Abstractions for Probabilistic Programming to Support Model Development

Ryan Bernstein

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2023
Abstract
Abstractions for Probabilistic Programming to Support Model Development
Ryan Bernstein

Probabilistic programming is a recent advancement in probabilistic modeling whereby we can express a model as a program with little concern for the details of probabilistic inference. Probabilistic programming thereby provides a clean and powerful abstraction to its users, letting even non-experts develop clear and concise models that can leverage state-of-the-art computational inference algorithms. This model-as-program representation also presents a unique opportunity: we can apply methods from the study of programming languages directly onto probabilistic models. By developing techniques to analyze, transform, or extend the capabilities of probabilistic programs, we can immediately improve the workflow of probabilistic modeling and benefit all of its applications throughout science and industry.

The aim of this dissertation is to support an ideal probabilistic modeling workflow by addressing two limitations of probabilistic programming: that a program can only represent one model, and that the structure of the model that it represents is often opaque to users and to the compiler. In particular, I make the following primary contributions: (1) I introduce Multi-Model Probabilistic Programming: an extension of probabilistic programming whereby a program can represent a network of interrelated models. This new representation allows users to construct and leverage spaces of models in the same way that probabilistic programs do for individual models. Multi-Model Probabilistic Programming lets us visualize and navigate solution spaces, track and document model development paths, and audit modeler degrees of freedom to mitigate issues like
$p$-hacking. It also provides an efficient computational foundation for the automation of model-space applications like model search, sensitivity analysis, and ensemble methods. I give a formal language specification and semantics for Multi-Model Probabilistic Programming built on the Stan language, I provide algorithms for the fundamental model-space operations along with proofs of correctness and efficiency, and I present a prototype implementation, with which I demonstrate a variety of practical applications. (2) I present a method for automatically transforming probabilistic programs into semantically related forms by using static analysis and constraint solving to recover the structure of their underlying models. In particular, I automate two general model transformations that are required for diagnostic checks which are important steps of a model-building workflow. Automating these transformations frees the user from manually rewriting their models, thereby avoiding potential correctness and efficiency issues. (3) I present a probabilistic program analysis tool, “Pedantic Mode”, that automatically warns users about potential statistical issues with the model described by their program. “Pedantic Mode” uses specialized static analysis methods to decompose the structure of the underlying model.

Lastly, I discuss future work in these areas, such as advanced model-space algorithms and other general-purpose model transformations. I also discuss how these ideas may fit into future modeling workflows as technologies.
# Table of Contents

Acknowledgments ................................................................. xv

Dedication .............................................................................. xvi

Chapter 1: Introduction ............................................................ 1

1.1 Probabilistic programming, briefly ........................................ 3

1.2 Modern best practices: the Bayesian Workflow ....................... 5

1.3 Supporting the Bayesian Workflow: Program Transformations .... 7

1.4 Contributions and roadmap ................................................... 8

Chapter 2: A Systematic Review of Static Analysis of Probabilistic Programs .................................................... 11

2.1 Introduction ........................................................................ 11

2.1.1 Introduction to Probabilistic Programming .......................... 11

2.1.2 Introduction to Static Analysis ........................................ 13

2.1.3 Purposes of Applying Static Analysis to Probabilistic Programming ........................................... 15

2.2 Technical background .......................................................... 18

2.2.1 Posterior inference .......................................................... 18

2.2.2 Static analysis ................................................................. 22

2.2.3 Probabilistic Programming from a Static Analysis and Programming Languages Perspective .............................. 28
3.3.2 Effects and Scope ........................................... 74
3.3.3 Program validity ........................................... 75
3.4 Modular Stan syntax ......................................... 75
3.5 Modular Stan semantics .................................... 78
  3.5.1 Basic operations on programs .......................... 79
  3.5.2 Structural constraints .................................... 81
  3.5.3 Module signatures and semantic constraints ............ 82
  3.5.4 Program validity ........................................ 85
  3.5.5 Selections ............................................... 85
  3.5.6 Semantics of modular programs and network operations ........................................ 87
3.6 Algorithms ...................................................... 88
  3.6.1 Concretize ............................................... 88
  3.6.2 ModelGraph .............................................. 99
  3.6.3 ModelNeighbors ......................................... 113
3.7 Additional features ......................................... 118
  3.7.1 Append blocks .......................................... 118
  3.7.2 Module fields .......................................... 120
3.8 Macros ........................................................ 122
  3.8.1 Collection holes ........................................ 123
  3.8.2 Indexed holes .......................................... 124
  3.8.3 Hole instances and hole copies ......................... 124
  3.8.4 Multi-ranges and range exponents ...................... 126
  3.8.5 Hole products and hole exponents ...................... 127
3.8.6 Example application of macros

3.9 Example applications

3.9.1 Interactive web interface

3.9.2 “Golf” case study: Modular Stan for ease and clarity of development

3.9.3 “Birthday” case study: Modular Stan as a platform for automation

3.9.4 Modular Stan as tool for multiverse and sensitivity analysis

3.10 Future work

Chapter 4: Automatic Transformations of Probabilistic Programs for Model Checking

4.1 Introduction

4.2 Background

4.2.1 Probabilistic programs

4.2.2 Efficiently sampling from model parameters

4.2.3 Translation challenges

4.2.4 Our approach

4.3 Methods

4.3.1 Factor Graphs

4.3.2 Directed Acyclic Probabilistic Graphical Models

4.3.3 A relation between Factor Graphs and DAGs

4.3.4 Defining a transformation from Factor Graphs to DAGs

4.3.5 Computing sound transformations between Factor Graphs and DAGs

4.4 Results

4.4.1 Stan implementation
List of Figures

1.1 A flow diagram of probabilistic programming. A program can be seen as a function from data and hidden parameters $\theta$ to a real number that is proportional to the joint distribution of the intended model. It is compiled into a program that accepts data and returns posterior draws. When combined with the data, inference is performed, and posterior draws are produced. ................................................................. 4

1.2 A flowchart representing a best-practices workflow for Bayesian modeling, from Gelman et al.. The green and yellow highlights show parts that are supported by the methods in this dissertation: diagnostic tests that utilize standardized model transformations are highlighted in green, and arbitrary, user-defined model transformations are highlighted in yellow. ................................................................. 6

1.3 The relation of scientific challenges, engineering methods, and probabilistic modeling applications in this dissertation. The two semantic limitations discussed at the beginning of this section correspond to the first and fourth boxes in the left column. The two types of model transformations are supported by the first and second boxes in the middle column. The yellow-highlighted parts of the modeling workflow in Figure 1.2 are supported by the first and second boxes in the right column, while the green-highlighted parts are supported by the third box in the right column. 9

2.1 On the left is shown the list of grammatical rules for writing (a limited subset of) expressions in Stan. Expressions are pieces of the language which evaluate to values, such as numbers or strings. On the right is shown the list of rules for writing (a limited subset of) statements in Stan. Statements are pieces that take some imperative action, such as setting the value of a variable or running other statements in a loop. Not included here are the rules for writing a whole Stan program, which would describe the various blocks that are allowed, and how the blocks are made up of statements. The rules are read as follows: to generate a value of the type on the left hand side of the ::=, such as an E, pick one of the instances on the right hand side (separated here by lines), such as c. This means that a constant such as 1 is a valid Stan expression. Figure taken from [54]. ............................ 14
2.2 This shows the steps of a trajectory of a particle simulated by NUTS for an underlying posterior distribution whose contour is shown in blue. The axes represent two arbitrary dimensions of the same scale. The trajectory is terminated approximately when the particle would start moving back on itself. NUTS samples efficiently from the eccentric ellipse-shaped distribution. Figure taken from [62].

2.3 Shown here are the Markov chain trajectories for two different MCMC variants given the same joint distribution (whose elliptical contour is shown as dotted lines). The axes represent two arbitrary dimensions of the same scale. Successive samples are shown as dots connected by lines. The random-walk Metropolis algorithm has a Gaussian proposal distribution at each step, where the Hamiltonian Monte Carlo moves each step according to a particle simulated under Hamiltonian dynamics after an initial random momentum. The posterior is a difficult shape for the random-walk algorithm to traverse, but the gradient-informed HMC easily takes steps across the space. Figure taken from [77].

2.4 Shown here are the results of applying two variants of variational inference (red) to an underlying bimodal posterior distribution (black). (a) should VI with KL(P||Q), which tends to give an estimate more spread out than the underlying distribution. (b) and (c) show two possible results of VI with KL(Q||P), which tends to give more peaked estimates than the underlying distribution.

2.5 Shown are the types allowed by an example probabilistic programming language [55]. In English, it says: “Types named $A$ or $B$ will be one of: a real number $R$, a probability distribution over another type $A$, the unit value (which is only ever one value), a pair of two values of types $A$ and $B$, or a type that can be any of the types $A_i$ for $i \in I$.” Effectively, this language allows real numbers, distributions, a singleton type, and any pair or disjunction of types. Figure taken from [55].
2.6 Shown are type rules sufficient to type-check the example probabilistic programming language [55]. A rule asserts that the statements over the line imply the statements below the line. $\Gamma \vdash_x t : A$ means “the context implies that an expression $t$ has type $A$”. The subscript $x$ is used to distinguish between the implication of deterministic expressions with $\vdash_d$ and probabilistic expressions with $\vdash_p$. Non-probabilistic programs would only have $\vdash_d$, and would not include rules for $\text{sample}$, $\text{score}$ or $\text{normalize}$. (a) shows a non-probabilistic example, which intuitively reads: “For some expression $t$ with type $A$ from the context $\Gamma$, $f(t)$ is of type $B$ in the context $\Gamma$ for some measurable function $f : A \rightarrow B$.” (b) shows the type rule for a $\text{sample}$ statement, which reads “For some expression $t$ which is a distribution over $A$, $\text{sample}(t)$ is a probabilistic expression of type $A$ in the context $\Gamma$. (c) shows the type rule for $\text{score}$, which reads “For some real number-valued expression $t$ in the context $\Gamma$, $\text{score}(t)$ is a probabilistic expression of which can only return one value (the unit value).” The other type rules define the rest of the type system in this way. The semantics of this language are discussed in a later section. Figure taken from [55].

2.7 This is adapted from the original presentation of consistency from the 1977 Abstract Interpretation paper by Cousot & Cousot [53]. $\alpha$ is the abstraction function, $\gamma$ is the concretization function, $\text{Init}$ is some program with maps inputs (subscripts $I$) to outputs (subscripts $O$). A line over a variable denotes that it is an abstracted version (the top part is abstracted). The $\supset$ sign denotes consistency, because there can be more than one concretization of an abstraction. An abstract interpretation is consistent if the result of flowing through the top, abstract path of the diagram produces a consistent result with flowing through the bottom, concrete part of the diagram. This is akin to a commutative diagram.

2.8 This shows how the Monotone Framework joins two branches of the control flow graph (such as after an $\text{if} \rightarrow \text{then} \rightarrow \text{else}$ statement) for some general abstract property. The transfer function is written in terms of $\text{gen}(l)$ and $\text{kill}(l)$, which denote the additions and removals of items from the abstract property set at the syntax element $l$. The $A_\bullet(l)$ shows the abstract property after executing the element $l$, and the $A_{\bullet}(l)$ shows the abstract property before the execution of the element $l$ as the combination of the output $A_{\bullet}s$ from previous nodes in the control flow graph. When this whole graph is drawn, the final $A_\bullet$ represents the final value of the abstract property for the whole program.

2.9 Shown here are examples of the inputs that produce each of four common (and very useful) classical static analyses. $L$ is the type of the property. $\sqcap$ is the way that properties are rejoined after control-flow branches. $\sqsubseteq$ and $\perp$ are the ordering relation and bottom element which define a lattice over properties, which is necessary for the Minimal Fixed Point Algorithm. $f_i$ shows the transfer function in terms of $\text{gen}$ and $\text{kill}$ functions.
2.10 This is a standard way of representing the semantics of a language: each element of the syntax is mapped, by the semantic bracket function, to an object in the semantic domain on the right hand side. In this case, the semantic domain is a probability kernel \([\Gamma] 
exists [A] = [\Gamma] \times \Sigma_{[A]} \to [0, \infty]\). Since the inputs (the program context \(\gamma\) and the event \(U\)) are supplied as subscripts on the left hand side, the right hand side is in \([0, \infty]\). (a) shows the semantics of \(\text{sample}\), and is read as “The density of \(\text{sample}(t)\) with context \(\gamma\) and event \(U\) is the meaning of \(t\) with context \(\gamma\) evaluated at \(U\).” Intuitively, the likelihood of drawing \(U\) from \(\text{sample}(t)\) is \(t(U)\), since \(t\) must be a distribution (from the typing rule marked (b) in Figure 2.6). (b) shows the semantics of \(\text{score}\), which is read as “The density of \(\text{score}(t)\) with context \(\gamma\) and event \(U\) is the value of \(t\) in the context \(\gamma\) if there is an event \(U\), otherwise 0.” Intuitively, \(\text{score}(t)\) has density \(t\) if it is evaluated to its return type (\(\cdot\)). The other statements define the rest of the semantics in a similar way. 

2.11 Shown here is the syntax of the untyped probabilistic lambda calculus in the same style as Figure 2.1. The Score and Random features are akin to the \(\text{score}\) and \(\text{sample}\) functions from the denotational semantics example language in Figure 2.10.

2.12 Shown here are rules for a call-by-value operational semantics for the untyped probabilistic lambda calculus. Rules are written in a similar style to typing rules, where the statements over the line imply the statement under the line. The \(A \Downarrow_D B\) syntax is read as “\(A\) evaluates to \(B\), adding \(C\) to the record of draws with density \(D\)”.

2.13 This is an example query from the Church language. The expression reads as: “Define a query over the model parameters \(\text{grass-is-wet}, \text{rain}, \text{and sprinkler}\), and return the distribution over \(\text{rain}\) given \(\text{day2}\) and conditioned on the predicate \((\text{grass-is-wet 'day2})\)”. Conditioning on this predicate almost always leads to a discontinuous density over the result of the query, even if the model parameters are continuous (they likely are not in this case). Figure taken from [52].

2.14 This shows an example usage of the C++ type \(\text{Uncertain<T>}\). Distance and speed are defined to be random variables. They are used to determine which branch of the \(\text{if}\) statement to follow, and so the only distributions that need to be inferred are the distributions over the predicate queries \((\text{Speed} > 4) \text{ and } (\text{Speed} < 4).\text{Pr}(0.9))\). Figure taken from [72].

2.15 This shows an example \(\text{Infer.NET}\) model which defines a graphical model with variables \(x, y, \) and \(mu\). \(\text{Infer.NET}\)’s primary backend uses variational inference. Figure taken from [74].
2.16 This shows a Pyro distribution. Pyro is written in Python for the Keras computational graph software platform. The function $normal_product$ defines two new, locally-defined model parameters. Stan, for example, does not allow parameters to be declared dynamically inside functions. Figure taken from [76].

2.17 On the left is shown a probabilistic program which computes the distribution of a random variable $l$. Since only $l$ is returned, we can eliminate computations that assign to variables which do not affect $l$. Figure 2.18 shows the probabilistic graphical model corresponding to the conditional dependence relationships between the variables in this program: using the usual rules for graphical models, we see that the value $s$ is independent of the value of $l$. The program on the right has the $s$ computations automatically removed but preserves the semantics of the full program.

2.18 The directed graphical model corresponding to the program in Figure 2.17. It encodes the following conditional independence: given $i$, $s$ is independent of $l$.

2.19 A Hakaru program. The first three lines are the type of the program. $noiseT$ and $noiseM$ are drawn from uniform distributions, $t1$, $t2$, $m1$ and $m2$ are drawn from normal distributions. The final line returns a joint distribution where $m1$ and $m2$ are observed variables and $noiseT$ and $noiseM$ are model parameters.

2.20 This is a Hakaru expression representing the program in Figure 2.19 conditioned on $m1$ and $m2$ (named here $x2$ and $x3$). The program will also be sliced with respect to the output variables $t1$ and $t2$ (now named $x4$ and $x6$). This representation is confusing to read because it has been automatically generated.

2.21 This is a Hakaru expression representing the conditional distribution in Figure 2.20 after being partially evaluated by Hakaru’s “Simplification” transformation, which applies algebraic reductions with the Maple software package. Again, this representation is difficult to read because it is automatically generated.

2.22 An R2 program before transformation. The sampling statements (e.g., line 6) are separated from the observation assertion (line 16), so some samples will disagree with the observations.

2.23 The R2 program after transformation. Each sampling statement is accompanied by the necessary conditions on the sample (for example, line 6 accompanied by line 7).

2.24 This is a deterministic part of the example program for the path sampling method.
2.25 This shows a probabilistic program in an imperative language designed to demonstrate the path sampling method. The underlined statements define verification queries. The queries ask whether the sensitivity of the \textit{estimateLogEGFR} function to its inputs meets a threshold. Figure 2.24 shows the full definition of the \textit{estimateLogEGFR} function. The key point from this function is that its behavior depends on branching conditions of its inputs.

2.26 Example before (a) and after (b) abstracting a probabilistic program into the predicate domain, considering the query $z = 0$. Figure taken from [65].

2.27 Enumerated here are the types of expressions in the SlicStan language. Each SlicStan expression is given a type $T$ which is a pair of a base type $\tau$ (taken directly from Stan) and a level type $\ell$. The level type indicates one of three options ($\text{DATA}$, \text{MODEL} and \text{GENQUANT}$), which indicate their block placement requirements.

2.28 Each Stan block, along with the Stan phase during which it is executed and the level of SlicStan statement that will be assigned to it. “Per chain” means that the block is executed every time the NUTS algorithm is restarted on a new starting point, which typically occurs many times during posterior inference. “Per leapfrog” means that the block is executed every time the NUTS algorithm takes a step, which occurs many times for each chain. As a result, for example, it is less efficient to place a statement in the model block than in the transformed data block.

2.29 On the left is shown a SlicStan program, which allows users to forgo blocks and define more flexible functions. On the right is shown the equivalent Stan program that will result from compilation. Only the necessary statements are included in the \textit{model} block of the resulting Stan program.

3.1 The correspondence between a Modular Stan program and a network of models. Holes are represented as empty shapes. Modules are represented as shapes filled with solid colors, having the same shape as the hole that they fill. (a) and (b) represent a Modular Stan program in its two parts, the base (a) and the module implementations (b). (c) represents the relationships between the base, holes and modules implementations. (d) represents the model graph, where each node is a model corresponding to a valid selection of modules, and each edge is labeled by the shape of the hole by which its endpoints differ, colored to show the change of modules. (e) shows the Stan program corresponding to a node; it is synthesized from the selected modules of the node.

3.2 The inputs to the \texttt{Concretize} operation are a modular program and a valid selection set (pictured here as a node in the model graph); the output is a Stan program (right).
3.3 The input to the ModelGraph operation is a module graph (left); the output is the corresponding graph of models (right).

3.4 Pathological module graphs. Round nodes represent the base and module implementations while rectangular nodes represent holes.

3.5 The inputs to the ModelNeighbors operation are the module graph (left) and a “center” model (circled in red); the output is the set of neighboring models (highlighted in red).

3.6 Collection hole module graph transformation. A collection hole \( H^+ \) produces a modified hole that lets a user select which implementations to include (yes) or exclude (no), and then concatenates the results (\( H_{merge} \)).

3.7 Indexed hole module graph transformation. An indexed hole \( H[1..N] \) produces \( N \) copies of its implementation.

3.8 Hole instance module graph transformation. When some code \( p_i \) refers to \( H<i> \), a new hole called \( H<i> \) is created with copies of the implementations of \( H \).

3.9 Hole product expansion shown as a module graph transformation. When some code \( p \) refers to a product \( H_1 * H_2 \), a new hole is created with the cartesian product of \( H_1 \) and \( H_2 \)’s implementations.

3.10 A labeled screenshot of the prototype web interface for Modular Stan.

3.11 Two views of the golf case study Modular Stan program. The models mentioned by the case study are labeled on the model graph in the bottom left of each panel. (a) shows a partial selection, \( \text{NSuccesses:binomial} \), and the model graph highlights the consistent subgraph. (b) shows a complete selection, which produces the concrete Stan program and its associated notes.

3.12 Taken from [32]. A visualization of time-series trends for the historical birth dataset, based on a Gaussian process model. The vertical axes are scaled so that values below 100 indicate a negative relationship with birth rate, while higher values indicate a positive relationship.

3.13 The model graph corresponding to the Birthday problem Modular Stan program. Each node represents a model and each edge represents one swapped-out module.

3.14 The red annotations show the ELPD scores for the assessed models. The search algorithm visited nodes along the red arrow path from starting at \([\text{START}]\) and terminating at \([\text{GOAL}]\).
3.15 This figure is taken from the “visualising-multiverse” vignette of [87], and is an example of a specification curve, first proposed by Simonsohn et al.. The first represents a multiverse analysis of a case study that examined the effects on a woman’s fertility due to various other variables, like political leaning and voting habits. The lower panels represent the various alternative options each model can take, for example, there were three alternative ways to include relationship status in each model. The dots on the horizontal axis show each model. The top panel arranges the model-dots vertically according to the strength of interaction they predict between fertility and relationship status. ............................................................... 144

3.16 This figure is taken from [93], and demonstrates the wide variety of possible visualizations for multiverse analyses. Subfigure e shows a specification curve of the same kind shown in Figure 3.15. ............................................................... 145

3.17 We compare the Valentine’s Day score of our candidate, “center” model (shown in black) with that of each neighboring model (shown in red). ............................................................... 148

3.18 We compare the November 1st score of our new center model (shown in black) with that of each neighboring model (shown in red). ............................................................... 149

4.1 A factor graph corresponding to a simple Stan program. The rectangular nodes correspond to the factors of the model’s density function, while the rounded nodes correspond to the model’s random variables, and edges are drawn between a factor and a variable if the factor includes the variable. ............................................................... 160

4.2 The factor graph form of a simple Stan program. ............................................................... 178

4.3 A DAG form of simple Stan program. ............................................................... 178

4.4 The factor graph form of the Stan program. ............................................................... 181

5.1 Two similar programs that result in different “prior” assignments by Pedantic Mode. 189

5.2 Shown here are factor graphs, where circular nodes represent random variables, rectangular nodes represent factors that contribute to the joint density of the variables, and edges represent a factor’s direct dependence on a variable. Filled circular nodes represent data variables, while empty circled nodes represent parameters. If an edge between a factor and a parameter is circled in blue dashes, it indicates that the factor is a prior for that parameter. Subfigures A and B represent programs A and B. In A, factors #1 and #2 are priors. In B, a new relationship is introduced between a and factor #3, which adds a path, shown in red arrows, from factor #2 to a the data variable x; factor #2 is therefore no longer a prior for b. ............................................................... 190
# List of Tables

2.1 This shows the properties for each of a small, diverse set of probabilistic programming languages. ............................... 37

3.1 Summary of macros .......................................................... 122

3.2 The change of the mean estimated effect of Valentine’s Day on birth rates, as each of the center model’s modules is changed. The left column shows the hole and module changed, and the right column shows the effect on the estimate. For example, the first row shows that the effect of changing the module that fills the DayOfWeekTrend hole from no to yes is to increase the estimate by 0.3268. . . 148

3.3 The change of the mean estimated effect of November 1st birth rates, as each of the center model’s modules is changed. ............................... 149
Acknowledgements

Thank you to my advisors, Jeannette Wing and Andrew Gelman, from whom I have learned so much and without whom I would have been in trouble. Thank you very much to my other thesis committee members: Baishakhi Ray, Matthijs Vákár, and Ronghui Gu. A huge thank you to Matthijs Vákár, Angie Moon, Aki Vehtari, Shinyoung Kim, Sangyoon Cha, Jonah Gabry, and many others, for fascinating discussions about the topics in this thesis. Thank you also to my former advisor, Itsik Pe’er, for all of his support.
Dedication

For my family, Alan, Sylvia and Kathryn, and for my future family, Talia.
Chapter 1: Introduction

Probabilistic modeling is the process of building, evaluating, and learning from mathematical models of systems that have random or uncertain aspects. A prototypical example of probabilistic modeling might go like this: a scientist has acquired some data, and they have only partial knowledge of the system from which the data was drawn. They want to leverage the data to make predictions or update their understanding of the system. They encode their partial knowledge as a mathematical model in terms of hidden random variables. They then combine their model with the data by a process called probabilistic inference, which results in new information about the system in terms of the likely values of the model’s hidden variables.

Probabilistic modeling is essential to science, and is also frequently used in industry and policy-making. For example, recently, probabilistic modeling has been used by researchers at the Laser Interferometer Gravitational-Wave Observatory (LIGO) to help confirm the detection of gravitational waves [89], by the Chicago Cubs to analyze their players’ performances \(^1\), and by the government of the United Kingdom to inform policy decisions regarding Covid-19 [88] \(^2\).

Because probabilistic inference is computationally challenging, especially with large modern datasets, most practical probabilistic modeling relies heavily on software tools. As a result, the nature and limitations of the software tools that are conveniently available have an impact on what models are built and how science progresses. These tools are developing rapidly.

The history of probabilistic modeling software has been driven mainly by computational constraints. It used to be that to do probabilistic modeling, one had to manually implement a numerical-analysis heavy program in a low-level programming language. Only later did statistical libraries in higher-level programming languages allow users to write models in terms of standard distributions

\(^1\)A representative of the Cubs contacted me and described their use of probabilistic modeling (and Stan).
\(^2\)Each of these applications used the probabilistic programming language Stan.
and random variables. Even when models could be expressed in probabilistic vocabulary, they still needed to be tailored for and intertwined with a probabilistic inference algorithm. Only recently has it become possible for users to express models with relatively little worry about the challenges of inference. These are the two trends we see as probabilistic modeling advances: firstly, users become more insulated from the computational details of inference; and secondly, they are afforded more powerful and expressive abstractions with which to describe their models.

Probabilistic programming is a recent technology that advances both of these trends. Probabilistic programming languages aim to let users write exactly their desired model in the form of a program, without any concern for the details of the inference algorithm. The compiler is then responsible for all of the details of translating the program into a form that can perform inference given data. With this tight program-to-model correspondence, probabilistic programming is perhaps the technology that promises the most semantic purity and expressive power to its users.

The purpose of this dissertation is to explore next steps in the trend towards more powerful probabilistic modeling abstractions. To that end, I apply methods and concepts from the study of programming languages to address current limitations of probabilistic programming. In particular, I approach two limitations of the semantics of probabilistic programs:

1. **A probabilistic program is a monolithic representation of a model.**

   The compiler views a probabilistic program as essentially a black-box function, which, when run a certain way, behaves like a probabilistic model.

   This view is enough to use the program for inference, but not to understand or manipulate the underlying model. If programming tools could instead interpret the model-level structure of a program, they could automatically analyze and even transform programs and the

---

3Probabilistic programs are technically many-to-one representations of probabilistic models, for the same reason that there can be many mathematical descriptions of the same probability distribution; rather they are many-to-one. What is important is that each probabilistic program represents one model without superfluous information. This is what we mean by “semantic purity”.

4This limitation is typical of, at least, major languages like Stan, due to their flexibility. Other languages, such as Infer.NET which uses a graphical model representation, have richer program semantics, but at the cost of expressive power. See Section 2.3.1 for more discussion of language flexibility and semantics.
models they represent, letting users interact with models at a higher level of confidence and abstraction.

2. **A probabilistic program can only represent a single model.**

Much of the work of probabilistic modeling does not take place within a single model, and so is currently beyond the scope of probabilistic programming. Model exploration, revision, optimization, evaluation, composition, and comparison are all modeling tasks that are done with multiple related models. If probabilistic programs could represent multiple related models, each of these tasks could be standardized, assisted, and even automated.

By addressing these two limitations, we can enrich the semantics of probabilistic programming, enabling software to handle more of the probabilistic modeling workflow.

In the next section, I give a brief introduction to probabilistic programming. (For a more thorough general introduction, see Section 2.1.1; for an simple example of a probabilistic model translated into a probabilistic program, see Section 4.2.1; and for an introduction to Stan, the particular language used throughout this dissertation, see Section 3.3.) In Section 1.2, I introduce a modern best practices probabilistic modeling workflow, and in Section 1.3, I describe how my contributions support that workflow. Section 1.4 lays out my contributions and the structure of the dissertation.

### 1.1 Probabilistic programming, briefly

Probabilistic programming languages are designed so that the meaning, or semantics, of a program is equivalent to a probabilistic model. There are various approaches to achieve this (see Section 2.3.1), but most designs are equivalent to augmenting a programming language with a notion of *sampling* a random variable from a distribution and *weighting* the likelihood of a configuration of random variables. These primitives let users describe the (unnormalized) joint probability distribution of their intended model, or, roughly, how likely it is for a particular dataset and a particular configuration of hidden variables to occur together. The joint distribution can also be split into
two intuitive factors: the *prior*, or, “how likely is a particular configuration of hidden variables, a-priori?”; and the *likelihood*, or, “how likely is the data, given particular hidden variables?” From this description of an (unnormalized) joint distribution, the compiler produces a new program that can accept an appropriate dataset and produce inferences about the *posterior* distribution of the hidden variables given the data. This process is pictured in Figure 1.1.

![Figure 1.1: A flow diagram of probabilistic programming. A program can be seen as a function from data and hidden parameters $\theta$ to a real number that is proportional to the joint distribution of the intended model. It is compiled into a program that accepts data and returns posterior draws. When combined with the data, inference is performed, and posterior draws are produced.](image)

The result is that a user can write a simple, expressive, and powerful representation of their model with little concern for the many details of the inference algorithm. In software engineering terms, the model specification is decoupled from the inference implementation, buying the user a valuable separation of concerns.

The semantic purity afforded by this separation of concerns makes probabilistic programs an effective medium to communicate models. While mathematics is the lingua franca of science, mathematical notation is not as unambiguous or practical as shared code. However, when publications share their modeling code, they often point to code repositories that are enormous, unstandardized, platform-dependent, mutable, and may not even match the given description. Probabilistic programs, by contrast, are often concise enough to embed into the publication body, resulting in a model specification that is machine-manipulable but also easy to understand and verify.

These are the promises of probabilistic programming, in any case. In reality, users do sometimes have to tailor their programs to work better with a particular inference algorithm, and sometimes general inference methods are not efficient enough for the task and users are forced to switch
paradigms. Regardless, probabilistic programming is perhaps the technology closest to the promise of fully decoupling the model description from inference methods.

1.2 Modern best practices: the Bayesian Workflow

To improve probabilistic modeling, we should aim to make best practices as easy as possible. A reasonable candidate for modern best practices, especially for Bayesian modeling, is the Bayesian Workflow [3].

The “Bayesian” in Bayesian Workflow refers to Bayesian statistics, which is a school of statistics that explicitly factors in a priori information and emphasizes the importance of considering the full uncertainty of variables, rather than aiming for point estimates of quantities.

A modeling workflow is a series of actions taken over the lifetime of a modeling project, starting with an initial dataset and some prior understanding of the system it was taken from, and ending with a probabilistic model, some inferred quantities like estimates of hidden variables or predictions of the future, and possibly real insights into the dataset and its generating process. We will focus in particular on the process of arriving at a model or models.

Figure 1.2 summarizes the Bayesian Workflow as a flowchart, starting with an initial model (upper left corner) and ending with multi-model comparisons and composites (lower right corner).

The first thing to notice about this workflow is that it has many cycles and encourages an iterative exploration of models. This is in the same spirit as the famous Box’s Loop [7]: we start with a model, we make inferences with it, we criticize those results, and we start again with new information or an improved model. The Bayesian Workflow goes further: we start with a model, we check whether it correctly encodes our a priori knowledge, or we iterate and start over. Then we check whether the model is computationally feasible with our inference algorithms; if not, we iterate and start over. Then we make inferences with it and criticize those results. Even if the model is reasonable, we may still keep looking for other reasonable options to compare the results.

The Bayesian Workflow recommends a variety of methods of criticism: we should perform diagnostics at every stage to check that the model works as expected and is consistent with our
Figure 1.2: A flowchart representing a best-practices workflow for Bayesian modeling, from Gelman et al. The green and yellow highlights show parts that are supported by the methods in this dissertation: diagnostic tests that utilize standardized model transformations are highlighted in green, and arbitrary, user-defined model transformations are highlighted in yellow.
available data and prior knowledge. Some of these diagnostics also hide program transformations (highlighted in green): the test is actually run using a separate but related probability distribution. Unfortunately, some of these tests can be cumbersome to set up and perform, especially for non-experts.

Many aspects of this workflow are challenging to perform, and may be cumbersome or error-prone for experts and out of reach for non-experts. For example, Simulation-based Calibration is a valuable diagnostic but it requires a particular rewrite of one’s model [79]. Multiverse analysis can give insight into the effects of modeling decisions, but the user will need to manage a potentially intractable “multiverse” of variations of their model [6]. The iterative model development process itself leaves users performing a blind, ad-hoc search of a potentially complex model space. New software methods can take the burden of complexity off the shoulders of users, just as probabilistic programming aims to do for probabilistic modeling and inference.

1.3 Supporting the Bayesian Workflow: Program Transformations

Many aspects of the Bayesian Workflow center around transformation of one model into another, closely related model. Each time a revision is made, such as in boxes 5 and 7 in Figure 1.2, a user transforms their model into another, likely very similar one. Diagnostic methods, like Simulation-based Calibration, Prior Predictive Checks, and Posterior Predictive Checks all involve drawing samples from a distribution that is closely related to, but distinct from, the model being tested. Methods that involve multiple models, such as those in the “Compare models (8)” box of the workflow, all operate on families of related models that can be seen as transformations of each other.

We can divide these transformations into two categories:

1. **Arbitrary, user-defined transformations.**

   These are the transformations that occur whenever a user updates their model or defines a model family (yellow highlights in Figure 1.2). While these updates often fall into certain
categories (adding prior information, reparameterizing, etc), they are so varied and context-specific that it is hard to categorize or automate them.

2. **Standard, generally-defined transformations.**

These are transformations with a widely applicable mathematical definition. In the Bayesian workflow, standardized transformations arise as parts of diagnostic tests (green highlights in Figure 1.2): Prior Predictive Checks, Simulation-based Calibration, and Posterior Predictive Checks all test a given model by deriving another model that is related in a standard way.

Model transformations are a central aspect of modeling workflow that has no support in our current software abstractions.

### 1.4 Contributions and roadmap

This dissertation focuses on exploring programming language methods to support arbitrary and standard model transformations, in order to automate and standardize the highlighted steps in Figure 1.2. I develop a method for representing arbitrary transformations of models as the swapping out of logical components of probabilistic programs, thereby bringing whole networks of models into the scope of probabilistic programming. I show a process for automating some standard model transformations, freeing users from manually (and possibly incorrectly or inefficiently) rewriting their models into diagnostic forms. I also describe a tool for performing automatic probabilistic program analysis using the same model understanding methods. Figure 1.3 shows the relationships of these contributions.

Here is a roadmap of the dissertation:

- Chapter 2 discusses the field of program analysis and transformation of probabilistic programs. I propose a taxonomy for analysis and transformation methods in order to categorize them according to their goals, flexibility, and the types of languages to which they can be applied. I then review a wide variety of existing methods according that taxonomy. This
Figure 1.3: The relation of scientific challenges, engineering methods, and probabilistic modeling applications in this dissertation. The two semantic limitations discussed at the beginning of this section correspond to the first and fourth boxes in the left column. The two types of model transformations are supported by the first and second boxes in the middle column. The yellow-highlighted parts of the modeling workflow in Figure 1.2 are supported by the first and second boxes in the right column, while the green-highlighted parts are supported by the third box in the right column.
section also serves as an introduction to probabilistic programming, static analysis, and their intersection.

- Chapter 3 presents Multi-Model Probabilistic Programming: an extension of probabilistic programming so that each program can represent networks of interrelated probabilistic models. Multi-Model Probabilistic Programming provides a way to represent arbitrary user-defined model transformations, gives first-class support for multiplicity in modeling, aids transparency of the model development process, and enables the automation of multi-model methods like those found in the “Compare models” box of Figure 1.2.

- Chapter 4 presents a method for automatically transforming probabilistic programs into certain diagnostic forms. These automated transformations let users avoid the error-prone and labor-intensive process of rewriting their model in a different form, letting them easily perform important correctness checks.

- Chapter 5 presents a probabilistic program analysis tool called “Pedantic Mode”, which uses the program decomposition methods described in Chapter 4 to analyze the structure of programs’ underlying models and provide automated statistical error-catching and feedback.
Chapter 2: A Systematic Review of Static Analysis of Probabilistic Programs

2.1 Introduction

The idea of statically analyzing the source code of probabilistic programs is relatively new, because probabilistic programming is relatively new. The idea has cropped up in the form of a disparate set of approaches that have been presented with varying purpose, technique and context of application. Thus far, there has been no systematic study of this field of static analysis of probabilistic programs as a whole. This section fills that role, with the following goals in mind:

- To give a context and motivation for the application of static analysis to probabilistic programs, such that the reader may understand and appreciate the important ideas in the field.
- To suggest some organization for the space of the field, as a means to locate ideas within the field.
- To present, categorize and discuss some existing ideas that have been explored in this space.
- To identify places where existing work could be fruitfully extended, and to suggest directions of potentially high impact for the future.

This chapter covers a representative set of probabilistic programming languages, but uses the popular Stan language [70] as the primary reference point. This chapter is based on my previously published work[97].

2.1.1 Introduction to Probabilistic Programming

Probabilistic programming provides a powerful means for specifying and computing with probabilistic models.
A probabilistic model is a probability distribution over some known variables, called data or observations, and unknown variables, called model parameters (or just parameters). Probabilistic models underlie statistical analyses and allow the analyst to generalize conclusions about quantities of interest beyond the observed data.

Often, the goal of writing a probabilistic model is to find the posterior probability distribution, also called the posterior distribution. The posterior distribution is the probability distribution over the parameters conditioned on the data, representing our updated beliefs about the model parameters given the new information found in the data.

A probabilistic program expresses a probabilistic model by defining a joint distribution. A joint distribution maps each possible pair of instantiations of data and parameters to a positive real number which corresponds to how “likely” that pair is to occur according to the model. This mapping is called the density function of the distribution\(^1\). When a distribution’s density sums (or integrates) to one, it is called a normalized distribution or a probability distribution. The joint distribution represented by a probabilistic program does not need to be normalized.

It is very convenient and flexible to write a probabilistic model as an unnormalized joint distribution. We can:

• Describe a process for generating the data in terms of the parameters, along with a distribution representing our prior beliefs about the parameters. This process corresponds to constructing a joint distribution by multiplying a likelihood and a prior.

• Simply specify a joint distribution directly as a function of the data and model parameters.

When the data are plugged into the (potentially unnormalized) joint distribution, we get an unnormalized distribution which is proportional to the posterior probability distribution. This is a corollary of Bayes’ rule.

\(^1\)Strictly speaking, the density function is a ratio between a distribution and a base measure, and some distributions do not have densities. Except when we deal explicitly in terms of measure theory, we will assume that every distribution has a density with respect to the Lebesgue measure. In that case, it is reasonable to think of the density as being a real number proportional to the probability.
This task of recovering the posterior probability distribution from a proportional unnormalized distribution is called \textit{posterior inference}, and it is the core computational challenge of probabilistic programming (and much of computational Bayesian statistics). Unfortunately, exact posterior inference is computationally intractable for most real-world problems, since it involves integrating over the whole space of parameters. We usually rely instead on approximate posterior inference methods. Some common approximate methods are described in section 2.2.1.

The advantages of probabilistic programs are enabled by the separation of concerns between the program, which is only concerned with representing probabilistic models, and the posterior inference engine, which is only concerned with calculating the posterior. This separation has a number of advantages over typical practices in applied statistics and machine learning:

- Probabilistic programs are freed of details of inference, letting users operate at a high level of abstraction. This makes models easier to write, debug, and iterate on. It also enables users to write programs which clearly communicate their domain knowledge.

- Improvements to posterior inference algorithms propagate for free to every probabilistic program, just by rerunning the model.

- As we will see, since probabilistic programs are written in standardized, computer-readable, and semantically meaningful format, we can apply static analysis for a number of additional benefits.

2.1.2 Introduction to Static Analysis

The term “static analysis” refers to a set of techniques for analyzing the source code of a program before it is run. The strategy is to enable as much work as possible at the point of compiling a program in order to gain some benefit every time the program is run.

Static analysis is pervasive in software engineering. For example, some common uses of static analysis are:
Figure 2.1: On the left is shown the list of grammatical rules for writing (a limited subset of) expressions in Stan. Expressions are pieces of the language which evaluate to values, such as numbers or strings. On the right is shown the list of rules for writing (a limited subset of) statements in Stan. Statements are pieces that take some imperative action, such as setting the value of a variable or running other statements in a loop. Not included here are the rules for writing a whole Stan program, which would describe the various blocks that are allowed, and how the blocks are made up of statements. The rules are read as follows: to generate a value of the type on the left hand side of the ::=, such as an E, pick one of the instances on the right hand side (separated here by lines), such as c. This means that a constant such as 1 is a valid Stan expression. Figure taken from [54].

- When a C program is made faster by GCC’s -O3 option, shaving seconds off the execution of a program that will be run a million times [82].

- When NASA engineers prove that the Mars rover’s piloting software will never divide by zero, ensuring the safety of a mission critical system [81].

- When a programmer’s code editor points out a mistake and suggests corrections in real time, saving hours of debugging time [83].

We will later see that these three examples are representative of three general purposes for static analysis: optimization, verification, and usability.

Approaches to static analysis vary, but one unifying theme is that they work by viewing a program as a syntax tree. Syntax is the set of grammatical rules that characterizes valid programs in a language, and allows each program to be structured into a tree with the whole program as the root and individual tokens as the leaves. For example, Figure 2.1 shows a subset of the expression and statement syntax of Stan.

