TECHNISCHE UNIVERSITEIT DELFT

MASTER OF SCIENCE THESIS IN COMPUTER SCIENCE

Automating Incremental Inference for Probabilistic Programs

Lola DEKHUIJZEN

Supervisor:
Dr. Sebastijan DUMANČIĆ

Daily Co-Supervisor:
Reuben GARDOS REID

22nd January 2026

Delft University of Technology

Automating Incremental Inference for Probabilistic Programs

Master's Thesis in Computer Science

Algorithmics group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Lola Dekhuijzen

22nd January 2026

Author
Lola Dekhuijzen

Title
Automating Incremental Inference for Probabilistic Programs

MSc presentation
29th January 2026

Graduation Committee

Dr. Sebastijan Dumančić (chair)	Delft University of Technology
Dr. Benedikt Ahrens	Delft University of Technology

Abstract

This thesis proposes a method that leverages incremental inference to improve inference efficiency in complex probabilistic programs, providing an algorithm-agnostic approach that does not rely on any single sampling method. The method builds on an existing incremental inference framework, which samples from one probabilistic program and uses the results to perform inference on a related program. Our approach uses this framework to sample from simplified programs, thereby bypassing direct inference on complex programs. These simplifications are constructed by automatically detecting certain program patterns, both structural and parametric, that increase program complexity, and applying corresponding changes to simplify them. On a set of input programs, we empirically show which patterns increase complexity and demonstrate how the constructed changes achieve more efficient inference than baseline sampling methods.

Preface

I would like to express gratitude to everyone who, in one way or another, supported me throughout my master's.

A special thanks to Sebastijan and Reuben for all your ideas, feedback, and patience.

To my family, for the endless support and for always having my back.

To friends, both those who were there from the beginning and those who were met along the way, for the memories, company, and welcome distractions.

And last but certainly not least, to Luka, not only for feeding me with delicious food but also for providing me with endless emotional support.

Lola Dekhuijzen

Amsterdam, The Netherlands
22nd January 2026

Contents

Chapter 1

Introduction

Alice is planning a visit to her grandma, who is of poor health, and wants to ensure beforehand that she does not have COVID-19. To do so, she identifies various factors that influence the likelihood of her being infected. Examples of such factors are whether she has knowingly been in contact with someone who is infected, whether she has been vaccinated, and whether she has shown any of the known symptoms. Alice cannot determine exactly whether she has COVID-19: the identified factors only tell her how likely it is that she does.

In order to combine the given information and form a prediction, one can make use of a statistical model. Such a model captures uncertainties by describing how the relevant factors influence both each other and the outcome we are interested in, even when that outcome cannot be determined exactly. In this case, whether Alice has COVID-19. Probabilistic inference is then the process of reasoning about such a model. Probabilistic programming languages (PPLs) [van de Meent et al., 2021] allow us to both represent probabilistic models and automate inference on them.

A naive method for inference is to consider all possible assignments of factors that influence the outcome, weigh them by their respective probabilities, and obtain a distribution over the possible outcomes. Consider the factor of whether Alice has been vaccinated. Possible assignments are yes and no, and in the scenario where Alice has been vaccinated, these assignments are weighted by one and zero, respectively. As the number of possible assignments grows, this procedure becomes intractable. Instead, we turn to approximate inference: rather than enumerating through all possible combinations, we draw a number of samples to approximate the distribution over potential outcomes.

Often, we obtain data that helps us determine the outcome. Alice might find out that her best friend has been infected, which increases the chance that Alice herself is infected. We are no longer interested in all possible assignments, but only those in which Alice's friend is infected. In other words, the distribution of possible outcomes changes after an observation. A simple sampling method rejects the samples that do not agree with the observations and approximates the outcome using the remaining samples. In trivial examples, such a method may suffice. However, when

observations are unlikely to occur, many samples are discarded. Consequently, we need more samples to still obtain enough information, or smarter sampling methods. Much research has been done to improve these methods [van Krieken et al., 2023, Li et al., 2025].

Which sampling method is best suited for a problem depends on the specifics of that problem. Rather than attempting to improve any of these specific methods, we turn to incremental inference, a framework that provides a way to more efficiently sample from non-trivial problems, i.e., in a way that requires fewer samples to obtain an accurate result, independent of which sampling method is used.

