1, \ldots, c_n, d_1, \ldots, d_m)$. This marginalisation, when performed as a program transformation, eliminates any discrete parameters from a model and thus enable us to infer the model using HMC or NUTS, as seen in section 3.3.7.
One possibility is to implement this program transformation in a naive fashion, where we sim-

ply sum each discrete variable d_i over all its possible assignments, so that we naively calculate $\sum_{d_1} \cdots \sum_{d_m} p(c_1, \ldots, c_n, d_1, \ldots, d_m)$. However, as we have seen before in section 2.3, this solution is very inefficient. Say that each discrete variable can take on l possible assignments. This then means that the ith summation over parameter d_i has a complexity of $\mathcal{O}(l^i)$, leading to an exponential complexity of $\mathcal{O}(l^m)$ for the entire marginalisation. The variable elimination algorithm is able to greatly improve on this complexity.

We start by explaining how variable elimination works in theory, on any mathematical probability density p with parameters $x_1, \ldots, x_n$ [49]. The algorithm assumes that the density is given as a product of factors:

$$p(x_1, \ldots, x_n) = \Pi_k \phi_k(x_1, \ldots, x_n). \tag{5.1}$$

Of course, not every factor ϕ_k is dependent on all the parameters. We can therefore rewrite the density as

$$p(x_1, \ldots, x_n) = \Pi_{k \in K} \phi_k(\bar{x}_k), \tag{5.1 revisited}$$

where each k in K contains the indices for the parameters that are relevant for factor ϕ_k and this factor ϕ_k maps the set of relevant parameters $\bar{x}_k = \{x_i \mid i \in k\}$ to a value. This set of relevant parameters is called the scope of the factor. Using this new notation for our probability density, we can now define two different operations on the factors. The first is the product of two factors: say we have factors ϕ_{k_1} and ϕ_{k_2}, we then define their product $\phi_{k_1} \times \phi_{k_2}$ as

$$\phi_{k_1}(\bar{x}_{k_1}) \times \phi_{k_2}(\bar{x}_{k_2}) = \phi_{k_1 \cup k_2}(\bar{x}_{k_1 \cup k_2}). \tag{5.2}$$

As we can see, the scope of the product of these factors is the union of their individual scopes. The second operation that we define on factors is the marginalisation operation. If we have a factor $\phi(\bar{x}_{k_1}, \bar{x}_{k_2})$ that has two distinct sets of variables ($\bar{x}_{k_1}$ and $\bar{x}_{k_2}$) in its scope, where one only contains discrete variables, we can create a new factor by marginalising out this discrete set (in this case, $\bar{x}_{k_1}$):

$$\phi'(\bar{x}_{k_2}) = \sum_{x \in assign(\bar{x}_{k_1})} \phi(x, \bar{x}_{k_2}) \tag{5.3}$$

Here, $assign(\bar{x}_k)$ represents the set of all possible joint assignments to the variables in the set $\bar{x}_k$. With these operations, we can now give the variable elimination algorithm. For each variable x_i that we want to eliminate, we perform the following steps: first we obtain the product of every factor ϕ_k, where $i \in k$. Then, we marginalise our variable x_i out of this factor, so that we obtain a new factor ϕ' that is independent of x_i. Finally, we remove all ϕ_k where $i \in k$ from the probability density and replace them with the single new factor ϕ'.

We illustrate this process with a simple example. Say we have three parameters x_1, x_2, x_3 and a probability density with the following factorisation:

$$p(x_1, x_2, x_3) = \phi_{\{1,2\}}(x_1, x_2)\phi_{\{2,3\}}(x_2, x_3)\phi_{\{1,3\}}(x_1, x_3). \tag{5.4}$$

We eliminate parameter x_2. Performing the steps as explained, we obtain the product of the relevant factors

$$\phi_{\{1,2\}}(x_1, x_2)\phi_{\{2,3\}}(x_2, x_3). \tag{5.5}$$

We then marginalise our parameter x_2 out of this product:

$$\phi'(x_1, x_3) = \sum_{x_2 \in assign(x_2)} \phi_{\{1,2\}}(x_1, x_2)\phi_{\{2,3\}}(x_2, x_3), \tag{5.6}$$

and finally, we replace these factors in our original density with our newly created factor ϕ', giving us the marginalised probability density

$$p(x_1, x_3) = \phi_{\{1,3\}}(x_1, x_3)\phi'(x_1, x_3) = \phi_{\{1,3\}}(x_1, x_3) \sum_{x_2 \in assign(x_2)} \phi_{\{1,2\}}(x_1, x_2)\phi_{\{2,3\}}(x_2, x_3). \tag{5.7}$$

The complexity of the variable elimination algorithm when compared to the naive marginalisation is significantly improved. If we take our earlier example, where we wish to eliminate m discrete parameters and each of these parameters has l possible assignments, the naive marginalisation gave us a complexity of $\mathcal{O}(l^m)$, which is exponential with respect to the amount of discrete parameters. Variable elimination has a complexity of $\mathcal{O}(ml^{M+1})$ in this case, where M is the largest size of the scope of any intermediate factor ϕ' that is constructed during the algorithm [49]. Of course, M is generally (much) smaller than m, making variable elimination more efficient than the naive approach. However, this M is dependent on the order in which the different variables are eliminated. This order is therefore quite important, but because many successful heuristics exist that can establish an efficient order, we will consider finding this order a solved problem in this thesis.

Now that we understand how to execute the variable elimination algorithm on any possible probability density, we can show how it can be used on models produced by our language. We want to eliminate certain parameters from the target density of our models. Our models define a clear factoring of this target density, as each factor ϕ_k is introduced by an individual sample operation in the model. Through our analysis for conditional dependencies in the previous chapter, we know for each of these factors which parameters the factor depends on. In other words, we know for each ϕ_k what indices are contained in k. This means that through our analysis, we satisfy the assumptions that are made when executing variable elimination, namely that we can correctly factorise our target density.
Now that we know that we can perform variable elimination, we can also describe what a successful variable analysis on a model in our language should do. If we eliminate some parameter x_i from our model, we require a program transformation that rewrites our model so that the target density changes in the way that the mathematical description of variable elimination describes. This means that the resulting target density should have all factors that were dependent on x_i replaced by the marginalised product factor of these factors as we have seen before.

5.2 Program transformation

We now have a clear idea of what a variable elimination program transformation should accomplish in terms of its effect on the target density of a program. The goal is to translate this effect on the target density to a concrete transformation that we can execute on the first-order abstract syntax tree of a *ProbProg* model. To do this, we examine the individual steps of the variable elimination and describe how each step could be realised as a syntax transformation when we eliminate some parameter x_1:

Collecting the relevant factors. A factor in the target density directly corresponds to a sample operation in a model. This means that in our program transformation, we need to collect all the different sample operations that are dependent on the parameter x_1. Again, to determine if a sample operation is dependent on x_1, we can simply use the result of our effect system analysis designed in the previous chapter. For each sample operation of the form `Sample` e_y e_f, we store the expression `Apply` e_f e_y, which represents the factor that the sample operation contributes to the target density.

Removing the relevant factors. The factors that depend on x_i will need to be removed from the target density. This is, we have to remove the sample operations from the model. Additionally, we can also remove the `Parameter` constructor that defines our eliminated parameter, seeing that it will not be used in the inference of the model after the variable elimination.

Replacing the factors with the marginalised product. The final and most difficult step is to introduce the new factor into the target density. We do this by multiplying all the expressions stored in the first step (which corresponds to creating the product of the factors) and marginalising the parameter x_1 out of this new expression, which can be done by summing over the

product of factors for each possible assigned value for x_1. This step results in a new factor expression, which we add to the target density by creating a new sample operation in which we place the newly created factor expression. The new sample expression then needs to be correctly inserted into the program.

The final step is the most difficult step of the program transformation because we need to ensure that, when combining the different collected expressions and moving them into one new sample operation, we create a program that is both syntactically sound and has no unexpected semantic changes. We must be careful as to not move variables out of scope, incorrectly bind free variables or build incorrect syntax when moving expressions throughout the syntax tree.

5.2.1 Implementation

We achieve the three steps for eliminating a parameter in a single function, `moveUp`. The name of this function describes its functionality: in one recursive sweep through the abstract syntax tree we identify the constructor of the parameter and any sample operations that depend on it, delete them at their current locations and move them up in the syntax tree. We then combine them into the marginalised product factor in a single sample operation, which we insert back into the program.
The type of the `moveUp` function is:

```
1 moveUp :: (ParamId, ConDepInfo) -> MoveEnv env env -> OpenExpr env a
2                                  -> TypeM (OpenExpr env a, ParamFound env, Move env)
```

The first argument to the function is a tuple of a `ParamId` and a `ConDepInfo`. `ParamId` is the type of the parameter that we are trying to eliminate, which means that it contains the parameter identifier π as determined by the effect system in the previous chapter. `ConDepInfo` contains a set of resolved subset constraints on the effect sets of various types that we have seen in section 4.1.1 and therefore the conditional dependence information of the model. It is produced as a result of running our effect system as described in the previous chapter on the full model. Using the knowledge stored in `ConDepInfo`, we can determine if a sample operation is dependent on the parameter that we wish to eliminate.
We discuss the second argument of the function at a later point in this section. First, we examine the third argument and the output. The third argument is the first-order abstract syntax expression that we wish to rewrite. The returned output then consists of three components, of which the first is the resulting transformed abstract syntax of this expression. This resulting expression represents the original expression, but with the variable elimination transformation applied to it, for as far as that is possible at the point that `moveUp` is called. Therefore, when `moveUp` is called on the full model, the resulting expression will contain the transformed model, where the correct parameter has been eliminated. However, for any recursive calls of `moveUp` on parts of the model, this resulting expression might contain an intermediate result where most steps of the transformation have been applied, but some additional steps might still need to be applied to this intermediate result at some point higher within the syntax tree of the full model.
The three output elements are wrapped in a typing monad, `TypeM`, which is used to obtain various identifiers – for example, those for parameters and sample operations – that match those in `ConDepInfo` throughout the transformation. This is done by incrementing several natural numbers that are stored in the `TypeM` monad whenever we encounter a relevant expression. These natural numbers can then be used as the identifiers for these expressions. This method is also the method in which these identifiers are determined in our implementation of the effect system, so the generated identifiers will match up with the identifiers retrieved from the effect system stored in `ParamId` and `ConDepInfo`.
The other two elements of the output triple require some more explanation. We start with `ParamFound env`. The idea is that this expression stores if we have found the parameter that we wish to eliminate in the input expression. However, if we wish to marginalise our final product factor, we need to know what assignments are possible for this variable.