These motivations for static analysis are easily extended to probabilistic programming. In-
ferring the posterior of a model can take a lot of computing time and power, and trusting that a model is correct can be expensive and/or risky. While typical programming can be said to have compile-time, run-time, and test-time phases, probabilistic programming can be said to have analogous compile-time, inference-time, and model checking-time phases. Each of these points could potentially benefit from static analysis using the new information available at each step.

2.1.3 Purposes of Applying Static Analysis to Probabilistic Programming

We can apply static analysis to probabilistic programs for a variety of reasons.

**Optimization**

The goal of optimization is, broadly, to get more done with fewer resources.

While optimization of traditional programs is done with respect to time or memory for a single execution of a program, optimization of probabilistic programs can consider the entire process of posterior inference, perhaps with respect to the average time or memory it takes to converge on a posterior distribution. This usually includes the traditional optimization of a program as a subproblem, but also considers how the model can be analyzed or transformed to better suit it for the particular inference method in use.

A second difference from optimization of traditional programs comes from the difference in the program semantics that need to be preserved. In traditional programming, the optimized program needs to have the same behavior when given the same inputs as the original. In probabilistic programming, the program needs to represent the same posterior distribution as the original. This difference leads to additional avenues for optimizing probabilistic programs.

The performance of probabilistic program inference is an important consideration. The performance advancements of the No-U-Turn Sampler algorithm [62], for example, have led to a significant number of practitioners turning to probabilistic programming for their probabilistic modeling tasks. The performance gains of variational inference, although often coming at a large cost of precision, have further expanded probabilistic programming into more applications. Fast model-
to-posterior times directly enable users to complete tasks that are otherwise infeasible, and lower barriers to users expressing models they believe are most correct. If we could perfectly optimize probabilistic programs, we could:

- Avoid all computation which is unnecessary, whether because it is irrelevant, overprecise, or redundant.
- Appropriately parallelize computation for hardware such as CPUs, GPUs and TPUs.
- Make all useful information available to the inference engine.

Verification

Software verification is the task of proving properties of programs. Examples of some typical types of properties are:

- A variable will never be negative;
- A loop will terminate;
- Some behavior is insensitive to perturbation in the program’s input.

Properties can be proven automatically by a variety of techniques, including by type systems or abstract interpretation, which are discussed in the technical background section.

When we extend verification to probabilistic programs, we can use the usual deterministic reasoning, but we can also extend both the properties and the proofs to include probabilistic reasoning. We can include random variables in the property, such as querying whether the expected value of a random variable will always be positive. We could also consider the probability of the truth of a property, such as asking whether a safety property holds with sufficient probability.

There are a number of types of errors in probabilistic programming that we might want to catch using verification.
• Depending on the probabilistic programming language, there might be any of the usual programming errors: infinite loops, null pointer references, buffer overflows and so on. This type of error can be caught by implementing the techniques of standard compilers.

• There might be an accidental mismatch between the program and the probabilistic model the user intended to represent. Software engineers call this a logic error. This type of error might be caught by warning the user of potential common errors or by checking the program against additional assertions supplied by the user.

• Even if a program perfectly represents the intended probabilistic model, the resulting posterior may still be incorrect if the model could not be inferred using the chosen inference method. Catching this type of error would be specific to the inference method in use.

Any progress toward catching these classes of error could automatically improve the modeling process for users en masse.

The ability in software engineering to automatically verify aspects of programs has become a vital safety feature for a variety of industries, such as healthcare, finance, and aerospace engineering. Probabilistic programming is in a unique position among machine learning and statistics tools to apply an analog of these automated proofs in data science. Given the calls for trustworthiness in machine learning and for robust statistical standards in science, any practical results in this space could have significant impact.

**Usability**

Static analysis can also be used to improve the human interface to probabilistic programming. In software engineering, static analysis is used to automatically generate, transform, and give immediate feedback on programs, and makes writing good programs easier and more accessible. This ease can certainly be translated to probabilistic programming. The main difference will be that interface improvements could potentially ease the entire modeling, inference and model criticism task, of which the programming task is a subset.
Advances in usability correspond to broadening the availability of probabilistic programming to a wider audience. It could also decrease the mental overhead of all users, allowing them to spend more resources on the quality of their models. Usability changes could also improve the interpretability of programs, which increases their usefulness as communication devices.

2.2 Technical background

2.2.1 Posterior inference

Posterior inference algorithms take in a probabilistic program and observations of the known variables, and produce a representation of the posterior distribution over the model parameters.

There are at least two popular flavors of approximate inference: sampling-based Markov Chain Monte Carlo methods and variational methods.

Markov-Chain Monte Carlo sampling

In Markov Chain Monte Carlo (MCMC) sampling methods, the goal is to draw random samples from the normalized posterior distribution by using some possibly unnormalized representation of the posterior. Sampling methods do not attempt to approximate the posterior as any closed-form function. MCMC is the only practical method available that produces an unbiased estimate of complicated probabilistic models.

If it is run forever, an MCMC algorithm will produce an unbiased estimate of the posterior distribution if the distribution is well-behaved. Sometimes distributions have disconnected regions, and so the process fails to discover and sample from some regions. Additionally, MCMC algorithms take some time to explore the space - initial samples will be highly correlated with the starting point and will not represent the distribution well. Because of this, the sampling process is usually allowed to run for a certain amount of time called a warm up period before the samples are

---

2 Some literature assumes that the data is embedded inside the probabilistic program, and so posterior inference would not need the data to be supplied separately. We often treat them as being supplied separately for clarity and to agree with the usage of languages like Stan. The two options are equivalent except when the data is needed for a compile-time static analysis.
Figure 2.2: This shows the steps of a trajectory of a particle simulated by NUTS for an underlying posterior distribution whose contour is shown in blue. The axes represent two arbitrary dimensions of the same scale. The trajectory is terminated approximately when the particle would start moving back on itself. NUTS samples efficiently from the eccentric ellipse-shaped distribution. Figure taken from [62].

recorded.

**Metropolis-Hastings.** Metropolis-Hastings (MH) algorithms are MCMC algorithms which have a particular method of sampling. The idea is to draw from a distribution called the *proposal distribution*, and then to either accept or discard that draw according to a certain probability called the *acceptance probability* [84]. The acceptance probability is a function of the unnormalized distribution that is carefully chosen to accept the appropriate distribution of draws according to the posterior distribution. Draws which move to higher density are more likely to be accepted. The proposal distribution at each step is a function of the previously accepted draw, making the sampling process into a Markov chain that explores the parameter space. There are many variants of MH, which differ mainly in their choice of proposal distribution.

In practice, MH with naive proposal distributions (such as Gaussian distributions centered at the last draw) can be very slow to explore high-dimensional spaces or spaces of difficult shapes, while more sophisticated variants like Hamiltonian Monte Carlo can achieve better performance.
Figure 2.3: Shown here are the Markov chain trajectories for two different MCMC variants given the same joint distribution (whose elliptical contour is shown as dotted lines). The axes represent two arbitrary dimensions of the same scale. Successive samples are shown as dots connected by lines. The random-walk Metropolis algorithm has a Gaussian proposal distribution at each step, where the Hamiltonian Monte Carlo moves each step according to a particle simulated under Hamiltonian dynamics after an initial random momentum. The posterior is a difficult shape for the random-walk algorithm to traverse, but the gradient-informed HMC easily takes steps across the space. Figure taken from [77].

Hamiltonian Monte Carlo and the No-U-Turn Sampler. Hamiltonian Monte Carlo (HMC) is a variant of MH that is particularly effective at quickly exploring difficult, high-dimensional distributions.

HMC works by using a connection between physics and probability theory: If a particle with random momentum is moving around a space, a snapshot of the particle’s location will be distributed according to a probability distribution that is related to the potential energy function of the space. So, to draw a random sample from a probability distribution, we set up a space with the appropriate potential energy function, we simulate a particle moving and sample the particle’s location. This process fits into the MCMC framework: the proposal distribution for the next sample is defined by the physical simulation from the last accepted sample.

The No-U-Turn Sampler (NUTS) (Figure 2.2) is an extension of HMC that automatically and dynamically selects runtime parameters of the HMC process, such as the length of each simulation between samples, to avoid inefficient particle movements.

HMC/NUTS is remarkably fast and accurate for high-dimensional problems, and is one of the standard workhorses in probabilistic programming. It tends to explore difficult and high-
Figure 2.4: Shown here are the results of applying two variants of variational inference (red) to an underlying bimodal posterior distribution (black). (a) should VI with KL(P||Q), which tends to give an estimate more spread out than the underlying distribution. (b) and (c) show two possible results of VI with KL(Q||P), which tends to give more peaked estimates than the underlying distribution.

MCMC algorithms are approximate and stochastic, and as such it can be difficult to assess the correctness of their posterior estimates. There are a number of standard statistical tools for diagnosing problems, such as posterior predictive checks, which check the posterior samples against the available data [80], and Simulation-Based Calibration, which simulates data using the generative distribution and checks the quality of the inferred posterior distributions against the simulated parameters [79]. There also exist diagnostic metrics specifically for measuring the convergence of MCMC algorithms [78].

In order to work, HMC/NUTS requires the gradient of the density function. This means that probabilistic programming languages which support HMC/NUTS usually implement automatic differentiation in their backend. It also means that those languages are restricted to differentiable density functions, which restricts the language semantics.

Other MCMC sampling algorithms require different structure from the density, such as the Hessian matrix at each point. Efficient Gibbs sampling requires a local statistical dependency set for each parameter called a *Markov blanket*, which can either be gained from language design or by static or dynamic analysis [66].
Variational methods

Variational inference (VI) attempts to find an approximation to the posterior distribution within a restricted solution space of distributions called a variational family. Parameters in the variational family are optimized to minimize a function called the Evidence Lower-Bound, which is a computationally tractable way of minimizing the KL-divergence, a measure of dissimilarity between the true posterior distributions and the approximation from the variational family. Figure 2.4 shows example results of VI.

VI requires some variational family to be specified, and its accuracy can depend heavily on the choice of family. VI will not converge to the true posterior if it is not included in the family, and the resulting approximation will be biased depending on the exact definition of KL-divergence used. However, VI often converges faster than sampling methods, and sometimes scales better for large datasets in practice [85].

Like MCMC algorithms, VI is an approximate method, and so assessing its correctness is also challenging. The standard posterior diagnostic tools such as posterior predictive checks and Simulation-Based Calibration also apply to VI. In addition, since VI is a (typically non-convex) optimization, converging to local optima is also a concern.

Some flavors of VI require first- or second- derivatives over the density, and so are subject to the same restrictions as MCMC algorithms which require the same.

2.2.2 Static analysis

A wide variety of techniques are available for static analysis. Exploring this toolkit helps us understand the space of possibilities that can be applied in a probabilistic setting. This section briefly covers three topics that come up repeatedly in the current approaches for probabilistic programs, and which will likely be important for future advancements.
Figure 2.5: Shown are the types allowed by an example probabilistic programming language [55]. In English, it says: “Types named $A$ or $B$ will be one of: a real number $R$, a probability distribution over another type $A$, the unit value (which is only ever one value), a pair of two values of types $A$ and $B$, or a type that can be any of the types $A_i$ for $i \in I$.” Effectively, this language allows real numbers, distributions, a singleton type, and any pair or disjunction of types. Figure taken from [55].

![Types](image)

Figure 2.6: Shown are type rules sufficient to type-check the example probabilistic programming language [55]. A rule asserts that the statements over the line imply the statements below the line. $\Gamma$ represents the context available from the rest of the program. $\Gamma \vdash x : A$ means “the context implies that an expression $t$ has type $A$”. The subscript $x$ is used to distinguish between the implication of deterministic expressions with $\vdash_d$ and probabilistic expressions with $\vdash_p$. Non-probabilistic programs would only have $\vdash_d$, and would not include rules for $B_0<?;4$, $B_2>A4$ or $\Rightarrow A<0;8I4$. (a) shows a non-probabilistic example, which intuitively reads: “For some expression $C$ with type $A$ from the context $\Gamma$, $\vdash_p f(t) : B$ is of type $B$ in the context $\Gamma$ for some measurable function $f : [A] \rightarrow [B]$ measurable.” (b) shows the type rule for a $sample$ statement, which reads “For some expression $t$ which is a distribution over $A$, $\vdash_p sample(t) : A$ is a probabilistic expression of type $A$ in the context $\Gamma$. (c) shows the type rule for $score$, which reads “For some real number-valued expression $r$ in the context $\Gamma$, $\vdash_p score(t) : 1$ is a probabilistic expression of which can only return one value (the unit value).” The other type rules define the rest of the type system in this way. The semantics of this language are discussed in a later section. Figure taken from [55].
Type systems

Types are descriptions associated with components of a program’s syntax tree. A component’s type describes properties of the value of the expression. For example, int and float are common names for types which correspond to integral and floating-point values.

Types are useful for restricting the set of programs that a compiler will allow. The compiler has a set of rules, called typing rules, which enumerate the ways that typed expressions and statements are allowed to fit together, and what their combined type is if they do fit together. If the compiler finds a point in a program which does not match any rule, the compiler rejects the program, and the program is said not to have type checked. For example, most compilers will reject statements like \( \text{int } x = 0.5; \) because they do not have a rule allowing the assignment of 0.5 to a variable of integer type.

Figure 2.5 shows the set of types allowed in the example probabilistic programming language from Staton [55], and Figure 2.6 shows the set of typing rules from the same language. These rules are sufficient to check if a program written in this language will type check and to infer the types of each part of the program.

If a program type checks, the programmer can be reasonably sure that the program is free of some class of bugs. Most compilers can promise that, for example, there are no floats masquerading as integers and each function is called with compatible parameters. The programmer can earn more peace of mind from more powerful type systems, such as ones that can express if a value falls within a certain range, or is a vector of a certain length. As type systems become more powerful, they can encode more of the specification of the program, and any program that successfully type checks is more likely to be correct. Encoding type systems so that they reject programs with important classes of bugs is a rich field of study.

If a program passes a type checker, the compiler has effectively proven something about the program. Writing type annotations into the program is akin to writing a proof about the program that can be checked automatically. Some compilers can infer some or all of the types of expressions based only on their usage - this is akin to the compiler automatically writing and checking a proof.
Figure 2.7: This is adapted from the original presentation of consistency from the 1977 Abstract Interpretation paper by Cousot & Cousot [53]. $\alpha$ is the abstraction function, $\gamma$ is the concretization function, Int is some program with maps inputs (subscripts I) to outputs (subscripts O). A line over a variable denotes that it is an abstracted version (the top part is abstracted). The $\geq$ sign denotes consistency, because there can be more than one concretization of an abstraction. An abstract interpretation is consistent if the result of flowing through the top, abstract path of the diagram produces a consistent result with flowing through the bottom, concrete part of the diagram. This is akin to a commutative diagram.

for the user. Compilers with advanced type systems often employ software such as Satisfiability Modulo Theory (SMT) solvers as workhorses for automated theorem proving in this manner.

Types can also be useful tools for compilers to take into account in order to generate efficient code.

Abstract Interpretation

Abstract Interpretation (AI) is a foundational and general framework for reasoning about properties of programs and program parts. It provides a vocabulary and some useful mathematical tools. Some work has already been done to extend it into probabilistic settings.

In AI, the full meaning of a program is called the concrete semantics. The concrete semantics captures the whole behavior of a program when it is executed. The concrete domain is the space of possible concrete semantics for programs.
An abstract domain is a simplification of the concrete domain that only captures some interesting aspect of the computation. The function that maps from a concrete meaning to an abstract meaning is called an abstraction function, and the reverse is called the concretization function. Given an abstraction function, a concrete program with concrete semantics can be transformed into an abstract program with abstract semantics. The abstract program does away with all details from the concrete program that do not affect the abstract domain. This abstract program is called an abstract interpretation of the concrete program.

An abstraction interpretation is called consistent if, for each possible input, the output of the abstract program is consistent with the output of the concrete program. A more formal statement is given in Figure 2.7.

For example ([53]), consider a program in an arithmetic language: \(-1515 \times 17\), and suppose we care only about the sign of the result. We can use the abstraction of signs: the concrete domain of integers is abstracted into the abstract domain of signs. The abstract program is then \(-(+) \times (+)\), where \(\times\) is redefined to work on signs, and it is now easy to compute that the outcome will be \((-)\). This abstract interpretation is consistent since, for all such arithmetic programs, the abstract result will have the sign of concrete result.

This style of reasoning can be extended to more sophisticated properties of programs, and serves as the foundation for other application approaches.

**Monotone Framework**

The Monotone Framework is a framework for applying the idea of abstract interpretation to programs in a mechanical way. Most of the standard set of program analyses fit nicely into this framework.

The Monotone Framework [57] is a process for finding abstract properties of the program at each point in its execution. The framework requires the following inputs:

- A function from a syntactic element (usually a statement) to a transformation of the abstract property. This is also called the transfer function. The transfer function is usually described
Figure 2.8: This shows how the Monotone Framework joins two branches of the control flow graph (such as after an \textit{if} -- \textit{then} -- \textit{else} statement) for some general abstract property. The transfer function is written in terms of \textit{gen}(l) and \textit{kill}(l), which denote the additions and removals of items from the abstract property set at the syntax element \textit{l}. The \(A_\ast(l)\) shows the abstract property after executing the element \textit{l}, and the \(A_\varnothing(l)\) shows the abstract property before the execution of the element \textit{l} as the combination of the output \(A_\ast s\) from previous nodes in the control flow graph. When this whole graph is drawn, the final \(A_\ast\) represents the final value of the abstract property for the whole program.

<table>
<thead>
<tr>
<th>Available Expressions</th>
<th>Reaching Definitions</th>
<th>Very Busy Expressions</th>
<th>Live Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>(L)</td>
<td>(\mathcal{P}(A\text{Exp}_\ast))</td>
<td>(\mathcal{P}(\text{Var}<em>\ast \times \text{Lab}</em>\ast^2))</td>
<td>(\mathcal{P}(\text{Var}_\ast))</td>
</tr>
<tr>
<td>(\sqsubseteq)</td>
<td>(\sqsupseteq)</td>
<td>(\sqsubseteq)</td>
<td>(\sqsubseteq)</td>
</tr>
<tr>
<td>(\sqcup)</td>
<td>(\cap)</td>
<td>(\cup)</td>
<td>(\cup)</td>
</tr>
<tr>
<td>(\bot)</td>
<td>(\mathcal{A}\text{Exp}_\ast)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(E)</td>
<td>({\text{init}(S_\ast)})</td>
<td>({\text{init}(S_\ast)})</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(F)</td>
<td>(\text{flow}(S_\ast))</td>
<td>(\text{flow}(S_\ast))</td>
<td>(\text{flow}^R(S_\ast))</td>
</tr>
<tr>
<td>(\mathcal{F})</td>
<td>({f : L \to L</td>
<td>\exists k, l_g : f(l) = (l \setminus l_k) \cup l_g})</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(f_t)</td>
<td>(f_t(l) = (l \setminus \text{kill}([B]^t)) \cup \text{gen}([B]^t)) where ([B]^t \in \text{blocks}(S_\ast))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.9: Shown here are examples of the inputs that produce each of four common (and very useful) classical static analyses. \(L\) is the type of the property. \(\sqsubseteq\) is the way that properties are rejoined after control-flow branches. \(\sqcup\) and \(\sqcap\) are the ordering relation and bottom element which define a lattice over properties, which is necessary for the Minimal Fixed Point Algorithm. \(f_t\) shows the transfer function in terms of \textit{gen} and \textit{kill} functions.
in terms of the *gene* function, which describes what is added to a property, and the *kill* function, which describes what is removed from a property.

- A way of combining abstract properties from different program branches of control flow (such as an *if* statement’s *then* and *else* branches). This is pictured in Figure 2.8.

- The syntax tree of a program.

- The control flow graph over the program, which specifies the branching structure of the statements.

The framework then produces a safe over- or under-approximation of the program property that holds at each point in the execution of the program, using the so-called Minimal Fixed Point Algorithm.

For example, consider the abstract property called the Reaching Definitions (RD) set. A reaching definition at a point is a definition (e.g. \( x = 5 \)) which has not been overwritten up to this point in execution. The RD set is a set containing each reaching definition. For some programming language, we could describe how the syntax generates reaching definitions by changing a variable definition (the *gen* function) or removes reaching definitions by overwriting a variable definition (the *kill* function). We could define the way that the RD property is joined together across branches as taking a set union. We could then use the Monotone Framework to find the RD set at each point in a given program. The RD property is a prerequisite for building a dependence analysis graph.

Figure 2.9 shows similar Monotone Framework inputs for three other classic analyses.

2.2.3 Probabilistic Programming from a Static Analysis and Programming Languages Perspective

In order to apply the rich theory of programming languages to probabilistic programs, we should first answer the fundamental question: what does a probabilistic program mean? When we type a probabilistic program into a text file, we know it represents an unnormalized probability distribution - but how exactly are the symbols that make up a program mapped to the mathematical
object of a distribution? If we can answer this question with sufficient rigor, we will have a good foundation to apply existing static analysis tools to probabilistic programs.

A formal language semantics is a mapping from each valid program to its meaning. The space of all meanings is called the semantic domain.

This level of rigor allows us to reason with mathematical rigor about a number of questions about correctness in probabilistic programming:

- Language correctness: is the language specification complete and correct?
- Correctness of posterior inference: will a posterior inference method given a program produce results consistent with the semantics of that program?
- Program transformation: When a transformation (such as a compiler optimization) is applied to some program A resulting in program B, will the semantics of A match the semantics of B?
- Program verification: What property can we prove about a program A that implies a property of the semantics of A?

For static analysis, we are especially interested in the help that formal semantics gives us with program transformation and verification. Keeping this formal foundation in mind helps to develop and prove correctness of static analyses.

There are two primary approaches that have been used most often to build formal semantics for probabilistic programming languages: denotational semantics and operational semantics. The following sections introduce them and give an example language and semantics for each.

**Denotational Semantics**

A denotational semantics assigns a mathematical object to each component of the program, such that the meaning of the program is the composition of the meanings of its components. A

---

3Axiomatic semantics would be a third example.
Figure 2.10: This is a standard way of representing the semantics of a language: each element of the syntax is mapped, by the semantic bracket function, to an object in the semantic domain on the right hand side. In this case, the semantic domain is a probability kernel
\[ J \Gamma \to \Sigma [A] \to [0, \infty]. \] Since the inputs (the program context \( \gamma \) and the event \( U \)) are supplied as subscripts on the left hand side, the right hand side is in \([0, \infty]\). (a) shows the semantics of \textit{sample}, and is read as “The density of \textit{sample}(t) with context \( \gamma \) and event \( U \) is the meaning of \( t \) with context \( \gamma \) evaluated at \( U \).” Intuitively, the likelihood of drawing \( U \) from \textit{sample}(t) is \( t(U) \), since \( t \) must be a distribution (from the typing rule marked (b) in Figure 2.6). (b) shows the semantics of \textit{score}, which is read as “The density of \textit{score}(t) with context \( \gamma \) and event \( U \) is the value of \( t \) in the context \( \gamma \) if there is an event \( U \), otherwise 0.” Intuitively, \textit{score}(t) has density \( t \) if it is evaluated to its return type \( () \). The other statements define the rest of the semantics in a similar way.

denotational semantics is defined by writing \( \llbracket S \rrbracket = V \) for each element \( S \) of the syntax, where \( V \) is the semantic value of the element in the semantic domain. Denotational semantics is most natural to define for expressions as opposed to imperative statements, and works especially well for languages where everything is an expression with a value.

1. Example: A measure theoretic semantics for a sample/score language

An example of an expression-only probabilistic programming language is given in [55], whose type system is shown in Figure 2.5 in section 2.2.2.

Staton [55] gives a denotational semantics for this language is given in terms of measure theory:

- A type \( A \) is interpreted in the semantics (written \( \llbracket A \rrbracket \)) as a \textit{measurable space}. A mea-
surable space is a set along with a $\sigma$-algebra $\Sigma_A$, which is a set of subsets of the space that can be interpreted as the set of subsets that can be formally assigned probabilities, called \textit{events}.

- A deterministically-valued expression which depends on some surrounding context is a \textit{measurable function} from the context to the expression type. A measurable function is a well-behaved function such that the preimage of a measurable set is also measurable. This can be written: $[\Gamma \vdash^d E] : [\Gamma] \rightarrow [A]$. The context $\Gamma$ is the set of variables bound in the surrounding program. $A$ is the type of the expression $E$.

- A probabilistically-valued expression which depends on some surrounding context is a \textit{measurable kernel} from the context to the expression type. A measurable kernel is a mapping from the context to a measure on the output. This is written: $[\Gamma \vdash^p E] : [\Gamma] \rightsquigarrow [A]$, which is equivalent to $[\Gamma] \times \Sigma_{[A]} \rightarrow [0, \infty]$.

- Since a whole program in this language is a probabilistic expression without a surrounding context, the whole program is a measure on the model variables.

The full denotational semantics for this language is shown in Figure 2.10.

If we repeatedly apply these rules, we reduce a program from a syntactic tree to a mathematical object - in this case, a measure on the model variables.

2. Example: Probabilistic graphical models as a semantic domain

Since Infer.NET programs correspond directly to directed, acyclic probabilistic graphical models, it would be reasonable to define a denotational semantics for Infer.NET with directed probabilistic graphical models as the semantic domain. Drawing a variable from a distribution would correspond to a node with edges to the nodes representing the distribution parameters. An example Infer.NET program is shown in Figure 2.15.
Figure 2.11: Shown here is the syntax of the untyped probabilistic lambda calculus in the same style as Figure 2.1. The Score and Random features are akin to the \textit{score} and \textit{sample} functions from the denotational semantics example language in Figure 2.10.

Figure 2.12: Shown here are rules for a call-by-value operational semantics for the untyped probabilistic lambda calculus. Rules are written in a similar style to typing rules, where the statements over the line imply the statement under the line. The $A \Downarrow_D B$ syntax is read as “$A$ evaluates to $B$, adding $C$ to the record of draws with density $D$”. (a) shows the rule for \textit{score}, which reads “For some constant $c \in (0, 1]$, \textit{score}(c) evaluates to \textit{true}, does not add any draws to the record, and has density $c$..” Score is used to explicitly scale the density at the present point in the distribution. (b) shows the rule for drawing random values from distributions, and reads “If $D(\widetilde{c})$ has density $w$ at $c$, $D(\widetilde{c})$ evaluates to $c$ with density $w$ and adds $c$ to the record of draws.” The other rules define the rest of semantics in this way.

\textbf{Operational Semantics}

An operational semantics assigns each component of the program to a mathematical object, but also describes how the component operates on some global meaning of the program. Operational semantics tend to be convenient to define for statements which can perform side-effects that change the meaning of the program, rather than just representing a value.

1. Example: Trace sampling semantics for a probabilistic lambda calculus

Borgström \textit{et al.} [52] define and provide an operational semantics for a probabilistic pro-
Programming language called the untyped probabilistic lambda calculus. The syntax for the language is shown in Figure 2.11. Roughly speaking, a lambda calculus is a simple language centered around defining and applying functions. The language in Figure 2.11 extends an untyped, call-by-value lambda calculus to include sampling random variables from distributions.

The full operational semantics for this language is shown in Figure 2.12. The semantic domain is a pair of: (1) a record of all of the random samples drawn, and (2) the density of the unnormalized joint distribution evaluated at those sampled points. The operational semantics defines how each element of the syntax contributes to this meaning by adding onto it.

Borgström et al. go on to use this operational semantics to prove that the posterior samples from running the Trace-MCMC algorithm on a program is consistent with the semantics of the program. The ability to give such a formal correctness proof is an example of the advantages of formal semantics for probabilistic programming languages.

2.3 Properties of PPLs salient for static analysis

Not all static analysis techniques make sense to apply to all probabilistic programming languages. This section enumerates a set of properties of probabilistic programming languages which, taken together, determine the applicability of a given static analysis approach. The author does not claim that this proposed set is complete or that it will suffice in the future, but at least that they are useful to keep in mind when exploring static analysis approaches. The properties of a small, diverse set of probabilistic programming languages are shown in Table 2.1.
Discontinuous density functions

Some languages and inference methods only work well (or at all) with distributions with densities that are continuous, differentiable, or differentiable to a higher order (as is the case with some MCMC variants). This requirement is sometimes incompatible with static analysis techniques that utilize discontinuous densities.

While it is possible to represent a Stan program with a discontinuous density function (for example, a program containing control flow that depends on a model parameter), Stan does not claim to handle this case well with NUTS. The particle trajectory that HMC simulates to draw samples is difficult to simulate with a discontinuous energy function. The language Church, on the other hand, was built to support conditioning on predicates over parameters (Figure 2.13), making discontinuities typical. Church generally uses non-HMC MCMC sampling.

Some static analysis techniques are not applicable to probabilistic programming languages that can not handle discontinuity, often because they utilize parameter-dependent control flow that introduces discontinuity.

Predicate query oriented

Some languages are designed to give a posterior probability of some predicate query over the parameters, rather than estimating the full posterior distribution of the parameters. One example
Figure 2.14: This shows an example usage of the C++ type Uncertain<T>. Distance and speed are defined to be random variables. They are used to determine which branch of the if statement to follow, and so the only distributions that need to be inferred are the distributions over the predicate queries (Speed > 4 and (Speed < 4).Pr(0.9)). Figure taken from [72].

```cpp
double dt = 5.0; // seconds
Uncertain<GeoCoordinate> L1 = GPS.GetLocation();
while (true) {
    Sleep(dt); // wait for dt seconds
    Uncertain<GeoCoordinate> L2 = GPS.GetLocation();
    Uncertain<double> Distance = GPS.Distance(L2, L1);
    Uncertain<double> Speed = Distance / dt;
    print("Speed: " + Speed.E());
    if (Speed > 4) GoodJob();
    else if ((Speed < 4).Pr(0.9)) SpeedUp();
    L1 = L2; // Last Location = Current Location;
}
```

Figure 2.15: This shows an example Infer.NET model which defines a graphical model with variables $x$, $y$, and $\mu$. Infer.NET’s primary backend uses variational inference. Figure taken from [74].

of such a language is Uncertain<T> (Figure 2.14).

Some static analysis techniques rely on this predicate query to structure the computation, and usually involve reducing the program to focus on only answering the query. It is worth noting that languages which infer full distributions can benefit from these approaches in the context of verification, where a predicate query is specified or generated separately from the program definition.

### Sampling-based inference

Some languages specialize in sampling-based inference (MCMC), where other languages focus on other methods (such as variational methods).

Stan is an example of language focused on its MCMC backend, while Infer.NET (Figure 2.15)
Figure 2.16: This shows a Pyro distribution. Pyro is written in Python for the Keras computational graph software platform. The function `normal_product` defines two new, locally-defined model parameters. Stan, for example, does not allow parameters to be declared dynamically inside functions. Figure taken from [76].

focuses on variational methods.

Some static analysis for probabilistic programming approaches specifically target the process of drawing samples in some way, and would not make sense to apply to languages focusing on other inference methods.

**Dynamic parameter declaration**

In most probabilistic programming languages, there is a constant set of model parameters defined over a program. This structure is not required in, for example, Pyro (Figure 2.16). Pyro allows new model parameters to be declared depending on the branch of execution.

Some probabilistic programming-specific static analysis approaches lack the flexibility to be compatible with this level of flexibility, such as those which build and analyze graphical models to transform the program. These methods could potentially be extended to handle this flexibility by making conservative approximations.

2.3.2 Properties of representative languages

Table 2.1 shows examples of how the properties apply to real-world languages.
Discontinuous
density
functions
Sampling-
focused
inference
Predicate
query
oriented
Conditional
variable
declaration

<table>
<thead>
<tr>
<th></th>
<th>Discontinuous</th>
<th>Sampling-</th>
<th>Predicate</th>
<th>Conditional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stan [70]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncertain&lt;T&gt; [72]</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>Infer.NET [74]</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Church [75]</td>
<td>✔️</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Pyro [76]</td>
<td>✔️</td>
<td></td>
<td></td>
<td>✔️</td>
</tr>
<tr>
<td>PSPs [73]</td>
<td>✔️</td>
<td></td>
<td></td>
<td>✔️</td>
</tr>
</tbody>
</table>

Table 2.1: This shows the properties for each of a small, diverse set of probabilistic programming languages.

2.4 Current techniques

2.4.1 Overview

There have been important and interesting contributions to the field of static analysis for probabilistic programming. They vary in their goal, approach and what they assume about the languages they target. It is not uncommon for a technique to be presented alongside an entirely new language on which to demonstrate it. This section categorizes these contributions first according to their purpose and broad strategy, then to describe their approach and enumerate the language properties that they assume.

2.4.2 Optimization

Perhaps the majority of current work focuses on the optimization of probabilistic programs.

Avoiding work

The primary strategy for classical optimization is to avoid doing work during program execution in some way. This idea extends to probabilistic programming by avoiding work during posterior inference. The probabilistic semantics do require some additional considerations.

1. Program slicing and dead code elimination
The idea of program slicing is to find the “slice” of a program which contributes to a value at some point. This task is highly related to the goal of dead code elimination, which aims to remove parts of the program which do not contribute to the final outcome of the execution. For our purposes, we can consider them to be the same problem, where dead code elimination preserves only the “slice” which contributes to the result of the program.

In the case of probabilistic programming, we seek to find the slice which contributes to the distribution that the program represents. We can apply the usual dead code elimination techniques to non-probabilistic portions of the program, but we can also sometimes eliminate probabilistic portions like extraneous model parameters. The examples below focus on eliminating unnecessary model parameters.

**Example: Slicing probabilistic programs.** Program slicing is made a little bit more complex in the probabilistic setting because there can be indirect dependencies between model parameters which are not reflected directly by the syntax. Hur et al. [51] provide a technique for program slicing which is aware of these indirect relationships, and can eliminate code to compute certain variables based on their independence from the model parameters being estimated. Figure 2.17 shows how a program is transformed.

**Example: Hakaru.** Another example of program slicing is implemented into the compiler of the probabilistic programming language Hakaru. During Hakaru’s “disintegration” phase, a program representing a joint measure is rewritten into a representation of a conditional distribution, and during this phase Hakaru drops pieces of the joint measure expression that do not contribute to the conditional distribution. Figure 2.19 shows a Hakaru program before disintegration and Figure 2.20 shows the representation after disintegration.

This technique is most effective when the user specifies a query that does not require inferring the full posterior of all model parameters. Stan does not have a facility for the user to specify queries, and so must assume that all parameters are important. Church explicitly
Figure 2.17: On the left is shown a probabilistic program which computes the distribution of a random variable $l$. Since only $l$ is returned, we can eliminate computations that assign to variables which do not affect $l$. Figure 2.18 shows the probabilistic graphical model corresponding to the conditional dependence relationships between the variables in this program: using the usual rules for graphical models, we see that the value $s$ is independent of the value of $l$. The program on the right has the $s$ computations automatically removed but preserves the semantics of the full program.

Figure 2.18: The directed graphical model corresponding to the program in Figure 2.17. It encodes the following conditional independence: given $i$, $s$ is independent of $l$. 

39
Figure 2.19: A Hakaru program. The first three lines are the type of the program. noiseT and noiseM are drawn from uniform distributions, t1, t2, m1 and m2 are drawn from normal distributions. The final line returns a joint distribution where m1 and m2 are observed variables and noiseT and noiseM are model parameters.

This technique requires a static analysis pass to find the dependency graph between variables. The dependency graph can be derived from the Reaching Definitions property found from the Monotone Framework. The dependency graph can be extracted from programs in any probabilistic programming language with varying levels of difficulty and specificity, and so program slicing can be applied in some form to any language.

2. Partial evaluation

Rather than avoiding computation altogether, partial evaluation attempts to compute as much as possible at compile-time. This is a very general and effective technique that is already applied as an optimization step in some probabilistic programming languages.

For example, the Hakaru language implements partial evaluation by translating pieces of the program into an algebraic language, passing it to an algebraic simplifier, and then translating back to the Hakaru representation. Hakaru calls this the simplification phase. A Hakaru representation before simplification is shown in Figure 2.20 and after simplification in Figure 2.21. This approach can afford Hakaru extra efficiency, depending heavily on the choices of variable distributions and the strength of the solver.

The implementation of partial evaluation probabilistic languages is not significantly differ-
Figure 2.20: This is a Hakaru expression representing the program in Figure 2.19 conditioned on \( m1 \) and \( m2 \) (named here \( x2 \) and \( x3 \)). The program will also be sliced with respect to the output variables \( t1 \) and \( t2 \) (now named \( x4 \) and \( x6 \)). This representation is confusing to read because it has been automatically generated.

Figure 2.21: This is a Hakaru expression representing the conditional distribution in Figure 2.20 after being partially evaluated by Hakaru’s “Simplification” transformation, which applies algebraic reductions with the Maple software package. Again, this representation is difficult to read because it is automatically generated.
ent from its implementation for non-probabilistic languages. The approach is to search for patterns of subexpressions in the program which can be reduced to some simpler form.

3. Simplification by abstract interpretation

When a probabilistic program is only being used to answer some query, it may be possible to simplify the program from the fully concrete domain into a more abstract domain that can still represent the query. This abstract program may allow faster posterior inference [65]. This idea is covered in 2.4.3 for the specific case of verifying a predicate query by abstracting the program into a predicate domain. In theory, it could be used to optimize programs for queries more complex than predicates but less than full parameter densities.

Inference-aware optimization

Another avenue for optimization, which is an opportunity unique to probabilistic programming, is to use static analysis to improve the inference process. In some circumstances, probabilistic programs can be transformed to be more efficient for a particular posterior inference method. Work on this approach is limited so far to improving the efficiency of sampling methods.

Example: Optimizing sampling with R2. Nori et al. [56] introduced an inference algorithm called R2 whose main innovation is overcoming a particular inefficiency in drawing samples from a probabilistic program.

An example of the situation that R2 attempts to improve is shown in Figure 2.22. The example is written in a language specified alongside R2. In this program, some binary random variables are first sampled from distributions. Observations of those samples are then specified (line 16) in the form of a predicate assertion. A naive sampling process for this program would generate the samples from the Bernoulli distributions, but they would very often not satisfy the observed condition, and so they would be rejected and work would be wasted.

Instead of this naive sampling process, R2 first transforms the program to specify the constraints on the samples at the same time they are drawn, as shown in Figure 2.23. Then, instead of
Figure 2.22: An R2 program before transformation. The sampling statements (e.g., line 6) are separated from the observation assertion (line 16), so some samples will disagree with the observations.

```plaintext
1: bool earthquake, burglary, alarm, phoneWorking, maryWakes, called;
2: earthquake = Bernoulli(0.001);
3: burglary = Bernoulli(0.01);
4: alarm = earthquake || burglary;
5: if (earthquake)
6:   phoneWorking = Bernoulli(0.6);
7: else
8:   phoneWorking = Bernoulli(0.99);
9: if (alarm && earthquake)
10:  maryWakes = Bernoulli(0.8);
11: else if (alarm)
12:  maryWakes = Bernoulli(0.6);
13: else
14:  maryWakes = Bernoulli(0.2);
15: called = maryWakes && phoneWorking;
16: observe(called);
17: return burglary;
```

Figure 2.23: The R2 program after transformation. Each sampling statement is accompanied by the necessary conditions on the sample (for example, line 6 accompanied by line 7).

```plaintext
1: bool earthquake, burglary, alarm, phoneWorking, maryWakes, called;
2: earthquake = Bernoulli(0.001);
3: burglary = Bernoulli(0.01);
4: alarm = earthquake || burglary;
5: if (earthquake) {
6:   phoneWorking = Bernoulli(0.6);
7:   observe(phoneWorking);
8: }
9: else {
10:  phoneWorking = Bernoulli(0.99);
11:  observe(phoneWorking);
12: }
13: if (alarm && earthquake){
14:  maryWakes = Bernoulli(0.8);
15:  observe(maryWakes && phoneWorking);
16: }
17: else if (alarm){
18:  maryWakes = Bernoulli(0.6);
19:  observe(maryWakes && phoneWorking);
20: }
21: }
22: else {
23:  maryWakes = Bernoulli(0.2);
24:  observe(maryWakes && phoneWorking);
25: }
26: called = maryWakes && phoneWorking;
27: return burglary;
```
sampling from independent distributions and rejecting many of the samples, R2 directly samples from distributions conditioned on the constraints being true. Since the sampling process itself is aware of the restrictions on the samples, no samples are rejected.

The program transformation is computed by cascading the logical implications of the observations backward through the program by finding the weakest precondition necessary for each successive command. This is called the preimage operator.

This technique relies heavily on restrictions in the demonstration language. The pre-image operator is not necessarily computable for general languages. It is shown only for Bernoulli distributions, but the authors suggest that the approach could be extended to other distributions by truncating their support to match conditions on the samples. This approach will probably only work with observations which reduce the support of the posterior.

Nonstandard interpretations. Non-standard interpretations are alternative ways of reading a program which give a different semantics to code that is already written. A typical strategy for non-standard interpretation is to replace an existing data type with a new data type that contains additional information. A common example of this is automatic differentiation [86].

Since non-standard interpretations do not necessarily happen at compile-time, they are not strictly static analyses. However, since in the probabilistic programming context they may happen before the inference phase, they may effectively work as static analyses, and so they are mentioned briefly here.

Non-standard interpretations have been used in the probabilistic program setting to add structure to the definition of the density function in order to enable an inference method. The most common example is automatic differentiation, which automatically provides a density gradient for inference methods such as HMC [66]. Another example is provenance analysis, which is a sort of ad-hoc dependence analysis, which can enable alternative MCMC algorithms [66]. A third example is a system called AutoConj [58], which searches the code of a density definition for conjugacy structure to make inference methods like Gibbs sampling simpler and more efficient.
Conclusions

Depending on the language, probably the most practical source of optimizations will be classical optimizations applied to the non-probabilistic programming specific portions of the pipeline, such as the density computation. Many classical optimizations can be applied without major modification.

The program slicing and partial evaluation ideas presented in this section are direct extensions of classical optimizations to probabilistic programming. Both of these ideas are likely to speed up any language they are applied to.