1.1 Why Incremental Inference?

The main reason to use incremental inference is that it allows us to approach a complex problem by first solving a simplified version of that problem and using the results to solve the more complex problem. Hereby, we avoid solving the complex problem directly. This is by no means a new concept: it occurs in various fields of physics. In perturbation theory, for example, a complex system is approximated by an exactly solvable, simplified version, to which perturbations are added step by step to build towards an approximate solution to the original problem [Fillion and Corless, 2022].

The core idea that makes up incremental inference is to take samples from one probabilistic program and reweigh them such that they can be used for inference on a similar program, thereby avoiding directly sampling from the first program. A simple use case of this technique is to be able to solve a set of similar programs without needing to begin from scratch each time, and save computation as a result. The key insight is that if we modify the complex program to create a similar program that is easier to sample from, we can use the same technique to perform inference on the complex program more efficiently, i.e., fewer samples need to be drawn to accurately represent its distribution.

To illustrate what simplifying a problem in this way may look like, we revisit the example of Alice. If Alice were to map out everyone she has been in contact with in the past week, from her mother to the grocery clerk, she would end up with a complex problem. If she instead decides to select only the few people whom she has seen for an extended period of time, she may not get a result that is as exact, but the simplified problem will be easier to solve. She could stop there and take the solution to the simplified program as her final answer. But Alice is determined to guarantee the health of her grandma and does not want to ignore any information that she has. Intuitively, we can think of incremental inference as taking the solution of the simplified problem and using the additional information we have to tweak the result such that it more closely resembles the result of the complex problem, without solving the complex problem directly.

In conclusion, the concepts that incremental inference relies on have been implemented successfully in different areas of research and have the potential to save

computation cost. An added benefit of the approach is that it does not depend on what sampling method is used to solve any given problem. This allows us to benefit from existing research that is focused on improving sampling methods, while simultaneously offering an algorithm-agnostic approach to further improving those methods.

1.2 Research Questions

In order to apply incremental inference effectively, we must know in what ways we can simplify a probabilistic program. This, in turn, requires knowing what makes a probabilistic program difficult. Once we identify general patterns that make a program difficult and find ways to modify the program to make it easier, we want to be able to automate this process of identification and modification for a given probabilistic program. To summarize, the research questions are as follows:

- What characteristics of a probabilistic program make inference difficult?
- How can a probabilistic program be modified to make inference more efficient, i.e., requiring fewer samples?
- How does the accuracy of inference compare to the baseline method for the same number of samples?
- How can difficulty detection and program modification be automated?

The rest of the thesis is structured as follows:

Chapter 2 provides background information about probabilistic programming and introduces various sampling methods. Chapter 3 discusses previous work that either serves as a stepping stone or is tangential to this work in some way. Chapter 4 provides the necessary notation and offers a formal problem definition. Chapter 5 outlines the devised algorithm and the intuition behind it. Chapter 6 describes the conducted experiments and their corresponding results, while Chapter 7 provides reasoning for those results. Finally, conclusions and directions for future work are discussed in Chapter 8.

Chapter 2

Background

2.1 Probabilistic Programming

Probabilistic Programming (PP) is a programming paradigm that allows us to automate the inference process of probabilistic models. In probabilistic programming languages (PPLs), probabilistic models are represented as functions within a program. These functions are different from normal functions in that they are stochastic – variables no longer have to be assigned a deterministic value, but may represent random variables instead. We refer to the assignments of these variables as sample statements. Whenever a function is called, each of these random variables may take on a different value, sampled from the distribution they represent. We refer to this as a trace, i.e., a trace consists of the specific values that are sampled during a single execution of a function. Alongside sample statements, probabilistic programs may also consist of observe statements. These allow us to fix random variables to specific values, thereby influencing the distribution of other random variables. The return statement contains the random variable whose posterior distribution we are interested in, conditioned on the observations. Listing 5.1 shows an example of a simple probabilistic program containing these elements.