To allow the user to specify the possible assignment values for a discrete parameter which we want to eliminate, we introduce a new variant of parameter constraints:

```
data Constraint env n = Constraint { transform :: OpenExpr env (Double -> Double),
                                     invert :: OpenExpr env (Double -> Double),
                                     densityFactor :: OpenExpr env (Double -> Double)
                                   }
                      | Discrete (Array env n Double)
```

The `Discrete` constraint contains an array of possible assignments for a discrete variable. These assignments are stored in a `Double` format so that they match the type of the original parameter, but this is only one of the possible representations for discrete assignments. As with the original constraints, we have also implemented a higher-order syntax equivalent, as well as the conversion to the higher-order syntax. Additionally, we have defined sugared syntax for discrete parameters:

```
parameterD :: Expr String -> Array n Double -> Expr (ModelResult Double)
parameterD s vs = Parameter s (Just (Discrete vs))
```

This sugared syntax allows users to easily define discrete parameters. Now that we know where we can obtain the possible assignments for a parameter that we wish to eliminate, we do not only want to store if we have found the parameter in `ParamFound`, but we also want to store the possible assignment values. We therefore define `ParamFound` as

```
data  ParamFound env where
   Hidden :: ParamFound env
   Found  :: OpenExpr env (ModelResult (V.Vector n Double)) -> ParamFound env
```

Where `Hidden` indicates that we have not found the parameter and `Found` indicates that we have, together with the vector of possible assignments. We use a GADT to prevent any unnecessary complications that might arise by propagating the static length type of the vector (given by n). The vector of assignments is wrapped in the `ModelResult` monad, because it can contain monadic effects that we wish to maintain. For example, take the situation where the possible assignments of a discrete parameter are determined by the value of an input of the model:

```
              ...       #>
              inputReal #> \t  ->
              parameterD "s" ([1,t] :: Array 2 Double) #> \s -> ...
```

In this case, we must not discard this monadic effect. This is a very common situation, that occurs in change point models, for example, as we will see in section 5.3.1.

The third member of the output tuple of `moveUp`, of type `Move env`, contains the possible factor that depends on the parameter chosen for elimination that might be defined in the input expression. It is defined as follows:

```
type Move env = Maybe (OpenExpr env (ModelResult Double))
```

We can then return `Nothing` if the input expression does not contain any relevant factors, otherwise we return the product of the factors as a single value. This value is again wrapped in a `ModelResult` monad. Later in this section we show how this helps us move the factor value through the abstract syntax tree.

We can now define the second input argument of the `moveUp` function: this is an environment, similar to the valuation environment shown in section 3.2 that contains various `Move` expressions:

```
data MoveEnv env env' where
  Empty :: MoveEnv () env'
  Push  :: MoveEnv env env' -> Move env' -> MoveEnv (env, t) env'
```

Notice that this `MoveEnv` has two distinct nested environment tuples: `env` represents the structure of the `MoveEnv` environment. Its internal types are never used; instead, it stores the environment as the untyped DeBruijn index that is on top of the stack. The other environment, `env'` represents the free variables of every contained `Move` expression. As in section 3.2, we define a way to add `Move` expressions to this environment and to project `Move` expressions out of the environment:

```
add :: MoveEnv env env' -> Move env' -> MoveEnv (env, t) env'
add env qt = env `Push` qt

prj :: Idx env t -> MoveEnv env env' -> Move env'
prj ZeroIdx        (Push _   qt)   = qt
prj (SuccIdx idx) (Push env _)    = prj idx env
```

This environment will help us maintain the `Move` factors of various free variables during the `moveUp` transformation. Again, we see that the types t that are stored in the first environment `env` and its accompanying indices are never specified: they represent untyped DeBruijn indices.
Now that we have an idea of the basic functionality, input and output of the `moveUp` function, we can describe the transformation done by the `moveUp` function for each element in our abstract syntax.

5.2.2 Invariants

We start our description of the `moveUp` transformation by showing its effect on constructors in the abstract syntax that do not need to change under the effect of the transformation. These are given by:

```
moveUp _ _ (Input i)      = return (Input i, Hidden, Nothing)
moveUp _ _ (Con c)        = return (Con c, Hidden, Nothing)
moveUp _ _ Nil            = return (Nil, Hidden, Nothing)
moveUp _ _ (PDF p)        = return (PDF p, Hidden, Nothing)
moveUp _ _ (PrimFunc pf) = return (PrimFunc pf, Hidden, Nothing)
```

We know that constants, input parameters and primitive functions (including the primitive probability distributions) can neither contain a parameter that we wish to eliminate or a sample statement that might define a factor that we wish to move. We can therefore leave these expressions as they are, returning empty values for the `Move` and `ParamFound` outputs.

5.2.3 Parameters

`Parameter` constructors can be divided into two cases. In the first case, we have a parameter with the `Discrete` constraint. These are handled as follows:

```
moveUp (s,_) _ p@(Parameter _ (Just (Discrete d))) =
    do t <- freshP
       if t == s
          then return (ScopeVar 0, Found (Return d), Nothing)
          else return (p, Hidden, Nothing)
```

We use a function `freshP`, which retrieves the correct identifier π for this parameter from the `TypeM` monad. We compare its result to the identifier of the parameter that we wish to eliminate: if the identifiers match, we have found the parameter. This means that we insert the array of possible values for the parameter into our resulting `ParamFound`. There is no factor that we want to move contained in this expression, so we return the empty `Move` expression. Notice that the expression that we return is some element of the abstract syntax that we have not seen before. We will explain this new constructor in section 5.2.6 and use it to turn the part of the syntax tree in which we discover the parameter that needs to be eliminated into a function into which we can later insert the different assignments for this parameter. It will become clear in section 5.2.8 how this helps us obtain the correct value for the final marginalised factor that we wish to add to the model.
In the second case, the type identifiers show that a `Parameter` constructor does not define the parameter that we wish to eliminate. We can therefore treat the parameter as an invariant expression. The same holds for parameters that are not marked as discrete by their constraint:

```
moveUp _ _ p@(Parameter _ _) = do t <- freshP
                                  return (p, Hidden, Nothing)
```

In this case, we still have to execute `freshP`, as this function has to increment the identifier stored in `TypeM`, but as we do not have to check if this identifier corresponds to the parameter that we wish to eliminate, we can safely discard t.

5.2.4 Sampling

When we encounter a sample operation, we need to check if it defines a factor that is dependent on the parameter that we want to eliminate. If it is, we need to relay that information upwards by storing the factor in the output `Move`. This results in the following implementation:

```
moveUp (s,c) env (Sample y f) =
    do t <- freshF
       if factorOf s t c
          then return (Return Nil, Hidden, Just (Return (Apply f y)))
          else return (Sample y f, Hidden, Nothing)
```

We start by mentioning that we do not have to call `moveUp` on the internal expressions of the `Sample` constructor. The reason for this is that expressions that are interesting for the `moveUp` transformation, such as a parameter or another sample statement, always have effects that are represented within the `ModelResult` monad and the inner types of the `Sample` constructor can never contain this monad. Next, we use the `freshF` function, which generates the correct type identifier for this sample operation (similar to how `freshP` works for parameters). We then use the function `factorOf`, which determines if this sample operation is dependent on the parameter that will be eliminated, using the type identifier for the sample operation, the identifier for the parameter and the information stored in the input `ConDepInfo`. If this is the case, we return the factor represented by the sample operation in the `Move` output and replace the sample operation with `Return Nil`, an expression of the same type as the original expression but with no effect on the target density, thus removing the factor from the density. If the sample operation is not dependent on the parameter, we can simply return the operation as it was before. In both cases we can ignore the `ParamFound` output and return `Hidden`, as a sample statement will never contain the relevant parameter constructor.

5.2.5 Let statements

Now that we have seen how the `moveUp` function collects the `ParamFound` and `Move` data from `Parameter` and `Sample` constructors, we show how this data is then moved through the abstract syntax tree. We start by examining how `moveUp` handles a `Let` construct:

```
moveUp sc env (Let e1 e2) =
    do (e1', b1, e1_es) <- moveUp sc env e1
       (e2', b2, e2_es) <- moveUp sc (incrVarsEnv (add env e1_es)) e2
       let es = case e2_es of
                  Nothing      -> Nothing
                  Just e2_es' -> Just (Let e1' e2_es')
       let b  = case b2 of
                  Hidden     -> b1
                  Found b2' -> b1 `orPF` Found (Let e1' b2')
       return (Let e1' e2', b, es)
```

First, we recursively call `moveUp` on the internal expressions $e1$ and $e2$. Notice that when we apply `moveUp` to $e2$, we add the `Move` result `e1_es` to the input `MoveEnv` environment. This allows us to retrieve this `Move` result when we encounter the bound variable in the body of the `Let` constructor, $e2$:

```
moveUp _ env (Var idx) = return (Var idx, Hidden, prj idx env)
```

When passing the `MoveEnv` to the `moveUp` call on $e2$, we must remember from the type signature of `moveUp` that we enforce that each element of the `MoveEnv` should have the same variable environment as $e2$. We want to enforce this to make sure that we never accidentally rebind any variables during the program transformation. In this case, if `e1 :: OpenExpr env a`, we know that `e2 :: OpenExpr (env, t) a` (this follows from the usage of the `Let` constructor). Just adding the `Move` result `e1_es` to the `MoveEnv`, gives us `add env e1_es :: MoveEnv (env, t) env`, meaning that the number of elements in the environment correctly corresponds to the variable environment of $e2$, but the variable environment of each `Move` in the `MoveEnv` does not.