R2 represents interesting theoretical work on optimization that takes advantage of the actual structure of the probabilistic programming pipeline. However, it is unlikely to be practically useful until it is extended to be compatible with popular languages and inference techniques.

2.4.3 Verification

In probabilistic programming, verification queries can be defined over probabilistic quantities. In such a case, we cannot necessarily answer the query by proof of disproof, but rather by estimating (or bounding) the probability that the query holds. One strategy for estimating the query probability is simply to sample from the program and observe how often the query is satisfied. This strategy can work, but the query may require rare events to be estimated, and so it may take an unreasonable number of samples to get sufficient certainty on the bound of the estimate. Most of the current work relating to the verification of properties of probabilistic programs deals with making this estimate feasible.

1. Dividing the space of executions

One strategy is to divide up the space of possible executions by the path taken through the program, and then evaluate each path separately. This strategy will also allow us to find formal bounds on the probabilities of validation predicates, given that we use a fairly restrictive language to make the proofs easier.
Consider the probabilistic program in Figure 2.25. Our goal is to find the probabilities of the predicates on lines 11 and 12. Ideally, we could sample executions of this program infinitely many times, taking new samples of the random variables each time, and we would know the predicate probabilities. Instead, the strategy will be:

(a) Sample some executions of the program, keeping track of the paths of the executions. A path is defined by the control-flow decisions that are made throughout the execution, such as the branch taken at each if-then-else statement. Continue sampling these paths until we are confident that, with high probability, the path of a newly sampled execution will fall into the already observed set.

(b) Find a proven lower bound on the probability of a newly sampled execution having a path in the observed set of paths.

(c) For each path in the observed set, compute upper and lower bounds on the probability of each validation predicate using the information that the path implies. For example, if the path includes the then branch of an if(X) statement, then X is known about all executions that take this path, and X may change the probabilities of the validation predicates. This extra information lets us compute tighter upper- and lower-bounds on the validation predicates.

(d) We can now compute upper- and lower- bounds on the predicates over all paths by assuming the opposite extreme (certainly true or certainly false) is taken with whatever probability is not certain to be covered by the path set.

This strategy relies heavily on the restrictions of the language used to demonstrate it. These restrictions include being limited to only basic arithmetic operations on variables, only integral and floating-point types, and limiting statements to assignments, conditionals and while-loops. The conditionals and validation predicates are restricted to linear assertions over reals and integers. Bounds on the probabilities of the predicates given paths are possible to compute because of these language restrictions. This strategy would be difficult to
Figure 2.24: This is a deterministic part of the example program for the path sampling method. translate directly to more flexible languages.

The strategy also depends heavily on the use of control-flow with model parameter predicates to break up the execution space. This advantage would be lost in languages that do not support discontinuous density functions, and thus have trouble representing such predicates at all.

2. Probabilistic abstract interpretation

The idea of abstract interpretation can be extended to probabilistic programs. We can extend the original definition of abstraction consistency (shown in Figure 2.7) to distributional consistency, which says that the the probability of an abstraction of the results of the concrete program should be identical to the probability of an abstraction resulting from the abstract program.

For the verification settings, we consider abstract interpretation of probabilistic programs into predicate domains. Predicate domains are semantic domains containing only boolean variables which correspond to the truth or falsity of predicates on variables in the concrete domain. The validation query can then be described in this domain. If we can construct an abstract program in the predicate domain which is consistent with the original program, we
Figure 2.25: This shows a probabilistic program in an imperative language designed to demonstrate the path sampling method. The underlined statements define verification queries. The queries ask whether the sensitivity of the \textit{estimateLogEGFR} function to its inputs meets a threshold. Figure 2.24 shows the full definition of the \textit{estimateLogEGFR} function. The key point from this function is that its behavior depends on branching conditions of its inputs.

```c
void compareWithNoise(real logScr, real age, 
    bool isFemale, bool isAA) { 
  f1 = estimateLogEGFR(logScr, age, isFemale, isAA); 
  logScr = logScr + uniformRandom(-0.1, 0.1); 
  age = age + uniformRandomInt(-1, 1); 
  if (flip(0.01)) 
    isFemale = not(isFemale); 
  if (flip(0.01)) 
    isAA = not(isAA); 
  f2 = estimateLogEGFR(logScr, age, isFemale, isAA); 
  estimateProbability(f1 - f2 >= 0.1); 
  estimateProbability(f2 - f1 >= 0.1); 
}
```

Figure 2.26: Example before (a) and after (b) abstracting a probabilistic program into the predicate domain, considering the query \( I = 0 \). Figure taken from [65].

\begin{align*}
1 & x \leftarrow \text{discrete\_dist}(); \\
2 & y \leftarrow \text{continuous\_dist}(); \\
3 & z \leftarrow x \times \text{floor}(y);
\end{align*}

(a) A concrete probabilistic program. Probabilistic sub-program \textit{discrete\_dist} returns a discrete random variable. Sub-program \textit{continuous\_dist} returns a continuous random variable. \textit{floor} rounds down to the nearest integer.

\begin{align*}
1 & \{x = 0\} \leftarrow \text{flip}(\theta_{x=0}) \\
2 & \{0 \leq y < 1\} \leftarrow \text{flip}(\theta_{0 \leq y < 1}) \\
3 & \{z = 0\} \leftarrow \{x = 0\} \lor \{0 \leq y < 1\}
\end{align*}

(b) A probabilistic program which captures the distribution only on the predicates \( \{x = 0\}, \{0 \leq y < 1\}, \) and \( \{z = 0\} \). A \textit{flip}(\theta) expression is true with probability \( \theta \).

can perform posterior inference on the abstract program in order to perform the verification.

Figure 2.26 shows an example of a probabilistic program before and after abstracting the program. The verification predicate we are considering is \( z = 0 \), so this is included as a variable in the abstracted program. The abstract program encodes the observation that \( z \) is zero if either \( x \) is zero or \( y \) rounds down to zero. If the functions \textit{discrete\_dist} or \textit{continuous\_dist} are expensive to compute, operating on the abstracted program instead will make this inference task feasible.
Conclusions

Verification for probabilistic programming is still a young idea, and is not yet especially practical - except in its classical form, which can still be used to prove non-probabilistic statements about non-probabilistic variables or probabilistic variables across a single evaluation of the density function.

Current probabilistic programming verification techniques mostly rely on restrictive demonstration languages for feasibility. Path sampling, for example, is only effective when the program includes many conditional branches which relate to the verification query, and can only handle very simple programming constructs. Probabilistic abstract interpretation has the potential to be a more general idea, but will likely require further development before it can be applied with any reasonable generality to real world probabilistic programming languages and scale to datasets.

2.4.4 Usability

Probabilistic programming languages tend to be much more user-friendly interfaces for machine learning than other approaches, since users can specify their models directly as programs. However, there are still major improvements to be made to the interface of the programs themselves. Some probabilistic programming languages have made design choices which favor a straightforward application of their inference method of choice over ease of programming. For example, Stan programs enforce a block structure in part of organize code around its sampling inference method. Static analysis can sometimes offer the best of both worlds by providing an easier interface for users and translating it into the original programming language, retaining its ease of inference. This enabling of user-centered design is the goal of usability methods. Another potential benefit of this translation step is to optimize the workflow that is automated away.

An example of this approach is SlicStan [54]. SlicStan is a probabilistic programming language which is similar to Stan except that it relaxes some of the restrictiveness of Stan’s syntax. SlicStan programs are compiled into Stan by re-organizing statements into a Stan program.

Stan programs are written in predefined blocks, such as the parameters block for declaring
model parameters and the model block and transformed parameters block for defining the log-density function. SlicStan’s primary contribution is to not require these blocks. Instead, SlicStan allows users to mix statements that would otherwise be in different blocks, and sorts them into their optimal block when the program is compiled to Stan. Figure 2.29 shows an example translation. By relaxing Stan’s block system, SlicStan also extends Stan’s user-defined function capabilities and makes composing programs more straightforward. One potential downside is that a user might lose some safety they would have by consciously structuring their program.

During the translation, SlicStan’s goal is to place each statement into the Stan block where that statement will be executed the fewest times while still executing correctly. Figure 2.28 shows the execution time for each block. For example, since the model block will (typically) execute each time the NUTS algorithm takes a step, SlicStan only assigns statements to the model block that must be there, while any statement that can go into generated quantities, which only runs once, should be placed there.

SlicStan works by augmenting Stan’s type system for expressions with additional information (Figure 2.27). SlicStan then infers the level type (also shown in Figure 2.27) for each statement using a set of typing rules. Statements which must be in the model block, such as ~ statements, are assigned the MODEL level type. Statements which receive information from MODEL level statements must be either MODEL or GENQUANT level statements, and statements which provide information to MODEL level statements must be either DATA or MODEL. Each statement is then assigned the least compute-intensive level that is permitted by these rules. This way, each statement is placed optimally without user input.

SlicStan is the only major example of static analysis of probabilistic programs for usability, perhaps since user experience can also be improved by writing powerful libraries and documentation. However, presenting users with easier and more powerful interfaces has potential impact across all programming languages.
Types, and Typing Environment:

<table>
<thead>
<tr>
<th>( \ell ) ::=</th>
<th>level type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>data, transformed data</td>
</tr>
<tr>
<td>MODEL</td>
<td>parameters, transformed parameters</td>
</tr>
<tr>
<td>GENQUANT</td>
<td>generated quantities</td>
</tr>
</tbody>
</table>

| \( n \) | size |
| \( \tau ::= \text{real} \mid \text{int} \mid \text{bool} \mid \tau[n] \) | base type |
| \( T ::= (\tau, \ell) \) | type: base type and level |
| \( \Gamma ::= x_1 : T_1, \ldots, x_n : T_n \ x_i \) distinct | typing environment |

\begin{align*}
\text{DATA} & < \text{MODEL} < \text{GENQUANT}
\end{align*}

Figure 2.27: Enumerated here are the types of expressions in the SlicStan language. Each SlicStan expression is given a type \( T \) which is a pair of a base type \( \tau \) (taken directly from Stan) and a level type \( \ell \). The level type indicates one of three options (\( \text{DATA} \), \( \text{MODEL} \) and \( \text{GENQUANT} \)), which indicate their block placement requirements.

<table>
<thead>
<tr>
<th>Block</th>
<th>Execution</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>—</td>
<td>DATA</td>
</tr>
<tr>
<td>transformed data</td>
<td>per chain</td>
<td>DATA</td>
</tr>
<tr>
<td>parameters</td>
<td>—</td>
<td>MODEL</td>
</tr>
<tr>
<td>transformed parameters</td>
<td>per leapfrog</td>
<td>MODEL</td>
</tr>
<tr>
<td>model</td>
<td>per leapfrog</td>
<td>MODEL</td>
</tr>
<tr>
<td>generated quantities</td>
<td>per sample</td>
<td>GENQUANT</td>
</tr>
</tbody>
</table>

Figure 2.28: Each Stan block, along with the Stan phase during which it is executed and the level of SlicStan statement that will be assigned to it. “Per chain” means that the block is executed every time the NUTS algorithm is restarted on a new starting point, which typically occurs many times during posterior inference. “Per leapfrog” means that the block is executed every time the NUTS algorithm takes a step, which occurs many times for each chain. As a result, for example, it is less efficient to place a statement in the model block than in the transformed data block.
Figure 2.29: On the left is shown a SlicStan program, which allows users to forgo blocks and define more flexible functions. On the right is shown the equivalent Stan program that will result from compilation. Only the necessary statements are included in the model block of the resulting Stan program.

2.5 Future directions

2.5.1 Extensions of current work

This section enumerates some directions that would be natural extensions of the current work.

Optimization

Automatic transformation of models to use parallel computing hardware. With sufficient understanding of the information flow within a probabilistic program, it should be possible to automatically transform probabilistic programs to take advantage of specialized computing hardware during posterior inference. This could include parallelism with the CPU, and GPU or TPU hardware. Some languages are already in a good position to implement this optimization, such as Edward and Pyro which target computation graph backends (Tensorflow and Keras, respectively) that already support specialized hardware. However, this optimization is applicable for almost any language.

For example, if a static analysis method could show that the iterations of a for loop in a Stan model block were independent, the loop could be automatically parallelized. Parallelism within the computation of a density would be the easiest to achieve, but parallelism across other parts of
the inference might also be possible for particular inference algorithms.

*Passing additional structure to the inference algorithm.* Non-standard interpretation of source code adds to the semantics of a program, which can provide more information to an inference algorithm. Current examples include automatic differentiation providing gradients to HMC in Stan [86]; provenance analysis to enable alternative MCMC algorithms [66]; and automatically discovered conjugacy structure to speed up various types of inference [58].

Advances in inference techniques could be coupled with new static analysis techniques which provide additional information from the program. For example, using dependence analysis on a Stan program to build a factor graph could enable Stan’s backend to utilize the Sum-Product algorithm, which might enable Stan to automatically marginalize out discrete valued variables [49]. Dependence analysis could also be used to find the Markov blanket for each parameter, enabling efficient Gibbs sampling.

*Partial evaluation of automatic differentiation.* Automatic differentiation is typically much more convenient and robust than user-provided derivatives, but it is sometimes less efficient. Attempting to find symbolic derivatives for expressions using algebraic solvers, in a similar style to Hakaru in 2.4.2, would sometimes speed up inference at the expense of compilation time and complexity.

**Verification**

*Relaxing the restrictions of verification methods.* Many current approaches to verification rely heavily on the restrictions of their demonstration languages. There are many potential ways to relax these restrictions which could allow these approaches to be applied more broadly. For example, in the paper introducing the R2 inference technique, Nori et al. suggest that R2’s ideas could be applied to Stan - however, since R2 relies on observations in the form of predicate assertions rather than data points, R2’s approach cannot be directly applied to Stan without expanding its notion of observation.
New probabilistic abstract interpretations. Holtzen et al. demonstrated the use of predicate domains for verification (2.4.3) in a relatively simple environment. Their work could be extended by implementing more sophisticated versions of the pre-image operator to infer the weakest precondition of more types of predicates and statements. For instance, if an implementation of the pre-image operator were to include sufficient statistical knowledge, then perhaps statistical predicates could be included and reasoned about in the abstract domain.

Usability

Restrictive mode. It could be useful to build restrictive subsets of existing probabilistic programming languages, which make it harder for users to make certain mistakes. This might be especially helpful to new users who might not understand the pitfalls of advanced features. This approach contrasts SlicStan, which aims to make the compilation process less restrictive.

For example, Stan includes a construct called \texttt{reject()}, which eliminates a NUTS trajectory and effectively sets the density at that point to zero. This feature has the potential to introduce discontinuity into the density which NUTS cannot effectively estimate. A restrictive mode in Stan might disallow \texttt{reject()} entirely.

In addition to being a helpful guide for usability, a restrictive mode might also be helpful for implementing many other static analyses discussed in this paper by disallowing problematic language features. For example, if some static analysis did not work with recursive functions, a restrictive mode could disallow recursive functions.

2.5.2 Unexplored directions

Listed here are directions that are of high potential and which are not directly reflected in the current literature. They are meant to be discussion points rather than detailed plans.
**Statistically-aware user feedback**

There is potential for some system, supported by static analysis, to provide the user with feedback about the statistical model that their program represents. This goal could fall under the general category of usability, because statistical feedback may make probabilistic programming easier for users who are not absolute experts in statistics. It could also fall under verification, because the feedback may be phrased in terms of evaluating the truth of statistical properties on a program.

For example, the compiler could find the conditional independencies encoded in a program and provide a representation such as a graphical model to the user. The user could then check this model against their intent.

Other properties of a statistical model from the literature could also potentially be provided, a compiler could attempt to determine the statistical power of a model, letting a user understand if their dataset is likely to be sufficient to evaluate a query to a sufficient level of certainty.

In the extreme, statistically-aware user feedback could allow a probabilistic program compiler to take on the role of an interactive statistical assistant. An assistant may not be able to supply the main ideas of a model, but it could catch common errors and find mismatches between the program and the user’s intentions. A conversation about the model could take place as an alternation of program iterations and compiler feedback.

**Validation of programs against other abstractions**

Typical program verification involves evaluating the truth of a predicate on the program variables.

When the user writes a probabilistic program, they may have some other description of the model in mind, which is then made more concrete in the form of the program. For example, a user might be trying to encode a directed graphical model with some conditional independence structure, or they have some knowledge of the causal relationships between the variables. If the program is written correctly, this additional information will be encoded in the program.

These additional representations could be considered abstractions of the probabilistic program
akin to the abstract programs in probabilistic abstract interpretation. They are not concrete enough to apply a posterior inference method, but they can be checked for consistency against a concrete program.

A user would provide an abstraction of their probabilistic model, such as a graphical model, in a specification language alongside their probabilistic program. A compiler could then check that the probabilistic program is consistent with the provided abstraction.

**Higher-level representations via transformation**

SlicStan provides a higher level alternative interface for Stan that alleviates some of the programming complexities of writing Stan code. It could be similarly useful to provide higher-level interfaces that alleviate some of the complex statistical considerations for the user. Any piece of statistical knowledge that can be encapsulated into a higher-level interface is likely to be easier to write and more likely to be correct.

There could be domain-specific languages for a variety of fields which are specialized to easily and precisely express domain knowledge in that field. For example, a domain-specific language for computational genetics might include convenient manipulation of sequences-valued random variables, and its compiler might translate that representation into a Hidden Markov Model represented in Stan code.

Another example of this could be automatically marginalizing discrete parameters.

**Data-scalable and data-absent analyses**

In the current literature, static analyses generally assume that observations are integrated into the body of the program, and so this data is available to be part of the analysis at compile time. There are two potential issues with this assumption:

- Some analyses, especially verification methods, will not scale well when the data becomes very large.
In some probabilistic programming languages such as Stan, the dataset is not provided to the compiler along with the source code and is instead fed into the program after compilation.

Static analysis methods could be developed with special attention to these issues. For example, interval analysis is a form of abstract interpretation where numeric types are abstracted as interval ranges representing their bounds. Interval analysis would be difficult to implement on a probabilistic program that has some values missing (in the case of absent data) or values that are resource intensive to traverse (in the case of large datasets). An alternative implementation of interval analysis could operate instead with some user-provided assumptions or an efficient process for summarizing the relevant features of the dataset. Examples of these summaries might be dimensions, moments or bounds of a data variable.

**Posterior inference failure prediction and diagnosis**

One class of error discussed in section 2.1.3 is a failure of the posterior inference algorithm to converge to a reasonable approximation of the true posterior. Static analysis could do something to predict this type of error for a given program, perhaps also informed by the dataset. Any such analysis would be specific to the inference algorithm being run.

In general, implementation approaches could be:

- Searching for specific predictors of bad inference behavior. For example, if the posterior inference algorithm were HMC, the compiler could warn the user when the program is likely to have a discontinuous density or produce a non-Ergodic Markov chain.

- Searching for sufficient indicators of good inference behavior (which will certainly be overly restrictive). For example, if the program is found to match a particular class of model known to be supported by the inference engine, the compiler could be confident that the inference task for that program is viable.

- Running posterior inference on automatically generated or subsampled datasets and checking for indications of failure. This may be a static analysis in the sense that it could be done
before the real inference task, but dynamic in the sense that it runs the program with an inference engine.

2.6 Conclusions

The current state of the field of static analysis for probabilistic programming is quite limited in scope. There is a disconnect between the language properties which theoretical static analyses assume and the languages which tend to scale well and have the most users. As a result, most of the immediate gains available to probabilistic programming language compilers are the same classical analyses available to other programming languages. However, current work has shown significant promise in the direction of probabilistic programming-specific static analyses.

Static analysis methods are already applied successfully in software engineering to provide high-level interfaces, automatic proofs-of-correctness, and optimizations without mental overhead. With advances in this field, applying static analysis methods to probabilistic programming has the unique potential to do the same for probabilistic modeling.
Chapter 3: Multi-Model Probabilistic Programming

3.1 Introduction

Probabilistic programming suggests an analogy between software engineering and probabilistic modeling: developing a model is like developing a program. However, there is a key difference: while there are only ever a handful of relevant versions of a program in development, there are often a multitude of alternative probabilistic models that remain relevant throughout development, communication and validation. Probabilistic programming systems currently ignore this multiplicity.

We present an extension of probabilistic programming that lets users encode networks of models: graphs of models connected by similarity. Just as probabilistic programs let users represent and query probabilistic models, these multi-model probabilistic programs let users represent and query networks of probabilistic models.

3.1.1 Uses of networks of models

We focus on four categories of use cases for networks of probabilistic models:

1. Automation in model-space

Networks of models allow us to deploy automation in model-space. For example, algorithms can search for models or neighborhoods in the network that optimize a real-valued model scoring function. We demonstrate a greedy graph search in Section 3.9.3 and more advanced search methods are discussed in Section 3.10. Other examples of automation on networks include:

- Stacking methods seek an optimal weighted subset of the models to combine into a
high-performing ensemble or “stacked” model [1].

- *Projection prediction* methods search for the “simplest” model that reproduces the predictions of a gold standard model or ensemble [2].

2. **Understanding the problem and its solution space.**

Statisticians often use multiple related models to gain insight into their problem and solution space [3]. Examples include:

- Plotting alternative models by evaluation metrics to understand trade-offs such as model complexity vs. accuracy [4].
- Comparing diagnostic samples, such as posterior predictive samples, to understand the impact of model decisions [5].
- Applying “multiverse” methods, which seek to quantify modeling uncertainty by sampling from a whole set of plausible models [6].

3. **Tracking and communicating the branching path of development.**

Probabilistic model development is an inherently iterative and branching process: models are improved by a cycle of criticism and adjustment [3, 7]. Development follows an often-backtracking path through model space; a path that model developers often have difficulty managing [8, 9, 10].

When the relevant region of model space is explicitly represented as a network of models, the path of development can also be explicitly documented. This documentation is useful for managing development [10], for third parties interested in learning from or extending the work, and, as discussed in case 4, as context for third-party auditors assessing the quality of the final model.

Standard version control tools like Git can serve this use case to some extent, but as the case study in Section 3.9.2 shows, the network of models abstraction earns more than version control: it keeps all relevant models available and tracks their semantic relationships. Two
surveys of data scientists, one by Guo and one by Kery et al., both report that version control is not typically used for managing exploratory data analysis [8, 10].


P-hacking [11] and the Garden of Forking Paths [12] are issues in statistics with a common root cause called modeler degrees of freedom [13]: when a modeling task, such as the analysis of scientific data, includes modeling decisions with multiple justifiable solutions, the modeler can, intentionally or not, tune their solutions to cherry-pick a desirable model, such as a model with a “significant” p-value.

There are two ways in which explicit networks of models can alleviate the issues of modeler degrees of freedom. The first way is to aid in reporting the set of analyses that were done. Wigboldus and Dotsch, discussing questionable research practices, argue: “the potentially (highly) questionable part of your actions as a researcher is not that you engage in all kinds of exploratory analyses. Instead, the questionable part is not reporting truthfully and explicitly the exploratory nature of these analyses” [14]. When researchers explicitly report their path of exploration through the network of models, their degrees of freedom become transparent.

The second way is to automate sensitivity analysis. Sensitivity analyses aim to “assess whether altering any of the assumptions made leads to different final interpretations or conclusions” [15]. They are broadly recommended for scientific reports such as clinical trials [16]. In a survey of solutions to researcher degrees of freedom, Rubin finds that sensitivity analyses “provide an effective solution to the p-hacking problem”, and also the Forking Paths problem in some cases¹ [17]. Explicit networks of models let both researchers and third-party auditors automate sensitivity analysis by a simple procedure: for each conclusion drawn from a final model, check the extent to which that conclusion is also drawn by the model’s neighbors. When each neighbor differs by one modeling decision, the analysis tells us the modeling decisions on which the conclusions depend.

¹Rubin distinguishes between result-biased and result-neutral Forking Paths, and finds that sensitivity analysis is not necessarily sufficient for result-neutral cases.
Some of these use cases, such as automated model search and visualizations of model space, are achieved in practice only by those with sufficient time and expertise to implement ad-hoc methods with hand-enumerated sets of models. Other use cases, such as explicit modeler degrees of freedom and automated sensitivity analysis, are rarely ever achieved in practice despite being valuable in theory. We argue that all of these use cases could become convenient and routine if we had a standard representation of networks of models.

We note that each of the four use cases above has a natural definition and utility for drawing edges between models:

- For use case 1 (automation), edges should be between the most similar models: the network is then more analogous to a continuous and differentiable space, and methods like greedy graph search more closely approximate gradient descent.

- For use case 2 (solution space mapping), again edges should be between the most semantically similar models; edges then provide a more consistent sense of distance and orientation.

- For use case 3 (tracking and communicating development), edges should bridge sequential versions of a model. Authors and auditors then can cleanly trace model development, with each of its decision points and model transformations, as a tree within the network.

- For use case 4 (explicit degrees of freedom), edges should be between model pairs that are one “decision” apart; then each edge is like a step in the Garden of Forking Paths.

Ideally, a standard construction of model networks should be compatible with all of these use cases.

3.1.2 Representing networks of models

To support all of the above use cases, representations of networks of models should support the following operations efficiently:

1. **Explicitly generate the network of models (if practical).**
2. **Given a node, generate its set of neighbors.** This operation becomes necessary when the network is too large to practically generate in its entirety; for instance, when we are searching through a large model space.

3. **Given a node, generate its corresponding probabilistic program.**

In addition, we would like our representation to be:

1. **Easy to read and understand.** Especially for use cases 3 and 4, clarity is the priority.

2. **Simple to write.** Probabilistic modelers may not be expert programmers.

3. **An efficient encoding.** If a representation is too redundant, large model spaces become too cumbersome to work with.

4. **Standardized.** A standard format enables a stable network-of-models API to easily build general model-space tools.

One obvious choice for representing multiple models is a directory of relevant probabilistic programs. According to surveys done by Guo and Kery et al., this is the typical approach among data scientists [8, 10].

While appealing for their simplicity, file collections are a poor solution because they become uninterruptible, unmanageable, and memory intensive as the number of models grows large, and because they discard the semantic relationships between models. Kery et al. also report that data scientists have issues with file naming, keeping track of the relationships between files, and maintaining a mental map of their code [10].

Another possible choice of representation is a program in a general-purpose language that generates the set of probabilistic programs. This approach is flexible and scalable, but its flexibility also makes it prohibitively difficult to write, understand, and standardize.

Our proposed representation is a middle ground: a meta-programming feature to augment existing probabilistic programming languages, so that meta-programs, which we call multi-model
programs, represent networks of probabilistic models. We argue that our meta-programming approach is nearly as flexible as general program generation and easier to write and understand than the directory-of-programs approach for nontrivial examples. We call our meta-programming feature *swappable modules*.

### 3.1.3 Swappable modules in Stan

To demonstrate swappable modules, we use Stan as our example host language, mainly because Stan is popular, performant, and has a clear, established semantics [18, 54]. Stan is also a highly structured and restrictive language, so by adding swappable modules to Stan, we are demonstrating an unusually difficult case that can easily be transferred to other languages.

We refer to the Stan language augmented with swappable modules as *Modular Stan*. We have built a prototype compiler (source code is available online\(^2\)) and an interactive visualization website (which can be visited online\(^3\)) for Modular Stan.

Figure 3.1 shows an abstracted Modular Stan program and two data structures derived from it.

A Modular Stan program is made of two parts. The first part is the *base*. The base is like a Stan program, except that its code can contain *holes*, which are syntactically similar to function calls but are more flexible. The second part is a list of *module implementations*. Each module implementation describes a new way to “fill” a particular hole. More than one module implementation can be specified to “fill” the same hole. Module implementations can themselves contain holes.

We can visualize the holes and implementations, with their implementation-fills-hole and code-contains-hole relationships, as a rooted directed acyclic graph of base, hole and module implementation nodes, where the root represents the program’s base, each base and module implementation node points to the holes it contains, and each hole node points to the module implementations that fill it. We refer to this graph as the *module graph* of a program. The module graph is not necessarily a tree because multiple pieces of code can contain the same hole.

To build a valid Stan program out of a Modular Stan program, we need to *select* one mod-

\(^2\)https://github.com/rybern/mstan  
\(^3\)http://ryanbe.me/modular-stan.html
Figure 3.1: The correspondence between a Modular Stan program and a network of models. Holes are represented as empty shapes. Modules are represented as shapes filled with solid colors, having the same shape as the hole that they fill. (a) and (b) represent a Modular Stan program in its two parts, the base (a) and the module implementations (b). (c) represents the relationships between the base, holes and modules implementations. (d) represents the model graph, where each node is a model corresponding to a valid selection of modules, and each edge is labeled by the shape of the hole by which its endpoints differ, colored to show the change of modules. (e) shows the Stan program corresponding to a node; it is synthesized from the selected modules of the node.

A Module implementation to fill each hole in its base, and one implementation to fill each hole in those implementations, and so on, until there are no holes. A minimal set of module implementations that leaves no empty holes is called a valid selection. Given a Modular Stan program and a valid selection, we can produce a valid Stan program by filling each hole with the given module implementation.

A Modular Stan program can therefore represent many Stan programs: one for each valid selection. If we consider each of these Stan programs to be a node, we find a natural network structure: draw an edge between two Stan program nodes when their selections differ at only one hole. Figure 3.1 shows such a graph with correspondence of each single node to a Stan program and thereby to a probabilistic model. We argue that this graph construction is a powerful way to represent networks of models.

Holes and module implementations are like function calls and function definitions, but with important differences:
• Selecting a module implementation can add model parameters and have other changes that are not local to their call site. Stan requires the set of parameters to be fixed at compile time; therefore modules are selected before Stan’s compile time.

• More than one module implementation can be specified to fill a hole, like function overloading but with identical type signatures.

• Because module implementation code is essentially inlined statically, holes can appear in places where function calls cannot.

By defining a module system for Stan, we are also incidentally letting users write their normal Stan programs in a more modular way. Modularity of this level is a feature that is both important for probabilistic modeling workflow [3] and conspicuously absent from probabilistic programming languages like Stan [19].

Probabilistic programming languages augmented with swappable modules meet all of our earlier criteria to represent networks of models:

• They are easy to read and write, because only one abstraction (the module) and a small amount of syntax is introduced on top of the host language, and the module abstraction already fits naturally into the probabilistic modeling workflow [3].

• They are scalable, because the combinatory nature of implementation selection can define many models with little code. The language extensions in Section 3.7 further increase that expressiveness.

• They are standardized, because compilers enforce well-defined languages.

• The required network-building, neighbor-finding and model-selection operations can be implemented efficiently.

In addition, the definition of edges as between models that differ by one choice of module implementation is compatible with all four use cases’ definitions of edges.
3.1.4 Key ideas

The following are the key points of Chapter 3:

- A standardized representation of networks of probabilistic models would benefit a variety of applications (Section 3.1.1).

- We can outline a set of desiderata and operational requirements of such an implementation (Section 3.1.2).

- Swappable modules is a meta-programming system for probabilistic programming languages that meets those desiderata and operational requirements. We demonstrate swappable modules by adding it to the Stan language, resulting in the Modular Stan language (Section 3.1.3).

- The semantics of Modular Stan programs correspond to networks of probabilistic models, and we can perform useful network-level operations with respect to that semantics (Section 3.5.6).

- The Concretize algorithm (Section 3.6.1) produces a Stan program from a node of the model graph. By proving the Concretize correct (Theorem 3.6.3), we show that a Modular Stan program indeed corresponds to a network of probabilistic models.

- The ModelGraph algorithm (Section 3.6.2) explicitly builds the graph topology represented by a Modular Stan program. Our implementation is correct in that it always produces networks that correspond to the program’s semantics (Theorems 3.6.4 and 3.6.5), and is efficient in that its time and space complexity scale linearly with the size of the output (Section 3.6.2), defeating naive solutions.

- The ModelNeighbors algorithm (Section 3.6.3) enumerates the neighbors of a model with computational complexity that is linear in the number of neighbors, not the total size of the network (Section 3.6.3), enabling practical use of very large model graphs.
• The Modular Stan language can be extended to be more expressive and user-friendly without adding any complexity to the `Concretize`, `ModelGraph` or `ModelNeighbors` algorithms (Section 3.7).

• “Macros” added on top of Modular Stan let it scale to combinatorially large spaces of model (Section 3.8).

• The graph interpretations of Modular Stan programs make useful visualizations of model spaces, as shown by our prototype web interface (Section 3.9.1).

• Case studies demonstrate some key motivations of Modular Stan (Section 3.9), including scientific communication, automatic model search, and model sensitivity analysis.

• There are many more potential applications of Modular Stan, including search, robustness, and visualization (Section 3.10).

Some sections are technical, dense, and only strictly necessary for understanding the formal semantics, algorithms and proofs. For a less technical reading, I suggest the following sections:

• Section 3.1

• First part of Section 3.3

• Skim Section 3.4 or skip to the case study in Section 3.9.2 for an example of Modular Stan syntax

• Skim Section 3.5, especially Section 3.5.5 and Section 3.5.6

• First parts of Section 3.7 subsections

• Section 3.8

• Section 3.9

• Section 3.10
For a technical reading that skips the (somewhat dense) correctness proofs in Section 3.6, the reader can safely skim the details of syntax, Stan program validity and Modular Stan program constraints.

### 3.2 Related work

This chapter is based on my previously published work[96].

The “network of models” is an established concept in statistics and machine learning, but explicit and general network of models abstractions are not supported within probabilistic programming languages or other statistical software. For instance, multi-model methods like ensemble methods and model search are common in machine learning, but are typically ad-hoc in that they do not start with a declarative network topology for their set of models. The closest method that we are aware of is perhaps grid search, which searches a pre-defined grid of model hyperparameters.

There are existing probabilistic programming systems which allow users to define their programs by combining probabilistic subcomponents. For example, Prophet [20] allows users to combine subcomponents into time-series models. This effect could also be achieved in embedded probabilistic programming languages whose host languages have sufficiently expressive module systems. We are not aware of any other module systems specialized to encapsulate flexible components of structured probabilistic programs.

Kery et al. developed a tool for code editors, Variolite, to support exploratory data science by tracking alternative snippets of code in version control. Variolite lets developers write, visualize and manage iterative and branching versions of data science pipelines in a similar way to our proposal. Variolite addresses what we call use case 3 (development tracking). Variolite differs from our system primarily in that it is a tool for code editors rather than a metaprogramming feature, it does not produce a network of models, and because it is language agnostic, it only supports swapping out regions of code rather than more general semantic units.

There are existing systems that allow users to specify a program’s components at compile-time. Backpack [21] is a build-system tool for the Haskell ecosystem that lets users swap out external
software libraries that implement a common interface. Much of the work done by Yang to introduce Backpack as a mixin linker also applies to our swappable module system. In practice, the C preprocessor is often used for this purpose; it can include or exclude sections of code depending on user flags or system environment properties. These systems are not commonly applied to probabilistic programs or to study networks of programs.

We discuss some other “multiverse” tools in Section 3.9.4.

3.2.1 Comparison to ML-like module systems

Our swappable module system bears some resemblance to ML-like module systems [22, 23]. We find OCaml to be a helpful comparison point [24]. The hole and module approach can be understood in the language of ML-like modules:

Each hole in a program, which is to say each unique hole identifier referenced in a program, can be thought of as declaring a module signature and a module-valued variable of that signature. Each statement and expression referencing a hole is like a reference to a field of that hole's corresponding variable. A hole's variable may take the value of any module that “implements” it. A hole's signature is inferred from its usage and implementations. The value assigned to a hole’s variable may either be specified outside of the program or left non-deterministic. Blocks of code that contain holes can then be thought of as ML functors in that their holes are like implicit module-valued arguments.

However, unlike ML-like modules:

1. Modules are not applied at any point in the program. Rather, the program represents the set of programs generated by any combination of module applications, as though module applications were non-deterministic; then the user can determine the modules to apply in an optional mode of compilation called concretization that we discuss in Section 3.6.1.

2. When a module is assigned to a hole, or selected, there can be global effects on the semantics of the program, because the module may add code to Stan’s top-level blocks. For instance, a module may introduce a new model parameter, which changes the domain
over which the program defines a joint distribution. This effect is especially noteworthy in languages like Stan in which model parameters are fixed at compile time, because it implies that module application must happen before compile time.

We are not aware of prior examples of inferred implicit module signatures, non-deterministic functor application semantics, or module application with non-local effects on the resulting program.

### 3.3 Background: Stan

Stan is our example host language, so we give a brief overview of Stan programs and of Stan’s syntax in this section.

Like all probabilistic programs, Stan programs represent probabilistic models. Stan programs are C-like, imperative, and are written as a sequence of top-level blocks. Here is a simple example of a Stan program:

```stan
data {
    int N;
    vector[N] x;
}
parameters {
    real mu;
    real sigma;
}
model {
    mu ~ normal(0, 1);
    sigma ~ lognormal(0, 1);
    x ~ normal(mu, sigma);
}
```
We see three of Stan’s blocks: data, parameters, and model. The data block declares the observed variables and the parameters block declares the unobserved variables. The model block defines the log-density of the joint distribution of the observed and unobserved variables. Each \( \sim \) statement implicitly increments a variable target that represents the value of the overall log-density function; for instance, \( \mu \sim \text{normal}(0, 1) \) could be rewritten as
\[
\text{target} += \text{normal_lpdf}(\mu, 0, 1);
\]

The above program represents the probabilistic model:

\[
\begin{align*}
\mu & \sim \text{Normal}(0, 1) \\
\sigma & \sim \text{LogNormal}(0, 1) \\
x & \sim \text{Normal}(\mu, \sigma)
\end{align*}
\]

where \( x \) is an observed variable. When the program is compiled and executed given data for \( x \), it should produce draws from the posterior distribution \( P(\mu, \sigma|x) \).

For our purposes, it is sufficiently precise to take the semantics of a Stan program \( p, [p] \), to be a joint distribution \( P(d, \theta) \) of the variables declared in the data block, \( d \), and parameters block, \( \theta \), of \( p \). That semantics can then be used to infer the posterior distribution \( P(\theta \mid d) \).

3.3.1 Syntax

In this section we give an incomplete and semi-formal syntax for the Stan language. This is included primarily to demonstrate how the syntax is extended by Modular Stan.

Stan programs are organized into blocks of statements that describe different aspects of a probabilistic model. Each is an ordered subset of blocks:

```
STAN_PROG: FUNCTIONS?
    DATA?
    TRANSFORMED_DATA?
    PARAMETERS?
```
This is a “rule” defining `STAN_PROG`, the top-level grammar of a Stan program, as a sequence of other rules, one for each block. Here `?` indicates that the block may or may not be present.

Below is the syntax of the `data` block:

```plaintext
DATA: data { STMT_DECL;* }  
STMT_DECL: TYPE identifier  
TYPE: int | real | vector | matrix | ...  
```

Here, `identifier` stands in for valid Stan variables names. The `*` symbols are Kleene stars to indicate an element that can be repeated or absent. This says that the `data` block contains a sequence of variable declarations.

The `parameters` block is similar to the `data` block:

```plaintext
PARAMETERS: parameters { STMT_DECL;* }  
```

The `model` block is more flexible, because it contains code to calculate the program’s log-density function:

```plaintext
MODEL: model { STMT_LPDF;* }  
STMT_LPDF: STMT_BASIC  
    | identifier ~ identifier( EXPR,* );  
    | target+= EXPR;  
STMT_BASIC: STMT_DECL | STMT_ASSIGNMENT | STMT_FOR  
    | STMT_IFELSE | STMT_FUNCTION_APPLICATION | ...  
```

---

4Strictly speaking, we should not allow discrete types like `int` to be declared as `parameters`, because Stan’s inference backends currently do not support sampling from discrete variables.
Here, `target` is a reserved variable in Stan that represents the accumulated log-value of the density function. The ~ and `target+=` statements serve to increment the value of the log-probability density function; the sum of these increments defines the joint probability distribution of the model.

`STMT_BASIC` and `EXPR` closely resemble C-like languages, so we omit their details here.

Stan allows user-defined function declared in the `functions` block:

```
FUNCTIONS: functions {FUNC_DECL*}
FUNC_DECL: RET_TYPE identifier ((TYPE identifier),*){STMT_FUNC;*}
STMT_FUNC: STMT_BASIC | return EXPR
RET_TYPE: TYPE | void
```

There are three more blocks, each containing statements with different purposes:

```
TRANSFORMED_DATA: transformed data { STMT_BASIC;* }
TRANSFORMED_PARAMETERS: transformed parameters { STMT_LPDF;* }
GENERATED_QUANTITIES: generated quantities { STMT_BASIC;* }
```

The `transformed data` and `transformed parameters` blocks let users define transformed versions of the observed and hidden variables in a way that works efficiently with the inference process. The `generated quantities` block lets users define output quantities calculated from the samples of the parameters.

### 3.3.2 Effects and Scope

The Stan language has some unusual rules around side effecting code and code scopes that are different from block to block. We define those rules here, and we avoid violating them later when we generate Stan programs.

Let `Block` be the set of Stan block types, `{data, parameters, model,...}`.

Stan statements and expressions are sometimes allowed to be impure in particular ways depending which block contains that code. We call those impurities `effects`. We say that when code uses the random number generator, it has the `RNG` effect, and when it increments the program’s
density function with $\sim$ or `target+=` statements, it has the LPDF effect. The set of effects \( \text{Eff} \) is then \{\text{RNG, LPDF}\}. Stan’s specification implicitly defines a mapping \( \text{effects} \) from some \( \text{block} \in \text{Block} \) to the set of effects allowed within that block, \( \text{effects}(\text{block}) \subseteq \text{Eff} \).

In Stan programs, the declarations that a statement may reference depends on the statement’s block and the declaration’s block. For instance, code in a transformed data block can reference top-level declarations in a data block but not in a parameters block. To know whether it is valid to insert new statements into a given block, therefore, we need to know which blocks’ declarations that statement is allowed to refer to. Stan’s specification implicitly defines a mapping \( \text{scope} \) from some \( \text{block} \in \text{Block} \) to the set of blocks whose top-level declared variables statements that block may reference, \( \text{scope}(\text{block}) \subseteq \text{Block} \).

### 3.3.3 Program validity

We define \( \text{valid}_\text{Stan} \) to denote whether a program is valid, or roughly whether we expect it to compile. Let \( \text{valid}_\text{Stan}(\text{SP}) \) for a Stan program \( \text{SP} \) if and only if all of the following are true:

1. Effects and scopes of code are available in their block.

2. \( \text{SP} \) typechecks as Stan code. Stan has a similar type rules to a subset of C.

### 3.4 Modular Stan syntax

Below is an example Modular Stan program:

```stan
data {
  int N;
  vector[N] x;
}

model {
  x ~ normal(Mean(), Stddev());
}
```
module "standard" Mean() {
    return 0;
}

module "standard" Stddev() {
    return 1;
}

module "normal" Mean() {
    parameters {
        real mu;
    }
    mu ~ normal(0, 1);
    return mu;
}

module "lognormal" Stddev() {
    parameters {
        real<lower=0> sigma;
    }
    sigma ~ lognormal(0, StddevInformative());
    return sigma;
}

module "yes" StddevInformative() {
    return 1;
In the above program, the base is made up of the data and model blocks, Mean, Stddev and StddevInformative are holes, and each block starting with the keyword module are the module implementations.

The above Modular Stan program has similar structure to the example in Figure 3.1. It also includes the example program in Section 3.3 as one of its nodes (where Mean is filled by normal, Stddev is filled by lognormal, and StddevInformative is filled by yes).

Modular Stan makes two additions to Stan’s syntax: holes and module implementations. Below, we extend the Stan syntax introduced in Section 3.3.1 with these additions.

Holes are statements or expressions that are syntactically similar to function applications. We define variants of the Stan syntax rules that are allowed to include holes. For each Stan grammar rule RULE that can directly or indirectly contain a STMT_BASIC, STMT_LPDF, or EXPR, we define a new rule RULE_M that replaces those rules with the following STMT_BASIC_M, STMT_LPDF_M, and EXPR_M rules, respectively. We can thereby generate new rules up to STAN_PROG_M.

STMT_BASIC_M: hole_identifier(EXPR_M,*);

STMT_LPDF_M: identifier ~ hole_identifier(EXPR_M,*);

STAN_PROG_M:
Here we use `hole_identifier` to stand in for valid hole names.

Module implementations are reminiscent of function definitions, and appear at the top level alongside blocks:

```stan
MODULE_IMPLEMENTATION_M:
  module "impl_identifier" hole_identifier((TYPE identifier,)*) {
    PARAMETERS?
    STMT_LPFD_M;*
    return EXPR_M;?
  }
```

The ? symbol indicates that the return statement is optional. We use `impl_identifier` to stand in for valid implementation names, while `module` is a new keyword.

A Modular Stan program is then:

```stan
MODULAR_STAN_PROG: STAN_PROG_M
  MODULE_IMPLEMENTATION_M*
```

We make two small additions to the syntax and capabilities of Modular Stan in Section 3.7, but this base syntax is sufficient to introduce its semantics and algorithms.

### 3.5 Modular Stan semantics

In this section we formalize the semantics of Modular Stan programs and the operations that they support.

We start by building a mathematical vocabulary of operations to work with the elements of modular programs. For example, we define a function `holes(P)` from a program `P` to the set of holes contained within `P`, and `sites(P)` to the set of textual locations in `P` where the holes appear.
We then use this vocabulary to give constraints that a “valid” modular program must conform to. These constraints focus the semantics of programs, and we also use them to prove the correctness of our algorithms in later sections.

Next, we formalize the notion of a “selection”, which represents a set of implementations that could be chosen by a user to realize a concrete Stan program. We differentiate between a “valid” selection, which corresponds exactly one-to-one with a valid Stan program, and a “partial” selection, which does not fully specify a Stan program and therefore corresponds to a set (or sub-network) of Stan programs.

Lastly, we formalize the desired network operations introduced in Section 3.1, and we give a formal semantics of modular programs as networks of probabilistic models.

3.5.1 Basic operations on programs

We assume that there is some parsing procedure $\text{Parse}$ such that, for all strings $F$ of the language defined by the syntax in Section 3.4, $P = \text{Parse}(F)$, where $P$ is some reasonable representation of $F$ that we refer to loosely as a “Modular Stan program”.

Since the actual representation of $P$ returned by $\text{Parse}$ is an implementation detail, we describe $P$ mathematically. $P$ is effectively a pair $P = (P_{\text{base}}, \text{impls}(P))$, where $P_{\text{base}}$ represents the Stan-like base of $P$ and $\text{impls}(P)$ is the set of all module implementations defined in $P$. $P$ also implicitly includes the set of all holes referenced by $P_{\text{base}}$ and $\text{impls}(P)$.

We likewise are not concerned with the representation details of implementations or holes, we only need to define operations on them.

Below are the basic operations we use to interact with programs $P$, implementations $i$, holes $h$, sets $I$ of implementations, and sets $H$ of holes:

$\text{impls}(h)$ is the set of implementations that implement a hole $h$.

$\text{impls}(P)$ is the set of all implementations defined in $P$.

$\text{impls}(H) = \bigcup_{h \in H} \text{impls}(h)$. 

79
holes($i$) is the set of holes referenced in the definition of an implementation $i$.

holes($P_{base}$) is the set of holes referenced in the base of $P$.

$$\text{holes}(I) = \bigcup_{i \in I} \text{holes}(i).$$

$$\text{holes}(P) = \text{holes}(P_{base}) \cup \text{holes}(\text{impls}(P)).$$

par($i$) is the hole that the implementation $i$ implements, also called the parent of $i$.

$$\text{pars}(I) = \bigcup_{i \in I} \text{pars}(i).$$

$$\text{pars}(P) = \text{pars}(\text{impls}(P)).$$

The above operations are specific to the context of a program $P$. Since the intended $P$ is usually clear, we only give the operation a subscript when disambiguation is necessary.

We note that $\text{par}$ and $\text{impls}$ operations are like inverses, so:

$$i \in \text{impls}(h) \iff h \in \text{pars}(i)$$

$$h \in \text{pars}(I) \iff \exists i \in I \text{ s.t. } i \in \text{impls}(h)$$

$$i \in \text{impls}(H) \iff \exists h \in H \text{ s.t. } h = \text{par}(i)$$

It is also be useful to note that, for all $I_1 \subseteq I_2$:

$$\text{holes}(I_1) \subseteq \text{holes}(I_2)$$

$$\text{impls}(I_1) \subseteq \text{impls}(I_2)$$

$$\text{pars}(I_1) \subseteq \text{pars}(I_2)$$

We also need to query certain syntactic elements of the code. Some of these operations are not fully detailed in the interest of brevity.
We refer to locations or sections of the original code as sites. We define a HoleSite as some data structure that captures the syntactic information of a hole called within code.

These are the operations on HoleSites $hs$, programs $P$, implementations $i$, and blocks $b$:

- $\text{sites}(i)$ or $\text{sites}(P_{\text{base}})$ is the set of hole sites in the code of $i$ or $P_{\text{base}}$, so that $|\text{sites}(c)| \geq |\text{holes}(c)|$.
- $\text{sites}(P)$ is the set of hole sites in all of the code of $P$.
- $\text{site}(b)$ is the site of the start of the code of block $b$.
- $\text{block}(hs)$ is the block that contains $hs$, if any.
- $\text{hole}(hs)$ is the hole that is called at $hs$.
- $\text{scope}(i)$ is the set of blocks whose top-level declarations $i$ references.
- $\text{effects}(i) \subseteq \text{Eff}$ is the set of effects whose top-level declarations $i$ references.

We can syntactically break down implementations $i$ into a triple, $(i_{\text{body}}, i_{\text{return}}, i_{\text{parameters}})$, so that:

- $i_{\text{body}}$ is the sequence of statements that makes up the code of the implementation,
- $i_{\text{return}}$ is the expression returned, if any,
- $i_{\text{parameters}}$ is the sequence of parameter declarations found in the implementation.

### 3.5.2 Structural constraints

Not all Modular Stan programs that can be parsed are valid; we will also impose certain constraints on the program’s structure and semantics. Input programs that do not meet the constraints are rejected by the compiler. These constraints keep programs straightforward and let us simplify and optimize network operation algorithms in Section 3.6.

Below are the structural constraints on a modular program $P$: 
1. The dependency graph of modules is acyclic:

   For any graph $G$, let $N(G)$ be the set of nodes and $E(G)$ be the set of edges. Let the
   \textit{module dependency graph} $MDG(P)$ be the directed graph with nodes $N(MDG(P)) = \text{impls}(P) \cup \{P_{\text{base}}\}$ and edges:

   $$E(MDG(P)) = \{ i_1 \rightarrow i_2 \mid i_1 \in \text{impls}(P) \cup \{P_{\text{base}}\}, i_2 \in \text{impls}(P), \text{par}(i_2) \in \text{holes}(i_1) \}$$

   We require that this graph is acyclic. We also refer to this property as $Acyclic(P)$.  

2. Every hole has an implementation. $\forall h \in \text{holes}(P), \text{impls}(h) \neq \emptyset$. 

3. Hole identifiers are unique and hole identifier plus implementation identifier pairs are unique.

   When a program $P$ meets these constraints, we say $valid_{\text{structure}}(P)$. 

3.5.3 Module signatures and semantic constraints

   We define the constraints on the semantics of a program in terms of \textit{module signatures}, which 
   are inferred information about the holes in a program. Module signatures are like embellished 
   function types. 

   We first give the definition and inference procedure for module signatures, then we use them 
   to give constraints for modular programs.

\textbf{Module Signatures}

To guarantee that any concrete Stan program generated from a Modular Stan program $P$ is 
valid, we need to understand the type, scope, and effect implications of filling holes with their 
implementations.

To that end, we attempt to infer a signature for every hole in $P$. A signature is like a function 
type plus extra information. When no signature can be inferred for a hole, we reject the program
as invalid. Signatures let us specify semantic constraints on input programs, and are also useful for generating Stan programs in Section 3.6.1.

We define a signature \( s \) to be a tuple:

\[
s = (s_{\text{arg-types}}, s_{\text{ret-type}}, s_{\text{effects}}, s_{\text{scope}})
\]

- \( s_{\text{arg-types}} \) specifies the argument types of a hole. \( s_{\text{arg-types}} \) is a sequence of Stan types, like the \textsc{type} syntactic element introduced in Section 3.3.1.

- \( s_{\text{ret-type}} \) specifies the return type of a hole. \( s_{\text{ret-type}} \) is a Stan return type, like the \textsc{ret-type} syntactic element introduced in Section 3.3.1.

- \( s_{\text{effects}} \) refers to the set of effects that a hole’s implementation may have.

- \( s_{\text{scope}} \) refers to the scope of non-local variables that a hole’s implementation may reference.

**Module Signature Inference**

Here we define a procedure for either inferring the signature of a hole given the type determinations at each of the hole’s call sites and all of the hole’s implementations, or rejecting the program as invalid.

We rely on an operation \textsc{returnType} that infers the type of the expression returned by implementation code, if any, given that the types of all other expressions in that code are available. This form of type inference is a standard operation for compilers of typed languages, including the Stan compiler [25]. We assume the following API:

\[
\text{ReturnType}(\text{arguments}, i)
\]

where \textit{arguments} is the collection of variable names and types available within the code of the implementation \( i \).

We visit each hole in topological order by dependency, such that when we visit a hole \( h \), for all \( i \in \text{impls}(h) \), all of the holes \( h' \in \text{holes}(i) \) have already been visited. This ordering is always
possible because of the \( \text{Acyclic}(P) \) property.

\[
\text{signature}(h) = (\text{argtypes}(\text{impls}(h)_0),
\]

\[
\text{ReturnType}(\text{argtypes}(\text{impls}(h)_0), \text{impls}(h)_0),
\]

\[
\bigcup_{i \in \text{impls}(h)} \text{effects}(i_{\text{body}}) \cup \bigcup_{h' \in \text{holes}(i)} \text{signature}(h')_{\text{effects}},
\]

\[
\bigcup_{i \in \text{impls}(h)} \text{scope}(i_{\text{body}}) \cup \bigcup_{h' \in \text{holes}(i)} \text{signature}(h')_{\text{scope}}
\]

\( \text{impls}(h)_0 \) is an arbitrary element of \( \text{impls}(h) \); \( \text{impls}(h) \) are never empty by structural constraint 2.

This way, \( s_{\text{arg-types}} \) is assigned to the argument types of any of its implementations, \( s_{\text{ret-type}} \) is assigned to the type that can be inferred from any of its implementations with the return types of dependent holes, and \( s_{\text{effects}} \) and \( s_{\text{scope}} \) are the unions of the effects and scopes required by any of the hole’s implementations or descendants.

**Semantic constraints**

We give a set of semantic constraints on programs in terms of the inferred signatures. Together, these constraints ensure that any valid combination of modules can be combined into a valid Stan program, as shown in Theorem 3.6.3.

1. Implementations match signature arg-types.

\[
\forall i \in \text{impls}(P), i_{\text{arg-types}} = \text{signature}(\text{par}(i))_{\text{arg-types}}
\]

2. Implementations match signature rettype.

\[
\forall i \in \text{impls}(P), i_{\text{ret-type}} = \text{signature}(\text{par}(i))_{\text{ret-type}}
\]

3. Effects and scopes of holes are available in their block or module signature. \( \forall i \in \text{impls}(P), \forall st \in \text{sites}(i): \)
(a) \( \text{signature}(\text{st\_hole})_{\text{effects}} \subseteq \text{signature}(\text{par}(i))_{\text{effects}} \)

(b) \( \text{signature}(\text{st\_hole})_{\text{scope}} \subseteq \text{signature}(\text{par}(i))_{\text{scope}} \)

\[ \forall \text{st} \in \text{sites}(\text{P\text{\_base}}) : \]

(a) \( \text{signature}(\text{st\_hole})_{\text{effects}} \subseteq \text{effects}(\text{st\_block}) \)

(b) \( \text{signature}(\text{st\_hole})_{\text{scope}} \subseteq \text{scope}(\text{st\_block}) \)

4. Effects and scopes of code are available in their block or module signature.

5. \( \text{P\text{\_base}} \) would typecheck under Stan if all holes \( h \) were function calls to function signatures with \( \text{signature}(h)_{\text{arg\text{-}types}} \) parameter types and \( \text{signature}(h)_{\text{ret\text{-}type}} \) return type.

6. The blockless statements of each implementation \( i \) would typecheck under Stan if it were the body of a function with \( \text{signature}(\text{par}(i))_{\text{arg\text{-}types}} \) parameter types and \( \text{signature}(\text{par}(i))_{\text{ret\text{-}type}} \) return type, and if the module-defined parameters \( i_{\text{parameters}} \) were included as model parameters.

When a program \( P \) meets these constraints, we say \( \text{valid}\text{\_semantics}(P) \).

Constraint 6 defines the scope available to code within modules: local variables, module arguments, module-defined global variables (such as parameters), and base-defined global variables (such as parameters).

3.5.4 Program validity

We say that a Modular Stan program \( P \) is valid if \( P \) meets both the structural constraints in Section 3.5.2 and the semantic constraints in Section 3.5.3: \( \text{valid}(P) = \text{valid}\text{\_structure}(P) \land \text{valid}\text{\_semantics}(P) \).

3.5.5 Selections

Selections are subsets of the implementations in a modular program. We call them “selections” because subsets of implementations are “selected” as components to build a concrete Stan model.
When a selection specifies a Stan program and has no extra implementations, the selection is valid.

Next we give formal criteria to recognize valid selections, $valid_p(I)$. We show in Section 3.6.1 that selections satisfying $valid_p(I)$ are exactly those needed to define concrete Stan programs. We first need to define siblings, an intersection-like operation on selections. $siblings(I_1, I_2)$ is the set of pairs of implementations across two sets that are different but share a parent:

$$siblings(I_1, I_2) = \{ (i_1, i_2) \mid i_1 \in I_1, i_2 \in I_2, i_1 \neq i_2, \text{par}(i_1) = \text{par}(i_2) \}$$

We note a useful property:

$$I_1 \subseteq I_2 \implies \forall I', siblings(I_1, I') \subseteq siblings(I_2, I')$$

The property $valid_p(I)$ is true for a selection $I$ if and only if all of the following three criteria are met:

1. The selection only includes implementations that are in the program:

$$I \subseteq impls(P)$$

2. Every hole in the program base and the selection has an implementation in the selection, and no extra implementations are included:

$$impls(I) = \text{holes}(P_{base}) \cup \text{holes}(I)$$

3. The selection does not include any pair of implementations that implement the same hole, as these would be contradictory definitions:

$$siblings(I, I) = \emptyset$$
The following are convenient restatements of the above properties 2 and 3:

1. Each hole found in the program base and the selection has exactly one implementation in the selection:

\[ \forall h \in \text{holes}(P_{\text{base}}) \cup \text{holes}(I), \ |\{ i \mid i \in I, \text{par}(i) = h \}| = 1 \]

2. \( \forall h \in \text{pars}(I), h \in \text{holes}(I) \) or \( h \in \text{holes}(P_{\text{base}}) \).

3.5.6 Semantics of modular programs and network operations

We are now equipped to formally define our high-level operations 1 to 3 and the semantic domain.

\textit{Concretize}(P, I) \text{ is the concrete Stan program that results from including each implementation from the set valid selection } I \text{ into the base of the valid modular program } P. \text{ We consider an implementation of Concretize correct if for all modular programs } P \text{ and selections } I \text{ such that } \text{valid}(P) \text{ and } \text{valid}_P(I):

1. \( \text{valid}_{\text{STAN}}(\text{Concretize}(P, I)) \)

2. \( \text{Concretize}(P, I) \) includes the same set of Stan statements in \( P_{\text{base}} \) and \( I \).

\textit{ModelGraph}(P) \text{ is the graph of all valid selections of } P, \text{ connected if they disagree on implementation of one hole.

\[ N(\text{ModelGraph}(P)) = \{ I \mid \text{valid}_P(I) \} \]

\[ E(\text{ModelGraph}(P)) = \{ (I_1, I_2) \mid I_1, I_2 \in N(\text{ModelGraph}(P)), |\text{siblings}(I_1, I_2)| = 1 \} \]

\textit{ModelNeighbors}(P, I) \text{ is the set of all selections that share an edge with } I \text{ in the model graph: ModelNeighbors}(P, I) = \{ I' \mid (I, I') \in E(\text{ModelGraph}(P)) \} \]
We can give a formal semantics of a valid modular program $P$ in these terms: $[P]$ has the same graph structure as $\text{ModelGraph}(P)$, but with the selection-valued nodes replaced by the corresponding probabilistic models:

$$N([P]) = \{ [\text{Concretize}(P, I)] | I \in N(\text{ModelGraph}(P)) \}$$

In this way, a valid modular program $P$ represents the graph of all probabilistic models that can be produced by recombination of modules, with connections between models that conflict by only one choice of module.

### 3.6 Algorithms

In this section we specify and analyze algorithms for network operations on Modular Stan programs: Concretize, which compiles a node in the model graph to a working Stan program, ModelGraph, which enumerates the node and edge sets of the network of models represented by a Modular Stan program, and ModelNeighbors, which enumerates the neighbors of a model in the network of models. Each algorithm is specified in a functional, mathematical style rather than in an imperative coding style to correspond to our prototype implementations in the functional programming language Haskell.

For each algorithm, we provide proofs of correctness in terms of the semantic definitions defined in Sections 3.3 and 3.5, and we analyze their efficiency.

#### 3.6.1 Concretize

In this section we develop a implementation of the Concretize operation, which produces a Stan program from a Modular Stan program and a valid selection set (shown in Figure 3.2). We write $\text{Concretize}(P, I)$ to mean the concrete Stan program that is derived from a valid Modular Stan program $P$ by applying the implementations of a valid selection set $I$. We define $\text{Concretize}$ by building up functions of increasing scope ($\text{ApplyImpl}$, $\text{ApplyImpls}$ and
then Concretize) and proving the necessary properties of each.

Figure 3.2: The inputs to the Concretize operation are a modular program and a valid selection set (pictured here as a node in the model graph); the output is a Stan program (right).

We build Concretize by careful use of function inlining as a subroutine. Function inlining is the process of removing a function call from a location inside of a program by replacing it with the body of the function. Function inlining is a standard operation in most optimizing compilers, including the Stan compiler [26], so we use it here without re-implementing it. We use the following simplified interface:

\[
\text{InlineFunction(Program, CallSite, Stmts, Params)} \\
\text{InlineFunction(Program, CallSite, Stmts, Params, Return)}
\]

Here, Program is the whole program to be updated, CallSite is a data structure indicating the span of code to be replaced, Stmts are a list of statements that make up the body of the function, Params is a list of the function’s parameters, and Return, when present, is return expression (modules can only have zero or one return statements). When we use InlineFunction,
we take \textbf{Program} to be a Modular Stan program, \textbf{CallSite} to be a HoleSite, \textbf{Stmts} to be \texttt{STMT\_FUNC\_M+}, \textbf{Params} to be \texttt{STMT\_DECL+}, and \textbf{Return} to be \texttt{EXPR\_M}.

\textbf{InlineFunction} assumes the following preconditions:

1. \textbf{Program} typechecks in the scope of \textbf{CallSite}.

2. \textbf{CallSite} supplies arguments matching \textbf{Params}.

3. The function represented by \textbf{Stmts}, \textbf{Params} and \textbf{Return} typechecks.

Then we assume the following properties of \textbf{Program'} = \textbf{InlineFunction}(\textbf{Program}, \textbf{CallSite}, \textbf{Stmts}, \textbf{Params}, \textbf{Return}):

1. References to \textbf{Params} in \textbf{Stmts} and \textbf{Return} are replaced by an argument expression of \textbf{CallSite}.

2. \textbf{Stmts} are inserted in order before \textbf{CallSite} in the same scope as \textbf{CallSite}.

3. When \textbf{Return} is provided, \textbf{CallSite} is replaced by \textbf{Return}, which does not change the expression type.

4. \textbf{Program'} typechecks wherever \textbf{Program} typechecks.

\textbf{ApplyImpl}

Our first step is to define the operation \textbf{ApplyImpl}(P, i): the result of taking a valid Modular Stan program \textit{P} and “applying” an implementation \(i \in \texttt{impls}(P)\), or using \textit{i} to “fill” all instances of the hole \textit{par}(i). The hole that \textit{i} fills, \textit{par}(i), no longer appears in the resulting program \textbf{ApplyImpl}(P, i).

The definition of “correctness” that we will require of \textbf{ApplyImpl} is as follows: for any valid Modular Stan program \textit{P} and \(i \in \texttt{impls}(P)\), \textbf{ApplyImpl}(P, i) is also a valid Modular Stan program such that \(\textit{par}(i) \notin \texttt{holes}(\textbf{ApplyImpl}(P, i))\).

This definition is more concise in imperative pseudocode:
ApplyImpl(P, i):

\[
\begin{align*}
\text{sites} & = \{\text{site} \mid \text{site} \in \text{HoleSites}(p), \text{site}_{\text{hole}} = \text{par}({\text{impl}})\} \\
P' & := P \\
\text{for site in sites:} \\
\quad P' & := \text{InlineFunction}(P', \text{site}, i_{\text{body}}, \\
& \quad \text{signature(st}_{\text{hole}})_{\text{arg-types}}, i_{\text{return}}) \\
\text{return InlineFunction}(P', \text{site}(\text{parameters}),\{\}, i_{\text{parameters}})
\end{align*}
\]

For each location site where the parent hole of i, \text{par}(i), appears, the variable \(P'\) is updated to inline the body of i into that location. Finally, i’s parameters block is appended onto the program parameter block.

ApplyImpl does not affect module signatures.

Correctness of ApplyImpl

In this section we prove the correctness of the above implementation of ApplyImpl.

After ApplyImpl, all holes \text{par}(i) in P are replaced by code containing the holes holes(i):

Lemma 3.6.1 (ApplyImpl\((P, i)\) replaces the each hole filled by i with i’s holes). \text{For all } i' \in \text{impls}(P) \cup \{P_{\text{base}}\}, \text{if } \text{par}(i) \in \text{holes}_P(i') \text{ then } \text{holes}_{\text{ApplyImpl}(P, i)}(i') = \text{holes}_P(i') \cup \text{holes}_P(i) - \{\text{par}(i)\} \text{ and } \text{holes}_{\text{ApplyImpl}(P, i)}(i') = \text{holes}_P(i') \text{ otherwise.}

Proof. If \text{par}(i) \notin \text{holes}_P(i'), ApplyImpl does not make any replacements in i', so \text{holes}_{\text{ApplyImpl}(P, i)}(i') = \text{holes}_P(i').
Otherwise, \(\text{ApplyImpl}(P, i)\) is made up of code from \(i\) and \(i'\), so:

\[
\text{holes}_{\text{ApplyImpl}(P,i)}(i') \subseteq \text{holes}_P(i) \cup \text{holes}_P(i')
\]

\[
= (\text{holes}_P(i) \cap \{\text{par}(i)\}) \cup (\text{holes}_P(i) - \{\text{par}(i)\})
\]

\[
\cup (\text{holes}_P(i') \cap \{\text{par}(i)\}) \cup (\text{holes}_P(i') - \{\text{par}(i)\})
\]

\[
\text{holes}_P(i) \cap \{\text{par}(i)\} = \emptyset, \text{ because otherwise } i \rightarrow i \in E(MDG(P)), \text{ which would violate } \text{Acyclic}(P) \text{ and } \text{valid}(P).
\]

\(\text{ApplyImpl}\) replaces all \(h \in \text{holes}_P(i') \cap \{\text{par}(i)\}\) with \(i_{\text{body}}\), so \(h \notin \text{holes}_{\text{ApplyImpl}(P,i)}(i')\).

Therefore:

\[
\text{holes}_{\text{ApplyImpl}(P,i)}(i') \subseteq (\text{holes}_P(i) - \{\text{par}(i)\}) \cup (\text{holes}_P(i') - \{\text{par}(i)\})
\]

\(\text{ApplyImpl}\) does not remove any \(\text{holes}_P(i') - \{\text{par}(i)\}\), so \(\text{holes}_P(i') - \{\text{par}(i)\} \subseteq \text{holes}_{\text{ApplyImpl}(P,i)}(i')\).

\(\text{ApplyImpl}\) inserts \(\text{holes}_P(i) - \{\text{par}(i)\}\) as part of \(i_{\text{body}}\), so \(\text{holes}_P(i) - \{\text{par}(i)\} \subseteq \text{holes}_{\text{ApplyImpl}(P,i)}(i')\).

Therefore:

\[
\text{holes}_{\text{ApplyImpl}(P,i)}(i') = (\text{holes}_P(i) - \{\text{par}(i)\}) \cup (\text{holes}_P(i') - \{\text{par}(i)\})
\]

\[
= \text{holes}_P(i) \cup \text{holes}_P(i') - \{\text{par}(i)\}
\]

\[\square\]

No code insertion violates the effects or scope constraints of the insertion site:

**Lemma 3.6.2** (ApplyImpl\((P,i)\) replaces the hole filled by \(i\) with code that does not introduce effect or scope violations). For all sites \(st \in \text{sites}(P)\) such that \(st_{\text{hole}} = \text{par}(i)\), the code \(i_{\text{body}}\) replacing \(st\) in ApplyImpl\((P,i)\) has \(\text{effects}(i_{\text{body}}) \subseteq \text{signature}(\text{par}(i))_{\text{effects}}, \text{scope}(i_{\text{body}}) \subseteq \text{signature}(\text{par}(i))_{\text{scope}}\).
signature(par(i))_{\text{scope}}, \text{and} \forall h \in \text{holes}(i),\ signature(h)_{\text{effects}} \subseteq \text{signature}(\text{par}(i))_{\text{effects}} \text{and} \text{signature}(h)_{\text{scope}} \subseteq \text{signature}(\text{par}(i))_{\text{scope}}.

Proof. By definition of signature:

\[
\text{signature}(\text{par}(i))_{\text{effects}} \subseteq \text{effects}(i_{\text{body}}) \cup \bigcup_{h' \in \text{holes}(i)} \text{signature}(h')_{\text{effects}}
\]

and

\[
\text{signature}(\text{par}(i))_{\text{scope}} \subseteq \text{scope}(i_{\text{body}}) \cup \bigcup_{h' \in \text{holes}(i)} \text{signature}(h')_{\text{scope}}
\]

All applications of InlineFunction are valid:

Lemma 3.6.3 (Preconditions of InlineFunction are met). When valid(P), the preconditions of the InlineFunction calls in ApplyImpl(P, i) are met.

Proof. Consider the first call:

\[
\text{InlineFunction}(P, \text{site}, i_{\text{body}}, \text{signature}(st_{\text{hole}})_{\text{arg-types}}, i_{\text{return}})
\]

1. If site is in the base of P, then the P typechecks in the scope of site by semantic constraint 5, otherwise it must be in an implementation body, in which case it typechecks by semantic constraint 6.

2. We know site looks like a function call because of the definition of HoleSites. Its arguments must match \text{signature}(\text{site}_{\text{hole}})_{\text{arg-types}} by semantic constraint 5.

3. Implied by semantic constraint 6.

On subsequent calls, P' is the output of a previous InlineFunction, which by induction meets its preconditions. P' typechecks by the same argument as P because of property 4. Both of the other preconditions hold by the same reasoning as the initial call.
Consider the final call:

\[ \text{InlineFunction}(P', \text{site}(\text{parameters}), \{\}, i_{\text{parameters}}) \]

1. \( P' \) typechecks at \( \text{site}(\text{parameters}) \) because \( P \) does by semantic constraint 5 and \( P' \) typechecks wherever \( P \) does.

2. \( \text{site}(\text{parameters}) \) is a location rather than a hole site and so does not supply any arguments, which matches \( \{\} \).

3. \( \text{Params} \) and \( \text{Return} \) are empty, and \( \text{Stmts} \) can only be a list of \( \text{STMT\_DECL} \) and so typechecks.

Finally, \( \text{ApplyImpl} \) meets our definition of correctness:

**Theorem 3.6.1** (\( \text{ApplyImpl} \) produces valid Modular Stan programs). \( \text{valid}(P), i \in P \implies \text{valid}(\text{ApplyImpl}(P, i)) \) and \( \text{par}(i) \notin \text{ApplyImpl}(P, i) \).

**Proof.** To prove that the resulting program is valid, we will prove each of the structural and semantic constraints defining validity.

**Structural constraints:**

1. *The dependency graph of modules is acyclic.*

Suppose there is a cycle \( C \) in \( \text{MDG}(\text{ApplyImpl}(P, i)) \).

Let \( i_0 \to i_1 \) be an edge in \( C \); then \( \text{par}(i_1) \in \text{holes}_{\text{ApplyImpl}(P, i)}(i_0) \). By Lemma 3.6.1, either \( \text{par}(i_1) \in \text{holes}_P(i_0) \), in which case \( i_0 \to i_1 \in E(\text{MDG}(P)) \); or \( \text{par}(i) \in \text{holes}(i_0) \) and \( \text{par}(i_1) \in \text{holes}_P(i_0) \), in which case \( i_0 \to i, i \to i_1 \in E(\text{MDG}(P)) \).

In either case, for all edges \( i_0 \to i_1 \in C \), there exists a path from \( i_0 \) to \( i_1 \) in \( \text{MDG}(P) \), so there exists a cycle in \( \text{MDG}(P) \). As that would contradict \( \text{valid}(P) \), there is no cycle \( C \) in \( \text{ApplyImpl}(P, i) \).
2. Every hole has an implementation.

ApplyImpl does not remove any implementations from or add any new holes to $P$, so any holes without implementations would also be without implementations in $P$, which would violate $\text{valid}(P)$.

3. Hole identifiers are unique and hole identifier plus implementation identifier pairs are unique.

ApplyImpl does not change any hole implementation identifier names, so any collisions would exist in $P$ and violate $\text{valid}(P)$.

Semantic constraints:

1. Implementations match signature arg-types.

ApplyImpl does not change signatures or implementation argument types.

2. Implementations match signature rettype.

$$\forall i' \in \text{impls}(P), i'_{\text{ret-type}} = \text{signature}(\text{par}(i'))_{\text{ret-type}}.$$  

If $i_{\text{return}}$ is a hole expression of $\text{par}(i)$, then it is replaced by $\text{ApplyImpl}(P, i)$. By Lemma 3.6.3, property 3 of InlineFunction for the replacement, so the type of $i_{\text{return}}$ is not changed.

If the type of $i_{\text{return}}$ depends on a hole expression of $\text{par}(i)$, by the same reasoning, the type does not change. Otherwise, it is unaffected by $\text{ApplyImpl}$.

3. Effects and scopes of holes are available in their block or module signature.

Suppose $\exists st \in \text{sites}(\text{ApplyImpl}(P, i))$ so that $st$ violates one of the conditions.

If $st \in \text{sites}(P)$, since $\text{block}$ and $\text{signature}$ are unaffected by $\text{ApplyImpl}$, then $st$ would violate $\text{valid}(P)$. Therefore, $st \notin \text{sites}(P)$ and must have been part of a replacement of $\text{par}(i)$ by $i_{\text{body}}$.

By Lemma 3.6.2, we have $\text{signature}(st_{\text{hole}})_{\text{effects}} \subseteq \text{signature}(\text{par}(i))_{\text{effects}}$ and $\text{signature}(st_{\text{hole}})_{\text{scope}} \subseteq \text{signature}(\text{par}(i))_{\text{scope}}$. Since the scope and effects of $st_{\text{hole}}$ are
subsets of the scope and effects of \( par(i) \), and \( par(i) \) passes the four conditions, \( st_{\text{hole}} \) must also pass the four conditions.

4. **Effects and scopes of code are available in their block or module signature.**

   By identical reasoning to the previous case, for any code in a block or module implementation, the code either existed in \( P \), in which case it must be valid, or it replaced \( par(i) \), in which case by Lemma 3.6.2 it requires only a subset of the scope and effects of \( par(i) \), and is therefore also valid.

5. \( P_{\text{base}} \) would typecheck under Stan if all holes \( h \) were function calls to function signatures with signature \( (h)_{\text{arg-types}} \) parameter types and signature \( (h)_{\text{ret-type}} \) return type.

   By Lemma 3.6.3, property 4 of InlineFunction applies for all updates to \( P \) in ApplyImpl.

   Since \( P_{\text{base}} \) typechecks, \( \text{ApplyImpl}(P, i)_{\text{base}} \) also typechecks.

6. **The blockless statements of each implementation \( i \) would typecheck under Stan if it were the body of a function with signature \( (par(i))_{\text{arg-types}} \) parameter types and with signature \( (par(i))_{\text{ret-type}} \) return type, and if the module-defined parameters \( i_{\text{parameters}} \) were included as model parameters.**

   By Lemma 3.6.3, property 4 of InlineFunction applies for all updates to \( i_{\text{body}} \) for any \( i \in \text{impls}(P) \) in ApplyImpl. Since \( i_{\text{body}} \) typechecks in \( P \) for all \( i \in \text{impls}(P) \), \( i_{\text{body}} \) also typechecks in \( \text{ApplyImpl}(P, i) \).

   The result \( par(i) \notin \text{ApplyImpl}(P, i) \) follows from the Lemma 3.6.1 and the definition of sites. \( \square \)

**ApplyImpls**

Next we define a \( \text{ApplyImpls} \) in terms of \( \text{ApplyImpl} \). Let \( \text{ApplyImpls}(P, I) \) be the result of applying \( \text{ApplyImpl}(P, i) \) for each \( i \in I \). Formally, if we let \( I = < i_1, \ldots, i_N > \), \( P_0 = P \), and \( P_j = \text{ApplyImpl}(P_{j-1}, i_j) \) for \( j \in [1, N] \), then \( \text{ApplyImpls}(P, I) = P_N \).
The property that we will require of \texttt{ApplyImpls} is that it adds $I$’s holes and removes the holes filled by $I$.

**Correctness of \texttt{ApplyImpls}**

**Theorem 3.6.2** (\texttt{ApplyImpls}(P, I) adds $I$’s holes and removes the holes filled by $I$). For all $i' \in \text{impls}(P) \cup \{P_{\text{base}}\}$, $\text{holes}_{\text{ApplyImpls}(P, I)}(i') \subseteq \text{holes}_P(i') \cup \text{holes}_P(I) - \text{pars}(I)$

**Proof.** Let $I_j = <i_1, \ldots, i_j>$ so that $I_N = I$. Note that $\text{ApplyImpls}(P, I_j) = P_j$.

We prove the following by induction on $j$:

$$\text{holes}_{P_j}(i') \subseteq \text{holes}_P(i') \cup \text{holes}_P(I_j) - \text{pars}(I_j)$$

For $j = 0$: $\text{holes}_{P_0}(i') \subseteq \text{holes}_P(i') \cup \text{holes}_P(I_0) - \text{pars}(I) = \text{holes}_P(i') \cup \emptyset - \emptyset = \text{holes}_{P_0}(i')$.

For $j > 0$: By Lemma 3.6.1, $\text{holes}_{P_j}(i') = \text{holes}_{\text{ApplyImpls}(P_{j-1}, i_j)}(i') \subseteq \text{holes}_{P_{j-1}}(i') \cup \text{holes}_{P_{j-1}}(i_j) - \text{par}(i_j)$.

By induction:

$$\text{holes}_{P_j}(i') \subseteq (\text{holes}_P(i') \cup \text{holes}_P(I_{j-1}) - \text{pars}(I_{j-1})) \cup \text{holes}_{P_{j-1}}(i_j) - \text{par}(i_j).$$

Again by induction, we can use $\text{holes}_{P_{j-1}}(i_j) \subseteq \text{holes}_P(i_j) \cup \text{holes}_P(I_{j-1}) - \text{pars}(I_{j-1})$:

$$\text{holes}_{P_j}(i') \subseteq (\text{holes}_P(i') \cup \text{holes}_P(I_{j-1}) - \text{pars}(I_{j-1})) \cup (\text{holes}_P(i_j) \cup \text{holes}_P(I_{j-1}) - \text{pars}(I_{j-1})) - \text{par}(i_j).$$

$$\text{holes}_{P_j}(i') \subseteq \text{holes}_P(i') \cup \text{holes}_P(I_j) - \text{par}(I_j).$$

\qed

**Concretize**

We define \texttt{Concretize} as $\text{Concretize}(P, I) = \text{ApplyImpls}(P, I)_{\text{base}}$ when $I$ is a valid selection.
We will see that the definition of validity for selections implies that Concretize meets our correctness criteria.

Correctness of Concretize

Concretize removes all holes from $P$:

**Lemma 3.6.4** (Concretize($P, I$) has no holes). If $\text{valid}(P)$ and $\text{valid}_p(I)$, then $\text{holes}(\text{Concretize}(P, I)) = \emptyset$.

**Proof.** By Theorem 3.6.2, $\text{holes}(\text{ApplyImpls}(P, I)_{\text{base}}) \subseteq \text{holes}(P_{\text{base}}) \cup \text{holes}(I) - \text{pars}(I)$. By $\text{valid}_p(I)$, $\text{holes}(P_{\text{base}}) \cup \text{holes}(I) = \text{pars}(I)$. Therefore, $\text{holes}(\text{ApplyImpls}(P, I)_{\text{base}}) \subseteq \emptyset$. $\square$

Concretize results in a valid concrete Stan program, and is therefore correct:

**Theorem 3.6.3** (Concretize($P, I$) is a valid Stan program). valid($P$) and valid$_p(I)$ implies valid$_{\text{Stan}}$(Concretize($P, I$)).

**Proof.** By Theorem 3.6.1 we have valid($\text{ApplyImpls}(P, I)$).

By Lemma 3.6.4, we have holes(Concretize($P, I$)) = $\emptyset$. Therefore, $SP = \text{Concretize}(P, I)$ is a concrete Stan program.

We can match the criteria of valid$_{\text{Stan}}$(SP) in Section 3.3.3:

1. Effects and scopes of code are available in their block.

   Implied by valid($\text{ApplyImpls}(P, I)$), semantic constraint 4.

2. SP typechecks as Stan code.

   Implied by valid($\text{ApplyImpls}(P, I)$), semantic constraint 5.

$\square$

This result shows that Concretize implements the one-to-one correspondence between valid selections and valid Stan programs.
User specification of a selection

This section deals with the practicality of implementing Concretize: in order to call Concretize, we need a way to write down a “selection”. To let users write selections, we define an additional piece of syntax for them. We refer to strings matching this syntax as *selection strings*.

**SELECTION**: hole_identifier:impl_identifier,+ 

Selection strings are therefore strings that references a set of module implementations within a given program. Our prototype implementation lets users provide selection strings to narrow down and index into the set of programs represented by a modular program.