In order to make sure that the variable environments match, we call `incrVarsEnv` on the environment. This function goes through each `Move` inside of the environment and increments the DeBruijn indices of any variables contained in this `Move` by one. This ensures that the variables in the environment correctly match their original bindings when used in the `moveUp` call on $e2$. We therefore have `incrVarsEnv (add env e1_es) :: MoveEnv (env, t) (env, t)`.
After applying `moveUp` on both $e1$ and $e2$, we have to combine both the `Move` and `ParamFound` outputs correctly. First, for the final `Move`, we can simply look at the `Move` output `e2_es` of the body of the `Let` constructor. If this is equal to `Nothing`, we can return `Nothing`. Otherwise, we have to return the resulting `Move` expression, but we add a `Let` binding to it, which binds $e1'$. The reason for this is that if we do not add this binding, we might pull any variables that refer to $e1'$ inside of the `Move` expression out of their scope. Notice that we do not have to use the `Move` result `e1_es` anymore, as we have added it to the environment earlier, meaning that if it was relevant, it would reappear in `e2_es`.
Finally, we combine the resulting `ParamFound` expressions in a similar way. The function `orPF` that is used to do this combines two `ParamFound` expressions as follows:

```
1 orPF :: ParamFound env -> ParamFound env -> ParamFound env
2 orPF Hidden Hidden       = Hidden
3 orPF (Found p) Hidden    = Found p
4 orPF Hidden (Found p)    = Found p
5 orPF (Found _) (Found _) = error "Could not complete variable elimination:
6                                  Duplicate discrete parameter found in program."
```

Notice that the last case throws an error: this is intended behaviour, as there should never be a situation in which we encounter the same parameter in the abstract syntax tree twice. As with the combining of the `Move` expressions, we again add a `Let` statement to $b2$ if needed, so that we do not move any variables contained within the `ParamFound` expression out of scope. The final output of the `moveUp` function is then the adjusted `Let` expression, together with these combined `Move` and `ParamFound` expressions.

5.2.6 Functions

The definition of `moveUp` of a function constructed using the `Func` constructor is similar to what we have seen for the `Let` constructor, but there are some important differences:

```
1 moveUp sc env (Func f) =
2     do (f', b, f_es) <- moveUp sc (incrVarsEnv (add env Nothing)) f
3        let f_es'' = case f_es of
4                          Nothing    -> Nothing
5                          Just f_es' -> Just (unscope f_es')
6        let b''    = case b of
7                          Hidden    -> Hidden
8                          Found b' -> Found (unscope b')
9        return (Func f', b'', f_es'')
```

As with the `Let` constructor, we start by recursively applying `moveUp` to the body expression of the `Func` constructor. When we did this for the `Let` constructor, we added the `Move` expression that we obtained from the bound expression in the constructor to our `MoveEnv` when calling `moveUp` on the body of the constructor, so that this `Move` expression would be retrieved when we encounter the bound variable in the body. Ideally, we would do this for the `Func` constructor as well and add the `Move` expression that corresponds to the variable bound to `Func` to the `MoveEnv`. However, we were not able to obtain this `Move` expression at this point: it would need to be obtained from the potential argument that the function built with this `Func` constructor is eventually applied to. Therefore, we add `Nothing` to the `MoveEnv` so that its environment has the correct type, but this solution is far from ideal and introduces some limitations to the `moveUp` algorithm, which we will see in section 5.2.9. Finally, we increment the `MoveEnv`, as we did when handling the `Let` constructor.
As with the `Let` constructor, we are not able to simply return the `Move` and `ParamFound` expressions that we get from calling `moveUp` on the internal expression, as this might mean that we move variables

within these expressions out of their scope. For the Let constructor, the variables that are at risk of being moved out of scope are the variables that binds to this Let constructor. Therefore, applying the Let constructor with its binding expression to the Move and ParamFound expressions establishes the correct scope in these expressions and prevents the variables from moving out of this scope. For the Func constructor, the variables that are at risk of being moved out of scope are also the variables bound to the constructor. However, unlike in our earlier solution, we cannot simply apply the Func constructor to Move or ParamFound, as these expressions cannot have a function type. A more complicated solution is required.
We introduce an addition to our abstract syntax that is only used in the intermediate representations formed during the program transformation:

```
data OpenExpr env a where
    ScopeVar  :: Int -> OpenExpr env a
```

This ScopeVar represents a variable that has temporarily been moved out of its original scope. The goal is then to replace the variables that move out of scope in Move and ParamFound with this ScopeVar, so that we can give these expressions the correct variable environment and temporarily move the variables out of scope. When we then, at a later point, encounter the Apply constructor where the argument for the function constructed by Func is introduced, we can then fix the scoping in a similar way as we have seen with the Let constructor: we turn the relevant instances of ScopeVar back into a variable that is placed on top of the environment stack and then consecutively apply the Func constructor and then the Apply constructor with its argument to the Move and ParamFound expressions, which correctly reestablishes the scope for the variables.
When moving the Move and ParamFound expressions upwards out of scope, we might encounter more Func constructors before we encounter the Apply constructor that can establish the scope. The integer k stored in a ScopeVar therefore indicates through how many Func constructors it has been moved: when an existing ScopeVar is moved up through a Func constructor, k is decreased by one (so, $k \leq 0$). We see this in action when we look at the function unscope: this function takes an expression and replaces the variables within it that are bound to the top of the environment stack with ScopeVar 0. It also changes the DeBruijn indices of the other variables to move them up one position on the stack, so that the top variable, which has become a ScopeVar, is eliminated from the environment. It decreases the integer in every already existing ScopeVar in the expression by one, thus indicating that these variables have been moved through another function constructor Func. We apply this unscope operation to both the Move and ParamFound expressions. Again, this means that we might temporarily move some variables out of their scope.

We now show the definition of moveUp on the Apply constructor, as this explains how we use the ScopeVar variables to reconstruct the Move and ParamFound expressions with the correct scope.

```
moveUp sc env (Apply f x) =
    do (f', bf, f_es) <- moveUp sc env f
       (x', bx, x_es) <- moveUp sc env x
       let es = case f_es of
                  Nothing    -> x_es
                  Just f_es' -> case rescopeStrict f_es' of
                                  Nothing     -> combineMove (Just f_es') x_es
                                  Just f_es'' -> combineMove
                                                     (Just (Apply f_es'' x')) x_es
       let b  = case bf of
                  Hidden    -> bx
                  Found bf' -> case rescopeStrict bf' of
                                 Nothing   -> Found bf' `orPF` bx
                                 Just bf'' -> Found (Apply bf'' x') `orPF` bx
       return (Apply f' x', b, es)
```

After calling moveUp on the internal expressions of the constructor, we now want to correctly combine the Move and ParamFound results of the function and the operand, before propagating them upwards. This includes potentially reconstructing the scope of a Func constructor, as we mentioned earlier in

this section. To do this, we introduce a counterpart to the `unscope` function: the `rescope` function. This function takes its expression argument, puts a new variable on the variable environment stack and changes all `ScopeVar(0)` variables inside of the expression to this new variable. (Recall that the `ScopeVar(0)` variables are the `ScopeVar` variables that have been introduced for the most recent `Func` constructor.) The function also increments the integers inside of other `ScopeVar` variables in the expression by one and finally wraps a `Func` constructor around the changed expression before returning it. This function will therefore bring any variables that were pulled out of scope in an earlier stage back into the scope of this new `Func` constructor. In the `moveUp` function on the `Apply` constructor, we specifically use `rescopeStrict`. This function only returns the expression that results from applying `rescope` if there were any `ScopeVar(0)` variables in the original expression and it returns `Nothing` otherwise.

We can now explain how the `moveUp` function on the `Apply` constructor builds the resulting `Move` and `ParamFound` expressions: using `rescopeStrict`, we check if there is a `ScopeVar(0)` in the expressions obtained by applying `moveUp` on the function f. If so, we know that we have moved a variable in this expression that should be bound to this `Apply` constructor out of scope. The `rescopeStrict` function then applies `rescope`, which transforms the relevant `Move` or `ParamFound` into a function, where the variable that was moved out of scope is bound to this function. Finally, we then apply the `Apply` constructor together with its operand on this rescoped expression. This gives us the resulting `Move` and `ParamFound` with the correct scope.

We then need to combine these results with the results of calling `moveUp` on the operand x, as this operand may also contain relevant `Move` or `ParamFound` expressions. This is done either by calling `orPF` for the `ParamFound` expression, or `combineMove` for the `Move` expression. This `combineMove` function allows us to combine the factors stored in two `Move` expressions into a single factor as follows:

```
1 combineMove :: Move env -> Move env -> Move env
2 combineMove Nothing Nothing     = Nothing
3 combineMove (Just e1) Nothing   = Just e1
4 combineMove Nothing (Just e2)   = Just e2
5 combineMove (Just e1) (Just e2) = Just $ e1 `Bind` Func
6                                   (incrVars e2 `Bind` Func
7                                   (Return (Apply (PrimFunc Mul)
8                                   (Pair (Var ZeroIdx) (Var (SuccIdx ZeroIdx))))))
```

In the situations where we only have one factor stored in the input `Move` expressions, we simply return this single factor. If both `Move` expressions contain a factor, we can combine the two factors into one by multiplying them and then returning the result, as seen in the final four lines of the function. This operation seems complicated in the code, but this is mostly because it is defined purely in abstract syntax elements without any sugaring.