### 3.6.2 ModelGraph

In this section we present an efficient algorithm to implement the ModelGraph operation, which produces a graph or network of models (shown in Figure 3.3). We defined ModelGraph in Section 3.5.6 as:

\[
N(\text{ModelGraph}(P)) = \{ I \mid valid_P(I) \} \\
E(\text{ModelGraph}(P)) = \{ (I_1, I_2) \mid I_1, I_2 \in N(\text{ModelGraph}(P)), |\text{siblings}(I_1, I_2)| = 1 \}
\]

This definition serves as the correctness criteria for our implementation. In addition, we will say that an implementation is *sound* when it includes only nodes or edges in the network of models: \( N(\text{ModelGraph}(P)) \subseteq \{ I \mid valid_P(I) \} \) and equivalent for edges. We will say that it is *complete* when it includes all of the nodes or edges in the network: \( N(\text{ModelGraph}(P)) \supseteq \{ I \mid valid_P(I) \} \) and equivalent for edges. An implementation is then correct when it is sound and complete.
Naive algorithms

To demonstrate what we mean by efficient, consider a naive construction of the model graph given a modular program \( P \):

\[
N(\text{NaiveModelGraph}(P)) = \left\{ \text{close}(P_{\text{base}}, I) \big| I \in \prod_{h \in \text{holes}(P)} \text{impls}(h), \text{valid}_P(\text{close}(P_{\text{base}}, I)) \right\}
\]

\[
E(\text{NaiveModelGraph}(P)) = \left\{ (I_1, I_2) \big| I_1, I_2 \in N(\text{NaiveModelGraph}(P)), |\text{siblings}(I_1, I_2)| = 1 \right\}
\]

Here, \( \text{close}(i, I) \) is the subset of \( I \) that is reachable from \( i \) by traversing edges in the module graph through \( I \). Thus \( \text{close}(P_{\text{base}}, I) \) effectively removes redundant implementations from \( I \). \text{NaiveModelGraph} works by enumerating all possible combinations of implementations and
model pairs and then filtering out the invalid options.

\[ E(\text{NaiveModelGraph}(P)) \] is exponentially inefficient when \( P \) is Figure 3.4a, because it would consider \( 2^N \) candidate neighbors for each node, where in reality each node has only \( N - 1 \) neighbors.

\[ N(\text{NaiveModelGraph}(P)) \] is exponentially inefficient when \( P \) is Figure 3.4b, because it would consider \( 2^N \) candidate nodes, where in reality there are only \( N \) nodes.

Another implementation would be a recursive construction. If the module graph is a tree, then we could recursively construct a network for the subtree under each hole node. We could then build up the network at any node by combining the networks of its children. The network of the sub-graph descending from each hole node is the graph join of the networks of the sub-graphs descending from its implementations, and the network of the sub-graph descending from each implementation node is the graph Cartesian product of the networks of the sub-graphs descending from its holes. The issue with this approach is how to handle situations like Figures 3.4c and 3.4d: the module graph is not always a tree, so hole and implementation nodes will sometimes combine networks with overlapping information. It is possible to modify the graph join and Cartesian product operations to handle the overlap correctly, but not without adding complexity and inefficiency that scales with the number of redundant paths in the module graph.

**Efficient algorithm**

Our approach is to build up the model graph’s node and edge sets simultaneously by “visiting” each of the program’s holes in a careful order. We track the set of model and edge “prefixes”: partial selections that only contain implementations for the holes visited thus far. We also track the set of holes required by the implementations in each prefix. When we “visit” a hole, we “expand” each prefix that requires that hole into additional prefixes, one for each implementation that could fill the hole. To ensure that no “prefix” discovers that it requires a hole after that hole has been visited, and to avoid other complex edge cases, holes are visited in a topological order of dependency.
Figure 3.4: Pathological module graphs. Round nodes represent the base and module implementations while rectangular nodes represent holes.
We first define the “expansion” operation, and then we specify the order that nodes will be visited and how the final result is arrived at.

$$\text{expand}(N, E, h) =$$

$$\bigcup_{(H, I) \in N} \text{expand-node}(H, I, h),$$

$$\bigcup_{(H, I) \in N} \text{new-edges}(H, I, h) \cup \bigcup_{(H_1, I_1, H_2, I_2) \in E} \text{expand-edge}(H_1, I_1, H_2, I_2, h)$$

$$\text{expand-node}(H, I, h) =$$

$$\begin{cases} 
\{(H \cup \text{holes}(i), I \cup \{i\}) \mid i \in \text{impls}(h)\} & \text{if } h \in H \\
\{(H, I)\} & \text{otherwise}
\end{cases}$$

$$\text{new-edges}(H, I, h) =$$

$$\begin{cases} 
\{(H \cup \text{holes}(i_1), I \cup \{i_1\}, H \cup \text{holes}(i_2), I \cup \{i_2\}) \mid (i_1, i_2) \in \binom{\text{impls}(h)}{2}\} & \text{if } h \in H \\
\emptyset & \text{otherwise}
\end{cases}$$

$$\text{expand-edge}(H_1, I_1, H_2, I_2, h) =$$

$$\begin{cases} 
\{(H_1, I_1, H_2, I_2)\} & \text{if } h \notin H_1, h \notin H_2 \\
\{(H_1 \cup \text{holes}(i), I_1 \cup \{i\}, H_2, I_2) \mid i \in \text{impls}(h)\} & \text{if } h \in H_1, h \notin H_2 \\
\{(H_1, I_1, H_2 \cup \text{holes}(i), I_2 \cup \{i\}) \mid i \in \text{impls}(h)\} & \text{if } h \notin H_1, h \in H_2 \\
\{(H_1 \cup \text{holes}(i), I_1 \cup \{i\}, H_2 \cup \text{holes}(i), I_2 \cup \{i\}) \mid i \in \text{impls}(h)\} & \text{if } h \in H_1, h \in H_2
\end{cases}$$

Let $G_{\text{Dep}}(P)$ be the graph of dependencies between holes of a program $P$ so that $N(G_{\text{Dep}}(P)) = \text{holes}(P)$ and $h_1 \rightarrow h_2 \in E(G_{\text{Dep}}(P))$ if and only if $\exists i \in \text{impls}(h_1)$ s.t. $h_2 \in \text{holes}(i)$. Let $\tilde{H}$ be a topological ordering of $G(P)$.

Let $(N_0, E_0) = ((\text{holes}(P_{\text{base}}), \emptyset), \emptyset)$. 

103
Let \((N_j, E_j) = \text{expand}(N_{j-1}, E_{j-1}, h_j), \forall j \in [1, |\tilde{H}|]\).

When \(j = |\tilde{H}|\), our set of nodes is \(\{I \mid (H, I) \in N_{|\tilde{H}|}\}\) and our set of edges is:

\[
\{(I_1, I_2) \mid (H_1, I_1, H_2, I_2) \in E_{|\tilde{H}|}\} = E(\text{ModelGraph}(P))
\]

**Proofs of soundness and completeness**

We show that \(\{I \mid (H, I) \in N_{|\tilde{H}|}\} = N(\text{ModelGraph}(P))\) and \(\{(I_1, I_2) \mid (H_1, I_1, H_2, I_2) \in E_{|\tilde{H}|}\} = E(\text{ModelGraph}(P))\).

Throughout this section, we assume that \(P\) is a modular program such that \(\text{valid}_{\text{structure}}(P)\). \(\text{valid}_{\text{semantics}}(P)\) is not necessary since we are dealing only with the module dependency structure of \(P\).

Firstly, we are guaranteed an appropriate topological ordering of holes:

**Lemma 3.6.5** (\(\tilde{H}\) is ordered by dependency). There exists a topological ordering \(\tilde{H} \Rightarrow G_{\text{Dep}}(P)\) and \(\forall j \in [1, |\tilde{H}|], \forall i \in \text{impls}(P) \Rightarrow \tilde{H}_j \in \text{holes}(i), \text{par}(i) \in \tilde{H}_{1:j-1}\) or \(\text{par}(i) \in \text{holes}(P_{\text{base}})\).

**Proof.** The graph \(G_{\text{Dep}}\) must be acyclic, because otherwise the holes of \(P\) would have cyclic dependencies and \(P\) would not be valid. Therefore \(G_{\text{Dep}}\) has at least one topological order \(\tilde{H}\).

For any such \(j\) and \(i\), we must have \(\text{par}(i) \rightarrow \tilde{H}_j \in E(G_{\text{Dep}}(P))\), by definition of \(G_{\text{Dep}}\). Then either \(\text{par}(i) \in \text{holes}(P_{\text{base}})\) and we are done, or \(\text{par}(i) \in \text{holes}(P)\), and so \(\text{par}(i) \in \tilde{H}\), so \(\tilde{H}_k\) for some \(k\). By the definition of a topological ordering, \(k < j\). Therefore, \(\text{par}(i) \in \tilde{H}_{1:j-1}\). \(\square\)

The useful invariants of the node “prefixes” that we collect are that they are exactly the intersections of the final nodes with the implementations of the set of holes that we have visited so far.

We show that the prefixes contain all of those intersections:

**Lemma 3.6.6** (Model prefixes have complete prefix hole sets). \(\forall j \in [1, |\tilde{H}|], \forall (H_{j-1}, I_{j-1}) \in N_{j-1}, \text{let } N^j = \{I' \mid I' \in N(\text{ModelGraph}(P)), I_{j-1} \subseteq I'\}\).
For all $j$, $\text{pars}(i') \cap H_{1:j} = H_{j-1} \cap H_{1:j}$.

**Proof.** This is a proof by induction on $j$.

We will simultaneously prove the following property, which we refer to as “consistency,” by induction for each $j$:

$$I' \cap \text{impls}(H_{1:j-1}) = I_{j-1} \cap \text{impls}(H_{1:j-1})$$

For $j = 1$:

Since $N_0 = \{\emptyset\}, I_{j-1} = \emptyset$, so $N^I = N(\text{ModelGraph}(P))$.

If $H_1 \in \text{pars}(I')$, then by valid$_P(I')$, $H_1 \in \text{holes}(P_{\text{base}})$, since if $\exists i \in \text{impls}(I')$ s.t. $H_1 \in \text{holes}(i)$, then by Lemma 3.6.5, par$(i) \in H_{1:0} = \emptyset$. Then since $\text{holes}(P_{\text{base}}) \subseteq \text{pars}(I')$ and $H_0 = \text{holes}(P_{\text{base}})$, par$(I') \cap H_{1:1} = H_0 \cap H_{1:1} = \emptyset$.

If $H_1 \notin \text{pars}(I')$, then $H_1 \notin \text{holes}(P_{\text{base}}) = H_0$, so $\text{pars}(I') \cap H_{1:1} = H_0 \cap H_{1:1} = \emptyset$.

Therefore the lemma holds for $j = 1$.

Trivially, consistency holds for $j = 1$ because $h \notin H_{1:0}$.

For $j > 1$:

Since $(H_{j-1}, I_{j-1}) \in N_{j-1}$ and $j \geq 2$, $\exists(H_{j-2}, I_{j-2}) \in N_{j-2}$ such that $(H_{j-1}, I_{j-1}) \in \text{expand-node}(H_{j-2}, I_{j-2}, H_{j-1})$.

If $H_{j-1} \in H_{j-2}$, then by induction of the lemma, $H_{j-1} \in \text{pars}(I')$, so $I' \cap \text{impls}(H_{j-1}) = \{i'\}$ for some $i'$. It also follows from the definition of expand-node that, for some $i \in \text{impls}(H_{j-1})$, $I_{j-1} = I_{j-2} \cup \{i\}$, and since $I_{j-2} \cap \text{impls}(H_{j-1}) = \emptyset$, $I_{j-1} \cap \text{impls}(H_{j-1}) = \{i\}$.

Since $I_{j-1} \subseteq I'$, $i \in I'$.

If not $i = i'$, then since $\text{par}(i) = \text{par}(i')$, $(i, i') \in \text{siblings}(I', I')$, but valid$_P(I')$ implies $\text{siblings}(I', I') = \emptyset$, so $i = i'$.

Thus, if $H_{j-1} \in H_{j-2}$, then $I' \cap \text{impls}(H_{j-1}) = I_{j-1} \cap \text{impls}(H_{j-1}) = \{i\}$. If $H_{j-1} \notin H_{j-2}$, then $H_{j-1} \notin \text{pars}(I')$ and $I_{j-1} = I_{j-2}$, so $I' \cap \text{impls}(H_{j-1}) = I_{j-1} \cap \text{impls}(H_{j-1}) = \emptyset$.

Starting from induction on consistency:

$$I' \cap \text{impls}(H_{1:j-2}) = I_{j-2} \cap \text{impls}(H_{1:j-2})$$
(I' ∩ impls(\(\tilde{H}_{1:j-2}\))) ∪ (I' ∩ impls(\(\tilde{H}_{j-1}\))) = (I_{j-2} ∩ impls(\(\tilde{H}_{1:j-2}\))) ∪ (I_{j-1} ∩ impls(\(\tilde{H}_{j-1}\)))

Because \(I_{j-2} = I_{j-1} ∩ impls(\(\tilde{H}_{1:j-2}\)):\)

(\(I' ∩ impls(\(\tilde{H}_{1:j-2}\))) ∪ (I' ∩ impls(\(\tilde{H}_{j-1}\))) = (I_{j-1} ∩ impls(\(\tilde{H}_{1:j-2}\))) ∪ (I_{j-1} ∩ impls(\(\tilde{H}_{j-1}\)))

\(I' ∩ impls(\(\tilde{H}_{1:j-1}\)) = I_{j-1} ∩ impls(\(\tilde{H}_{1:j-1}\))\)

Thus, consistency holds.

If \(\tilde{H}_j \in holes(P_{\text{base}})\), then the lemma is true by the same reasoning as the \(j = 1\) case.

Suppose \(\tilde{H}_j \in H_{j-1}\) and \(\tilde{H}_j \notin holes(P_{\text{base}})\). By the construction of \(H_{j-1}\) and the definition of expand-node, \(\exists i \in I_{j-1}\) such that \(\tilde{H}_j \in holes(i)\). \(I_{j-1} \subseteq I', i \in I', \text{ so } \tilde{H}_j = \text{par}(i) \in \text{pars}(I')\).

Suppose \(\tilde{H}_j \in \text{pars}(I')\) and \(\tilde{H}_j \notin holes(P_{\text{base}})\). By valid\(_p(I')\) and Lemma 3.6.5, there exists some \(k < j\) such that \(\text{par}(i) = \tilde{H}_k\) and \(i \in impls(\tilde{H}_k)\). By consistency, \(I' \cap impls(\tilde{H}_{1:k}) = I_k \cap impls(\tilde{H}_{1:k})\), so \(i \in I_k \subseteq I_{j-1}\). By the definition of expand-node(\(H_{k-1}, I_{k-1}, \tilde{H}_k\), \(i \in I_k \implies holes(i) \subseteq H_k\), so \(\tilde{H}_j \in H_k \subseteq H_{j-1}\). \(\square\)

Each model prefix can be expanded with any implementation that fills a hole in its set of hole requirements and remain a valid model prefix:

**Lemma 3.6.7** (Model prefixes can be expanded with any implementation). \(\forall j \in [1, |\tilde{H}|], \forall I \in N(ModelGraph(P)), \forall h \in holes(I \cap impls(\tilde{H}_{1:j}) \cup holes(P_{\text{base}})) - \tilde{H}_{1:j}, \forall i \in impls(h), \exists I' \in N(ModelGraph(P))\) s.t. \(I \cap impls(\tilde{H}_{1:j}) \cup \{i\} \subseteq I'\).

**Proof.** We will construct a selection set that satisfies the property by modifying \(I\).

Let \(I_1 = I - impls(h) \cup \{i\}\). Let \(I_2 = I_1 \cup \{\text{any}(impls(h)) \mid h \in holes(P) - \text{pars}(I_2)\}\), where \(\text{any}(s)\) is an arbitrary element of a set. The sets \(impls(h)\) are never empty by valid\(_p(P)\).

Then, \(\text{pars}(I_2) = holes(P)\) and \(I \cap impls(\tilde{H}_{1:j}) \cup \{i\} \subseteq I_2\).

Let \(f(I) = \{i \mid i \in I, \text{par}(i) \in holes(I) \cup holes(P_{\text{base}})\}\). Let \(I_3\) be the fixed point applying \(f\) to \(I_2\). We know \(f\) converges to some \(I_3\) because \(|I|\) is finite, cannot go below 0, and decreases monotonically under \(f\) until convergence.
I_3 meets the criteria for valid_P(I_3):

1. \( I_3 \subseteq \text{impls}(P) : I_3 \subseteq I_2 \subseteq \text{impls}(\text{holes}(P)) \cup I \cup \{i\} \subseteq \text{impls}(P) \) by valid_P(I).

2. By \( I_3 = f(I_3) \) we know \( \text{pars}(I_3) \subseteq \text{holes}(I_3) \cup \text{holes}(P_{base}) \). Since \( \text{pars}(I_2) = \text{holes}(P) \supseteq \text{holes}(I_3) \cup \text{holes}(P_{base}) \) and \( f \) does not remove any \( h \in \text{holes}(I_3) \cup \text{holes}(P_{base}) \), we have \( \text{pars}(I_3) = \text{holes}(I_3) \cup \text{holes}(P_{base}) \).

3. \( \text{siblings}(I_3, I_3) = \emptyset \), because only one \( i \in \text{impls}(h) \) for each \( h \notin \text{pars}(I) \) is added to \( I \) in the construction of \( I_1 \) and \( I_2 \).

By construction, \( I \cap \text{impls}(\tilde{H}_{1:j}) \cup \{i\} \subseteq I_3 \). □

Our set of prefixes at each expansion step therefore exactly matches the intersections of the final nodes:

**Lemma 3.6.8** (\( N_j \) represents exactly the model prefixes). \( \forall j \in \mathbb{N} \), \( \{I \mid (H, I) \in N_j\} = \{I \cap \text{impls}(\tilde{H}_{1:j}) \mid I \in N(\text{ModelGraph}(P))\} \).

**Proof.** This is a proof by induction on \( j \).

For \( j = 0 \), \( \{I \mid (H, I) \in N_0\} = \{\emptyset\} = \{I \cap \emptyset \mid I \in N(\text{ModelGraph}(P))\} \).

For \( j \geq 1 \):

Suppose \( \exists I \in \{I \cap \text{impls}(\tilde{H}_{1:j}) \mid I \in N(\text{ModelGraph}(P))\} \) such that \( I \notin \{I \mid (H, I) \in N_j\} \).

Let \( I' = I \cap \text{impls}(\tilde{H}_{1:j-1}) \). By induction, \( I' \in \{I \mid (H, I) \in N_{j-1}\} \), so \( \exists H' \) s.t. \( (H', I') \in N_{j-1} \).

If \( \tilde{H}_j \in H' \), then by Lemma 3.6.6, \( \tilde{H}_j \in \text{pars}(I) \), so \( \exists i \in I \) s.t. \( i \in \text{impls}(\tilde{H}_j) \). By the definition of \( \text{expand-node}(H', I', \tilde{H}_j) \), \( \forall i' \in \text{impls}(\tilde{H}_j), I' \cup \{i'\} \in \{I \mid (H, I) \in N_j\} \), so \( I' \cup \{i\} = I \in \{I \mid (H, I) \in N_j\} \), which contradicts the construction of \( I \).

If \( \tilde{H}_j \notin H' \), then by Lemma 3.6.6, \( \tilde{H}_j \notin \text{pars}(I) \), so \( I' = I \). By the definition of \( \text{expand-node}(H', I', \tilde{H}_j) \), \( (H', I') \in N_j \), so \( I \in \{I \mid (H, I) \in N_j\} \), which again contradicts the construction of \( I \).
Therefore, there is no such I, and \( \{ I \cap \text{impls}(\tilde{H}_{1:j}) \} \subseteq \{ I \mid (H, I) \in N_j \} \).

Suppose \( \exists I \in \{ I \mid (H, I) \in N_j \} \) such that \( I \notin \{ I \cap \text{impls}(\tilde{H}_{1:j}) \} \subseteq \{ I \mid (H, I) \in N(\text{ModelGraph}(P)) \} \).

By the definition of \( N_j \), \( \exists (H', I') \in N_{j-1} \) s.t. \( (H, I) \in \text{expand-node}(H', I', \tilde{H}_j) \). By induction, \( I' \in \{ I \cap \text{impls}(\tilde{H}_{1:j-1}) \mid I \in N(\text{ModelGraph}(P)) \} \), and therefore \( \exists I^* \in N(\text{ModelGraph}(P)) \) s.t. \( I' \subseteq I^* \).

If \( \tilde{H}_j \notin H' \), then by the definition of \( \text{expand-node} \), \( I = I' \), and by Lemma 3.6.6, \( \tilde{H}_j \notin \text{pars}(I^*) \); so \( I = I^* \cap \text{impls}(\tilde{H}_{1:j-1}) = I^* \cap \text{impls}(\tilde{H}_{1:j}) \), so \( I \in \{ I \cap \text{impls}(\tilde{H}_{1:j}) \mid I \in N(\text{ModelGraph}(P)) \} \).

If \( \tilde{H}_j \in H' \), then by the definition of \( \text{expand-node} \), \( \exists i \in \text{impls}(\tilde{H}_j) \) s.t. \( I = I' \cup \{i\} \), and by Lemma 3.6.6, \( \tilde{H}_j \in \text{pars}(I^*) \).

Because \( \tilde{H}_j \in \text{holes}(I^* \cap \text{impls}(\tilde{H}_{1:j-1})) \), by Lemma 3.6.7, \( \exists I' \in N(\text{ModelGraph}(P)) \) s.t. \( I \cap \text{impls}(H_{1:j-1}) \cup \{i\} \subseteq I' \), so \( I' \cup \{i\} = I \subseteq I' \), so \( I \in \{ I \cap \text{impls}(\tilde{H}_{1:j}) \mid I \in N(\text{ModelGraph}(P)) \} \).

Therefore, \( \{ I \mid (H, I) \in N_j \} \subseteq \{ I \cap \text{impls}(\tilde{H}_{1:j}) \mid I \in N(\text{ModelGraph}(P)) \} \).

Therefore, at the final stage when all holes have been visited, the set of prefixes is the desired set of nodes:

**Theorem 3.6.4** \( (N_{\tilde{H}} | \tilde{H}|) \) is sound and complete. \( \{ I \mid (H, I) \in N_{\tilde{H}} \} = N(\text{ModelGraph}(P)) \).

**Proof.** By Lemma 3.6.8,

\[
\{ I \mid (H, I) \in N_{\tilde{H}} \} = \{ I \cap \text{impls}(\tilde{H}_{1::\tilde{H}}) \mid I \in N(\text{ModelGraph}(P)) \}
\]

Then, because \( \tilde{H}_{1::\tilde{H}} = \text{pars}(P) \), \( \{ I \cap \text{impls}(\tilde{H}_{1::\tilde{H}}) \mid I \in N(\text{ModelGraph}(P)) \} = N(\text{ModelGraph}(P)) \). □

We will follow similar logic for our edge set. We start by showing that, at every expansion step, the endpoints of edge prefixes are also in the set of node prefixes:
Lemma 3.6.9 (Edge prefix endpoints are model prefixes). \( \forall j \in [0, |\vec{H}|], \forall (H_1, I_1, H_2, I_2) \in E_j, (H_1, I_1) \in N_j \text{ and } (H_2, I_2) \in N_j. \)

Proof. This is a proof by induction on \( j \).

For \( j = 0 \), the lemma is trivially satisfied because \( E_0 = \emptyset \).

For \( j \geq 1 \):

By definition of expand, \((H_1, I_1, H_2, I_2)\) came either from \text{new-edges} expand-edge.

If \((H_1, I_1, H_2, I_2) \in \text{new-edges}(H', I', \vec{H}_j)\) for some \((H', I') \in N_{j-1}\), then \(\vec{H}_j \in H'\) and \((H_1, I_1) = (H' \cup \text{holes}(i_1), I' \cup \{i_1\})\), \((H_2, I_2) = (H' \cup \text{holes}(i_2), I' \cup \{i_2\})\), so \((H_1, I_1), (H_2, I_2) \in \text{expand-node}(H', I', \vec{H}_j) \subseteq N_j. \)

If \((H_1, I_1, H_2, I_2) \in \text{expand-edge}(H'_1, I'_1, H'_2, I'_2, \vec{H}_j)\) for some \((H'_1, I'_1, H'_2, I'_2) \in E_{j-1}\), then by induction \((H'_1, I'_1), (H'_2, I'_2) \in N_{j-1}\). If \(\vec{H}_j \in H'_1\), then \(\exists i \in \text{impls}(H'_1) \text{ s.t. } (H_1, I_1) = (H'_1 \cup \text{holes}(i), I'_1 \cup \{i\}) \in \text{expand-node}(H'_1, I'_1, \vec{H}_j) \subseteq N_j.\) If \(\vec{H}_j \notin H'_1\), then \((H_1, I_1) = (H'_1, I'_1) \in \text{expand-node}(H'_1, I'_1, \vec{H}_j) \subseteq N_j.\) The same reasoning applies for \((H_2, I_2)\).

Just like our desired network edges, edge prefixes have endpoints that differ at exactly one hole:

Lemma 3.6.10 (Edge prefix endpoints share one pair of siblings). \( \forall j \in [0, |\vec{H}|], \forall (H_1, I_1, H_2, I_2) \in E_j, |\text{siblings}(I_1, I_2)| = 1. \)

Proof. This is a proof by induction on \( j \).

For \( j = 0 \), the lemma is trivially satisfied because \( E_0 = \emptyset \).

For \( j \geq 1 \):

By definition of expand, \((H_1, I_1, H_2, I_2)\) came either from \text{new-edges} expand-edge.

If \((H_1, I_1, H_2, I_2) \in \text{new-edges}(H', I', \vec{H}_j)\) for some \((H', I') \in N_{j-1}\), then \(\exists (i_1, i_2) \in (\text{impls}(\vec{H}_j)/2)\) such that \(I_1 = I' \cup \{i_1\}, I_2 = I' \cup \{i_2\}, \text{par}(i_1) = \text{par}(i_2) = \vec{H}_j\), so \((i_1, i_2) \in \text{siblings}(I_1, I_2)\). Since \(I_1\) and \(I_2\) are otherwise identical, \(\text{siblings}(I_1, I_2)\) can have no other elements.

If \((H_1, I_1, H_2, I_2) \in \text{expand-edge}(H'_1, I'_1, H'_2, I'_2, \vec{H}_j)\) for some \((H'_1, I'_1, H'_2, I'_2) \in E_{j-1}\), then by induction \(\text{siblings}(I'_1, I'_2) = \{(i_1, i_2)\}\) for some \(i_1, i_2\). Since \((I_1, I_2)\) is either \((I'_1, I'_2)\) or
($I'_1 \cup \{i\}, I'_2 \cup \{i\})$ for some $i \in \text{impls}(\bar{H}_j)$, $I_1$ and $I_2$ cannot have gained or lost any siblings, so $\text{siblings}(I_1, I_2) = \{(i_1, i_2)\}$.

We show a consequence of the definition of siblings and validity of selections - if two prefixes of valid selections, such as the endpoints of an edge prefix, have an empty sibling set, then they must be equal:

**Lemma 3.6.11** (Model prefixes without siblings are equal). \(\forall I_1, I_2 \in N(\text{ModelGraph}(P)), \forall j \in [1, |\bar{H}|], \) let \(I'_1 = I_1 \cap \text{impls}(\bar{H}_{1:j})\) and \(I'_2 = I_2 \cap \text{impls}(\bar{H}_{1:j})\). If $\text{siblings}(I_1, I_2) = \emptyset$, then $I'_1 = I'_2$.

**Proof.** Let $i$ be the element of $I'_1 - I'_2$ so that $\text{par}(i) = \bar{H}_k$ with minimal $k$. Since $i \notin I'_2$ and $\text{siblings}(I'_1, I'_2) = \emptyset$, $\bar{H}_k \notin \text{pars}(I'_2)$. Since $\text{valid}_P(I_1)$ and $\bar{H}_k \in \text{pars}(I'_1)$, $\bar{H}_k \in \text{holes}(P_{\text{base}}) \cup \text{holes}(I')$. Since $\text{valid}_P(I_2)$, $\text{holes}(P_{\text{base}}) \subseteq \text{pars}(I'_2)$, so $\bar{H}_k \notin \text{holes}(P_{\text{base}})$, so $\exists i' \in I'_1$ s.t. $\bar{H}_j \in \text{holes}(i')$. By the topological sorting of $\bar{H}$, $\text{par}(i') = \bar{H}_m$ for some $m < k$. Since $k$ is minimal in $I'_1 - I'_2$, $i' \notin I'_1 - I'_2$, so $i' \in I'_2$. Then $\bar{H}_k \in \text{holes}(i') \subseteq \text{holes}(I'_2) \subseteq \text{pars}(I'_2)$, but $\bar{H}_k \notin \text{pars}(I'_2)$. Therefore $I'_1 - I'_2$ is empty. The same reasoning applies to show $I'_2 - I'_1$ is empty, so $I'_1 = I'_2$. \qed

At each expansion step, our set of edge prefixes includes the prefix of each edge in the final edge set:

**Lemma 3.6.12** ($E_j$ is a complete set of edge prefixes). \(\forall j \in [1, |\bar{H}|], \) \(\{ (I_1 \cap \text{impls}(\bar{H}_{1:j}), I_2 \cap \text{impls}(\bar{H}_{1:j})) | I_1, I_2 \in N(\text{ModelGraph}(P)), \text{siblings}(I_1, I_2) = \{(i_1, i_2)\}, \text{par}(i_1) \in \bar{H}_{1:j} \} \subseteq \{ (I_1, I_2) | (H_1, I_1, H_2, I_2) \in E_j \} \).

**Proof.** This is a proof by induction on $j$.

For $j = 0$:

\[
\{ (I_1 \cap \emptyset, I_2 \cap \emptyset) | I_1, I_2 \in N(\text{ModelGraph}(P)), \text{siblings}(I_1, I_2) = \{(i_1, i_2)\}, \text{par}(i_1) \in \emptyset \}
\]

\[
= \emptyset \subseteq E_0
\]

\[
= \emptyset
\]
For \( j \geq 1 \):

We consider some \((I_1, I_2) \in \{(I_1 \cap \text{impls}(\tilde{H}_{1:j}), I_2 \cap \text{impls}(\tilde{H}_{1:j}))\} | I_1, I_2 \in N(ModelGraph(P)), \text{siblings}(I_1, I_2) = \{(i_1, i_2)\}, \text{par}(i_1) \in \tilde{H}_{1:j}\) and show that \((I_1, I_2) \in \{(I_1, I_2) | (H_1, I_1, H_2, I_2) \in E_j\}\). Let \{(i_1, i_2)\} = \text{siblings}(I_1, I_2).

If \( \text{par}(i_1) = \text{par}(i_2) = \tilde{H}_j \), then since \( \text{siblings}(I_1 - i_1, I_2 - i_2) = \emptyset \), by Lemma 3.6.11, \( I_1 - i_1 = I_2 - i_2 \). Since \( I_1 - i_1 \subseteq \text{impls}(\tilde{H}_{1:j-1}) \), by Lemma 3.6.8, \( \exists H' \) s.t. \( (H', I_1 - i_1) \in N_{j-1} \), therefore \( \text{new-edges}(H', I_1 - i_1, \tilde{H}_j) \subseteq E_j \), and because \( \tilde{H}_j \in \text{pars}(I_1) \) implies \( \tilde{H}_j \in H' \) and \((i_1, i_2) \in \{(\text{impls}(\tilde{H}_j))\}, (H' \cup \text{holes}(i_1), I_1 - i_1 \cup i_1 = I_1, H' \cup \text{holes}(i_2), I_1 - i_1 \cup i_2 = I_2) \in \text{new-edges}(H', I_1 - i_1, \tilde{H}_j)\).

If \( \text{par}(i_1) = \text{par}(i_2) \neq \tilde{H}_j \), then by induction, \( (I_1 \cap \text{impls}(\tilde{H}_{1:j-1}), I_2 \cap \text{impls}(\tilde{H}_{1:j-1})) \in \{(I_1, I_2) | (H_1, I_1, H_2, I_2) \in E_{j-1}\}\), so \( \exists (H'_1, I'_1, H'_2, I'_2) \in E_{j-1} \) such that \( I'_1 = I_1 \cap \text{impls}(\tilde{H}_{1:j-1}) \) and \( I'_2 = I_2 \cap \text{impls}(\tilde{H}_{1:j-1})\).

If \( \tilde{H}_j \notin H'_1 \) and \( \tilde{H}_j \notin H'_2 \), then by Lemma 3.6.9 and Lemma 3.6.6, \( I_1 = I'_1, I_2 = I'_2 \), and by the definition of \( \text{expand-edge} \), \( (H'_1, I'_1, H'_2, I'_2) = (H'_1, I_1, H'_2, I_2) \in \text{expand-edge}(H'_1, I'_1, H'_2, I'_2, \tilde{H}_j) \subseteq E_j. \)

If \( \tilde{H}_j \in H'_1 \) and \( \tilde{H}_j \notin H'_2 \), then \( \exists i \in \text{impls}(\tilde{H}_j) \) such that \( I_1 = I'_1 \cup \{i\} \) and \( I_2 = I'_2 \), and \( \forall i' \in \text{impls}(\tilde{H}_j), s(H'_1 \cup \text{holes}(i'), I'_1 \cup \{i'\}, H'_2, I'_2, I_2) \in \text{expand-edge}(H'_1, I'_1, H'_2, I'_2, \tilde{H}_j) \subseteq E_j. \) Similarly, if \( \tilde{H}_j \notin H'_1 \) and \( \tilde{H}_j \in H'_2 \), then \( I_1 = I'_1, I_2 = I'_2 \cup \{i\} \) and \( s(H'_1, I'_1, H'_2 \cup \text{holes}(i), I'_2 \cup \{i\}) = (H'_1, I_1, H'_2 \cup \text{holes}(i), I_2) \in \text{expand-edge}(H'_1, I'_1, H'_2, I'_2, \tilde{H}_j) \subseteq E_j. \)

Finally, because each edge prefix set is complete, and each edge is valid, the final edge set must be both sound and complete:

**Theorem 3.6.5** \((E_{\tilde{H}})\) is the sound and complete edge set. \( \{(I_1, I_2) | (H_1, I_1, H_2, I_2) \in E_{\tilde{H}}\} = E(ModelGraph(P)). \)
Proof. \( \forall (H_1, I_1, H_2, I_2) \in E_{\tilde{H}} \), \( I_1, I_2 \in N_{\tilde{H}} \) by Lemma 3.6.9, so \( I_1, I_2 \in N(\text{ModelGraph}(P)) \) by Theorem 3.6.4, and also \( |\text{siblings}(I_1, I_2)| = 1 \) by Lemma 3.6.10. Therefore \( (I_1, I_2) \in E(\text{ModelGraph}(P)) \), so \( \{(I_1, I_2) \mid (H_1, I_1, H_2, I_2) \in E_{\tilde{H}} \} \subseteq E(\text{ModelGraph}(P)) \).

Because \( \tilde{H}_{1;\tilde{H}} = \text{pars}(P) \),

\[
\left\{ \begin{array}{ll}
(I_1 \land \text{impls}(\tilde{H}_{1;\tilde{H}}), I_2 \land \text{impls}(\tilde{H}_{1;\tilde{H}})) & I_1, I_2 \in N(\text{ModelGraph}(P)), \\
\text{siblings}(I_1, I_2) = \{(i_1, i_2)\}, & \text{par}(i_1) \in \tilde{H}_{1;\tilde{H}}
\end{array} \right.
\]

\[
= \{(I_1, I_2) \mid I_1, I_2 \in N(\text{ModelGraph}(P)), |\text{siblings}(I_1, I_2)| = 1 \}
\]

\[
= E(\text{ModelGraph}(P))
\]

So by Lemma 3.6.12, \( E(\text{ModelGraph}(P)) \subseteq \{(I_1, I_2) \mid (H_1, I_1, H_2, I_2) \in E_{\tilde{H}} \} \) \( \square \)

Efficiency

We can analyze the efficiency of this algorithm in terms of the number of nodes and edges in the final network of models.

For a modular program \( P \), let \( N = |N(\text{ModelGraph}(P))| \) and \( E = |E(\text{ModelGraph}(P))| \).

Index the graph nodes as \( \{n_i \mid i \in [1 \ldots N] \} = N(\text{ModelGraph}(P)) \) and the edges as \( \{(e_{i,1}, e_{i,2}) \mid i \in [1 \ldots E] \} = E(\text{ModelGraph}(P)) \). Let \( H = |\text{holes}(P)| \).

The minimum size possible of the explicit representation of \( \text{ModelGraph}(P) \) is then

\[
\sum_{i \in [1 \ldots N]} |n_i| + \sum_{j \in [1 \ldots E]} |e_{j,1}| + |e_{j,2}| \text{ identifiers.}
\]

1. Runtime efficiency. Consider the selection set insertion operations \( I \cup \{i\} \) in \text{expand-node}. Since implementations are never removed from any selection sets, and selection sets are never removed from model prefix sets, the number of such insertions must be \( O\left(\sum_{i \in [1 \ldots N]} |n_i| \right) \).

Consider the hole union operations \( H \cup \text{holes}(i) \) in \text{expand-node}. Since each \( h \in \)
holes\((i)\) is eventually replaced by exactly one implementation, the amortized cost is again \(O(\sum_{i=1...N} |n_i|)\).

The same arguments apply to the edge operations in \texttt{new-edges} and \texttt{expand-edge}: the number of implementation set insertions must be \(O(\sum_{j \in [1...E]} |e_{j,1}| + |e_{j,2}|)\), and the amortized cost of each hole union operation is also \(O(\sum_{j \in [1...E]} |e_{j,1}| + |e_{j,2}|)\).

The set inclusion checks \(h \in H\) result in a small constant overhead for \texttt{expand-node}, \texttt{new-edges} and \texttt{expand-edge}. Since \texttt{expand} is called \(H\) times with at most \(N\) nodes and \(E\) edges, the number of inclusion checks is \(O(H \times N + H \times E)\).

The resulting runtime complexity is then \(O(H \times N + H \times E)\).

2. **Space efficiency.** Because the nodes \(N_j\) and edges \(E_j\) are the entire program state, \(|N_j| \leq |N(\text{ModelGraph}(P))|\) and \(|E_j| \leq |E(\text{ModelGraph}(P))|\), and the elements of \(N_j\) and \(E_j\) are less than or equal to the elements of \(N(\text{ModelGraph}(P))\) and \(E(\text{ModelGraph}(P))\), the program state never exceeds the size of the output, so space complexity is \(\Theta(\sum_{i \in [1...N]} |n_i| + \sum_{j \in [1...E]} |e_{j,1}| + |e_{j,2}|)\).

3. **Notes on efficiency.** Since \(N_{\tilde{I}}\) does not depend on any \(E_j\), when we only calculate the graph nodes we have a runtime complexity of \(O(H \times N)\) and space complexity of \(O(\sum_{i \in [1...N]} |n_i|)\).

This method can efficiently handle the extreme cases in Figure 3.4. We have no asymptotic time or space overhead for the example Figure 3.4b, and for Figure 3.4a, we have no space overhead but an asymptotic time overhead of \(O(H \times H)\) set inclusion checks. This is a naive algorithm of Section 3.6.2.

3.6.3 ModelNeighbors

The \texttt{ModelNeighbors} operation enumerates the neighbors of a model within a network of models (shown in Figure 3.5). We defined it formally in Section 3.5.6 as: \(\text{ModelNeighbors}(P, I) = \{ I' \mid (I, I') \in E(\text{ModelGraph}(P)) \}\). In this section we will use \(\text{Nei}(I, P)\) as a shorthand for \(\text{ModelNeighbors}(P, I)\).
In this section we develop a function \texttt{ModelNeighbors} by feeding a modified input to the \texttt{ModelGraph} function from the previous section. We define soundness, completeness, and correctness with respect to this definition in an analogous way to \texttt{ModelGraph}.

Let \texttt{Limit}(I, P), for a valid modular program \( P \) and a (possibly partial) selection \( I \), be equal to \( P \), except that, for all holes \( h \in \text{pars}(P) \):

\[
\text{impls}_{\text{Limit}(I, P)}(h) = \begin{cases} 
\text{impls}_P(h) \cap I & \text{if } h \in \text{pars}(I) \\
\text{impls}_P(h) & \text{otherwise}
\end{cases}
\]

Intuitively, \texttt{Limit}(I, P) restricts \( P \) so that, for each implementation \( i \) in \( I \), \( i \) is the only option to fill \( i \)'s parent. If \( I \) were a valid selection, this would restrict \( P \) so that \( I \) would become the only valid selection. We will use \texttt{Limit} with selections that are just short of being valid, so that the valid selections become exactly the neighbors of a valid selection.

We can compute \texttt{Nei} efficiently in terms of \texttt{Limit} and our previously defined \texttt{ModelGraph} algorithm:
The total set of neighbors is exactly the union of the model sets of these restricted programs over all holes and all alternative implementations.

Proofs of soundness and completeness

Our overall strategy will be to show that each valid selection of $\text{Limit}(I - \{i\} \cup \{i'\}, P)$ is indeed a neighbor of $I$ in $P$ (soundness), and then that each neighbor of $I$ in $P$ has some sibling $i$ and $i'$ such that it is valid in $\text{Limit}(I - \{i\} \cup \{i'\}, P)$ (completeness).

First we confirm that our definition of $\text{ModelGraph}$ will always produce a valid program, since $\text{ModelGraph}$ should only be applied to valid programs.

**Lemma 3.6.13** (Limit$(I, P)$ is a valid modular program). $\text{valid}_{structure}(P)$, $\text{valid}_{P}(I)$ implies $\text{valid}_{structure}(\text{Limit}(P, I))$.

Proof. Requirements 1 and 3 are implied by $\text{valid}_{structure}(P)$ and $\text{impls}(\text{Limit}(P, I)) \subseteq \text{impls}(P)$.

Requirement 2 is $\forall h \in \text{holes}(P) \cup \text{holes}(P_{\text{base}}), \text{impls}_{\text{Limit}(P, I)}(h) \neq \emptyset$. For every $h$, if $h \notin \text{pars}(I)$, then $\text{impls}_{\text{Limit}(P, I)}(h) = \text{impls}_{P}(h)$, which must be non-empty given $\text{valid}_{structure}(P)$. If $h \in \text{pars}(I)$, then $\text{impls}_{\text{Limit}(P, I)}(h) = \text{impls}_{P}(h) \cap I$, which must have one element by the definition of $\text{valid}_{P}(I)$. □

**Lemma 3.6.14** (Neighbors of $I$ are valid under $\text{Limit}(I - \{i\} \cup \{i'\}, P)$). $\forall I' \in N(\text{ModelGraph}(P))$ such that $\text{siblings}(I, I') = \{(i, i')\}, \text{valid}_{\text{Limit}(I - \{i\} \cup \{i'\}, P)}(I')$

Proof.

1. $I' \subseteq \text{impls}(\text{Limit}(I - \{i\} \cup \{i'\}, P))$: Suppose $\exists i'_{\_} \in I'$ such that $i'_{\_} \notin \text{impls}(\text{Limit}(I - \{i\} \cup \{i'\}, P))$. It must be that $\text{par}(i'_{\_}) \in \text{pars}(I - \{i\} \cup \{i'\})$, because otherwise
impls_{\text{Limit}(I - \{i\} \cup \{i'\}, P)}(i'_{\_}) = \text{impls}_{P}(i'_{\_})$ by the definition of $\text{Limit}$. Then $\exists i_{\_} \in I - \{i\} \cup \{i'\}$ s.t. \text{par}(i_{\_}) = \text{par}(i'_{\_})$, so $(i_{\_}, i'_{\_}) \in \text{siblings}(I - \{i\} \cup \{i'\}, I')$ and $(i_{\_}, i'_{\_}) \neq (i, i')$.

Since $i' \in I'$, $i_{\_} \neq i'$, so $i_{\_} \in I'$. Then $(i'_{\_}, i_{\_}) \in \text{siblings}(I, I')$, which contradicts $\text{siblings}(I, I') = \{(i, i')\}$, so there is no such $i'_{\_}$.

2. $\text{pars}(I') = \text{holes}(I') \cup \text{holes}(\text{Limit}(I - \{i\} \cup \{i'\}, P))_{\text{base}}$ follows from $\text{valid}_{P}(I')$ and Lemma 3.6.13 because $\text{Limit}$ does not modify base, holes or pars.

3. $\text{siblings}(I', I') = \emptyset$ by $\text{valid}_{P}(I')$.

\[ \square \]

We show soundness:

\textbf{Lemma 3.6.15} (Selections that are valid under $\text{Limit}(I - \{i\} \cup \{i'\}, P)$ are neighbors of $I$). $\forall I' \in N(\text{ModelGraph}(\text{Limit}(I - \{i\} \cup \{i'\}, P)))$, $\text{valid}_{P}(I')$ and $\text{siblings}(I, I') = \{(i, i')\}$.

\textbf{Proof.} Because $\text{impls}(\text{Limit}(I - \{i\} \cup \{i'\}, P)) \subseteq \text{impls}(P)$ and Lemma 3.6.13, $\text{valid}_{\text{Limit}(I - \{i\} \cup \{i'\}, P)}(I')$ implies each requirement of $\text{valid}_{P}(I')$.

Suppose $\text{siblings}(I, I') = \emptyset$. Because $I, I' \in N(\text{ModelGraph}(P))$, by Lemma 3.6.11, $I = I'$. Then, $i \in I \implies i \in I'$. However, by $\text{valid}_{\text{Limit}(I - \{i\} \cup \{i'\}, P)}(I')$, $I' \subseteq \text{impls}(\text{Limit}(I - \{i\} \cup \{i'\}, P)) \neq i$, which is a contradiction. Therefore $\text{siblings}(I, I') \neq \emptyset$.

Let $(i_2, i'_2) \in \text{siblings}(I, I')$, so $\text{par}(i_2) = \text{par}(i'_2)$. If $\text{par}(i'_2) \neq \text{par}(i)$, then by the definition of $\text{Limit}$, $\text{impls}_{\text{Limit}(I - \{i\} \cup \{i'\}, P)}(\text{par}(i'_2)) = \{i_2\}$, so $i_2 = i'_2$, which contradicts $(i_2, i'_2) \in \text{siblings}(I, I')$. If $\text{par}(i'_2) = \text{par}(i)$, then $\text{impls}_{\text{Limit}(I - \{i\} \cup \{i'\}, P)}(\text{par}(i'_2)) = \{i'\}$, so $(i_2, i'_2)$ must be $(i, i')$. Therefore, $\text{siblings}(I, I') = \{(i, i')\}$.

\[ \square \]

We show the correctness of the union:

\textbf{Theorem 3.6.6} (The models of $\text{Limit}(I - \{i\} \cup \{i'\}, P)$ are exactly the neighbors of $I$ with $\text{siblings}(I, I') = \{i, i'\}$, so the union over possible siblings is correct).

\[ \text{Nei}(I, P) = \bigcup_{i \in I} \bigcup_{i' \in \text{impls}(\text{par}(i)) - \{i\}} N(\text{ModelGraph}(\text{Limit}(I - \{i\} \cup \{i'\}, P))) \]

116
Proof. By Lemmas 3.6.14 and 3.6.15, \( \{ I \mid I \in N(ModelGraph(P)), siblings(I, I') = \{(i, i')\} \} = N(ModelGraph(Limit(I - \{i\} \cup \{i'\}, P))) \).

\[
\bigcup_{i \in I} \bigcup_{i' \in impls(par(i)) - \{i\}} N(ModelGraph(Limit(I - \{i\} \cup \{i'\}, P)))
\]

\[
= \bigcup_{i \in I} \bigcup_{i' \in impls(par(i)) - \{i\}} \{ I \mid I \in N(ModelGraph(P)), siblings(I, I') = \{(i, i')\} \}
\]

\[
= \{ I \mid I \in N(ModelGraph(P)), |siblings(I, I')| = 1 \}
\]

\[= Nei(I, P)\]

\[\square\]

Efficiency

Since we are using the ModelGraph implementation from Section 3.6.2, the runtime complexity of \( N(ModelGraph(P)) \) is \( O(H * N) \) for \( H = holes(P) \) and \( N = |N(ModelGraph(P))| \).

The runtime complexity of Nei is then:
Our space complexity is the size of the output.

The important feature of this implementation is that the time and space complexity does not grow with the total size of the model graph, only on the number of neighbors. This lets us use $Nei$ even in very large model spaces.

### 3.7 Additional features

This section presents two extensions of the module system described in Sections 3.4 and 3.5, giving users more expressive power to build their desired networks of models.

#### 3.7.1 Append blocks

In the base syntax, we allow modules to define a parameters block that upon concretization is appended onto the base parameters block. In the same way, we can allow modules to add on to each of the other Stan blocks (except for data, which is fixed for the whole network of models). The extended syntax generalizes `MODULE_IMPLEMENTATION_M`:

```plaintext
MODULE_IMPLEMENTATION_M:

module "impl_identifier" hole_identifier((TYPE identifier,)*) {  
```

```plaintext
O(Nei(I, P)) = O\left(\bigcup_{i \in I} \bigcup_{i' \in implex(par(i)) \cup \{i\}} \text{N}(\text{ModelGraph}(\text{Limit}(I - \{i\} \cup \{i', P\}))\right)

= O\left(\sum_{i \in I} \sum_{i' \in implex(par(i)) \cup \{i\}} \text{N}(\text{ModelGraph}(\text{Limit}(I - \{i\} \cup \{i', P\}))\right)

= O\left(\sum_{i \in I} \sum_{i' \in implex(par(i)) \cup \{i\}} H|\text{N}(\text{ModelGraph}(\text{Limit}(I - \{i\} \cup \{i', P\}))|\right)

= O\left(\sum_{i \in I} \sum_{i' \in implex(par(i)) \cup \{i\}} |\text{N}(\text{ModelGraph}(\text{Limit}(I - \{i\} \cup \{i', P\}))|\right)

= O(H|\text{Nei}(I, P)|)
```
We refer to these module blocks as append blocks because they are appended to the end of their corresponding Stan blocks upon concretization.

The rules for scope and available effects are the same for within-module blocks as they are for the corresponding Stan blocks as described in Section 3.3.2. For example, variables defined in a module `transformed parameters` block can be referenced inside of that module’s `model` block but not its `transformed data` block. Module arguments are not available inside append blocks.

ApplyImpl is updated to include InlineFunction calls for each append block in the same way that it is used for parameters.

Concretization is still correct:

- Scope is preserved: For any statement in a module that could reference a global variable `g` in module append block `B`, `g` will still be a valid reference after concretization because `B` must also be included. For any statement `S` in a module append block that references a global variable `h` in its corresponding base block, `h` will still be a valid reference after concretization because `M` will be inserted after the declarations of the base block.

- Type correctness and effect correctness are preserved because appends are statements and must have the same effect requirements as the blocks into which they are inserted, as usual.
• All other arguments about statement order, well-typedness, and lack of cycles are unaffected by this change.

Since the inclusion of append blocks has no effect on the module dependency graph, this change does not change the correctness of the network of models or neighbor algorithms.

Use cases of append blocks include:

• Defining functions for use in the module and as arguments to its descendants.

• Defining transformed data or transformed parameters that are used in the module.

• Adding a prior distribution to the model block for each parameter defined in the module, without requiring that the module signature have the \texttt{LPDF} effect.

• Emitting the likelihood of the data under a distribution defined the module as a generated quantity (as in the case study in Section 3.9.3).

### 3.7.2 Module fields

Sometimes, two or more behaviors only make sense to include together.

For example, suppose we want to define a change of variables transformation. We need to define a \texttt{transform} function and corresponding \texttt{inverse} function. Suppose we want to abstract the transformation into a hole so that it can be swapped out. If we were to define separate holes for \texttt{Transform} and \texttt{Inverse}, we may end up with implementations that do not match, because they can be selected independently. We would rather have one hole, \texttt{Transformation}, that packages compatible \texttt{transform} and \texttt{inverse} implementations together.

We can achieve this coupling by letting modules contain multiple named behaviors called \texttt{fields}. With fields, we would define one hole, \texttt{Transformation}, with two fields, \texttt{forward} and \texttt{reverse}, referenced in code as \texttt{Transformation.forward} and \texttt{Transformation.reverse}.

Fields of a module also share the append blocks (and therefore global scope) of a module.
The syntax for associated behavior introduces a second, optional variant of

\texttt{MODULEIMPLEMENTATIONM} where arguments are moved into field declarations:

\begin{verbatim}
MODULEIMPLEMENTATIONM: module "impl_identifier" hole_identifier {
  FUNCTIONS_M?
  TRANSFORMED_DATA_M?
  TRANSFORMED_PARAMETERS_M?
  PARAMETERS?
  MODEL_M?
  GENERATED_QUANTITIES_M?

  field field_identifier?(TYPE identifier*) {
    STMT_LPDF_M;*
    return EXPR_M;?
  } +
}
\end{verbatim}

We use \texttt{impl_identifier} to stand in for valid field names, while \texttt{field} is a new keyword.

Each of a hole’s fields must be implemented by a corresponding field block and implementations as though it were a new hole. A field \( f_i \) in hole \( h \) is referenced as \( h.f_i(\ldots) \), except when the field identifier \( f_i \) is empty, in which case the reference is \( h(\ldots) \).

Each of a hole’s fields is treated like a separate hole for the purposes of concretization, and they have no differences in terms of the correctness of the resulting Stan program. Holes with fields are treated like holes without fields for the purpose of building the module graph, model graph and neighbor sets; an implementation has a dependency on a hole \( h \) if any of its fields depends on \( h \).

Coupling behavior into the fields of a module is only one constraint that we could impose on co-selection of implementations. We discuss a more general constraint logic in Section 3.10.
3.8 Macros

While the module system described so far is in theory flexible enough to describe arbitrarily complex networks of models, it may sometimes be too verbose to be practical.

For instance, suppose we want to build a regression model, but we do not know which of $N$ features to include. Our model space of interest then consists of one model for each subset of features. Since we want to choose inclusion or exclusion for each feature, we will need at least $N$ holes, each with two implementations, so we will need to write $2^N$ implementations. This is cumbersome and repetitive for large $N$.

To make the language more expressive, we define a series of “macros” as shorthand ways of generating larger modular programs. Each macro is translated by the compiler back into the basic module language, so we can then use the same algorithms defined in Section 3.6.

Table 3.1 is a summary of the macros described in the following sections.

<table>
<thead>
<tr>
<th>Name</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection hole</td>
<td>$\mathbb{H}^+$</td>
<td>Select a subset of implementations rather than one</td>
</tr>
<tr>
<td>Indexed hole</td>
<td>$\mathbb{H}[i..j]$</td>
<td>Copy implementations of $H$ for each index in range</td>
</tr>
<tr>
<td>Hole instance</td>
<td>$\mathbb{H}&lt;j&gt;$</td>
<td>Copy a hole; same selection but new parameters</td>
</tr>
<tr>
<td>Hole copy</td>
<td>$\mathbb{H}«j»$</td>
<td>Copy a hole; independent selection</td>
</tr>
<tr>
<td>Ranged versions</td>
<td>$\mathbb{H}&lt;i..j&gt;$</td>
<td>Apply the macro for each index in range, collecting an array of results</td>
</tr>
<tr>
<td>Multi-ranges</td>
<td>$i..j,k..l,..$</td>
<td>Same as a range, but for each index combination</td>
</tr>
<tr>
<td>Range exponent</td>
<td>$(i..j)^e$</td>
<td>Same as a range, but $e$ indices without replacement</td>
</tr>
<tr>
<td>Hole product</td>
<td>$\mathbb{H}_1\mathbb{H}_2(\ldots)(\ldots)$</td>
<td>New hole with implementations $impls(H_1) \times impls(H_2)$</td>
</tr>
<tr>
<td>Hole exponent</td>
<td>$\mathbb{H}^j$</td>
<td>Same as a hole product but without replacement</td>
</tr>
</tbody>
</table>

Table 3.1: Summary of macros

The expansion of each macro transforms the program’s module graph in some way, it may synthesize new modules with new (Stan) code, and it may define a translation of selection strings (Section 3.6.1) to apply to the post-expansion module graph.
3.8.1 Collection holes

A “collection hole” is a hole that can be filled with any number of implementations, rather than exactly one.

Consider our regression model example where we want to use a subset of $N$ features. For each feature, we need a hole with one implementation that includes the feature and another that does not. Then, we need to collect all of the holes into an array to pass into the regression. The “collection hole” macro automates this pattern, so that the user can instead write one “collection hole” with $N$ implementations instead of $N$ holes and $2N$ implementations.

Collection holes are identified by a + at the end of a hole identifier. The value of a non-\texttt{void} collection hole is an array containing the values of the selected implementations in an undefined order.

Figure 3.6 shows how expansion of collection holes modifies a program’s module graph.

For each implementation $i$ of $h+$, the implementation \texttt{yes} is the same as $i$ but wraps its result in a singleton array, while \texttt{no} does nothing and returns an empty list. The implementation \texttt{merge_h} returns a concatenation of the arrays returned by each new hole.

Users can then select a subset of implementations in their selection strings, as in 
\begin{verbatim}
..h:[i_1,i_2,...].. 
\end{verbatim}
If a collection hole $h$ has a set of implementations $C$, then a selection string $s_{before}, h:[I], s_{after}$ for some list of implementations $I \subseteq C$ is translated to $s_{before}, h:\texttt{merge_h}, \bigcup_{i \in I} h_{i:yes}, \bigcup_{i \in C-I} h_{i:no}, s_{after}$.
The network produced by a collection hole includes an edge between two nodes if they differ by exactly one inclusion. 

3.8.2 Indexed holes

An “indexed hole” is a hole that can generate additional implementations. Indexed holes are identified by adding a range \([i..j]\), where \(i\) and \(j\) non-negative integer literals, to the end of a hole identifier. Implementations of indexed holes accept an index as an extra argument, denoted in brackets at the end of the hole identifier in their definition, such as in module "i" h[j](..) { .. }. Here, \(j\) is the index, and can be used as an integer literal within the module, because it will be replaced by each integer in the range upon macro expansion. In this way, each written module implementation serves as a template for generating more. Figure 3.7 shows how expansion of indexed holes modifies a program’s module graph.

![Indexed hole module graph transformation](image)

Figure 3.7: Indexed hole module graph transformation. An indexed hole \(H[1..N]\) produces \(N\) copies of its implementation.

Users can then specify an indexed implementation to fill the indexed hole, as in ..h:i[5].. 

3.8.3 Hole instances and hole copies

So far, the design of holes effectively assumes that they represent decisions about individual subcomponents of a model. For example, when a parameter is defined within a hole, it only ever translated to a single parameter in the resulting program. It may be instead that a hole should be repeated in multiple places. 

---

5Like a Hamming graph.
For example, suppose a hole \( h \) represents a model of a storm cloud. What if we have data from two storm clouds? There are three possibilities:

1. We only want one copy of \( h \), and data from both clouds will be used to estimate the parameters in \( h \).

2. We want two copies of \( h \), one to model each cloud, with identical implementations but separate parameters.

3. We want two copies of \( h \), one to model each cloud, but which may have different implementations.

Case 1 corresponds to the basic semantics: users can call \( h() \) multiple times, but all references to \( h \) are filled with the same implementation, and the append blocks of \( h \)’s implementation are only added once, so all references use the same parameters.

Case 2 is the motivation for a macro called *hole instances*.

Case 3 is the motivation for a macro called *hole copies*.

**Hole instances**

Hole instances are identified by the syntax \( \text{hole_identifier}<j> \), where \( j \) is a non-negative integer literal. Hole instances are transformed in the module tree in the following way:

Figure 3.8 shows how expansion of hole instances modifies a program’s module graph.

```
Figure 3.8: Hole instance module graph transformation. When some code \( p_i \) refers to \( H<i> \), a new hole called \( H<i> \) is created with copies of the implementations of \( H \).
```

Here, \( h[1] \ldots h[N] \) are identical copies of \( h \) except that their local and global variables are given unique names. This way, there are \( N \) copies of the implementation of \( h \), each re-
ferring to its own set of new global variables such as parameters. Each \( j \) found in a reference 
\texttt{hole_identifier<j>} produces a new instance.

Users specify selections in the same way as if \( h \) were not copied: \( \ldots, h : i, \ldots \) A selection 
string \( s_{\text{before}}, h : i, s_{\text{after}} \) is translated into \( s_{\text{before}}, \bigcup_{j \in 1 \ldots N} h_{j : i}, s_{\text{after}} \).

Optionally, module implementations can accept an index as an extra argument, denoted in angle 
brackets at the end of the hole identifier in their definition, such as in \texttt{module "i" h<j>(..)} 
(\ldots). When that module implementation is used as a hole instance, the variable \( j \) can be used as an integer literal. This way, \( j \) can specify hole instances within the module, copying deeper into the module graph.

**Hole copies**

Hole copies are identified by the syntax \texttt{hole_identifier<j>}, where \( j \) is a non-negative integer literal. Hole copies behave the same way as hole instances, except that implementations must be selected independently for each generated hole \( h<j> \), as in:

\[ \ldots, h<i>:1, \ldots, h<i>:N, \ldots \]

**Ranged hole instances and copies**

To generate many hole instances or copies automatically, \( h[1:N](\ldots) \) is translated into a Stan array \{ \( h[1](\ldots) \), \( h[2](\ldots) \), \ldots, \( h[N](\ldots) \) \} and \( h<1:N>(\ldots) \) is translated into \{ \( h<1>(\ldots) \), \( h<2>(\ldots) \), \ldots, \( h<N>(\ldots) \) \}.

### 3.8.4 Multi-ranges and range exponents

Macros that can have ranges, namely \( H[i..j], H<i..j>, \) and \( H<i..j> \), can also accept multi-ranges. Multi-ranges are the same as ranges except that they produce one result per combi-
nation of their ranges. For example, a 1..3,1..5 produces (1,1), (1,2), \ldots (3,5). Implementations that accept indices as extra arguments, such as h<i,j>, must then accept h<i, j>.

Ranges exponents come in three variants. R\(^n\) is equivalent to a multi-range with \(R\) repeated \(n\) times, for example, \((1..3)^2\) is equivalent to 1..3,1..3. R\(^Pn\) is like R\(^n\) except that it gives ordered permutations without replacement, for example \((1..3)^P3\) does not include (1,1), (2,2), or (3,3). R\(^Cn\) is like R\(^Pn\) except that it gives unordered combinations without replacement, for example \((1..3)^C3\) does not include (2,1), (3,1), or (3,2).

Multi-ranges and range exponents make it easier to generate holes and implementations that represent combinations.

3.8.5 Hole products and hole exponents

A hole product \(H_1 \times H_2 (\ldots, \ldots)\) is a hole that combines the implementations of the holes \(H_1\) and \(H_2\). For each \(i_1 \in impls(H_1)\) and \(i_2 \in impls(H_2)\), \((i_1, i_2)\) is an implementation of \(H_1 \times H_2\) that returns a tuple of the results of \(i_1\) and \(i_2\) The arguments lists of \(H_1\) and \(H_2\) are concatenated together to make the argument list of \(H_1 \times H_2\).

Figure 3.9 shows how expansion of hole products modifies a program’s module graph.

![Figure 3.9: Hole product expansion shown as a module graph transformation. When some code \(p\) refers to a product \(H_1 \times H_2\), a new hole is created with the cartesian product of \(H_1\) and \(H_2\)’s implementations.](image)

Hole exponents are analogous to range exponents from Section 3.8.4. H\(^n\) is equivalent to the product of \(H\) with itself \(n\) times. H\(^Pn\) gives the ordered \(n\)-permutations of impls(\(H\)) without replacement, while H\(^Cn\) gives the unordered \(n\)-combinations of impls(\(H\)) without replacement.
3.8.6 Example application of macros

Recall the regression example in which we want to include some subset of \( N \) variables in our model. Suppose \( N = 100 \). To encode this model space in the module system requires \( 2N = 200 \) module implementations. Using an indexed collection hole, we can reduce this to one handwritten module implementation.

data {
    int N;
    matrix[100, N] x;
    vector[N] y;
}
parameters {
    real sigma;
}
model {
    y ~ normal(sum(Feature[1..100]+(x)), sigma);
}

module "f" Feature[n](x) {
    parameters {
        real theta;
    }
    return theta*x[n,:];
}

This program represents a family of regression models on \( y \) given the features \( x \), each including a different subset of features. \( y \) is modeled with a normal distribution centered on the sum of the subset of features, \( \text{Feature}[1..100]+(x) \), which is an indexed collection hole: the range...
copies the Feature module implementation for n=1 to 100, and the + indicates that each implementation is either included or excluded from the result. An individual Stan program can be generated from this family by supplying a selection string, such as: Feature: [1, 2, 3], which includes only the first three features.

Now suppose we also want our regression to include some subset of 2- and 3-way interactions between variables, such as $x[3] \times x[9]$ or $x[4] \times x[10] \times x[99]$. To encode this model space requires $2 \times (100 + \binom{100}{2} + \binom{100}{3}) = 333500$ module implementations. We can use macros to reduce this to two or three handwritten module implementations. We will show two alternative implementations.

The first way makes use of range exponents:

```stan
data {
  int N;
  matrix[100, N] x;
  vector[N] y;
}
parameters {
  real sigma;
}
model {
  y ~ normal(sum( Feature[1..100]+(x) )
                + sum( FeaturePair[(1..100)^C2]+(x) ),
                + sum( FeatureTriplet[(1..100)^C3]+(x) ),
                sigma);
}

module "f" Feature[n](x) {
  parameters {
```
This program uses indexed collection holes with range exponentials to collect and sum a subset of the features, feature pairs, and feature triplets. An individual Stan program can be generated by supplying a selection string such as: `Feature:[1,2,3], FeaturePair:[(1,2),(1,4)], FeatureTriplet:[(1,2,3),(4,10,99)]`.

The second way makes use of hole products and exponents:

data {
  int N;
  matrix[100, N] x;
  vector[N] y;
}
parameters {
    real sigma;
}

model {
    vector[N] total = rep_vector(0, 100);
    for ((t, r) in Theta*Col[1..100]+()) {
        total += t * r;
    }
    for ((t, r1, r2) in Theta*Col[1..100]^C2+()) {
        total += t * r1 .* r2;
    }
    for ((t, r1, r2, r3) in Theta*Col[1..100]^C3+()) {
        total += t * r1 .* r2 .* r3;
    }
    y ~ normal(total, sigma);
}

module "t" Theta() {
    parameters {
        real theta;
    }
    return theta;
}

module "r" Col[n]() {

return x[n];
}

By taking a product $\Theta \times \text{Col}[1..100]$, we are generating an implementation of $\text{Col}$ for each index 1 to 100, and then producing a new parameter for each of those implementations. By taking a product of $\Theta$ with the exponent $\text{Col}[1..100]^2$, we are producing a new parameter for each pair of indices. Each of these products returns an array of tuples of values which must then be multiplied and summed into the total vector. The selection strings for this program look like: $\Theta \times \text{Col}: [(t,1),(t,2),(t,3)], \Theta \times \text{Col}^2: [(t,1,2),(t,1,4)], \Theta \times \text{Col}^3: [(t,1,2,3),(t,4,10,99)]$.

These programs have $\approx 333500$ module implementations and represent networks with $2^{100+{100 \choose 2}+{100 \choose 3}} \approx 10^{50000}$ models. How can we use such large programs and spaces? By never explicitly representing them. Macros instantiate synthetic modules lazily, only when they are selected. Large networks can be explored efficiently by only enumerating neighbors with ModelNeighbors; in this case, the network diameter and branching factor are a more manageable 166750.

### 3.9 Example applications

In this section we present three small but real-world probabilistic modeling case studies that we have translated into the Modular Stan language, and we discuss their benefits. These case studies showcase only motivating use cases 1 (automation), 3 (tracking), and 4 (researcher degrees of freedom) listed in Section 3.1; the rest we for future work.

In addition, we present a web interface that can be used to follow along with our two examples.

#### 3.9.1 Interactive web interface

We have built a prototype web interface for development and interactive visualizations of Modular Stan. Figure 3.10 shows its interface.
Users can write a Modular Stan program or load an example program and compile it at (a). When they do, interactive visualizations are produced: (b) the module graph and (c) the model graph.

The page keeps track of a module selection set. Users can modify the selection by: selecting or deselecting implementations of the module graph (b), selecting complete models in the model graph (c), editing the selection string directly (d), or selecting a previously labeled model (d). When the selection is modified, the model graph (c) highlights nodes compatible with that selection, and when the selection is valid, the corresponding concrete Stan program is displayed (e) and labels and notes associated with that program can be edited (f). The set of model labels and notes can be saved and loaded as a text file.

Users can also bookmark and annotate nodes in the model graph. Annotations can be saved and loaded as files separate from the Modular Stan file, in a format that maps between the model’s unique selection set and model labels and annotations.

Interactive versions of the following two case studies can be found online:\(^6\).

\(^6\)Golf case study: http://ryanbe.me/modular-stan.html?example=golf; birthday case study:
3.9.2 “Golf” case study: Modular Stan for ease and clarity of development

This section gives a basic demonstration of how Modular Stan can cleanly support and express a typical model development workflow, as an example of application 3.

The “Golf” case study [27] follows the development of a Bayesian statistical model for describing the probability that a professional golfer will sink a shot given their distance from the hole.

We represent the modeling process as a single Modular Stan program. The base of the program is the part that remains constant throughout development:

data {
  int J;     // Number of distances
  vector[J] x; // Distances
  int n[J];   // Number of shots at each distance
  int y[J];   // Number of successful shots at each distance
}

tmodel {
  y ~ NSuccesses(n, PSuccess(x));
}

The data block describes J distances, where the jth distance is x[j] feet, and y[j] shots out of n[j] were successful.

The model block describes an abstracted modeling approach: we model the number of successes y as being drawn from some distribution NSuccesses parameterized by the number of attempts n and the probability of success PSuccess, which itself is a function of the distance x. NSuccesses and PSuccess are holes.

A natural distribution to choose for NSuccesses is the binomial distribution. We express this as a module:

http://ryanbe.me/modular-stan.html?example=birthday
module "binomial" NSuccesses(y | n, p) {
    y ~ binomial(n, p);
}

A simple way to take a real value like x to a probability is the logit function, so we choose to explore that option first for PSuccess:

module "logistic" PSuccess(x) {
    parameters {
        real a;
        real b;
    }
    return logit(a + b*x);
}

If we stop here, we have only one choice of implementation for each of our holes, so our Modular Stan program defines only one valid Stan program:

data {
    int J; // Number of distances
    vector[J] x; // Distances
    int n[J]; // Number of shots at each distance
    int y[J]; // Number of successful shots at each distance
}
parameters {
    real a;
    real b;
}
model {
    y ~ binomial(n, logit(a + b*x));
This program implements logistic regression, which is the first model explored in the case study.

The next step in the probabilistic workflow or Box’s loop is to criticize our model by applying it to the data. We find that the model fit is lacking and the parameters $a$ and $b$ have no obvious physical interpretation.

This criticism motivates us to try a more sophisticated, more physically realistic implementation for $P_{\text{Success}}$. If we suppose that a shot will be successful if its trajectory angle is sufficiently precise, and we suppose that the angle is normally distributed, then we can write our model in terms of angle variance:

```stan
// A shot’s angle is good if the center of the ball would roll
// over the hole.
module "angle_success" PSuccess(x) {
  parameters {
    real sigma_angle;
  }
  real r = (1.68 / 2) / 12; // ball radius
  real R = (4.25 / 2) / 12; // hole radius
  vector[J] threshold_angle = asin((R-r) ./ x);
  vector[J] p_angle = 2*Phi(threshold_angle / sigma_angle) - 1;
  return p_angle;
}
```

We find this model has superior fit and interpretability and we continue iterating by adding on modules in this fashion.

The completed representation of the case study can be found with our source code\textsuperscript{7} or at the web interface along with visualizations\textsuperscript{8}. Here it is presented by the web interface:

\textsuperscript{7}https://github.com/rybern/mstan/blob/master/examples/golf.m.stan
\textsuperscript{8}http://ryanbe.me/modular-stan.html?example=golf
Figure 3.11: Two views of the golf case study Modular Stan program. The models mentioned by the case study are labeled on the model graph in the bottom left of each panel. (a) shows a partial selection, $\text{NSuccesses:binomial}$, and the model graph highlights the consistent subgraph. (b) shows a complete selection, which produces the concrete Stan program and its associated notes.

Though this is a small example, we already see some benefits to clarity:

- There is only a single, minimal source file;
- We have a standard, integrated way to draw the development path and document the decision-making evidence and rationale at each step;
- The modular organization makes the solution space easier to understand and extend.

3.9.3 “Birthday” case study: Modular Stan as a platform for automation

This section gives an example of how the network of models provides a platform for automation (use case 1). We demonstrate constructing a network, defining an evaluation metric, and performing a simple graph search. We use a moderately sized network of models from a case study of incremental model improvements. This approach would also apply to other multiple-model contexts, such as feature selection and symbolic regression.

The “Birthday” case study [28, 32] follows the development of a statistical model of the number of babies born in the US on a given day, given birth data from 1969-1988. The authors used a time-
series Gaussian process approach. A visualization of the time-series analysis appears on the cover of *Bayesian Data Analysis* by Gelman et al., shown in Figure 3.12.

![Figure 3.12: Taken from [32]. A visualization of time-series trends for the historical birth dataset, based on a Gaussian process model. The vertical axes are scaled so that values below 100 indicate a negative relationship with birth rate, while higher values indicate a positive relationship.](image)

Like many probabilistic modeling case studies and publications, the Birthday case study presents a series of models that differ by incremental variations. The authors explicitly evaluate nine models, but the model variations they present implicitly define a much larger set of models that it would be reasonable to explore: what if a different combination of variations were applied, or in a different way? To feasibly explore that larger set of models, we need automation, and for automation, we need an explicit representation of the model space, like the network of models offered by a Modular Stan program.

We start by translating the case study into a single Modular Stan program. The translation reduces the number of lines of code from 1098 to 270 while increasing the number of models
represented from nine to 120.

The translation process is largely mechanical, and involves encapsulating the variation between the given models into modules. For example, model 2 adds a days-of-the-week Gaussian process component onto model 1, model 3 adds a long-term-trend component onto model 2, etc. Each of these variations becomes one or more modules in the Modular Stan program. The full translation of the modular program can be found with our source code\textsuperscript{9} or at the web interface along with visualizations\textsuperscript{3, 10}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{model_graph.png}
\caption{The model graph corresponding to the Birthday problem Modular Stan program. Each node represents a model and each edge represents one swapped-out module.}
\end{figure}

Figure 3.13 shows the model graph produced by the modular program.

To automatically search these 120 models for high-quality options, we must first define “quality” by choosing a model-scoring metric. One reasonable approach is to measure a model’s pre-

\textsuperscript{9}\url{https://github.com/rybern/mstan/blob/master/examples/birthday/birthday.m.stan}

\textsuperscript{10}Our Modular Stan program would have been significantly more concise if we used the collection hole feature described in Section 3.8.1, but they were not implemented in our prototype compiler at the time of writing.
dictive accuracy by computing its Expected Log-Posterior Density (ELPD). ELPD approximates the leave-one-out prediction accuracy of a model for a dataset [29].

While ELPD is relatively efficient, it could still take a long time to accurately compute ELPD for every model in our network. Our goal, then, is to find high-quality models with as few ELPD evaluations as possible. Here we take the simplest approach, a greedy graph search, and leave more sophisticated search methods for future work.

Our greedy graph search loops over the following steps, given an arbitrary starting point:

1. Score the neighbors of the current model.

2. Move to the highest scoring model seen so far, or if that is the current model, return it.

This search algorithm is not guaranteed to be optimal; it is analogous to a gradient descent of a (likely non-convex) space.

The search algorithm is implemented as a short Python script that uses the prototype compiler’s implementation of the ModelNeighbors algorithm described in Section 3.6.3. Its source code is available online².

Starting from the case study’s first model, the greedy graph search followed the path shown in Figure 3.14:

The search performed 47 ELPD evaluations. The search agreed with the case study authors’ final model, confirming that it has (at least locally) optimal predictive performance.

While greedy maximization of ELPD is a naive statistical workflow and shouldn’t be blindly trusted to give a final model, it is at least useful for finding promising neighborhoods, especially for large model spaces.
Figure 3.14: The red annotations show the ELPD scores for the assessed models. The search algorithm visited nodes along the red arrow path from starting at [START] and terminating at [GOAL].
In this section we discuss using Modular Stan for sensitivity analysis and multiverse analysis, concluding with a small demonstration.

We briefly introduced sensitivity analysis as use case 4 in Section 3.1. Multiverse methods are similar to sensitivity analysis, and seek to understand the implications of alternative modeling decisions in a “multiverse” of models or data pipelines by evaluating many or all of those models or data pipelines.

To show the merits of Modular Stan for sensitivity and multiverse analysis, we will compare it to existing methods: there have recently been a few such attempts to design software tools to streamline multiverse methods for statistical analyses. Although these other methods are fundamentally different from Modular Stan in design and execution, they overlap in the goals of enumerating alternative models and evaluating them for sensitivity to modeling decisions. In the following sections, we briefly introduce these recent approaches, and discuss how they contrast with Modular Stan for specifying alternatives and doing multiverse analysis. Next, we give a case study to show how sensitivity analysis can utilize Modular Stan’s network-of-models abstraction, which the alternative multiverse approaches lack.

**Multiverse analysis with Modular Stan and other multiverse tools.**

Out of the recent attempts to support multiverse analysis in software, two representative examples are **boba** [92] and **multiverse** [87].

**boba** is a tool that lets users write a program template, labeled program snippets, and a constraint file, and can then generate all of the programs that can be produced by inserting the snippets into the template according to the constraints. The underlying language of the template and the snippets is a general-purpose scripting language like Python. The snippets are inserted into the template as lines of text.

Similarly, **multiverse** is an R package that lets a user specify a “multiverse” of R data pipelines (which can include data pre-processing, modeling, inference and evaluation) by adding
alternative implementations of expressions to a multiverse object. The user can then generate the results of running every valid data pipeline.

boba and multiverse are alike in goals, structure and design: they each let the user swap out components of a program, specify constraints on which components can co-exist, and then generate all of the valid combinations for analysis. boba and multiverse both focus on user experience and producing visualizations of whole-multiverse analyses. We will refer to these and similar tools as multiverse tools.

Modular Stan can also be used this way. However, there are fundamental differences in how Modular Stan specifies and generates “multiverses” of models:

• Modular Stan is an abstraction for probabilistic programming. On the one hand, the multiverse tools would not work on probabilistic programming languages such as Stan; on the other hand, while some multiverse tools include alternative methods of data pre-processing, inference, and evaluation in their “multiverse” abstraction, Modular Stan only allows alternative models in the form of Stan code, which is somewhat more limited.

• While the multiverse methods provide textual swapping of statements or expressions as their units of abstraction, Modular Stan provides modules, which have more expressive power.

• The multiverse tools will not scale well for certain model families; see the issues raised in Section 3.6.2.

• The multiverse tools are designed to generate all valid models, where Modular Stan is designed to provide a network-of-models abstraction (which also includes efficiently generating all models). The multiverse tools cannot efficiently implement other network operations. This is because the multiverse tools specify component constraints with a set of user-written “structural constraints”, rather than with Modular Stan’s module graph abstraction. The former is more general, but requires more manual specification and can be less efficient. We could also add “structural constraints” into Modular Stan to get the best of both worlds.
Despite these differences, Modular Stan can specify and evaluate “multiverses” of models in the same way as these existing multiverse tools: Modular Stan specify, (lazily) generate, and evaluate a “multiverse” of models, which could then be used to generate statistics and visualizations that show the relationships between component inclusions and modeling outcomes.

As an example of how the results of that automated multiverse analysis might look, Figure 3.15 shows an example visualization produced from the output of the multiverse package. The figure visualizes the relationship between a regression coefficient and p-value across a “multiverse” of 210 alternative models.

Figure 3.15: This figure is taken from the “visualising-multiverse” vignette of [87], and is an example of a specification curve, first proposed by Simonsohn et al. The first represents a multiverse analysis of a case study that examined the effects on a woman’s fertility due to various other variables, like political leaning and voting habits. The lower panels represent the various alternative options each model can take, for example, there were three alternative ways to include relationship status in each model. The dots on the horizontal axis show each model. The top panel arranges the model-dots vertically according to the strength of interaction they predict between fertility and relationship status.
In *A Survey of Tasks and Visualizations in Multiverse Analysis Reports*, Hall et al. provide a wide variety of visualizations that can be generated from multiverse analyses. Figure 3.16 shows a selection of example visualizations taken from their paper.

Figure 3.16: This figure is taken from [93], and demonstrates the wide variety of possible visualizations for multiverse analyses. Subfigure e shows a specification curve of the same kind shown in Figure 3.15.

We could set up each of these multiverse analyses and visualizations to be automatically generated from a Modular Stan program.

*“Local” sensitivity analysis with networks of models.*

The multiverse analysis tools and methods in the previous section evaluate all models in the “multiverse” of reasonable alternative models (or modeling pipelines) to understand the effects of modeling decisions. We will refer to this as a *global* analysis.

One potential issue with global analysis is that it scales very poorly with large model spaces or computationally intensive analyses.

Another potential issue with global analysis is that it does not necessarily focus on the relevant part(s) of the model space. Consider a case where a model developer has evaluated many options for each modeling decision, but has finally arrived at a model they think best represents reality. The model developer would like to evaluate the sensitivity of their model’s results to each component included in the final model. A global analysis will consider every combination of every alternative of every decision the developer considered, regardless of whether those models are remotely similar to the model under question.
For these reasons, it will sometimes be more meaningful or feasible to consider the results’ sensitivity to model decisions only among similar models. We will refer to this as local sensitivity analysis. Global analyses answer the question: “How does this decision influence the results, across all reasonable models for the data?” Local analyses answer the question: “How does this decision influence the results of this model, assuming all other decisions are fixed?” The latter question is both easier to answer and may be more relevant when a particular model is being evaluated, such as when auditing a published model.

Local analysis fits naturally into the network of models abstraction, because edges in the network approximate one changed modeling decision. So, for a simple local analysis of a modeling decision, we can compare a model to each of its neighbors that has an alternative decision.

By comparing a modeling result against each of a model’s neighbors, we also get a sense of how “unusual” that model is among its close peers with respect to that particular result.

A generalization of this neighborhood sensitivity analysis is to compare against all models within \( n \) steps in the network. For example, an \( n \) of 2 represents the sensitivity of the results of each decision or pair of decisions. As \( n \) increases, the sphere of models considered grows, and the analysis more closely approximates global multiverse analysis.

Modular Stan can retrieve the neighborhoods and spheres centered at a given model in linear time, regardless of the complexity of the model space (see Section 3.6.3), making it an efficient platform for local sensitivity analysis.

**Sensitivity analysis with the “Birthday” case study.**

We give two (contrived) scenarios to demonstrate this “local” sensitivity analysis. In each case, we use the data and model space from the “Birthday” case study from Section 3.9.3.

**Scenario 1: Is Valentine’s Day extra prolific?**

Suppose that, while we are exploring the implications of the model we discovered in Section 3.9.3, we notice that Valentine’s Day seems to be an especially popular day to be born. We should be skeptical of this claim, but how should we test it?
One way to test it would be a local sensitivity analysis, to answer the question: “Is one of my modeling decisions causing this effect to appear in the results?”

To accomplish this with Modular Stan, we first write an evaluation script that accepts a Stan model and produces the degree to which Valentine’s Day appears extra popular under that model. This is straightforward in our case: the script executes the Stan model with the “Birthday” dataset, and then outputs the mean estimate of the parameter of the model that represents the extra popularity of Valentine’s Day. The popularity of a particular day of the year is represented by the \texttt{beta_f4} parameter array, which is introduced in the \texttt{DayOfYearTrend} module. The evaluation script transforms the appropriate element of \texttt{beta_f4} into a percentage scale, 100 meaning that the day has no effect that is not explained by other trends. If a particular module does not include the \texttt{DayOfYearTrend} module, it is ignored.

The evaluation script is then handed to Modular Stan’s sensitivity analysis tool, along with the “Birthday” Modular Stan program, and the identifier of model of interest or “center” model. The sensitivity analysis script generates and evaluates the center model and compares it against each of its neighbors.

In this case, the central model (the model we found in Section 3.9.3) has a mean posterior for Valentine’s Day of 103.268. This can be interpreted approximately as: even accounting for all other trends that the model considers, babies are still slightly more likely to be born on Valentine’s Day. But is this conclusion robust? We compare the result to each of its neighbors (pictured in Figure 3.17); the results of the sensitivity analysis are shown in Table 3.2.