5.2.7 Control Flow

Next, we look at applying `moveUp` on the `If` constructor:

```
1 moveUp sc env (If c e1 e2) =
2     do (e1', b1, e1_es) <- moveUp sc env e1
3        (e2', b2, e2_es) <- moveUp sc env e2
4        return (If c e1' e2', b1 `orPF` b2, moveIf c e1_es e2_es)
5   where moveIf :: OpenExpr env Bool -> Move env -> Move env -> Move env
6         moveIf _ Nothing Nothing          = Nothing
7         moveIf c (Just es') Nothing       = Just (If c es' (Return (Con (LReal 1))))
8         moveIf c Nothing (Just es')       = Just (If c (Return (Con (LReal 1))) es')
9         moveIf c (Just es1') (Just es2') = Just (If c es1' es2')
```

As always, we recursively apply `moveUp` to the internal expressions of the constructor. However, we do not have to apply it to the condition of the `If` constructor, because we know that this condition cannot contain any relevant expressions. This is because, as we have seen before when we discussed the `Sample` constructor, the type of the condition cannot contain the `ModelResult` monad. Combining

the resulting transformed expressions and the ParamFound expressions is simple, but when we combine the resulting Move expressions, we must make sure that the factor stored in the output Move expression includes the If statement if this is needed.
This is where we use the moveIf function. If the Move results of branches of the If constructor are not empty, we must make sure that the factor stored within them is only used in the target expression if the condition holds. If both branches have a non-empty Move result, we can simply apply the If constructor with these resulting factors as branches. If only one branch has a non-empty Move result, we must make sure that the other branch has no effect on the target density. We therefore apply the If constructor and return the neutral element 1 as that respective branch, with the non-empty Move result as the other branch.

5.2.8 Monads

The moveUp function is defined on the Return constructor as follows:

```
moveUp sc env (Return a) = do (a', b, a_es) <- moveUp sc env a
                              return (Return a', b, a_es)
```

We simply apply the moveUp function to the internal expression and then propagate the results upwards in the abstract syntax tree.
The definition of the moveUp function on the Bind constructor is a special case. We know that the variable elimination algorithm requires us to, at one point, marginalise the factor that we have collected throughout the syntax tree using the Move expressions and add it to the target density of the model. Because we have to use a sample operation to add the factor and sample operations are monadic, we want to insert this new sample operation at the location of the Bind constructor. Specifically, we know that in our models we will have some Bind a f, where a contains the parameter that we wish to eliminate and f contains various sample operations in which we use this parameter. This is the Bind constructor where we would like to insert the sample operation for our new factor. We therefore handle the Bind constructor as follows:

```
moveUp sc env (Bind a f) =
do  (f', bf, f_es) <- moveUp sc env f
    (a', ba, a_es) <- moveUp sc env a
    case ba of
      Found vs -> case combineMove f_es a_es of
                    Nothing  -> return
                                (Bind (Apply (rescope a') (Return (Con (LReal 0)))) f',
                                Hidden, Nothing)
                    Just es' -> let fac = makeFactor vs es' a'
                                    b   = Bind (Apply (rescope a')
                                                (Return (Con (LReal 0)))) f'
                                 in return (Bind fac (rescope b), Hidden, Nothing)
      Hidden   -> let es = case f_es of
                             Nothing    -> a_es
                             Just f_es' -> case rescopeStrict f_es' of
                                             Nothing     -> combineMove
                                                              (Just f_es') a_es
                                             Just f_es'' -> combineMove
                                                              (Just (Bind a' f_es'')) a_es
                      b  = case bf of
                             Hidden    -> ba
                             Found bf' -> case rescopeStrict bf' of
                                            Nothing   -> Found bf' `orPF` ba
                                            Just bf'' -> Found (Bind a' bf'') `orPF` ba
                  in return (Bind a' f', b, es)
```

We start by recursively calling moveUp on the internal elements. We then have two cases: the resulting ParamFound either indicates that a contains the parameter that we wish to eliminate, or that it does not. If it does not, we see that we encounter a situation that is familiar to what we have seen in the definition of moveUp on the Apply constructor. Just as with that constructor, there might

be a possibility that the Move or ParamFound expressions resulting from calling moveUp on f might contain some ScopeVar(0) variables that were bound to this Bind constructor. We therefore use the same combination of rescopeStrict, combineMove and orPF that we used for the Apply constructor in order to apply the Bind constructor correctly when needed before we return the new Move and ParamFound expressions.
If a does contain the parameter we wish to eliminate, we know that any possible factors containing the parameter must be in a or f: as the parameter is introduced in a, only a and f are within its scope. We again have two possible situations. Either combineMove f_es a_es returns Nothing, meaning that f and a contain no relevant factors. In this case, we do not have to add a new sample statement and only have to remove the parameter from the model. Remember that when we applied moveUp to the parameter that we want to eliminate in section 5.2.3, we replaced it with a ScopeVar in the transformed expression. This means that if we now rescope this transformed expression a', we obtain a function into which we can insert values for the eliminated parameter. Because we know that the parameter is never used in any sample expressions contained in a or f, we can simply return the Bind statement, but with the parameter replaced by the constant value 0. As the parameter does not occur in a or f, we can safely substitute it with any value without changing the semantics of the Bind statement. We choose the value 0 because it matches the original Double type of the parameter, which prevents any type errors in the *Haskell* compiler.
The other case is that we do have accumulated relevant factors through the resulting Move expressions. We can still return the Bind statement with the constant value 0 inserted for the parameter (we know that we have removed all the sample operations that use the parameter from a and f), but in addition to this we also need to insert the sample operation for the new factor. This operation is constructed using the makeFactor function:

```
1  makeFactor :: OpenExpr env (ModelResult (V.Vector n Double))
2  -> OpenExpr env (ModelResult Double) -> OpenExpr env a -> OpenExpr env Model
3  makeFactor vs m p =
4      let pf   = rescope (Apply (rescope m) p)
5          ds   = vs `Bind` rescope (Apply (PrimFunc Sequence)
6                                     (Apply (PrimFunc Map) (Pair pf (ScopeVar 0))))
7          add = Func (Func (Apply (PrimFunc Add)
8                             (Pair (Var ZeroIdx) (Var (SuccIdx ZeroIdx)))))
9          sum = Apply (PrimFunc Foldr) (Pair add (Pair (Con (LReal 0)) (Var ZeroIdx)))
10         samp = Func (Sample sum (Func (Var ZeroIdx)))
11     in  ds `Bind` samp
```

This function takes our vector of assignments for the parameter vs, the accumulated factor that has been moved upwards m and the transformed expression on the left-hand side of the bind p, that once contained the parameter, but in which this parameter has been replaced with a ScopeVar. With these three inputs, we then construct the final marginalised factor. We first create pf, which is a function of type OpenExpr env (ModelResult Double → ModelResult Double), that maps parameter assignments to the correct value of the factor. It is obtained through the function composition of $pf = m \circ p$, where we rescope the functions where needed. Next, this function is mapped over the possible assignments in vs and the results of this map are sequenced, resulting in the ds expression. Using add and sum, we can then sum over this ds vector, obtaining the final marginalised factor. We use a sample operation defined in samp to add this factor to our target density.
This sampling operation returned by makeFactor can then be inserted in our transformed Bind expression. This then concludes our variable elimination: we have removed the parameter and its relevant factors and replaced these factors in the target expression with the marginalised product factor.

5.2.9 Limitations

There are some limitations to the current design of the moveUp function. In this subsection we discuss each of these limitations and show in which situations they might introduce problems when applying the transformation.

As mentioned before in section 5.2.6, we have to resort to adding `Nothing` to the `MoveEnv` environment when calling `moveUp` on the internal expression of a `Func` constructor. This leads to some limitations when evaluating certain programs. Take the following model:

```
model = Let (Func (\x -> x <~ normal 0 2)) $ \f ->
            parameter "x1" #> \x1 ->
            parameterD "x2" ([0, 1] :: Array 2 Double) #> \x2 ->
            x1 <~ normal 0 1 #-
            Apply f x2
```

Say we wish to eliminate parameter $x2$. In the first line of the model, we have a sample statement that eventually uses this parameter. This sample statement seems to lie out of the scope of the transformation, as we only move up sample statements within the expressions inside of the bind where the parameter is introduced (in the fifth line). However, because we have added the `Move` expression that contains the factor for this sample operation to the `MoveEnv` environment when calling `moveUp` on the accompanying `Let` constructor, we know that the sample operation will still be moved upwards through the transformation, as we re-obtain this `Move` expression when calling `moveUp` on the variable f later.
Now we take the same model, but replace the `Let` constructor with a structure where we apply the body as a function to its operand.

```
model = Apply (Func (\f ->
            parameter "x1" #> \x1 ->
            parameterD "x2" ([0, 1] :: Array 2 Double) #> \x2 ->
            x1 <~ normal 0 1 #-
            Apply f x2
        )) (Func (\x -> x <~ normal 0 2))
```

We are unable to add the aforementioned `Move` expression to the `MoveEnv` environment in this case, as we always add `Nothing` when calling `moveUp` on the `Func` constructor. Because of this, the sample operation described on the sixth line will only be removed, without being included in the new marginalised product factor that we create.
This means that models of this form – that have a relevant `Sample` statement that is located inside of a function argument, but outside of the scope of the `Bind` operation that introduces the parameter that needs to be eliminated – will not produce the intended result when `moveUp` is called on them.
There are several solutions for this. One solution is to hide the `Apply` constructor from the user. The sugared syntax of the *ProbProg* language provides enough functions to create plenty of models and the example model on which the transformation fails does not resemble a program that a user would generally write. A second solution is to find a way to pass the operand of an `Apply` constructor into the `Func` constructor so that it can be added in the `MoveEnv` environment at that point. This could mean that we would have to expand the `Move` datatype in order to encode unapplied functions, for example by including expressions of type `a` → `Move env` as a possible `Move` expression. We were unable to fully explore this solution due to time constraints, but later in this section we see that this idea might also solve other limitations in the transformation.

A second limitation occurs when we try to define `moveUp` on the `Pair` constructor:

```
moveUp sc env (Pair e1 e2) =
    do (e1', b1, e1_es) <- moveUp sc env e1
       (e2', b2, e2_es) <- moveUp sc env e2
       return (Pair e1 e2, b1 `orPF` b2, combineMove e1_es e2_es)
```

Combining the two different elements in the pair using `combineMove` and `orPF` seems intuitive, but this might cause problems when we look at the following example:

```
model = parameter "x1" #> \x1 ->
        parameterD "x2" ([0, 1] :: Array 2 Double) #> \x2 ->
        x1 <~ normal 0 1 #-
        fst (Pair (x2 <~ normal 0 1) (x2 <~ normal 0 1))
```

In this case, we see that the sample operation in the second element of the pair defined in the fourth line does not contribute to the target density of the model. However, when we run `moveUp` on this model to eliminate the $x2$ parameter, our definition of `moveUp` on the `Pair` constructor causes us to incorporate this sample operation into our new factor anyway. This means that the final target density after variable elimination is incorrect.
This problem is caused by the fact that we can not encode the `Move` expression that we retrieve from calling `moveUp` on a `Pair` constructor in such a way that applying the `Fst` primitive operator only retrieves the `Move` result of the first element. If our `moveUp` algorithm could be adjusted so that this would be possible, this limitation would no longer exist. We believe that this would require some extension of the `Move` datatype. This extension should enable that we can encode a product of two `Move` expressions in some way, but also that we could encode the effect that a primitive function might have on the `Move` result of some expression when it is applied to it. Ideally, this extension can be combined with the earlier extension for the `Move` datatype that we have suggested for the limitations with the `Func` constructor. The underlying idea is similar; we wish to somehow store and apply the effect that functions have on the factors that we are moving through the abstract syntax tree.

Finally, we have not defined the `moveUp` function for the array constructs of our language, even though we do have the conditional independence information on these arrays through our static analysis. The reason that these array constructs are incompatible with the current definition of our `moveUp` becomes clear if we look at some useful array operators in our sugared syntax.
An array of parameters, for example, will be defined by the user by using the `parameterA` function. As we have seen in section 3.4.3, these arrays are generated using the `Generate` primitive function. This means that if we were to marginalise our parameter out of the newly introduced factor as in section 5.2.8, we cannot simply insert the given possible assignments for a discrete parameter into the location of the `Parameter` constructor. If we would, we would assign this single assignment value to every parameter in the entire array. Instead, we would need a way to rewrite such constructs so that all possible combinations of parameter assignments within the array are assigned.
Secondly, if we look at some of our example models that contain arrays in section 3.5, we see that we often use the `modelFor` functions to map a sample operation over an array of parameters. This results in a similar issue to what we have seen with the `Pair` constructor, where we cannot store the effect that this `modelFor` function might have on the target expression. Therefore, when we move this sample operation upwards out of the application of the `Map` and `Sequence` primitive operations in the `modelFor` function, we lose the syntactic effect of these primitive operators on this sample operation. This means that in the final marginalised product factor, the factors that the `Map` primitive operation creates for each element of the array will not be incorporated correctly: we only move a single factor upwards in the model, whereas applying `modelFor` with a sample operation on an array represents multiple factors (one for each array element).
Resolving these issues for the array constructs of the languages will likely be more complicated than resolving the earlier two limitations. However, we do believe that the core of this issue still lies with the fact that we need some representation of the effects that functions within our language could have on the factors stored in the `Move` expressions. Being able to correctly determine, store and apply these effects might be the key to solving the current limitations of the `moveUp` algorithm. Looking into the possibilities for such a representation might therefore be a good starting point for any further research on the implementation of this transformation.

5.3 Support for discrete parameters

Now that we have fully described the `moveUp` function and its limitations, we are able to eliminate a single discrete parameter from a (supported) model as follows: given the parameter that we want to eliminate, we run the static analysis as described in the previous chapter. Next, we call the `moveUp` function on the full model, passing in the results of the static analysis, the identifier for the parameter and an empty `MoveEnv` environment. This function will then return the transformed expression where

the given parameter has been eliminated. This process can then be repeated on this new expression for any other discrete parameters that we also want to eliminate. This concludes our implementation of the variable elimination algorithm.

5.3.1 Examples

To finish this section, we give some examples of the transformation working on actual models. We start with a simple example where we have two parameters. One parameter is continuous and the other is a discrete parameter with two possible assignments.

```
model = parameter "x1" #> \x1 ->
        parameterD "x2" ([0, 1] :: Array 2 Double) #> \x2 ->
        x1 <~ normal 0 1 #-
        x2 <~ normal 0 2
```

When we translate the sugared representation of this example model to the first-order abstract syntax, the model is given as

```
Parameter(LString "x1") `Bind`
Func (Parameter(LString "x2") `Bind`
Func (Var(1) <~ NormalPDF(LReal 0.0, LReal 1.0) `Bind`
Func (Var(1) <~ NormalPDF(LReal 0.0, LReal 2.0))))
```

Some details, such as the constraints of the parameters have been left out to improve readability. We can then apply our variable elimination algorithm in order to eliminate the discrete parameter $x2$, which transforms this first-order abstract syntax into the following code:

```
Parameter(LString "x1") `Bind`
Func (Return ([LReal 1.0,LReal 0.0]) `Bind`
Func (Apply Sequence (Apply Map (Pair (
    Func (Apply (Func (
        Return (Apply (NormalPDF(LReal 0.0, LReal 2.0)) (Var(0))))) (Var(0))),
    Var(0))))) `Bind`
Func (Apply Foldr (Pair (
    Func (Func (Apply Add (Pair (Var(0),Var(1))))),
    Pair (LReal 0.0,Var(0)))) <~ Func (Var(0))) `Bind`
Func (Apply (Func (Var(0))) (Return(LReal 0.0)) `Bind`
Func (Var(2) <~ NormalPDF(LReal 0.0, LReal 1.0) `Bind`
Func (Return(Nil)))))
```

The raw abstract syntax can seem complicated, but we will give a line-by-line explanation of the code: first, we see that the first line contains the first parameter, which remains unchanged. Next, in lines 2 to 6, we take the given possible assignments for the discrete parameter $x2$ and insert these values in our accumulated factor. In this case, this factor consists of the normal distribution that was originally applied to $x2$ in a sample statement (in line 4 of the original model). We then sequence the resulting array. Then, in lines 7 to 9, we calculate the sum of this array and with this we finish the marginalisation of the new factor over the possible assignments of $x2$. We then use a sample operation in order to add our new factor to the target density. Finally, we see that in line 10 we have code that binds the value 0 into the next expressions, but this value is never used again. This is where the `Parameter` constructor was originally located, but as we have seen in section 5.2.8, we have replaced it with an unused value. Similarly, we see that in line 12 we now return an empty value at the location where the sample operation involving $x2$ used to be. Line 11 contains a sample operation that did not involve $x2$, so this sample operation remains unchanged.
The same result is visible if we present a pretty-printed version of the abstract syntax tree of the model before the transformation:

```
Parameter ('x1') `Bind`
(\v0 -> Parameter ('x2') `Bind`
(\v1 -> v0 <~ NormalPDF(0.0, 1.0) `Bind`
(\v2 -> v1 <~ NormalPDF(0.0, 2.0))))
```

As we can see, the syntax tree has been simplified and variables are now named, instead of showing their DeBruijn index. We can also show a simplified version of the resulting abstract syntax tree after the transformation:

```
Parameter ('x1') `Bind`
(\v0 -> Return [1.0, 0.0] `Bind`
(\v1 -> sequence
    (map (\v2 -> ((\v3 -> Return ((NormalPDF(0.0, 2.0)) v3))) v2) v1)) `Bind`
(\v1 -> (foldr (\v2 -> (\v3 -> v3 + v2)) 0.0 v1) <~ (\v2 -> v2)) `Bind`
(\v1 -> (\v2 -> v2) (Return 0.0) `Bind`
(\v2 -> v0 <~ NormalPDF(0.0, 1.0) `Bind`
(\v3 -> Return Nil))))
```

This simplified version illustrates the same transformation process: the array of possible values for $x2$ is introduced in line 2, the accumulated factor function is mapped over this array and the array is then sequenced in lines 3 and 4. After this, in line 5, the sum of the resulting array becomes our new factor, which is added to the target density. As before, we also see the `Return Nil` statement in line 8 where the factor dependent on parameter $x2$ was originally located.
The analysis of the transformed code for this example therefore shows that our transformation correctly eliminates the discrete parameter. However, there is room for improvement, as we see that there are many instances of unused code that are inserted during the transformation. One solution for this problem would be to run a standard dead code elimination pass after performing our variable elimination transformation. Doing so would ensure that the syntax tree of a program remains clean, even after the variable elimination.

For our final example we look at the simplified change point model, as seen earlier in section 3.5. We slightly rewrite the model in order to use our newly introduced `Discrete` constraint:

```
changePoint :: Expr Model
changePoint = inputReal #> \re ->
              inputReal #> \rl ->
              inputReal #> \t  ->
              inputInt  #> \d  ->
              parameterC "e" (lower 0) #> \e ->
              parameterC "l" (lower 0) #> \l ->
              parameterD "s" ([1,t] :: Array 2 Double) #> \s ->
              e <~ exponential re #-
              l <~ exponential rl #-
              s <~ uniformD ([1, t] :: Array 2 Double) #-
              d <~ poisson (ifThenElse (t |<| s) e l)
```

This model has a slightly more complicated first-order abstract syntax tree:

```
Input `Bind`
Func (Input `Bind`
Func (Input `Bind`
Func (Input `Bind`
Func (Parameter(LString "e") `Bind`
Func (Parameter(LString "l") `Bind`
Func (Parameter(LString "s") `Bind`
Func (Var(2)<~ExponentialPDF(Var(6)) `Bind`
Func (Var(2)<~ExponentialPDF(Var(6))) `Bind`
Func (Var(1)<~UniformDPDF([Var(5),LReal 1.0])) `Bind`
Func (Var(4)<~PoissonPDF(
    If Var(5) |>=| Var(1)
       then Var(3)
       else Var(2))))))))))
  where a |>=| b = Apply Not (Apply Not (Apply And (Pair (
             Apply Not (Apply Not (Apply (<=) (Pair (a, b)))),
             Apply Not (Apply (==) (Pair (a, b)))))))
```