These results can be (tentatively) interpreted as: Valentine’s Day has more births than expected, but only if you take into account days of the week and long-term (multi-year) trends. We should now consider whether this story is reasonable. It does sound plausible that, for example, Valentine’s Day does not have an overall higher birth rate, but that it overcomes any slumps in birth rate due to the day of the week. To be more confident in the conclusion, we could perform more analyses to check if that is a reasonable explanation.
Figure 3.17: We compare the Valentine’s Day score of our candidate, “center” model (shown in black) with that of each neighboring model (shown in red).

<table>
<thead>
<tr>
<th>Change of module</th>
<th>Change of estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>DayOfWeekTrend: no → yes</td>
<td>+03.268</td>
</tr>
<tr>
<td>DayOfWeekWeights: uniform → weighted</td>
<td>+01.231</td>
</tr>
<tr>
<td>DayOfYearHierarchicalVariance: no → yes</td>
<td>-00.092</td>
</tr>
<tr>
<td>DayOfYearNormalVariance: no → yes</td>
<td>-00.273</td>
</tr>
<tr>
<td>HolidayTrend: no → yes</td>
<td>+00.814</td>
</tr>
<tr>
<td>LongTermTrend: no → yes</td>
<td>+03.265</td>
</tr>
<tr>
<td>SeasonalTrend: no → yes</td>
<td>-00.031</td>
</tr>
</tbody>
</table>

Table 3.2: The change of the mean estimated effect of Valentine’s Day on birth rates, as each of the center model’s modules is changed. The left column shows the hole and module changed, and the right column shows the effect on the estimate. For example, the first row shows that the effect of changing the module that fills the DayOfWeekTrend hole from no to yes is to increase the estimate by 03.268.
Scenario 2: What about November 1st?

Now suppose we had arrived at a different model, which does not use the DayOfYearHierarchicalVariance:yes or DayOfYearNormalVariance:yes modules. This model suggests that a different day is extra prolific: November 1st. When we run the same analysis as before, we find that our central model produces a mean posterior estimate of 103.213 for the effect of November 1st, and its neighbors (pictured in Figure 3.18) produce the sensitivity results shown in Table 3.3.

![Figure 3.18: We compare the November 1st score of our new center model (shown in black) with that of each neighboring model (shown in red).](image)

<table>
<thead>
<tr>
<th>Change of module</th>
<th>Change of estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>DayOfWeekTrend: no → yes</td>
<td>+02.988</td>
</tr>
<tr>
<td>DayOfWeekWeights: uniform → weighted</td>
<td>+03.723</td>
</tr>
<tr>
<td>HolidayTrend: no → yes</td>
<td>+03.563</td>
</tr>
<tr>
<td>SeasonalTrend: no → yes</td>
<td>+05.498</td>
</tr>
<tr>
<td>LongTermTrend: yes → no</td>
<td>+00.367</td>
</tr>
<tr>
<td>DayOfYearNormalVariance: yes → no</td>
<td>+04.287</td>
</tr>
<tr>
<td>DayOfYearHierarchicalVariance: yes → no</td>
<td>+03.220</td>
</tr>
</tbody>
</table>

Table 3.3: The change of the mean estimated effect of November 1st birth rates, as each of the center model’s modules is changed.

We can (tentatively) read these results as: November 1st has a more births than expected, but only if we consider a day of the week trend, and we give the day-of-week effects a non-uniform
weighted prior, and we consider holiday trends, and seasonal trends, and we do not use a normal or hierarchical model for the day-of-the-year trend. This sensitivity analysis suggests that the result is highly sensitive to many of the modeling decisions made, in ways that do not seem like they should be relevant to the conclusion drawn. And in fact, this model and day were cherry-picked.

These two analyses were kept very simple. A next step to improve these analyses would be to include the distribution of result posterior draws, in this case the distribution of $\text{beta}_f^4$ posterior draws, rather than only their mean. The analyses could also consider an $n$-sphere around the model of interest, representing $n$ module changes, instead of only the immediate neighborhood.

### 3.10 Future work

As discussed in Section 3.1.1, there are many motivating use cases for the network of models. We hope to demonstrate and build tooling for more of these use cases, using Modular Stan as a foundation:

- **Multi-model ensemble methods.** As mentioned in Section 3.1.1, multi-model ensemble methods like stacking and multiverse analysis could be applied to multi-model programs. Bayesian stacking methods, like those discussed by Yao et al., are especially promising.

- **More advanced model search.** Section 3.9.3 gave a simple algorithm for greedy search that only used the network topology. We hypothesize that more efficient search methods could leverage the module structure, treat search as an exploration/exploitation problem, or utilize methods from the symbolic regression literature.

- **Model space navigation tools.** Model developers must decipher promising directions in which to iterate. If we could annotate the edges between models with meaningful joint metrics, we could provide developers with direction. We could also use model statistics or edge weights to embed network-space into visualizations.

In addition, we hope to expand the capabilities of our swappable module system.
• Explicit model signatures. Though we believe that our module signature inference scheme is more beginner-friendly, optional explicit signatures may make module reuse across applications easier by giving a name to the interface to be provided by libraries and consumed by programs.

• Implementation co-selection logic. The structure of module graphs allow users to encode modules that are mutually exclusive: “include either module A or module B”. The module fields feature allows users to encode the constraint that a set of implementations should always be selected together: “if A then B and if B then A”. There are other constraints a user might want, such as “if A and B, then not C”. These selection sets could get arbitrarily complex, so to be fully flexible, we could augment our swappable module system with a predicate logic of implementation selection.
Chapter 4: Automatic Transformations of Probabilistic Programs for Model Checking

4.1 Introduction

Representing our probabilistic models as programs affords us the opportunity to apply code analysis and transformation techniques from decades of computer science research. In much the same way that software engineers have tools to check or generate code, data scientists could have tools to verify or construct their models. This intersection of static analysis and probabilistic programming is a largely unexplored area that could make data science simultaneously more reliable, efficient and approachable.

In this section we apply code analysis and transformation techniques to automate large parts of two model checking techniques: Prior Predictive Checking (PPC) [43] and Simulation-based Calibration (SBC) [79]. PPC addresses the question, “Does the model as we have written it match our actual domain knowledge?”; SBC addresses the question “Can our inference algorithm fit our specified model reliably?”. Together, these checks make up a vital part of a robust Bayesian workflow [45], and they are used ubiquitously within the Bayesian probabilistic modeling community.

PPC and SBC both require random draws from the prior and prior-predictive distributions. The prior is the distribution over model parameters prior to the data being considered, while the prior-predictive is the distribution over the data implied by the prior.

Currently, to apply PPC or SBC, users manually write sampling code which must correspond exactly to the prior and prior-predictive distributions of their particular model. At best, this process is time consuming, requires expertise, and duplicates the information and maintenance cost of the original program. At worst, the produced prior-predictive distribution is wrong, causing PPC and SBC to produce misleading results, which defeats the purpose of these quality assurance methods.
To mitigate this issue, we present a method to transform a probabilistic program into one which efficiently draws samples from the prior and prior-predictive distributions of the model specified by the original program. Our process either produces a guaranteed-correct and efficient program or, in rare cases, rejects the task as impossible, all with minimal user involvement. We focus on the Stan language for our implementation and primary reference point, as Stan is one of the most widely used probabilistic programming languages, as measured by package downloads and citations [42].

Central in our approach is a novel translation from a factor graph – a probabilistic graphical model giving a convenient intermediate representation of a probabilistic program – into a directed acyclic graphical (DAG) model, which admits efficient forward sampling.

This chapter makes the following core contributions:

1. It presents a technique for automatically computing all possible DAGs which represent the same conditional independence information as a given factor graph, together with a proof of its soundness and completeness.

2. It shows how to use this technique to derive efficient forward sampling code from a probabilistic program presented in a relatively unstructured representation, whenever possible.

3. It presents an implementation targeting the Stan probabilistic programming language, automating large parts of a robust Bayesian workflow for a wide community of probabilistic programming users.

The source code of our implementation is available online\(^1\). This chapter is based on previous work done by the author with help from Matthijs Vákár and Jeannette Wing[98].

4.2 Background

4.2.1 Probabilistic programs

Probabilistic models represent a joint probability distribution over a set of unobserved parameters and observed data. One representation of a probabilistic model is as a sequence of parameters

\(^1\)https://github.com/rybern/factor-graph-to-dag
drawn from individual distributions. For example:

\[
\begin{align*}
\mu & \sim \text{Normal}(0, 1) \\
\sigma & \sim \text{LogNormal}(1, 3) \\
x & \sim \text{Normal}(\mu, \sigma)
\end{align*}
\]

This represents a probabilistic model in which \(\mu\), \(\sigma\) and \(x\) are drawn from distributions. We can efficiently draw samples of \(\mu\), \(\sigma\) and \(x\) with the following process:

1. draw from a \textit{Normal}(0, 1) distribution; bind the result to the variable \(\mu\).
2. draw from \textit{LogNormal}(1, 3); bind the result to \(\sigma\).
3. draw from \textit{Normal}(\mu, \sigma); bind the result to \(x\).

We call this a forward sampling representation of the probabilistic model: each variable can be drawn individually in sequence.

However, suppose instead that we want to draw \(\mu\) and \(\sigma\) but the variable \(x\) is observed. We then need to perform probabilistic inference to determine the posterior distributions for \(\mu\) and \(\sigma\) given the observed \(x\), and we should draw \(\mu\) and \(\sigma\) from this posterior.

Probabilistic programs represent probabilistic models, such as the example above, in a way that facilitates inference algorithms to draw from the posterior distributions of the unobserved parameters.

In the Stan probabilistic programming language, models are represented in terms of the joint density function over all observed and unobserved variables. This function of the variables, called the log-probability density function or \textit{lpdf}, is then typically passed to a Markov-chain Monte Carlo (MCMC) algorithm which provides draws from the posterior distribution [38]. We could represent the example probabilistic model as the following Stan program:

data {
real x;
}
parameters {
  real mu;
  real sigma;
}
model {
  target += normal_lpdf(mu | 0, 1);
  target += lognormal_lpdf(sigma | 1, 3)
  target += normal_lpdf(x | mu, sigma)
}

This program defines a joint lpdf function of the (observed) data x and the (unobserved) model parameters mu and sigma. The special variable target holds the value of the lpdf and is implicitly initialized to 0 and returned at the end of model. Each variable contributes to target according to its distribution’s density, and the final joint density is the product (or sum, in log space) of these contributions.

This density representation of the program allows for powerful posterior inference capabilities, but it is no longer straightforward to follow the efficient forward-sampling procedure. The challenge we address is to allow the user to write their program once as a density representation, and to transform it automatically to its corresponding a forward-sampling representation.

4.2.2 Efficiently sampling from model parameters

In the modeling process, we sometimes want to sample directly from our model of the variables, for example to perform PPC and SBC.

A practical barrier to using tools like SBC is the time it takes to naively sample a large number of parameters from the model, when it is presented in a density representation. In our example, the sampling process may be written as:

\[
\begin{align*}
\text{target} &+\text{normal}\_\text{lpdf}(\mu \mid 0, 1) \\
&+\text{lognormal}\_\text{lpdf}(\sigma \mid 1, 3) \\
&+\text{normal}\_\text{lpdf}(x \mid \mu, \sigma)
\end{align*}
\]
Here, \texttt{sample} is some sampling algorithm, such as an MCMC method, which takes an \texttt{lpdf} function of the distribution of the variables and produces draws. Running this algorithm is not an especially efficient way to sample. Instead, the following would be ideal:

\begin{verbatim}
(mu, sigma, x) ~ sample(normal_lpdf(mu | 0, 1)
   + lognormal_lpdf(sigma | 1, 3)
   + normal_lpdf(x | mu, sigma))
\end{verbatim}

mu ~ sample(normal_lpdf(mu | 0, 1));
sigma ~ sample(lognormal_lpdf(sigma | 1, 3));
x ~ sample(normal_lpdf(x | mu, sigma));

This sampling method is potentially faster in two ways:

1. It decomposes the sampling process to draw each variable individually. This scales and parallelizes easily.

2. It isolates the distributions of each variable, allowing us to potentially recognize common distributions with known efficient sampling algorithms, such as the Normal distribution.

This sequential approach, called forward sampling, is often significantly faster than sampling from the program as a whole using an MCMC method, and has better convergence properties [38].

4.2.3 Translation challenges

Ideally, we would like to translate programs written in a general \texttt{lpdf}-function form into a form amenable to forward sampling. There are a number of issues that make this translation challenging:

1. Programmers are free to write the \texttt{lpdf} function as unconstrained code, with intermediate variables and program structures that muddle the dependencies between the variables.
There may be more than one way to produce a valid translation. For example, consider the following \texttt{lpdf} function:

\begin{verbatim}
parameters {
  real x, y;
}
model {
  target += f1(x);
  target += f2(y);
  target += f3(x, y);
}
\end{verbatim}

There are at least two valid interpretations of this \texttt{lpdf} program as sequential sampling statements:

\begin{verbatim}
x ~ sample(f1(x))
y ~ sample(f2(y) + f3(x, y))
\end{verbatim}

and

\begin{verbatim}
y ~ sample(f2(x))
x ~ sample(f1(x) + f3(x, y))
\end{verbatim}

There is not enough information in the original program to determine which of these distinct sampling distribution assignments is correct.

There are some \texttt{lpdf} functions which cannot be decomposed into a sequence of sampling statements:

\begin{verbatim}
parameters {
\end{verbatim}
real x, y, z;
}
model {
  target += f1(x, y)
  target += f2(x, z)
  target += f2(y, z)
}

Which parameter should be drawn first? As every parameter depends on another, this represents a directed graphical model with a cycle. Hence, forward sampling is not possible.

4.2.4 Our approach

We present a process for translating probabilistic programs in \texttt{lpdf} form into forward-sampling programs. We start by observing that, at compile-time, the conditional independence structure of an \texttt{lpdf}-function probabilistic program can be distilled into a factor graph, which is an undirected graph of factors (such as \(f_1\) and \(f_2\) in the examples above) and variables \((x, y,\) and \(z)\). Further, we represent probabilistic models which can be forward sampled by the more informative abstraction of a directed acyclic graphical model. Using these abstractions as intermediates between probabilistic programs and forward-sampling programs, our approach works as follows:

1. Extract a factor graph from the \texttt{lpdf}-function program by applying dependency analysis to the program source code.

2. Perform a graph transformation from the factor graph to a directed acyclic graph (DAG), when possible. We describe this process in the methods section.

3. Produce forward-sampling code for the variables in topological order of the DAG.
4.3 Methods

In this section, we present our procedure for translating a factor graph derived from a probabilistic program into a DAG from which we can efficiently sample. We define the necessary terms below, then define the two graph abstractions, present a sequence of algorithms, and prove soundness and completeness properties of the methods.

For an input probabilistic program $P$, let $S(P)$ be the set of statements in $P$ and let $V(P)$ be the set of identifiers for variables in $P$.

Intuitively, a factor is a fragment of the program $P$ that directly contributes to the joint density that $P$ represents. To be precise, a factor $f$ is a statement $s \in S(P)$ along with the set of statements on which $s$ depends. Let $F(P)$ be the set of all factors in a program, then $f \in F(P) \subseteq S(P) \times \mathcal{P}(S(P))$, where $\mathcal{P}$ indicates the powerset.

When we write $f(v_1, v_2, \ldots)$ with $v_1, v_2, \cdots \in V(P)$, we interpret $f$ as the function which corresponds to the density contribution calculated by $f$, given specified values for $v_i$.

Note that throughout Section 4.3, we will work with pdf functions rather than log pdf functions, despite Stan factors being in lpdf form. We find that it is more intuitive to reason about densities outside of log-space. All of the results in this section can mechanically be translated to log-space, if desired.

4.3.1 Factor Graphs

A factor graph represents a probability distribution as a product of factors that make up the distribution’s density function. A factor graph $G_{Fac}$ is an undirected, bipartite graph with two sets of vertices: the sets of variables $V(G_{Fac})$ and factors $F(G_{Fac})$. For a program $P$, we use $V(G_{Fac}) = V(P)$ and $F(G_{Fac}) = F(P)$. The edge set $E(G_{Fac}) \subseteq V(G_{Fac}) \times F(G_{Fac})$ connects each factor to the variables that it depends on. Let $\text{Nei}(G_{Fac}, f) = \{v \mid (v, f) \in E(G_{Fac})\}$ be the set of variables which are $f$’s neighbors in $G_{Fac}$. A factor graph then defines a joint density.
function:

$$pdf_{G_{Fac}}(V(G_{Fac})) = \prod_{f \in F(G_{Fac})} f(\text{Nei}(G_{Fac}, f))$$

An example factor graph that corresponds to the first simple Stan program in 2 is shown in Figure 4.1.

![Factor graph](image)

Figure 4.1: A factor graph corresponding to a simple Stan program. The rectangular nodes correspond to the factors of the model’s density function, while the rounded nodes correspond to the model’s random variables, and edges are drawn between a factor and a variable if the factor includes the variable.

### 4.3.2 Directed Acyclic Probabilistic Graphical Models

A directed acyclic probabilistic graphical model (DAG) represents a joint probability distribution as the product of conditional density functions which are associated with each variable. A DAG $G_{DAG}$ consists of sets $V(G_{DAG})$ of vertices and $E(G_{DAG}) \subseteq V(G_{DAG}) \times V(G_{DAG})$ of directed edges, together with, for each $v \in V(G_{DAG})$, a real-valued function $D_v$. We think of $D_v$ as an (unnormalized) conditional density for $v$ given its parents in the graph. The graph then represents a joint density over the variables as:

$$pdf_{G_{DAG}}(V(G_{DAG})) = \prod_{v \in V(G_{DAG})} D_v(v \mid \text{Par}(G_{DAG}, v))$$

For a program $P$, we can choose $V(G_{DAG}) = V(P)$ and $E(G_{DAG}) \subseteq V(G_{DAG}) \times V(G_{DAG})$. Given a function $A_{G_{DAG}} : V(G_{DAG}) \to \mathcal{P}(F(P))$ to assign to each variable the factors that make
up its conditional density function, we can define

\[ D_v(v \mid \text{Par}(G_{DAG}, v)) = \prod_{f \in \overline{\text{G}_{DAG}(v)}} f(\text{Par}(G_{DAG}, v)). \]

We will refer to variables without any parents in a DAG as root variables and variables without any children in a DAG as leaf variables. Let \( \text{Par}(G_{DAG}, v) = \{ v' \mid (v', v) \in E(G_{DAG}) \} \) be the set of parents of \( v \) in \( G_{DAG} \).

We call a density \( D_v(v \mid v_1, \ldots, v_N) \) constant-normalized if its total mass is constant with respect to the variables being conditioned on:

\[ \forall i \in [1, \ldots, N], \quad \frac{\partial}{\partial v_i} \left( \int D_v(v \mid v_1, \ldots, v_N) \, dv \right) = 0. \]

Let a DAG be called a constant-normalized DAG if all \( D_v \) are constant-normalized. We note that the typical definition of a DAG requires that each \( D_v \) be a conditional probability distribution, and so it would trivially fit what we call a constant-normalized DAG.

**Sampling from constant-normalized DAGs**

We show that sampling from a constant-normalized DAG \( G \) in topological order is consistent with the joint density of the graph.

Let \( \text{sample} \) be a procedure for sampling a value from a (not necessarily normalized) density function \( pdf \):

\[ P(v \leftarrow \text{sample}(pdf)) \propto pdf(v) \]

We define a conditioned DAG \( G\mid_\{v \leftarrow \bar{v}\} \) for some DAG \( G \) and root variable \( v \in V(G) \) to be a DAG with variables \( V(G\mid_\{v \leftarrow \bar{v}\}) = V(G) \setminus v \), edges \( E(G\mid_\{v \leftarrow \bar{v}\}) = \{(v', v'') \mid (v', v'') \in E(G), v' \neq v\} \), and conditional densities \( D_{v'}(v' \mid \text{Par}(v')) = D_{v'}(v' \mid \text{Par}(v'), \bar{v}) \).

Let \( N = |V(G)| \) be the number of variables in \( G \), and let \( v_1, \ldots, v_N \) be any topological order of the variables in \( G \), so that \( v_1 \) is a root and \( v_N \) is a leaf. Let \( V_i \) represent the set of variables
\{v_1, \ldots, v_i\}, and \tilde{V}_i represent a set of draws \{\tilde{v}_1, \ldots, \tilde{v}_i\} for these variables. Then, we iterate the previous construction and write \(G|_{\tilde{V}_0 \leftarrow \tilde{v}_0} = G\) and \(G|_{\tilde{V}_i \leftarrow \tilde{v}_i} = G|_{V_{i-1} \leftarrow \tilde{v}_{i-1}}|_{V_i \leftarrow \tilde{v}_i}\).

Let \(P_G(v), P_G(v_1, \ldots v_N),\) and \(P_G(v_\ldots)\) be marginal, joint and conditional distributions of the variables under the joint distribution of \(G\) defined by the joint density \(pdf_{G_DAG}(V(G))\).

**Lemma 1.** For any root variable \(v_r\) in a constant-normalized DAG \(G, D_{v_r}(v_r) \propto P_G(v_r)\).

**Proof.** Let \(v_\ell\) be any leaf variable in \(G\) such that \(v_\ell \neq v_r\):

\[
P_G(v_r) = \int_{V(G) \setminus v_r} pdf_{G_{DAG}}(V(G)) \prod_{v \in V(G) \setminus v_r} dv
\]

\[
= D_{v_r}(v_r) \int_{V(G) \setminus v_r} \prod_{v \in V(G) \setminus v_r} D_v(v | Par(v)) dv
\]

\[
= D_{v_r}(v_r) \int_{V(G) \setminus v_r} \left( \int_{v_\ell} D_{v_\ell}(v_\ell | Par(v_\ell)) dv_\ell \right) \prod_{v \in V(G) \setminus v_r \setminus v_\ell} D_v(v | Par(v)) dv
\]

\[
= \left( \int_{v_\ell} D_{v_\ell}(v_\ell | Par(v_\ell)) dv_\ell \right) D_{v_r}(v_r)
\]

\[
= \int_{V(G) \setminus v_r \setminus v_\ell} \prod_{v \in V(G) \setminus v_r \setminus v_\ell} D_v(v | Par(v)) dv
\]

\[
\propto D_{v_r}(v_r) \int_{V(G) \setminus v_r \setminus v_\ell} \prod_{v \in V(G) \setminus v_r \setminus v_\ell} D_v(v | Par(v)) dv
\]

\[
= P_G|_{v_r}(v_r).
\]

Here, the fourth equality follows by constant-normalization.

We then induct with a new \(v_\ell\) until there are no non-\(v_r\) leaf variables. At that point we have \(P_G(v_r) \propto P_{G_{v_r}}(v_r)\), where \(G_{v_r}\) is \(G\) without any descendants of \(v_r\). Since \(v_r\) is then both a root and a leaf variable in \(G_{v_r}\):

\[
P_G(v_r) \propto P_{G_{v_r}}(v_r) \propto D_{v_r}(v_r)
\]

**Lemma 2.** For topologically ordered variables \(v_1, \ldots, v_{i-1}\) in a constant-normalized DAG \(G\),

\[
P_{G|_{V_{i-1} \leftarrow \tilde{v}_{i-1}}}(v_i) = P_G(v_i | \tilde{V}_{i-1})
\]

162
Proof.

\[ P_G(v_i \mid \tilde{v}_{i-1}) \]
\[ = \int_{V(G) \setminus \{v_i\}} \prod_{v \in V(G) \setminus \{v_i\}} D_v(v \mid \text{Par}(v) \setminus \{v_i\}, \tilde{v}_{i-1}) \, dv \]
\[ = \int_{V(G) \setminus \{v_i\}} \prod_{v \in V(G) \setminus \{v_i\}} D_v(v \mid \text{Par}(v) \setminus \{v_{i-1}\}, \tilde{v}_{i-1}) \, dv \]
\[ = P_{G|_{V_{i-1} \setminus \tilde{v}_{i-1}}}(v_i) \]

**Theorem 3.** When each of the variables of \( G \ v_1, \ldots, v_N \) are drawn in topological order according to \( \tilde{v}_i \leftarrow \text{sample}(D_{v_i}|_{V_{i-1} \setminus \tilde{v}_{i-1}}) \), the draws \( \tilde{v}_N \) are distributed according to \( P_G(\tilde{v}_N) \).

**Proof.** We calculate

\[ P(\tilde{v}_1 \leftarrow \text{sample}(D_{v_1}), \ldots, \tilde{v}_N \leftarrow \text{sample}(D_{v_N})) \]
\[ = P(\tilde{v}_1 \leftarrow \text{sample}(D_{v_1})). \]
\[ P(\tilde{v}_2 \leftarrow \text{sample}(D_{v_2}) \mid \tilde{v}_1). \]
\[ \vdots \]
\[ P(\tilde{v}_N \leftarrow \text{sample}(D_{v_N}) \mid \tilde{v}_{N-1}). \]

Since \( v_1 \) is a root variable in \( G \), then, by the definition of \text{sample} and Lemma 1, \( P(\tilde{v}_1 \leftarrow \text{sample}(D_{v_1})) \propto D_{v_1}(\tilde{v}_1) \propto P_G(v_1) \), so \( P(\tilde{v}_1 \leftarrow \text{sample}(D_{v_1})) = P_G(\tilde{v}_1) \). By the same reasoning, since each \( v_i \) is a root variable in \( G|_{V_{i-1} \setminus \tilde{v}_{i-1}} \), \( P(\tilde{v}_i \leftarrow \text{sample}(D_{v_i}|_{V_{i-1} \setminus \tilde{v}_{i-1}})) \propto D_{v_i|_{V_{i-1} \setminus \tilde{v}_{i-1}}}(\tilde{v}_i) \propto P_{G|_{V_{i-1} \setminus \tilde{v}_{i-1}}}(v_i) \), so \( P_{G|_{V_{i-1} \setminus \tilde{v}_{i-1}}}(v_i) = P(\tilde{v}_i \leftarrow \text{sample}(D_{v_i}|_{V_{i-1} \setminus \tilde{v}_{i-1}})) \). Making those substitutions:

\[ P(\tilde{v}_1 \leftarrow \text{sample}(D_{v_1}), \ldots, \tilde{v}_N \leftarrow \text{sample}(D_{v_N})) \]
\[ = P_G(\tilde{v}_1); \]
\[ P_{G|_{V_{1-1} \setminus \tilde{v}_{1-1}}}(\tilde{v}_2). \]
\[ \vdots \]
\[ P_{G|_{V_{N-1} \setminus \tilde{v}_{N-1}}}(\tilde{v}_N). \]
Then by Lemma 2, for each $i$: $P_{G|V_{i-1}=\tilde{v}_{i-1}}(\tilde{v}_i) = P_G(\tilde{v}_i|\tilde{V}_{i-1})$.

$$P(\tilde{v}_1 \leftarrow \text{sample}(D_{v_1}), \ldots, \tilde{v}_N \leftarrow \text{sample}(D_{v_N}))$$

$$= P_G(\tilde{v}_1);$$

$$P_G(\tilde{v}_2|\tilde{v}_1).$$

$$\ldots$$

$$P_G(\tilde{v}_N|\tilde{V}_{N-1})$$

$$= P_G(\tilde{v}_1, \tilde{v}_2, \ldots, \tilde{v}_N).$$

Constant-normalization is necessary for Lemma 1. If any $D_v$ is not constant-normalized, the sampling process may not be consistent with the joint distribution.

We define, in pseudocode, $\text{sampleG}(G)$, a procedure which draws a sample for each variable in a DAG $G$ in topological order. Writing $++$ for the sequence concatenation operator,

$$\text{sampleG}(G) = \text{let } v_r \in \text{roots}(G), \tilde{v}_r \leftarrow \text{sample}(A_G(v_r)) \text{ in}$$

$$[\tilde{v}_r] ++ \text{sampleG}(G|_{v_r \leftarrow \tilde{v}_r})$$

We pass $\text{sample}$ the set $A_G(v_r)$ of assigned factors to allow for flexibility of implementation. The graph is thus reduced in (any) topological order, with each variable replaced by a draw from its marginal distribution.

4.3.3 A relation between Factor Graphs and DAGs

Factor graphs and DAGs are both representations of the joint density of a set of variables. We say that a DAG $G_{\text{DAG}}$ is a sound transformation of a factor graph $G_{\text{Fac}}$ if:

1. $V(G_{\text{DAG}}) = V(G_{\text{Fac}})$

2. $pdf_{G_{\text{DAG}}}(V(G_{\text{DAG}})) = pdf_{G_{\text{Fac}}}(V(G_{\text{Fac}}))$

Since $G_{\text{DAG}}$ is a DAG, it also must be acyclic.
Let the relation $\sigma(G_{Fac}, G_{DAG})$ be true if and only if $G_{DAG}$ is a sound transformation of $G_{Fac}$.

4.3.4 Defining a transformation from Factor Graphs to DAGs

Our strategy to transform factor graphs into DAGs will be to reinterpret each factor as part of the conditional density function of some variable.

A factor can only be part of the conditional density of a variable if that factor is a function of that variable, so factors will always be assigned to variables with which they share an edge in the factor graph. The assignment of factors to conditional densities can thus be thought of as selecting a subset of edges from the factor graph, which we call the edge selection set, $s \subseteq E(G_{Fac})$.

To produce a DAG given a factor graph $G_{Fac}$ and an edge selection set $s$, we define a function $G_{DAG} = C(G_{Fac}, s)$, which we call the contraction function. The contraction function produces a DAG with a vertex for each variable and directed edges built according to the edge selection set:

$$
V(C(G_{Fac}, s)) = V(G_{Fac})
$$

$$
E(C(G_{Fac}, s)) = \{(v_a, v_b) \mid v_a, v_b \in V(G_{Fac}), \exists f \in F(G_{Fac}), (v_b, f) \in s, (v_a, f) \in E(G_{Fac})\}
$$

$$
A_{C(G_{Fac}, s)}(v) = \{f \mid (v', f) \in s, v' = v\}
$$

The contraction function contracts all of the edges in the selection set, adding each variable which contributes to a contracted factor as a parent. The joint density of the resulting DAG then is

$$
pdf_{G_{DAG}}(V(C(G_{Fac}, s)))) = \prod_v D_v(v \mid Par(C(G_{Fac}, s), v))
$$

$$
= \prod_v \prod_{f \mid (v, f) \in s} f(\text{Nei}(G_{Fac}, f))
$$
4.3.5 Computing sound transformations between Factor Graphs and DAGs

Our goal is to find the set $S^*$ of edge selection sets $s$ which produce a sound and constant-normalized DAG $C(G_{Fac}, s)$. Our strategy will be as follows:

1. **Identify the set $r$ of recognizable edges.** Use heuristics to immediately identify some factors as constant-normalized conditional densities for a neighboring variable, producing the set of recognizable edges, $r$. Discussed in section Section 4.3.5.

2. **Construct the set $S$ of valid edge selection sets.** Construct the set $S$ of edge selection sets $s$ which produce a sound DAG $C(G_{Fac}, s)$ such that $r \subseteq s$. We accomplish this by use of a SAT solver. Discussed in section Section 4.3.5.

3. **Query the user when necessary.** When it is ambiguous whether a conditional density is constant-normalized, query the user for an assertion of constant-normalization. Then, filter the set $S$ down to the subset $S^*$ of those edge selection sets consistent with the query results. This is necessary because we require that the DAG only contains constant-normalized densities in order to sample from it (Section 4.3.2). Discussed in Section 4.3.5.

4. **Generate sampling code.** Generate a forward sampling program which draws each variable $v$ from its constant-normalized conditional density function $D_v$ determined by $s \in S^*$. Since this step is implementation specific, we discuss this in section Section 4.4.1 along with our Stan implementation.

**Theorem 4.** The edge selection sets $s$ in $S^*$ produce DAGs which are equal to each other up to proportionality.

**Proof.** Let $s_1, s_2 \in S^*$ and $v \in V$.

Since $s_1$ and $s_2$ are both constant-normalized, $D_{v,C(G_{Fac}, s_1)}(v)$ and $D_{v,C(G_{Fac}, s_2)}(v)$ are both proportional to the marginal distribution for $v$ according to Theorem 3, and are therefore also proportional to each other.
Due to Theorem 4, any edge selection set $s \in S^*$ will result in the same sampling distribution.

**Recognizing constant-normalized conditional densities**

Since we are unable to classify constant-normalization statically from source code in general, we use simple pattern heuristics to identify a conservative subset of cases which we call *recognizable edges*. When we recognize an edge $(v, f)$ of the factor graph, we assert that $f$ makes up the entire conditional density for $v$.

In our Stan implementation, we can recognize a factor $f$ as a constant-normalized conditional density for $v$ if the statement of $f$ takes the form of `target + dist\_lpdf(v | ..)=` or `v \sim dist(..)`, where `dist\_lpdf` is either one of Stan’s built-in probability distributions[36] or a user defined distribution (whose name ends in \_lpdf). User-defined function names with this suffix are considered to be user annotations that assert that the function implements a constant-normalized distribution.

**Computing sound edge selection sets**

Our strategy is to leverage a SAT solver into producing $S$, the set of edge selection sets $s$ such that $\sigma(G_{Fac}, C(G_{Fac}, s))$. SAT solvers take as input a propositional formula\(^2\) over a set of atomic propositions, and return the set of subsets of atomic propositions which satisfy the formula. We will construct a propositional formula and set of atomic propositions to encode $\sigma$ and then translate the set of solutions into the set $S$.

First, we construct a set of atomic propositions to represent an edge selection set $s$ and useful properties of $s$. The set of atomic propositions $Atom(G_{Fac})$ is the union of the following:

1. **An edge $(v, f)$ is selected.** $Sel_{v, f}$ asserts that the edge $(v, f)$ is included in the selection set.

\[
\{Sel_{v, f} \mid v \in V(G_{Fac}), f \in F(G_{Fac})\}
\]

\(^2\)The SAT solver interface we use allows us to input general propositional formulae, which are then automatically translated to Conjunctive Normal Form.
2. There exists a path between vertices. $P_{v_1 \rightarrow v_2}$ asserts the existence of a path from $v_1$ to $v_2$ in $C(G_{Fac}, s)$.

$$\{ P_{v_1 \rightarrow v_2} \mid v_1, v_2 \in V(G_{Fac}) \}$$

Let $\alpha : \mathcal{P}(\text{Atom}(G_{Fac})) \rightarrow \mathcal{P}(E(G_{Fac}))$ be a function that reconstructs an edge selection set from a SAT solution: $\alpha(A) = \{(v, f) \mid Sel_{v, f} \in A\}$.

Next we construct a propositional formula, $prop_{G_{Fac}}(A)$, to be true if and only if $\sigma(G_{Fac}, C(G_{Fac}, \alpha(A)))$ and $r \subseteq \alpha(A)$. The formula $prop_{G_{Fac}}$ is the conjunction of the following rules:

1. The resulting DAG is acyclic. $C(G_{Fac}, s)$ does not include any cycles, so no variable has a path to itself.

$$\forall v \in V(G_{Fac}), \neg P_{v \rightarrow v}$$

2. All factors are covered. $s$ includes at least one edge that includes each factor in $F(G_{Fac})$.

$$\forall f \in F(G_{Fac}), \exists v \in V(G_{Fac}), Sel_{v, f}$$

3. Factors are not covered more than once. $s$ does not include more than one edge that includes each factor in $F(G_{Fac})$. Therefore, selecting one edge with a factor $f$ implies that no other edges which include $f$ are selected.

$$\forall f \in F(G_{Fac}), v_1 \in V(G_{Fac}), v_2 \in V(G_{Fac}),$$

$$\quad (v_1 \neq v_2) \land Sel_{v_1, f} \quad \Rightarrow \quad \neg Sel_{v_2, f}$$

4. All variables are covered. $s$ includes at least one edge that includes each variable in $V(G_{Fac})$.

$$\forall v \in V(G_{Fac}), \exists f \in F(G_{Fac}), Sel_{v, f}$$

168
5. Edges in \( r \) are included.

\[ \forall (v,f) \in r, \, Sel_{v,f} \]

6. Variables covered by \( r \) are not included again. Since in a recognizable edge \( (v,f) \) the factor \( f \) is asserted to be the entire conditional density for \( v \), no other factors will contribute to \( v \)'s conditional density.

\[ \forall (v,f_1) \in r, f_2 \in F(G_{Fac}), f_1 \neq f_2 \implies \neg Sel_{v,f_2} \]

7. Selecting an edge creates edges in the DAG. There will be an edge from variable \( v_1 \) to variable \( v_2 \) in \( C(G_{Fac}, s) \) if, for some factor \( f \), there is an unselected edge \( (v_1, f) \) (implying that \( v_1 \) contributes to \( f \)) and a selected edge \( (v_2, f) \).

\[ \forall f \in F(G_{Fac}), v_1, v_2 \in V(G_{Fac}), \]

\[ (v_1, f), (v_2, f) \in E(G_{Fac}), \]

\[ \neg Sel_{v_1,f} \land Sel_{v_2,f} \implies P_{v_1 \rightarrow v_2} \]

8. Paths between variables compose.

\[ \forall v_1, v_2, v_3 \in V(G_{Fac}), \, P_{v_1 \rightarrow v_2} \land P_{v_2 \rightarrow v_3} \implies P_{v_1 \rightarrow v_3} \]

We now construct the set \( S \) of sound edge selection sets by:

\[ S = \{ \alpha(A) \mid A \in SAT(prop_{G_{Fac}}) \} \]

**Theorem 5** (Soundness). For any sound edge selection set \( s \in S \), we have that \( r \subseteq s \) and \( C(G_{Fac}, s) \) is a sound transformation of \( G_{Fac} \).
Proof. By rule 4 of $\text{prop}_{\text{Fac}}$, each variable in $G_{\text{Fac}}$ will appear in $s$, and so $V(G_{\text{DAG}}) = V(C(G_{\text{Fac}}, s))$.

By rules 2 and 3 of $\text{prop}_{\text{Fac}}$, each factor will be included in exactly one edge in $s$, so the joint density does not change:

\[
\text{pdf}_{G_{\text{DAG}}}(V(C(G_{\text{Fac}}, s))) = \prod_v \prod_{f \in \text{Nei}(G_{\text{Fac}}, f)} f(\text{Nei}(G_{\text{Fac}}, f))
\]

\[
= \prod_f f(\text{Nei}(G_{\text{Fac}}, f))
\]

\[
= \text{pdf}_{\text{Fac}}(V(G_{\text{Fac}}))
\]

Theorem 6 (Completeness). For any edge selection set $s$ such that $C(G_{\text{Fac}}, s)$ is a sound transformation of $G_{\text{Fac}}$ and $r \subseteq s$, $\text{prop}_{\text{Fac}}(s)$ is true.

Proof. Suppose that there were some $s = \alpha(A)$ for which $G = C(G_{\text{Fac}}, s)$ is a sound transformation of $G_{\text{Fac}}$, but $\text{prop}_{\text{Fac}}(A)$ were false. $\text{prop}_{\text{Fac}}(A)$ being false implies that $s$ broke at least one rule:

If $s$ broke rule 1, $G$ would be cyclic and therefore not a DAG.

If $s$ broke rule 2, $G$ would not include some factor in its joint density function and would not have the same joint density as $G_{\text{Fac}}$, and so would not be sound.

If $s$ broke rule 3, $G$ would include some factor in its joint density function more than once and would not have the same joint density as $G_{\text{Fac}}$, and so would not be sound.

If $s$ broke rule 4, $G$ would not include some variable in $V(G_{\text{DAG}})$, and so would not be sound.

If $s$ broke rule 5, $r \not\subseteq s$.

If $s$ broke rule 6, some $v$ covered by $r$ would have another edge, so the variables covered by $r$ would be otherwise assigned.

Rules 7 and 8 enforce the invariant that the $P_{\rightarrow}$ propositions reflect the paths in the DAG produced by the $\text{Sel}$ variables, and they cannot in themselves reject any potential $s$.

Breaking any of these rules results in a contradiction, so no such $s$ exists.

Consequently, $\text{prop}_{\text{Fac}}(A) \iff \sigma(G_{\text{DAG}}, C(G_{\text{DAG}}, \alpha(s))) \land r \subseteq s$. Therefore, $\{s(A) \mid A \in$
SAT(prop_{G_{Fac}}) = \{ s \mid \sigma(G_{DAG}, C(G_{DAG}, s)) \} is a sound and complete set of edge selection sets.

Querying the user with ambiguous conditional densities

When it cannot be gleaned whether a conditional density is constant-normalized, either from a recognizable edge or from the topology of the factor graph, then it is necessary to ask the user if they can assert constant-normalization. This is necessary because it is not possible to perfectly identify constant-normalization from code, while we require all densities to be constant-normalized in order to sample from the DAG (Section 4.3.2).

There are alternatives to avoid user queries: (1) Wrap each constant-normalized conditional density function in a recognizable user-defined function before the tool is applied. This will disambiguate all conditional densities. (2) Fail when conditional densities cannot be automatically proven constant-normalized. This will limit the scope of application of the tool.

We describe our strategy for user queries below.

Given the set $S$ of sound edge selection sets and the set $R$ of recognizable edges, our goal is to filter out edge selection sets in $S$ which contain conditional density assignments that are not constant-normalized.

We will query the user with each density assignment, allowing them to assert or not assert that the presented set of factors is constant-normalized. We will not need to query assignments which are included in $R$ since those are already assumed to be constant-normalized. We will also not need to query any variables which are root variables in all DAGs arising from $S$. We can then filter out edge selection sets with assignments which the user did not assert are constant-normalized.