Remember that the lesser-than operator that we see in the last line of the sugared syntax is combined using various comparison and Boolean operators. This is why we have added the extra sugaring for

readability in the final line of the unsugared syntax. Applying the transformation to eliminate s then transforms the syntax of the change point model into the following abstract syntax tree:

```
Input `Bind`
Func (Input `Bind`
Func (Input `Bind`
Func (Input `Bind`
Func (Parameter(LString "e") `Bind`
Func (Parameter(LString "l") `Bind`
Func (Return([Var(3),LReal 1.0]) `Bind`
Func (Apply Sequence (Apply Map (Pair (Func (Apply (Func (Return(Apply (PoissonPDF(
    If Var(6) |>=| Var(0)
       then Var(4)
       else Var(3)
)) (Var(5))) `Bind`
Func (Return(Apply (UniformDPDF([Var(7),LReal 1.0])) (Var(1))) `Bind`
Func (Return(Apply Mul (Pair (Var(0),Var(1)))))))) (Var(0))),Var(0))))) `Bind`
Func (Apply Foldr (Pair (
    Func (Func (Apply Add (Pair (Var(0),Var(1))))),
    Pair (LReal 0.0,Var(0)))) <~ Func (Var(0))) `Bind`
Func (Apply (Func (Var(0))) (Return(LReal 0.0)) `Bind`
Func (Var(3)<~ExponentialPDF(Var(7)) `Bind`
Func (Var(3)<~ExponentialPDF(Var(7))) `Bind`
Func (Return(Nil)) `Bind`
Func (Return(Nil)))))))))
```

We can see many similarities when comparing the transformation with our earlier example. The final two lines show how we again have replaced the sample operations containing s with an empty expression. Line 18 shows how we also again have replaced the eliminated parameter s with the unused value of 0. Lines 7 to 14 again show how we map the product factor over the possible assignments for s and lines 15 to 17 show how we sum these values and use a sample operation to add this sum to the target density. One difference from the earlier model is that in this case we have two factors that contained s in the original model; we can see that we therefore needed to multiply these two factors during the transformation in line 14. Finally, we can also see that any inputs, parameters or sample operations that did not involve s still remain unchanged in the transformed model.
As with our previous example, we also present the pretty-printed version of the change point model:

```
Input `Bind`
(\v0 -> Input `Bind`
(\v1 -> Input `Bind`
(\v2 -> Input `Bind`
(\v3 -> Parameter ('e') `Bind`
(\v4 -> Parameter ('l') `Bind`
(\v5 -> Parameter ('s') `Bind`
(\v6 -> v4 <~ ExponentialPDF(v0) `Bind`
(\v7 -> v5 <~ ExponentialPDF(v1)) `Bind`
(\v7 -> v6 <~ UniformDPDF([v2, 1.0])) `Bind`
(\v7 -> v3 <~ PoissonPDF(If not (not (not (not (v2 <= v8)) and not (v2 == v8)))
                                then v4
                                else v5)))))))))
```

After applying the variable elimination transformation, we then get the following pretty-printed result:

```
Input `Bind`
(\v0 -> Input `Bind`
(\v1 -> Input `Bind`
(\v2 -> Input `Bind`
(\v3 -> Parameter ('e') `Bind`
(\v4 -> Parameter ('l') `Bind`
(\v5 -> Return ([v2, 1.0]) `Bind`
(\v6 -> sequence (map (\v7 -> (\v8 -> Return ((PoissonPDF(
    If not not not not v2 <= v8 and not v2 == v8
       then v4
       else v5
)) v3) `Bind`
```

```
13 (\v9 -> Return ((UniformDPDF([v2, 1.0])) v8) `Bind`
14 (\v10 -> Return (v10 * v9)))) v7) v6)) `Bind`
15 (\v6 -> (foldr (\v7 -> (\v8 -> v8 + v7)) 0.0 v6) <~ (\v7 -> v7)) `Bind`
16 (\v6 -> (\v7 -> v7) (Return 0.0) `Bind`
17 (\v7 -> v4 <~ ExponentialPDF(v0) `Bind`
18 (\v8 -> v5 <~ ExponentialPDF(v1)) `Bind`
19 (\v8 -> Return Nil) `Bind`
20 (\v8 -> Return Nil)))))))))
```

As with the previous example, this pretty-printed result shows the steps that we have discussed when examining the regular transformed abstract syntax tree more clearly. For example, we see the replacement of s with the value 0 in line 16, the mapping of the factor function over the possible values for s in lines 8 to 12 and the sampling of the final factor that is created by summing over the resulting array in line 15.

5.3.2 Conclusion

The two examples in the previous section illustrate how our variable elimination transformation works in practice: we have shown that it works on a simple example, but also on a model that is commonly used in probabilistic programming. This means that despite the limitations, the `moveUp` function in combination with the static analysis introduced in the previous chapter provides us with a useful variable elimination transformation that can automatically eliminate discrete parameters from some models in the *ProbProg* language.
Unfortunately, one limitation does greatly restrict the range of useful models on which we can perform discrete parameter elimination: the fact that the `moveUp` function does not work on models containing arrays of discrete parameters. As we have seen in section 3.5, many models that are used in practice use these arrays of parameters. However, unlike the limitations on these types of models in the similar program transformation for *SlicStan* developed by Gorinova et al. [11], our limitations do not stem from a lack of conditional independence information for these models. Further research might therefore be necessary to see if the `moveUp` function can be extended in the manner that we have suggested in section 5.2.9 in order to make use of this conditional dependence information.

5.4 Further usages

As hinted at in chapter 2, we believe that there are several more cases for which the conditional dependence information of a model could prove useful. For example, in our second research question we ask if it might be possible to improve the efficiency of the inference of models. We were not able to construct an elaborate implementation for such improvements as we have done with our variable elimination program transformation. Instead, in this section we list several ideas for such improvements for the inference of models that we believe might be possible based on our review of the relevant literature in chapter 2 and could be useful starting points for further research.

We will describe two sets of possible improvements that could increase inference efficiency. Firstly, there are improvements that might help for a single inference method. We have seen several examples of this in section 2.2. In Gibbs sampling, using the Markov blanket of a model (and thus the conditional dependence information) can help give a good decomposition of variables into blocks that improves the performance of a blocked Gibbs sampler. As we have seen in section 2.2, there is existing research showing that this is possible once the Markov blanket is obtained from the model [22, 23]. In variational message passing, the conditional dependence information may be used to automatically rewrite incompatible (parts of) models so that these models can be inferred using variational message passing. This is of course a more experimental idea, which would require more research in order to determine if it is actually feasible. We can therefore see that these types of improvements exist for various different inference methods and may differ widely in their possible feasibility.

Secondly, there might be room for improvement in the efficiency of inference if we would combine different inference algorithms during the inference of one model. The conditional dependence information of that model could then be used to determine what inference methods should be used on specific parts of the model. In section 2.2, we have seen that blocked Gibbs sampling can be used to combine different inference algorithms to infer a single model, where one inference method is chosen for each block. We can then construct these different combinations using the conditional dependence information: for example, by detecting parts of the model that can be inferred with variational message passing or inferred exactly, which might increase the accuracy of the inference when compared to methods such as HMC or NUTS. Of course, this set of improvements would also require a lot of further research to determine the feasibility and efficiency of different combinations of inference algorithms.

Finally, we mention one more change with potential to improve the modelling workflow instead of the inference efficiency of models. As mentioned in section 2.4, there are several checks that are routinely performed on models as part of the modelling workflow. In the section, we explain that some of these checks require the model to be written in the so-called forward or ancestral sampling mode of a model. Rewriting the model into this mode is a task that is often still left to the user. Therefore, performing the necessary rewrite automatically would spare users a significant amount of work. Of course, this differs from our earlier suggested improvements in that it will likely not require knowledge on the conditional dependencies within the model. However, the automatic rewrite will likely need to be implemented as a program transformation, meaning that knowledge of our `moveUp` program transformation might be useful in its implementation. Additionally, there might be benefits to implementing this specific transformation in a functional probabilistic programming language: because the purely functional aspect of such a language allows us to easily track the effects of adding to the target density, it might be easier to localise statements that perform such operations and rewrite them to their generative counterparts.

Chapter 6

Conclusion

In the introduction of our thesis we set out to find ways in which we could improve the current state of probabilistic programming. We asked ourselves if we could somehow use the functional programming paradigm to design a language that facilitates useful improvements to the probabilistic programming workflow, perhaps in a manner that is not possible in the traditional imperative form of probabilistic programming languages. Our emphasis for these improvements was on the inference and usage of conditional dependence between the variables of probabilistic models.
After giving the relevant background information in chapter 2, we started our research into this subject. In chapter 3, we designed a small functional probabilistic programming language, the *ProbProg* language. With several examples, we have shown that a number of useful probabilistic models can be implemented in this language (in section 3.5). We then formulated and implemented an effect system that performs a static analysis that obtains the conditional dependence structure of models implemented in the *ProbProg* language (in chapter 4). We added an informal extension to this effect system that allows us to also perform this static analysis on models that contain arrays in section 4.3. Finally, in chapter 5, we examined how we could use this conditional dependence information to improve our probabilistic programming language. The main result was an automatic program transformation that implements the variable elimination algorithm, as seen in section 5.2. This transformation then allows us to automatically marginalise discrete parameters out of our models. We described several limitations of this transformation in section 5.2.9. Although we saw in section 5.3.1 that the algorithm works on some practical examples, such as the change-point model, it does not work on models that contain arrays of discrete parameters. Additionally, we mentioned several other possible use-cases for the conditional dependence information in section 5.4.

In the introduction we formulated two relevant research questions that we set out to answer with our thesis. Using our aforementioned findings, we can now answer these research questions.

Research question 1. *How can a functional probabilistic programming language be designed so that it can efficiently extract conditional independence relationships between variables from implemented models?*

In the implementation of our static analysis that obtains the conditional dependence relationships of a model, we have used several beneficial properties of the *ProbProg* language. Firstly, as we have mentioned in section 4.4, the fact that *ProbProg* is a purely functional language with referential transparency makes the design of our effect system very straightforward.
Secondly, there are some specific design decisions with respect to array operators and constructors that we have made for the *ProbProg* language that allow us to extend this effect system to models with arrays. These largely amount to limiting the expressivity of the user with regards to arrays, for example by only allowing the user to have access to the `Generate` constructor through the sugared syntax. However, as we have seen in section 3.5, this does not limit us from building useful models

with arrays. This technique is largely possible because functional languages use array operators such as `map` and `foldr`, that effect the conditional dependence structure of a model in clear patterns. These clear patterns allow us to easily infer the effects of these operators, whereas it would have been difficult to implement an effect system in a language with `for` and `while` loops and free indexing of arrays. Other functional domain specific languages that deal with arrays, such as the *Accelerate* language [41], also benefit from the idea that functional programming languages work with these specific array operators, instead of the more expressive imperative array operations.

Research question 2. *How can knowledge on the conditional dependencies between variables within a model in this functional language be used to simplify the implementation of models and improve the efficiency of the inference in the language?*

Our answer to this question is that we were able to simplify the implementation of models in *ProbProg* using a automatic program transformation that implements the variable elimination algorithm. This transformation can then be used to marginalise any discrete parameters out of a model, which makes it possible to perform the HMC or NUTS inference methods on models that contain discrete parameters. This allows the user to implement these models with discrete parameters without having to perform any complicated rewrites of model code. This greatly improves the modelling workflow, as it reinforces the idea of probabilistic programming that writing probabilistic models and inferring these models should be separated: the user should not actively have to think about this inference when implementing a model. As mentioned before in this conclusion, this program transformation does have limitations: we are unable to perform the variable elimination in some cases; most importantly in models that contain arrays of discrete parameters. Next to our implemented transformation, we also give an idea for other improvements that might simplify the modelling workflow in section 5.4. Unfortunately, due to a lack of time, we were unable to implement methods that use the conditional dependence information to increase the efficiency of the inference of models. In section 5.4 we have suggested some possible ways in which such an improvement might be possible, based on our current knowledge of the inference methods in section 2.2.

To conclude, we have found that there is definitely a benefit to designing a probabilistic programming language in a way so that the conditional dependence relationships of a model can easily be extracted automatically. As we have seen, using the purely functional programming paradigm can help achieve such a design. This conditional dependence information can then be used to make practical improvements to the usability of the language. Although we have only implemented one single limited improvement that uses the conditional dependence information in this thesis, we believe that there are possibilities for further improvements that use this information as well.

6.1 Discussion

In this section we give further final insights as we reflect on our implemented work: the *ProbProg* language, our static analysis and our program transformation. We discuss design decisions, limitations, and possibilities for improvements or starting points for further research.

6.1.1 Language design

We have made several important choices in the design of the *ProbProg* language. Some design choices were made deliberately because they were necessary for our research, such as the choice to design a purely functional language. Other choices were made for pragmatic reasons, such as the choice to develop a deeply embedded domain specific language instead of developing a standalone language, which would have been a much larger task. We discuss some insights that we have developed through these choices and give several ideas that could be explored in further research on functional probabilistic programming.

Functional probabilistic programming

In our research questions, we explicitly ask how we could use the functional setting of our language to capture and use the conditional dependence information. In the introduction we even mention that because there is not much existing research on functional probabilistic programming, there might be some unknown improvements for probabilistic programming that could only be achieved in a functional language, as opposed to in an imperative language. We have seen several concrete benefits of our functional setting:

- The referential transparency of a purely functional language helps determine the parameters that an expression is dependant on in our static analysis. These dependencies propagate through our models in fairly simple ways: a dependency is never introduced through a side-effect of the model, for example.
- The referential transparency also allows us to move expressions around in the abstract syntax during our program transformation, without having to keep any side effects in mind.
- The functional type system that we use for *ProbProg* is the basis for our effect system that we use to obtain the conditional dependencies.
- Finally, our functional setting lets us limit any array transformations in *ProbProg* to the application of several functions that have a clear pattern through which dependence information spreads. This allows us to obtain the conditional dependencies of models that contain array expressions using our effect system.

However, while these benefits might simplify our static analysis and our program transformation, we have also seen that both a similar static analysis and a similar program transformation can also be implemented in an imperative language [11]. We are therefore not fully convinced that a functional probabilistic programming language would have any decisive advantages over an imperative probabilistic programming language. More research needs to be done in order to determine these advantages, if they exist. Good starting points for this research would be the benefits that we have found, because these give an insight in the strengths of functional probabilistic programming, as well as the other aspects of language design that we mention in this section. Of course, our benefits focus on our specific improvements through our conditional dependency effect system and variable elimination program transformation: further research can also look into different methods of improvement that might only be possible in a functional (or an imperative) setting.

Embedded language

The choice to implement *ProbProg* as a deeply embedded domain specific language was useful to prevent the extensive task of building a custom compiler from taking over the bulk of the work for this thesis. However, building an embedded language comes with additional benefits: for example, we might be able to run useful analyses and transformations at the run-time compilation, we can use several elements from the host language such as the type system during compilation and users might be able to iterate on their probabilistic models faster than if they would need to recompile an executable every iteration.
We therefore believe that it might be useful to investigate if designing a functional probabilistic programming language on a larger scale as an embedded domain specific language could be beneficial. Standalone domain specific probabilistic programming languages such as *Stan* can greatly limit how a model should be built, but lose some of the benefits of having a host-language and being able to perform just-in-time compilation. On the other hand, many widely used probabilistic programming languages are embedded within a general purpose programming language. *Pyro* and *PyMC* extend *Python* and *Church* extends *Scheme*, for example [1, 6]. Ideally, an embedded probabilistic programming language can maintain the limits for constructing a model that a standalone language has by not making the embedding too shallow and controlling access to the general purpose parts of its host

language. Designing a deeply embedded domain specific probabilistic programming language might thus provide us with the best of both worlds.

Representation of probabilistic side-effects

Currently, our language captures the probabilistic side-effects of introducing parameters and performing sampling operations within a monad, which stores the parameters and the target density as a state. Of course, this is not the only possible representation for these effects. For example, another possibility could be to build a language that is purely functional in all regards, except for when parameters are introduced or the sample operation is called, giving the language two impure operations. This example would reduce some of the complexity of the language that is introduced by the monadic aspects.
We believe that the choice of representation for the probabilistic effects within a functional programming language is an important aspect of the language design that warrants further research. Of course, this research should keep in mind how changes in this representation effect both the complexity of the language for the user, but also the complexity of implementing automatic improvements such as static analyses and program transformations, as these are two important factors that contribute to the overall usability of the language. Ideally, reducing the complexity of one of these should also reduce the other, but this might not always be the case.

Expressivity

Throughout our thesis, we saw that at some points a reduction in expressivity was needed in the *ProbProg* language to accomplish our intended goals, for example in the case where we had to reduce the access to the `Generate` constructor in section 4.3.1 in order to define the type inference for our effect system on that constructor. Still, the language is too expressive in some respects. In section 5.2.9, we saw that it allows users to construct models without arrays that can not be transformed correctly using our `moveUp` algorithm. These models were constructed by using the `Func` and `Pair` constructors in a manner that is unlikely to be useful for any practical models. In summary, we find that allowing too much expressivity might harm attempts at creating useful program transformations or static analyses.
From this point of view, the *ProbProg* language is too expressive. We think that further research could explore what level of expressivity is necessary for a functional probabilistic programming language. This requires many possible aspects to be researched, of which we give two examples. Firstly, *ProbProg* is a fully higher-order language, but this might not be needed. Restricting the complexity of functions that are applied or returned in a higher-order manner simplifies program transformations. This could be a solution to the issues that occur in the `moveUp` algorithm because of the `Func` operator, for example. Secondly, parameters can currently be defined anywhere in a model. Instead, *Stan* requires users to define all latent and observed parameters as global variables in a preamble in advance. This would also be useful in the *ProbProg* language: it would remove the entire complex procedure of the `moveUp` algorithm explained in section 3.3.5 of having to find discrete parameters and their assignments within a model. Clues on how to effectively limit the expressivity can thus be taken from existing probabilistic programming languages.

Additional models

Throughout our thesis we have given several example models that have been implemented in the *ProbProg* language. Although we have chosen some very commonly used models, such as hidden Markov models, for example, there is a very large set of models used in probabilistic programming that we have not tested. These models can range from very simple graphical models to complex models that incorporate techniques from other disciplines, such as machine learning [6]. Further research into the design of functional probabilistic programming languages should examine what language features are needed to implement certain models, simply by trying to implement more example models.

This should also guide the different design choices of the language. As an example, we can look at the simplified change point model introduced in section 3.5 that is also used as an example for the variable transformation in section 5.3.1. The simplification that we used is that the range of the change point s was decreased from $s \in \{1, \ldots, T\}$ to $\{1, T\}$. It seems as if moving from this simplification to a regular change point model is easy. However, choices made in the early design stage of *ProbProg* will make this difficult: as the arrays in *ProbProg* have a static size, we cannot easily create an array $a = [1, \ldots, T]$, as the size of this array depends on the dynamic value T. This means that we cannot create a discrete parameter s, as this would require us to specify its concrete values with the array a. This flaw in the design of *ProbProg* shows the importance of making design choices with the needs for probabilistic models in mind.
Of course, adding support for more models is very related to the problem of expressivity as discussed in the previous section: reducing the expressivity of the language will naturally also reduce the possible models that can be implemented. However, we believe that the goal in this further research should be to find a balance between expressivity and practicality. We should allow users to implement as many practically useful models as possible, while still limiting the expressivity to facilitate convenient improvements through static analyses and program transformations. These ideas are not even always opposed to each other: we allow the user to conveniently implement models with discrete parameters through a static analysis and program transformation, for which we need to limit the expressivity of *ProbProg*.