We use a query procedure, `query`, which takes as input a variable $v$ and a set of assigned factors $F$ and returns `true` if and only if the user asserts that the assigned density $D_v(v \mid \ldots) = \prod_{f \in F} f(Nei(G_{Fac}, f))$ is constant-normalized.
We construct the set of queries \( Q \):

\[
V' = V \setminus \{ v \mid (v, f) \in r \}
\]

\[
F(v, s) = \{ f \mid (v, f) \in s \}
\]

\[
Q = \{(v, F(v, s)) \mid v \in V', s \in S \}
\]

The set of affirmative queries \( Q' \) is:

\[
Q' = \{(v, F) \mid (v, F) \in Q \text{ query}(v, F)\}
\]

We can then produce the set of edge selection sets which are consistent with the affirmative queries, \( S^* \):

\[
S^* = \{s \mid s \in S \ \forall v \in V', (v, F(v, s)) \in Q'\}
\]

When the set \( S^* \) is empty, we can conclude that no DAG can be derived from the factor graph with the given constant-normalized densities.

### 4.4 Results

#### 4.4.1 Stan implementation

We implemented a pipeline that (1) extracts a factor graph from a Stan program, (2) produces a constant-normalized DAG\(^3\) when possible using a SAT solver and possibly user queries, and then (3) constructs a forward-sampling program. Parts (1) and (3) are specific to Stan (but could be implemented for any Stan-like language), while part (2) could readily be applied to a factor graph from any source\(^4\).

\(^3\)Since multiple valid DAGs will have proportional conditional densities (Theorem 4), it is not necessary to return the whole set.

\(^4\)The only Stan-specific part of the DAG transformation is the heuristic to recognize constant-normalized edges; this would need to be adapted in an implementation for another language.
Extracting a factor graph from a Stan program

Factor graphs are derived from Stan programs by a pipeline of static analyses which we have built into the open-source Stanc3 compiler [37].

Consider a Stan program $P$ with statement set $S(P)$ and a set of free variables $V(P)$. Let $V_S(S)$ be the set of free variables in the statement $S$.

We first derive a dependency graph $D(P)$ which is a directed graph between the statements $S(P)$. The dependency graph has a directed edge $(s_1, s_2) \in E(D(P))$ if $s_1$ directly or indirectly influences the behavior of $s_2$. The dependency graph is built using a reaching definitions monotone framework analysis on the control flow graph [44]. As is standard, the resulting static analysis is sound but not complete. As such it will detect a superset of the true dependencies present in the program, resulting in a conservative approximation.

From the dependency graph $D(P)$ and a statement $S$, we construct a set $V_D(S)$ to be the set of free variables that $S$ depends on:

$$V_D(s_1) = V_S(s_1) \cup \{v \mid (s_2, s_1) \in E(D(P)), v \in V_S(s_2)\}$$

Next, we collect all factors in the program, $F(P)$. Factors are statements which have a direct effect on the density that the Stan program represents. These take the form of:

1. `target +=` and `x ~` statements, which directly increment the `target` value.
2. Function calls which can affect `target`. In Stan, only functions which the suffix `_lp` can affect `target`.
3. `reject` statements. In Stan, when a `reject` statement is executed, the current sample point is thrown out, which implicitly sets the density of the point to zero.

For each statement $s$ of the above form, we decorate $s$ with its statement dependencies to
produce each factor $f \in F(P)$:

$$f = (s, \{s' \mid (s', s) \in E(D(P))\})$$

We can now construct a factor graph $G_{Fac}(P)$:

$$V(G_{Fac}(P)) = V(P)$$
$$F(G_{Fac}(P)) = F(P)$$
$$E(G_{Fac}(P)) = \{(v, f) \mid f \in F(P), v \in V_D(f)\}$$

**Producing a constant-normalized DAG**

We implement the algorithm described in section Section 4.3.5 with a simple Haskell program using the MiniSat package [47, 31]. The implementation parses the factor graph produced by the Stanc3 compiler, applies a SAT solver, and queries the user when the constant-normalization of a density is ambiguous (section Section 4.3.5).

If this process returns no solutions, we can conclude that no sound DAG can be produced from the original factor graph with the given constant-normalized densities, and our process terminates.

**Generating sampling code**

Finally, we construct a sampling program using the DAG $G$ and the program statements which decorate the factors. For a set $A$ of factors, let $Stat(A)$ be the sequence of statements in $P$ corresponding to the factors and write $Slice(A)$ for their sequence of statement dependencies in $P$.

We first define $sample(A)$ which generates the necessary code for $A$, the set of factors assigned to the conditional density function. $sample(A)$ returns a list of statements which either generate samples with a Stan builtin _rng function and are decorated with an RNG label, or cal-
ulate a density function by affecting Stan’s target variable and are decorated with a PDF label:

$$\text{sample}(A) =$$

if $A$ is a singleton factor $\{f\}$ with $f$ of the form:

$$\text{target} += \text{dist}_lpdf(\mu \mid . . .) \text{ or}$$

$$\mu \sim \text{dist}(..)$$

then

$$(\text{RNG}, \text{Slice}(\{f\}) \Rightarrow \{\mu = \text{dist}_r\text{ng}(..)\})$$

else

$$(\text{PDF}, \text{Slice}(A) \Rightarrow \text{Stat}(A))$$

We can now apply the procedure sample defined in section Section 4.3.2, which applies sample to each variable’s assigned factors in topological order and returns a sequence of the results. The result is a sequence of code segments decorated with RNG or PDF.

**Synthesizing sampling programs as Stan programs**

The way that we can turn the sequence of code segments into executable Stan programs depend on the number of segments labeled as PDF. These statements can only be written in Stan’s model block, while segments labeled as RNG can only be written in Stan’s transformed data or generated quantities blocks, which are executed before and after model, respectively. Since Stan programs only have one model block, we will need one Stan program for each PDF-labeled segment.

For example, suppose we have the following sequence of code segments:

$$(\text{RNG}, S_1), (\text{RNG}, S_2), (\text{PDF}, S_3), (\text{RNG}, S_4)$$

$S_3$ must go in the model block. Since $S_1$ and $S_2$ must go before $S_3$, they go in the transformed
data block. Since $S_4$ must go after $S_3$, it goes in the generated quantities block.

**Synthesizing Stan programs for prior predictive sampling**

Prior predictive sampling is done by sampling from the model parameters without influence from the data, and then sampling synthetic data from the likelihood, using the model parameter draws.

We generate the sequence $Sam_{prior}$ of sampling code for the prior samples by removing each data variable and its neighbors from the factor graph, and then following our method. We then generate the sequence $Sam_{predictive}$ of sampling code for the data variables by removing each model parameter variables from the factor graph, and then following our method. We then synthesize a Stan program from $Sam_{prior} \leftrightarrow Sam_{predictive}$.

**Synthesizing Stan programs for Simulation-based Calibration**

Simulation-based Calibration is performed by sampling from the prior predictive distribution, then sampling from the posterior distribution given the synthetic data, and finally calculating the rank statistics of the generating prior samples among the posterior samples. This procedure can be achieved within Stan by using the model block to draw from the posterior and the generated quantities block to calculate the rank statistics [95]. The prior predictive sampling code is generated in the same way as section Section 4.4.1. Because SBC requires the use of a model block to sample from the posterior, the whole process cannot be fit into one Stan program unless there are no PDF code segments in the prior predictive sampling code. Otherwise it will need one Stan program for each PDF-labeled segment plus one.

**4.4.2 Stan program example**

We will step through two examples of translating Stan programs into forward sampling form. The first is a simple example where most edges are recognizable, no user queries are necessary, and a single-program SBC can be synthesized. The second is a more complex example where most
edges are not recognizable and a user query is necessary.

Eight schools example

Suppose we have the following Stan program, which is modified\(^5\) from the well-known eight schools problem [32]:

```stan
data {
  int<lower=0> J;
  real y[J];
  real<lower=0> sigma[J];
}
parameters {
  real mu;
  real theta[J];
  real<lower=0> tau;
}
model {
  target += - (mu - 1) ^ 2;
  target += normal_lpdf(tau | 1, 1);
  target += normal_lpdf(theta | mu, tau);
  target += normal_lpdf(y | theta, sigma);
}
```

Our goal will be to produce a new Stan program which draws from the prior predictive distribution for \(y\) [43].

The Stanc3 compiler emits the following factor graph \(G_{Fac}\):

\(^5\)The non-distribution expression on line 12 was added to make an instructive example, and the distribution on line 13 was added to give \(tau\) an arbitrary prior.
To translate this factor graph to a DAG, our implementation follows the steps from Section 4.3.5:

1. Identify the set of recognizable edges, \( r \).
   
The recognizable set is \( r = \{ (\text{theta}, \text{normal\_lpdf(} \text{theta}, \text{mu, tau}) ), (\text{tau}, \text{normal\_lpdf(} \text{tau}, 1, 1) ) \} \).

2. Construct the set of valid edge selection sets, \( S \). Given the recognizable edges, the remaining variables to be covered are \( \text{mu} \) and \( \text{tau} \). Each only has one option, so \( S = r \cup \{ (\text{mu}, -(\text{mu} - 1)^2) \} \).

3. Query the user when necessary. We do not need to query the user in this case, because all variables either have recognizable edges (\( \text{theta} \) and \( \text{tau} \)) or are root variables in all DAGs arising from \( S \) and are therefore trivially constant-normalized (\( \text{mu} \) and \( \text{tau} \)). Since we are now confident that each assigned density in \( S \) is constant-normalized, we can assert that the element \( s \) of \( S \) will produce a constant-normalized DAG \( G_{DAG} = C(G_{Fac}, s) \):

![Diagram](image_url)
4. **Generate sampling code.** We first transform and decorate each factor as in Section 4.4.1, producing the sequence $SAM_{prior}$:

\[
(PDF, \text{target} += -(\mu - 1)^2)
\]
\[
(RNG, \tau = \text{normal}_\text{rng}(1, 1))
\]
\[
(RNG, \theta = \text{normal}_\text{rng}(\mu, \tau))
\]

To produce PPC code as in Section 4.4.1, we repeat the above process starting with a factor graph that includes the desired data variables as vertices and holds the other data variables and model parameter variables constant. We find that this second DAG is built from one recognizable edge $(y, \text{normal}_\text{lpdf}(y, \theta, \sigma))$. We can transform and decorate this factor to produce the sequence $SAM_{predictive}$:

\[
(RNG, y = \text{normal}_\text{rng}(\theta, \sigma))
\]

We now generate code for $SAM_{prior} \ LIFT \ SAM_{predictive}$. Since there is only one PDF element, we can produce prior predictive sampling code in a single Stan program:

```stan
data {
  int<lower=0> J;
  real<lower=0> sigma[J];
}
transformed data {
  real mu;
  real theta[J];
  real<lower=0> tau;
  real y[J];
}
model {
```
target += - (mu - 1) ^ 2;
}
generated quantities {
    tau = normal_rng(1, 1);
    theta = normal_rng(mu, tau);
    y = normal_rng(theta, sigma);
}

This was a simple example where we produce a sampling program automatically.

**Query example**

Next consider the following Stan program:

data {
    real a;
}
parameters {
    real b;
    real c;
    real d;
    real e;
}
model {
    target += normal_lpdf(a | b, 1);
    target += normal_lpdf(b | 1, e);
    target += -c ^ 2;
    target += -d ^ 2;
    target += 0.5 * log(d / 2 * pi() * e^3);
target += -d * (e - c)^2 / (2 * c^2 * e);
}

Our goal will be to produce a constant-normalized DAG without the data variables, which could then produce a sampling program. Each line from 12 to 16 represents a non-data factor. We will refer to these factors by line number, e.g., $F_{12} = \text{normal}\_\text{lpdf}(b \mid 1, e)$.

The factors $F_{15}$ and $F_{16}$ together make up an inverse Gaussian distribution on $e$ with mean $c$ and shape parameter $d$.

The Stanc3 compiler emits the a factor graph $G_{Fac}$ (after renaming), shown in Figure 4.4.

![Factor graph](image)

**Figure 4.4:** The factor graph form of the Stan program.

Again we follow the steps from section Section 4.3.5:

1. Identify the set of recognizable edges, $r$.

   The recognizable set is $r = \{(b, F_{12})\}$.

2. Construct the set of valid edge selection sets, $S$. Our implementation finds two valid edge selection sets, one in which $F_{16}$ is matched with $e$ and one in which it is matched with $c$. All other ambiguity is eliminated automatically: for example, matching $F_{16}$ or $F_{15}$ with $d$ would imply a cycle of $d$ and $e$. 
3. **Query the user when necessary.** To disambiguate between the two elements of $S$, it is necessary to query the user to determine the true constant-normalized distributions for either $d$ or $e$ (the distribution for one is sufficient to infer the other). Our implementation prompts:

Which of the following is a constant-normalized density for $e$?

0: None of the below

1: $\text{lpdf}(e \mid c, d) =$ 
\[
\frac{\text{FMinus__(d)} \ast ((e - c)^2)}{((2 \ast (c^2)) \ast e)}
\]
\[
+ (0.5 \ast \log((((d / 2) \ast \text{pi()}) \ast (e^3))))
\]

2: $\text{lpdf}(e \mid d) =$ 
\[
(0.5 \ast \log((((d / 2) \ast \text{pi()}) \ast (e^3))))
\]

Enter a number 0-2:

If the user selects 0, no DAG will be produced; if the user selects 1, the correct DAG will be produced ((a) below); if the user selects 2, an alternate DAG will be produced ((b) below).

4.5 Discussion

One limitation of our approach is that we rely on a SAT solver to perform the graph transformation between factor graphs and DAGs. Since the SAT problem is NP-Hard, we cannot provide a polynomial computational bound on the graph transformation nor size bound on the number of edge selection sets, so sufficiently large or pathological programs will not be practical applications of this approach.

An additional limitation is that we rely on static analysis to derive the dependency structure of Stan programs. This static analysis has a number of limitations, all of which can jeopardize the efficiency but not the correctness of our approach:

- Container types such as arrays are treated as monolithic variables, which potentially overestimates dependence. This will be mitigated by further engineering effort on the Stan compiler, for example by leveraging well-known techniques like polyhedral dependence analysis [46].
• Control flow and other value-dependent structure cannot be perfectly predicted due to the halting problem, which can hide dependence structure.

• There is independence structure inside some Stan math distributions which is not accounted for. For example, a multivariate Gaussian distribution with diagonal covariance matrix does not introduce dependency between each dimension of its variate.

Our method sometimes requires user interaction (avoiding user interaction was discussed in Section 4.3.5). Since the user interaction is minimal and avoidable, we consider this a small limitation. We expect that users will typically find the answer to a query to be obvious, since the code was likely produced with a probability distribution in mind for each variable, as in section Section 4.4.2. If the user is unsure, they can simply make no claims of constant-normalization and result will be a loss of efficiency rather than loss of correctness. Only in cases where the user makes an incorrect assertion will the correctness of the result be compromised, and that case a hand-built sampling program would certainly contain the same error.

We do not believe that these limitations will prevent this method from being applicable in the vast majority of practical cases. We hope that this tool will fit into the workflow of many Stan users, allowing them to efficiently automate methods like SBC and PPC and thereby produce more reliable results.

We also note that although parts of our implementation are specific to Stan, the idea is applicable more generally to Stan-like density-based probabilistic programming languages [34]. There may also be other applications for the production of DAGs from Stan-like programs. For example, for directly checking the structure of the written model. When a Stan user writes a Stan program, they will sometimes have expectations about the independence structure in terms of a directed graph, since directed probabilistic graphical models are common representations in machine learning [35] and effective at communicating modeling ideas. When this is the case, we could automatically check the user’s Stan program against their assumptions by providing a visualization of the DAG, validating the program against a representation of the DAG, or verifying certain graphical properties of the program like its hierarchical structure.
Chapter 5: “Pedantic Mode”: Detecting Statistical Issues in Probabilistic Programs

This section presents “Pedantic Mode”, an extension for the Stan compiler that uses Factor Graph extraction and other static analysis methods in order to warn users about potential statistical issues in their program source code. Pedantic Mode combs over a Stan program looking for indications of common errors, in order to expose any problems with the user’s probabilistic model or implementation before those problems cause erroneous results after inference. Pedantic Mode is akin to a linting tool, except that it looks for probabilistic modeling and inference issues rather than programming issues, and some of its analyses involve the whole program while linters typically only use local context. By automatically discovering potential problems in candidate models without having to wait for the results of inference or post-inference diagnostic methods, we gain faster development iteration times, a lower barrier to entry for new users, and a lower likelihood of misleading results.

Pedantic Mode is composed of several individual analyses that each look for different issues with different static analysis methods. Some of those analyses evaluate the whole-model structure of the program using the method introduced in Chapter 4: Factor Graph extraction. By reducing the program to an undirected graphical model, we can look for graphical patterns that may indicate issues with the probabilistic model.

For a toy example, consider the following Stan program:

data {
    int N;
    real x[N];
}

parameters {

This program models the elements of a real vector $x$ as being normally distributed around $1000$ with an unknown standard deviation $\sigma$. When pedantic mode is turned on, the compiler will produce the following warnings:

Warning:

The parameter $\sigma$ has no priors.

Warning at ‘ped-mode-ex1.stan’, line 10, column 14 to column 18:

Argument $1000$ suggests there may be parameters that are not unit scale; consider rescaling with a multiplier (see manual section 22.12).

Warning at ‘ped-mode-ex1.stan’, line 10, column 20 to column 25:

A normal distribution is given parameter $\sigma$ as a scale parameter (argument 2), but $\sigma$ was not constrained to be strictly positive.

Pedantic Mode has been included as an optional part of the Stan compiler since 2020, and has been active by default for users of Stan’s Python interface since 2021. Its source code is available online\(^1\).

### 5.1 Pedantic Mode analyses

The following sections discuss the types of issues found by Pedantic Mode. For each type of issue we give example code, explain the static analysis methodology, and discuss how the analysis differs from typical linting.

Of particular interest is Section 5.1.3, which uses Factor Graph extraction to detect cases where a prior distribution is missing or is over-defined.

\(^1\) [https://github.com/stan-dev/stanc3/blob/master/src/analysis_and_optimization/Pedantic_analysis.ml](https://github.com/stan-dev/stanc3/blob/master/src/analysis_and_optimization/Pedantic_analysis.ml)
The set of analyses included in Pedantic Mode was gathered from expert users such as Andrew Gelman [91], as well as a poll of the online Stan community.

5.1.1 Random variable is missing from the joint distribution

A Stan program should represent a joint distribution of its data and all of its parameters. If a parameter has no effect on the density of the joint distribution, Stan’s inference algorithm has no information about that parameter’s distribution, which may cause numerical instability. In addition, a parameter with no distribution is useless, so the user likely made a mistake. A trivial example:

```stan
parameters {
   real a;
   real b;
}
model {
   a ~ normal(1, 1);
}
```

Here, $a$ participates in the density function but $b$ does not. Pedantic Mode produces the following warning.

Warning:
The parameter $b$ was declared but was not used in the density calculation.

It is not trivial in general to find the set of parameters that do not affect the joint density: false-negatives can occur when a parameter is used, but not in a meaningful way; false-positives can occur when the parameter affects the density through a chain of other variables. To distinguish these cases, Pedantic Mode uses a program-wise dependence analysis to trace any dependence from the special target variable to on each parameter.

5.1.2 Random control flow

Stan’s inference algorithm assumes that the joint density function is differentiable in each of its parameters. Stan, however, is a very general programming language, and it is very possible
to introduce discontinuity in the density via control flow features like if, for and while. For example, a program with a parameter \( x \) will be discontinuous at \( x=10 \) if a user writes

\[
\text{if}(x<10) \quad \{ \text{target} += 1 \} \quad \text{else} \quad \{ \text{target} += 2 \}
\]

On the other hand, control flow is safe and useful when the branching statements do not depend on parameters.

Pedantic Mode generates a warning when any branching condition may depend on a parameter value. For example, consider the following program:

```plaintext
parameters {
    real a;
}
model {
    // x depends on parameter a
    real x = a * a;

    int m;

    // the if-then-else depends on x which depends on a
    if(x > 0) {
        // now m depends on x which depends on a
        m = 1;
    } else {
        m = 2;
    }

    // for loop depends on m -> x -> a
    for (i in 0:m) {
        a ~ normal(i, 1);
    }
}
```

The if and for statements are control flow statements that depend (indirectly) on the value of the parameter \( m \). Pedantic Mode produces the following warning.
Warning at ‘param-dep-cf-warn.stan’, line 11, column 2 to line 16, column 3:
A control flow statement depends on parameter(s): a.
Warning at ‘param-dep-cf-warn.stan’, line 19, column 2 to line 21, column 3:
A control flow statement depends on parameter(s): a.

Just like the missing-parameter analysis, this method uses dependence analysis to predict issues of probabilistic inference.

5.1.3 Parameters with zero or multiple priors

In Bayesian modeling, each parameter is assigned a prior distribution. In Stan, when a prior distribution is omitted, the prior is assumed to be uniform, but it is better practice to write out the prior explicitly. Further, it is rarely meaningful to assign more than one prior distribution to a parameter. Therefore, whenever a parameter has no prior or multiple priors, it is an indication of a modeling mistake or a coding mistake.

The difficulty is in defining a “prior” to the compiler. Statistically, a prior is a term $P(\theta)$ as in the equations $P(\theta \mid D) = \frac{P(D \mid \theta)P(\theta)}{P(D)}$ and $P(\theta, D) = P(D \mid \theta)P(\theta)$, for data $D$ and parameters $\theta$. One reasonable definition of a prior is then in contrast to the likelihood: a prior is a factor of the joint distribution $P(\theta, D)$ that does not depend on the data $D$.

A further difficulty is in how to define “data” for the purposes of finding a prior. One reasonable condition for whether an expression is “data” is whether its value depends on any of the variables defined in Stan’s data or transformed data blocks. However, sometimes parameters do not contribute to the model likelihood by directly modeling data, but rather by modeling other parameters, which themselves model the data. We should also treat these “closer to the data” parameters as data for the purposes of distinguishing likelihood terms from prior terms.

Consider the two Stan programs in Figure 5.1. In program A, the second sampling statement, $b \sim \text{normal}(a, 1)$, is considered a prior for $b$ because it has no dependence path to data $x$ except through $b$. In program B, however, that same sampling statement now has a dependence path to $x$ through $a$, since $a$ has a direct connection to $x$ through the statement $x \sim \text{normal}(b,$
Figure 5.1: Two similar programs that result in different “prior” assignments by Pedantic Mode.

(a) Program A

(b) Program B

Because we have defined a prior as a factor with a particular relationship to other factors, the Factor Graph of a program has exactly the information we need. Our procedure will then be to build a Factor Graph in the same way as in Chapter 4, and then search it for patterns that match our definition of a prior.

The pattern is as follows: a factor $F$ that neighbors a parameter variable $P$ is called a prior factor of $P$ if there is no path in the factor graph from $F$ to any data variable except through $P$. This pattern matches our definition of “data”: if there is a path from $F$ to a data variable $D$ that does not pass through $P$, then the value of $F$ depends on a data variable or a parameter that itself depends on data. Once we have collected all prior factors in this way, we can report a warning for each parameter that has zero or multiple prior factors.

As an example, Figure 5.2 shows the factor graphs for the example programs A and B, along with the path that disqualifies the second sampling statement of program B from being a prior.

There are two likely points of confusion with this definition of a “prior”:

1. data variables are sometimes not “data” variables.

Unfortunately, Stan does not offer a way to distinguish whether a data variable is a true, “modeled” data variable intended to represent a dataset or a variable intended to be a hyper-parameter or data size. So, this analysis will sometimes mark a factor as a “likelihood” factor.
Figure 5.2: Shown here are factor graphs, where circular nodes represent random variables, rectangular nodes represent factors that contribute to the joint density of the variables, and edges represent a factor's direct dependence on a variable. Filled circular nodes represent data variables, while empty circled nodes represent parameters. If an edge between a factor and a parameter is circled in blue dashes, it indicates that the factor is a prior for that parameter. Subfigures A and B represent programs A and B. In A, factors #1 and #2 are priors. In B, a new relationship is introduced between a and factor #3, which adds a path, shown in red arrows, from factor #2 to the data variable x; factor #2 is therefore no longer a prior for b.

because it depends on a hyper-parameter, thus causing confusion. This could be avoided with a language feature to distinguish true data variables.

2. Undirected interpretation of parameter relationships.

In Stan, distributions can be assigned in two ways: by ~ statements, or by target+= statements. The former is more similar to statistical notation, but the latter is more flexible, and is close to what most of the compiler actually uses.

When a user writes \( a \sim \text{normal}(b, 1) \), they see this as \( b \) affecting the distribution of \( a \), rather than the other way around. However, Stan converts this statement to a function call, \( \text{target}+= \text{normal}_\text{lpdf}(a, b, 1) \), which muddles that directionality. Further, a target+= statement can be used with any expression, like \( a * b \), that lacks any directionality. The factor graph, and therefore this analysis, takes the view that all factors, even apparently directed statements like \( a \sim \text{normal}(b, 1) \), are undirected.

This fact can cause some unintuitive results. For example, \( a \sim \text{normal}(b, 1) \) can be considered a “prior” for \( b \), rather than \( a \), if \( a \) has no connection to a data variable. Another
example is that \( a \sim \text{normal}(b, 1) \) could be considered a “likelihood” for \( a \) rather than \( b \), if \( b \) has a connection to a `data` variable, where normally a likelihood has the “data” variable on the left-hand side.

There are two solutions to this confusion. The first is to attempt to determine the “directionality” of factors by the methods laid out in Chapter 4. However, that method can be computationally expensive, complex to implement, and does not work automatically in every case. These drawbacks make it inconvenient for a tool like Pedantic Mode.

The second solution is to check whether the undirected prior analysis agrees with the directionality of sampling ~ statements written in the original program, and if it does not, then to warn the user that the results may be confusing.

For an example of this analysis, consider the following program:

```plaintext
data {
    real x;
}
parameters {
    real a;
    real b;
    real c;
    real d;
}
model {
    a ~ normal(0, 1); // this is a prior
    x ~ normal(a, 1); // this is not a prior, since data is involved

    b ~ normal(x, 1); // this is also not a prior, since data is involved
    // this is not a prior for c, since data is involved through b
    // but it is a prior for b, since the data is only involved through b
```
c ~ normal(b, 1);

//these are multiple priors:
d ~ normal(0, 1);
l ~ normal(d, 1);
}

One prior is found for a and for b, while c only has a factor that touches a data variable and
d has multiple priors. Pedantic Mode produces the following warning:

Warning:
The parameter c has no priors.
Warning:
The parameter d has 2 priors.

5.1.4 Ill-advised use of statistical distributions

Stan includes a large library of pre-defined statistical distributions. Pedantic Mode looks for
two types of issues with how a program uses a distribution.

Firstly, Pedantic Mode attempts to warn the user when they violate distribution assumptions. A
statistical distribution may come with a set of assumptions about the arguments it is given and the
variate whose distribution it describes. Often, those assumptions cannot be expressed within the
Stan type system, so users can accidentally violate them without compilation errors. For example,
consider the following program:

parameters {
    real unb_p;
    real<lower=0> pos_p;
}
model {
    1 ~ poisson(unb_p);
    1 ~ poisson(pos_p);
}
The parameter of poisson should be strictly positive, but the parameter unb_p is not constrained to be positive. Pedantic Mode produces the following warning:

Warning at 'ex-dist-args.stan', line 6, column 14 to column 19:
A poisson distribution is given parameter unb_p as a rate parameter (argument 1), but unb_p was not constrained to be strictly positive.

Pedantic Mode will also check whether each argument and variate’s subtype (such as the simplex type, which imposes additional constraints on a variable) and value (if it is known at compile-time) are compatible with the specification of the distribution.

It is not possible to know the values of some variables at compile-time, so Pedantic Mode must sometimes give false-negatives when checking variable values and bounds.

Secondly, Pedantic Mode will attempt to point out when a user applies a distribution in certain ways that experts would not advise. For example a user might give a variable a uniform prior distribution to act as an “uninformative” prior, but the uniform distribution can create non-differentiable boundary conditions which cause issues during sampling, unless the variable was declared with matching bounds. Consider the following program:

```stan
parameters {
  real a;
  real<lower=0, upper=1> b;
}
model {
  a ~ uniform(0, 1);
  b ~ uniform(0, 1);
}
```

Pedantic Mode produces the following warning:

Warning at 'uniform-warn.stan', line 6, column 2 to column 20:
Parameter a is given a uniform distribution. The uniform distribution is not recommended, for two reasons: (a) Except when there are logical or
physical constraints, it is very unusual for you to be sure that a parameter will fall inside a specified range, and (b) The infinite gradient induced by a uniform density can cause difficulties for Stan’s sampling algorithm. As a consequence, we recommend soft constraints rather than hard constraints; for example, instead of giving an elasticity parameter a uniform(0,1) distribution, try normal(0.5,0.5).

Pedantic Mode will also produce warnings for Gamma and Inverse-Gamma distributions, which are sometimes used to assign improper priors to a parameter, and for \texttt{lkj_corr} distributions, which usually have a more numerically stable alternative.

The static analysis methods necessary for this section are not unusual among other linters, but the motivations, especially for special-case distributions, are unusual in that they aim to help the user avoid difficult-to-spot, modeling-specific pitfalls. For example, the warnings for uniform distributions warn the user about issues specific to Stan’s inference algorithm.

5.1.5 Poorly scaled random variables

Scaling a random variable means multiplying and offsetting it so that its mean and variance are similar to a standard normal distribution. Scaling is considered a best practice for numerical stability. Pedantic Mode will point out when a user appears to be assigning a distribution to a variable without first scaling that variable to the unit range. For example:

```stan
parameters {
  real x;
  real y;
}
model {
  x ~ normal(-100, 100);
  y ~ normal(0, 1);
}
```

The constants \(-100\) and 100 suggest that \(x\) is not unit scaled. Pedantic Mode produces the following warning:
Again, the static analysis methodology here is not unusual, but the aim is unusual: tailoring the program to work well with Stan’s probabilistic inference methods.

5.1.6 Parameters with multiple sampling statements

Typically in statistical modeling, a random variable is only assigned one distribution. While convention has less meaning in Stan, which cares only about densities, more than one assigned distribution may indicate that the user has made a mistake. This provides overlapping but distinct functionality to the “Zero-or-multiple Priors” analysis.

Pedantic Mode will warn the user when they have repeated sampling-style statements. For example:

```stan
data {
  real x;
}
parameters {
  real a;
  real b;
}
model {
  a ~ normal(0, 1);
  a ~ normal(x, 1);
  b ~ normal(1, 1);
}
```
Pedantic Mode produces the following warning.

Warning at ‘multi-twiddle.stan’, line 9, column 2 to column 19:
The parameter a is on the left-hand side of more than one twiddle statement.

This is a very simple static analysis, but it is unusual in that it codifies a statistical convention to avoid accidents.

5.1.7 Strict or nonsensical parameter bounds

Except when there are logical or physical constraints, it is very unusual for a model to be certain that a parameter will fall inside a specified range. Pedantic Model generates a warning for all parameters declared with the bounds <lower=.., upper=..>, except for <lower=0, upper=1> or <lower=-1, upper=1>, which are considered the standard ranges for typical data types like binary data or credences.

For example, consider the following program.

parameters {
  real<lower=0, upper=1> a;
  real<lower=-1, upper=1> b;
  real<lower=-2, upper=1012> c;
}

model {
  c ~ normal(b, a);
}

Pedantic Mode produces the following warning.

Warning:
Your Stan program has a parameter c with a lower and upper bound in its declaration. These hard constraints are not recommended, for two reasons:
(a) Except when there are logical or physical constraints, it is very unusual for you to be sure that a parameter will fall inside a specified
range, and (b) The infinite gradient induced by a hard constraint can cause
difficulties for Stan’s sampling algorithm. As a consequence, we recommend
soft constraints rather than hard constraints; for example, instead of
constraining an elasticity parameter to fall between 0, and 1, leave it
unconstrained and give it a normal(0.5,0.5) prior distribution.

5.2 Limitations and discussion

Pedantic Mode has a number of limitations that may cause false-positives or false-negatives. For example, some analyses depend on the compiler knowing the value of an expression, such as the distribution arguments in Section 5.1.5. Pedantic Mode attempts to evaluate expressions down to literal values so that they can be used to generate warnings. For example, in the code
\texttt{normal(x, 1 - 2)}, the expression \texttt{1 - 2} will be evaluated to \texttt{-1}, which is not a valid variance argument, so a warning is generated. However, this strategy is inherently limited; it is very often impossible to fully evaluate expressions at compile time.

The above limitation may cause false-negatives. Other causes of false-negatives including limitations in the Stan implementation: for container types, index ranges are not considered, so containers are considered monoliths; for data-block variables, declaration bounds are not kept, so incompatibilities may not be cause; for user-defined functions, it is sometimes difficult to know which arguments are parameters, so control flow that depends on arguments may not throw an appropriate warning.

While Pedantic Mode’s false-negatives tend to come from fundamental constraints of static analysis, most of the false-positives (or perhaps “unexpected positives”) come from an ambiguity in the way the Stan language represents user intent. These ambiguities, such as whether a data-block variable is true “modeled” data or not, and whether a given factor should be considered a “directed” distribution, could be resolved by adding compiler annotations to Stan.

Luckily, unlike some other applications of static analysis, linter-like tools can have false-positives and false-negatives like these while remaining useful. Its limitations notwithstanding, Pedantic Mode appears anecdotally to have been a practical addition to Stan, and has received
generally positive feedback from its user base.

From a technical perspective, the majority of the analyses in Pedantic Mode require only static analysis methods that would be found in a typical linter. However, some of the challenges, like the factor graph evaluation in Section 5.1.3 and the random variable control flow in Section 5.1.2, are unique to the considerations of probabilistic programming and Stan’s structure and inference algorithm.
Chapter 6: Future Work

There are many ideas that could not be crammed into this dissertation. Mostly, they involve model-space operations which use multi-model probabilistic programming as a foundation. In this section, I discuss some of those ideas, as well as other extensions of this work, and how it might be integrated with emerging technologies.

6.1 Advanced model-space search and ensemble methods

In Section 3.9.3, I presented a naive, brute-force method of discovering models in model space: greedy graph search. While this straightforward approach can be useful, it is also likely subject to local optima, may select over-fit models, and can only optimize a metric, which is not always the goal of model discovery.

One of the problems with the greedy graph search as presented was the search criteria being optimized, ELPD. Optimization of ELPD is likely to over-fit. In addition, ELPD does not always translate between models, so it will not be an appropriate choice for all model families. While it may always be best for the user to tailor their search criteria to their particular problem, it may also be a fruitful area of research to find model evaluation metrics that tend to produce useful models.

Another approach to the search criterion is to use concepts like projection prediction [2]. In projection prediction methods, a user defines a “gold standard” model that performs well but is in some way too cumbersome (as in computational performance or conceptual complexity, such as an ensemble model), and aims to find a “simple” model that most accurately replicates the performance of the gold standard. Searching model space for this “simple” model should reduce the risk of over-fitting to the data, since we are instead trying to match another model.

Greedy graph search is in part inefficient because it does not try to predict which next models
will score well; it only guesses that neighbors of high-scoring models might also be high-scoring. Instead, our search could attempt to predict whether the next model will be high-scoring based on the other information available about it: the set of modules that it includes. Over the course of search, as we assign scores to models, we could attempt to learn which modules tend to be included in higher-scoring models, and then use that knowledge to find the next candidate models. This learning could be done with its own probabilistic model, like a Bayesian regression where the contributions of each module are represented by random variables, so that we could infer a posterior distribution over the score of each unevaluated model.

If we consider the time spent evaluating models as a kind of resource, we can view model search as a many-armed bandit problem: we have many possible levers to pull (models to evaluate), each one costs a coin to operate (evaluation time), and when we pull a lever, we learn more about the lever’s reward (we narrow our distribution of the model’s true score). This is an exploration vs. exploitation problem: we want to spend more time evaluating high-scoring models (exploitation) but we also want to learn more about unknown models and modules (exploration). With this view, instead of naively searching model space, we can employ a procedure such as Thompson sampling [90] to optimally balance exploration and exploitation.

We can also view the model search problem as a kind of symbolic regression for probabilistic models. Some existing methods of symbolic regression may be stolen for model search, like genetic algorithms to find module compositions that produce high scores.

An alternative to discovering models through search is to assemble new models out of those found in the network: these are called ensemble models or “stacks” of models. Given weights for each of a set of models, stacking methods can produce a new model that combines their inference power. The challenge is to select a set of models from the network of models and assign them weights. Some of the approaches that work for model search, like assigning models a predicted quality score by its composition of modules, may also be useful for this challenge.
6.2 Model-space navigation and visualization tools

As model developers iterate on their models, they can be seen as navigating the model-space of solutions to their problem. Multi-model programs can serve as a kind of map of this model space. Future methods could help make this map richer and easier to read.

For example, the network of models produced by Modular Stan does not label its edges, except to say what code has changed between the two endpoints. In theory, we could answer many questions about the relationship between two similar models: How do they differ in their prior or posterior predictions of the parameters? Of the data? Which model is more flexible? If fake data were drawn from one model, could the other recover its parameters? We could annotate the edges with the answers to any of these questions, giving the user a better sense of their relationship and of the space as a whole.

Further, when edge annotations take any form that could be considered a “distance”, we can treat model space as a metric space, and produce visualizations for the user that show similar models clustered together.

6.3 Engineering improvements

With the exception of Pedantic Mode, the implementations presented in this dissertation should be considered incomplete prototypes. Following are some of the directions in which multi-model probabilistic programming could be extended:

Explicit module type signatures. Though implicit module signatures, such as those found in Modular Stan, are beginner-friendly, having the option for explicit signatures may make module reuse easier, by better specifying their interface.

Module libraries. With explicit type signatures, it should be possible to build libraries of standard components of probabilistic models.

Structural constraints. While the module graph and macro systems offered by swappable mod-
ules and Modular Stan are expressive ways to specify a family of models, there could in theory be model families with arbitrarily complex rules about which modules make sense to co-exist within a model. Modular Stan could be extended to offer a logic of “structural constraints” for more flexible module coexistence rules.

**Sequential inference.** Some work has been done within the Stan community to improve the efficiency of probabilistic inference when highly similar models are learned one after the other: the inferred posterior of the earlier model acts as a kind of “warm-up” for the later ones. This kind of optimization could be used automatically for model-space algorithms like search that perform inference for many related models in sequence.

### 6.4 Going beyond Stan

This dissertation demonstrates multi-model probabilistic programming by implementing “swappable modules” in Stan because Stan is a popular and powerful language, however, “swappable modules” is just as applicable to other languages as it is to Stan. Augmenting any other language would let that language represent networks of models in the same way.

### 6.5 Broader vocabulary for automated program transformations

The system of automatic model transformations demonstrated in Chapter 4 could in theory be used for a wide variety of applications. The particular transformations demonstrated are already practical applications, but we could use a similar method to implement many other universally defined transformations.

Further, if we had well-defined vocabulary for the typical transformations that models undergo over the course of development, like “add a prior to a parameter”, or “move from partial pooling to full pooling”, we could apply this kind of automation for model development itself.
6.6 Large language models and program synthesis

We should expect that large language models will soon be able to generate and iterate on scientific hypotheses in the form of probabilistic models written as programs. For a given problem description, they will be able to generate many plausible models and model components.

This wealth of potentially useful but unproven models will only increase the importance of model diagnostics, code quality feedback, expressive representations of alternative models, and methods to explore and evaluate model spaces.
Conclusion

Throughout this dissertation I presented abstractions and algorithms to support model development with probabilistic programming. In particular, I aimed at two kinds of limitations with the abstraction presented by traditional probabilistic programming. The first limitation is that probabilistic programs only represent one model. In Chapter 3, I presented a method to augment probabilistic programming languages with a meta-programming abstraction called Swappable Modules, so that programs can efficiently represent networks of probabilistic models. In Section 3.6, I specified algorithms, along with proofs of correctness and efficiency, that enable programs with Swappable Modules to be used as undirected graphs of programs. I implemented Swappable Modules for Stan, resulting in Modular Stan, and used Modular Stan to demonstrate some of myriad of useful applications for multi-model probabilistic programs: representation, presentation and organization of model development, automated model-space tasks like model search, and model-space diagnostic methods like sensitivity analysis. These multi-model program semantics bring more of the modeling workflow into the scope of software.

The second limitation is that probabilistic programs, especially those written in relatively flexible languages like Stan, are unstructured. In Chapter 4, I described a way to decompose a Stan program into the factors that contribute to its density function, and arrange them into a factor graph. In Section 4.4.1, I presented a method to re-assemble those factors into a new form, thereby transforming the original model. In Chapter 5, I presented a method to give users feedback on the structure of their model based on the topology of the factor graph. Also in
Chapter 5, I detailed a variety of static analysis methods that decipher aspects of the model underlying a Stan program to give users statistical feedback.

The multi-model probabilistic programs of Chapter 3 afford users a precise representation of each transformation they apply to their models as they iterate through the workflow, resulting in a concrete representation of the model space for their task, where normally they would be left searching without a map. This concrete representation is invaluable for humans, but also for algorithms that can automate other parts of the workflow like multiverse analysis and model stacking.

The automated transformations of Chapter 4 make it easier for users to implement diagnostic methods, thereby lowering the barrier to entry for robust model development.

The automated user feedback of Chapter 5 may catch a modeling or inference issue any time a user writes or iterates on a model; when it does, it shortens the iteration time of the workflow, and may even prevent erroneous results.

Taken together, these methods can alleviate much of the barrier to entry for the Bayesian Workflow noted in Section 1.2.

While I expect that the technical landscape surrounding probabilistic modeling will continue to change, and that many details of these methods will become obsolete, I expect that the goals and considerations of these methods will remain relevant for as long as we use software to assist with probabilistic reasoning.
References


[12] A. Gelman and E. Loken, “The garden of forking paths: Why multiple comparisons can be a problem, even when there is no "fishing expedition" or "p-hacking" and the research hypothesis was posited ahead of time,” 2019.