6.1.2 Conditional dependence analysis

The extension to our effect system that allows us to obtain the conditional dependencies of models containing arrays presented in section 4.3.1 is still an informal algorithmic extension. A possibility for further research could be to adapt this extension so that it can be formalised as a proper extension to the theoretical effect system, for example by devising type judgements for the array constructors and operators.
Additionally, as we have mentioned earlier, there is also a lot of possible further research that could be done into the usages of conditional dependence information, either by exploring use-cases that we have mentioned in section 5.4, or by finding entirely new use-cases. This research might also provide new insights into the storage for the conditional dependence information: currently we store our solved constraints, which, as explained in chapter 4, induce the factor graph, but this method of storage was designed specifically so that our `moveUp` algorithm in chapter 5 can extract the needed conditional dependence information. Other use-cases for the conditional dependence information that arise from further research might require a completely different representation for this information.

6.1.3 Program transformation

As described extensively in section 5.2.9, our `moveUp` algorithm has several limitations. Most importantly, we are unable to automatically marginalise arrays of discrete parameters out of programs, which is a limitation similar to that seen in earlier work on variable elimination program transformations by Gorinova et al. [11]. Solving this issue is thus an important goal for any further research.
In section 5.2.9, we have provided some leads on how we believe it is possible to overcome the limitations in the `moveUp` algorithm specifically. However, it might also be needed to make a fundamental adjustment to the design of the *ProbProg* language as a whole, as we have mentioned in section 6.1.1, in order to build a program transformation that is able to eliminate arrays of discrete parameters. As mentioned in that section, it will likely be useful to limit the expressivity with respect to arrays and only allow users to implement constructs that propagate dependence information in predictable ways.

Bibliography

[1] N. Goodman, V. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum, "Church: A language for generative models," *arXiv preprint arXiv:1206.3255*, 2012.

[2] F. Wood, J. W. Meent, and V. Mansinghka, "A new approach to probabilistic programming inference," in *Artificial Intelligence and Statistics*, 2014, pp. 1024–1032.

[3] T. Minka, J. Winn, J. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill, "Infer.net 2.7.(2018)," *URL http://research. microsoft. com/infernet. Microsoft Research Cambridge*, vol. 1, 2018.

[4] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, and A. Riddell, "Stan: A probabilistic programming language," *Journal of statistical software*, vol. 76, no. 1, 2017.

[5] D. Tran, M. D. Hoffman, R. A. Saurous, E. Brevdo, K. Murphy, and D. M. Blei, "Deep probabilistic programming," *arXiv preprint arXiv:1701.03757*, 2017.

[6] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman, "Pyro: Deep universal probabilistic programming," *The Journal of Machine Learning Research*, vol. 20, no. 1, pp. 973–978, 2019.

[7] G. Baudart, J. Burroni, M. Hirzel, K. Kate, L. Mandel, and A. Shinnar, "Extending stan for deep probabilistic programming," *arXiv preprint arXiv:1810.00873*, 2018.

[8] A. Bryant, M. Alter, and A. Metcalf. (2018). "Rainier: Bayesian inference in scala," [Online]. Available: `https://github.com/stripe/rainier/` (visited on 01/13/2021).

[9] P. Narayanan, J. Carette, W. Romano, C.-c. Shan, and R. Zinkov, "Probabilistic inference by program transformation in hakaru (system description)," in *International Symposium on Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, Springer, 2016, pp. 62–79. DOI: `10.1007/978-3-319-29604-3_5`. [Online]. Available: `http://dx.doi.org/10.1007/978-3-319-29604-3_5`.

[10] A. Ścibior, O. Kammar, and Z. Ghahramani, "Functional programming for modular bayesian inference," *Proceedings of the ACM on Programming Languages*, vol. 2, no. ICFP, pp. 1–29, 2018.

[11] M. I. Gorinova, A. D. Gordon, C. Sutton, and M. Vakar, "Conditional independence by typing," *arXiv preprint arXiv:2010.11887*, 2020.

[12] S. Staton, "Commutative semantics for probabilistic programming," in *European Symposium on Programming*, Springer, 2017, pp. 855–879.

[13] R. Culpepper and A. Cobb, "Contextual equivalence for probabilistic programs with continuous random variables and scoring," in *European Symposium on Programming*, Springer, 2017, pp. 368–392.

[14] M. F. Cusumano-Towner, F. A. Saad, A. K. Lew, and V. K. Mansinghka, "Gen: A general-purpose probabilistic programming system with programmable inference," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 221–236.

[15] J. Carette and C.-c. Shan, "Simplifying probabilistic programs using computer algebra," in *International Symposium on Practical Aspects of Declarative Languages*, Springer, 2016, pp. 135–152.

[16] C.-K. Hur, A. V. Nori, S. K. Rajamani, and S. Samuel, "Slicing probabilistic programs," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 133–144, 2014.

[17] R. Bernstein, "Static analysis for probabilistic programs," *arXiv preprint arXiv:1909.05076*, 2019.

[18] W. K. Hastings, "Monte carlo sampling methods using markov chains and their applications," 1970.

[19] A. E. Gelfand, S. E. Hills, A. Racine-Poon, and A. F. Smith, "Illustration of bayesian inference in normal data models using gibbs sampling," *Journal of the American Statistical Association*, vol. 85, no. 412, pp. 972–985, 1990.

[20] A. E. Gelfand, "Gibbs sampling," *Journal of the American statistical Association*, vol. 95, no. 452, pp. 1300–1304, 2000.

[21] I. Yildirim, "Bayesian inference: Gibbs sampling," *Technical Note, University of Rochester*, 2012.

[22] C. Riggelsen, "Mcmc learning of bayesian network models by markov blanket decomposition," in *European Conference on Machine Learning*, Springer, 2005, pp. 329–340.

[23] J.-N. Rivasseau, "From the jungle to the garden: Growing trees for markov chain monte carlo inference in undirected graphical models," Ph.D. dissertation, University of British Columbia, 2005.

[24] R. M. Neal, *Probabilistic inference using Markov chain Monte Carlo methods.* Department of Computer Science, University of Toronto Toronto, Ontario, Canada, 1993.

[25] R. M. Neal *et al.*, "Mcmc using hamiltonian dynamics," *Handbook of markov chain monte carlo*, vol. 2, no. 11, p. 2, 2011.

[26] M. D. Hoffman and A. Gelman, "The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo.," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1593–1623, 2014.

[27] J. Salvatier, T. V. Wiecki, and C. Fonnesbeck, "Probabilistic programming in python using pymc3," *PeerJ Computer Science*, vol. 2, e55, 2016.

[28] J. Winn and C. M. Bishop, "Variational message passing," *Journal of Machine Learning Research*, vol. 6, no. Apr, pp. 661–694, 2005.

[29] Y. Yao, A. Vehtari, D. Simpson, and A. Gelman, "Yes, but did it work?: Evaluating variational inference," in *International Conference on Machine Learning*, PMLR, 2018, pp. 5581–5590.

[30] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[31] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *stat*, vol. 1050, p. 1, 2014.

[32] M. I. Jordan, Z. Ghahramani, T. S. Jaakkola, and L. K. Saul, "An introduction to variational methods for graphical models," *Machine learning*, vol. 37, no. 2, pp. 183–233, 1999.

[33] R. Bernstein, M. Vákár, and J. Wing, "Transforming probabilistic programs for model checking," in *Proceedings of the 2020 ACM-IMS on Foundations of Data Science Conference*, 2020, pp. 149–159.

[34] F. A. Saad, M. C. Rinard, and V. K. Mansinghka, "Exact symbolic inference in probabilistic programs via sum-product representations," *arXiv preprint arXiv:2010.03485*, 2020.

[35] A. Gelman, A. Vehtari, D. Simpson, C. C. Margossian, B. Carpenter, Y. Yao, L. Kennedy, J. Gabry, P.-C. Bürkner, and M. Modrák, "Bayesian workflow," *arXiv preprint arXiv:2011.01808*, 2020.

[36] J. Gabry, D. Simpson, A. Vehtari, M. Betancourt, and A. Gelman, "Visualization in bayesian workflow," *Journal of the Royal Statistical Society: Series A (Statistics in Society)*, vol. 182, no. 2, pp. 389–402, 2019.

[37] S. D. Team. (2019). "Stan user's guide," [Online]. Available: `https://mc-stan.org/docs/2_25/stan-users-guide/index.html` (visited on 11/11/2020).

[38] J. Hermaszewski. (2016). "Vector-sized: Size tagged vectors," [Online]. Available: `https://hackage.haskell.org/package/vector-sized-1.4.1.0` (visited on 05/01/2020).

[39] mniip. (2018). "Finite-typelits: A type inhabited by finitely many values, indexed by type-level naturals.," [Online]. Available: `https://hackage.haskell.org/package/finite-typelits` (visited on 06/01/2020).

[40] T. Altenkirch and B. Reus, "Monadic presentations of lambda terms using generalized inductive types," in *International Workshop on Computer Science Logic*, Springer, 1999, pp. 453–468.

[41] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier, "Optimising purely functional gpu programs," *ACM SIGPLAN Notices*, vol. 48, no. 9, pp. 49–60, 2013.

[42] B. O'Sullivan and A. Khudaykov. (2014). "Statistics: A library of statistical types, data, and functions," [Online]. Available: `https://hackage.haskell.org/package/statistics` (visited on 05/01/2020).

[43] C. J. Fonnesbeck, A. Patil, D. Huard, and J. Salvatier, "Pymc documentation," 2017.

[44] D. McDermott, "Reasoning about effectful programs and evaluation order," University of Cambridge, Computer Laboratory, Tech. Rep., 2020.

[45] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer Science & Business Media, 2004.

[46] L. Damas and R. Milner, "Principal type-schemes for functional programs," in *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1982, pp. 207–212.

[47] M. P. Jones, "Ml typing, explicit polymorphism and qualified types," in *International Symposium on Theoretical Aspects of Computer Software*, Springer, 1994, pp. 56–75.

[48] N. L. Zhang and D. Poole, "A simple approach to bayesian network computations," in *Proc. of the Tenth Canadian Conference on Artificial Intelligence*, 1994.

[49] V. Kuleshov and S. Ermon. (2017). "Cs228 notes," [Online]. Available: `https://ermongroup.github.io/cs228-notes/` (visited on 06/01/2020).