Listing 2.1: Alice's example

```
illness = bernoulli(0.01)
sneeze = illness ? bernoulli(0.9) : bernoulli(0.01)
observe(sneeze = true)
return illness
```

2.2 Inference Methods

PP allows us to not only encode probabilistic models, but also offers built-in inference algorithms. Probabilistic inference is the process of calculating the posterior distribution we are interested in. Revisiting Alice's dilemma, Listing 5.1 models a simple scenario in which the posterior distribution tells us how likely it is that Alice

is ill, given the observation that she has sneezed. The most naive inference method is enumeration, which consists of considering every possible trace that agrees with the observation, and weighing them by how likely they are to occur. While Listing 5.1 consists of only four traces, and half of those agree with the observation, enumeration quickly becomes intractable as models grow in complexity.

Rather than summing over all possible traces, we use sampling to obtain an approximation of the posterior distribution instead. Rejection sampling is a simple sampling method that draws samples from the prior and only keeps those samples that agree with the observations. In our example, most samples will assign `illness = false`, and consequently `sneeze = false`. Most samples will disagree with the observation that `sneeze = true` and will therefore be rejected. As a result, when an observation is unlikely to occur, a large number of samples are required to obtain a good approximation of the posterior, making rejection sampling inefficient for complex probabilistic programs.

Importance Sampling (IS) addresses this issue by a weight to each sample, reflecting its likelihood under the posterior, rather than discarding it. This weighting allows us to better approximate the posterior, often with fewer samples. While IS offers a general improvement upon rejection sampling, more advanced methods, such as Markov Chain Monte Carlo (MCMC) and Sequential Monte Carlo (SMC), can be more efficient in certain settings, given that they are suitable for the problem and properly tuned.

SMC differs in that it maintains a collection of samples, called particles, and updates their weights sequentially. Each time an observation occurs, the particles are reweighted according to how well they match the observation. A resampling step samples each particle with a probability proportional to its weight, leading to low-weight particles being discarded and high-weight particles being duplicated. Due to its sequential nature, SMC is only suitable when the model can be executed incrementally.

MCMC constructs a chain of dependent samples, where each new sample depends only on the previous one. At each step, a new sample is proposed by modifying a subset of the random choices of the previous sample and is accepted or rejected according to a rule that guarantees convergence to the posterior. Convergence can be slow, depending on how the changes, meaning which random choices are updated and how, are determined. Different strategies correspond to different MCMC variants, and the choice of which to use depends on the specific problem.

Rather than selecting a more advanced method and tuning it for a specific problem, we use IS and draw on the intuition of simpler sampling methods to develop a generalized method for identifying difficult parts of a program and designing the appropriate modifications.

Chapter 3

Related work

3.1 Incremental Inference

While incrementalization has been considered previously in the context of probabilistic programming, notably in the form of sequential Monte Carlo (SMC) which accounts for evidence in an incremental manner, Cusumano-Towner et al. [2018] were the first to introduce a generalized framework of incremental inference for probabilistic programs. Stuhlmüller et al. [2015] achieved a similar goal, though their focus is narrowed to sequences of programs in which the data is generated in increasing detail. Finally, the work of Zhang et al. [2017] differs from the aforementioned in that it takes a static approach.

Incremental Inference for Probabilistic Programs

Often, when we have an existing probabilistic model, we may need to adapt it upon changes in the data the model is conditioned on or changes in the prior knowledge. To avoid redoing inference from scratch each time, Cusumano-Towner et al. [2018] introduced a framework for performing incremental inference on probabilistic programs. The main idea is to adapt samples of some program and use those for inference on a second program that corresponds to the first in some way, thereby avoiding sampling from the second program directly.

The approach builds on top of a correspondence between pairs of programs P and Q, where random choices in P are linked to random choices in Q. The paper identifies one way to obtain such a correspondence: through program edits, i.e. program P differs from program Q by one or more small changes. Translation between these programs can be done efficiently by building a dependency graph of each trace and only reevaluating nodes that are changed, propagating changes along the dependency graph.

A use-case not explicitly mentioned by the authors is the following: we can use incremental inference to perform inference on a complex probabilistic program more efficiently. We simplify the program in some way, then perform inference

on the simplified program, and finally translate its traces to the more complex program. We can repeat these three steps multiple times, each time changing the program slightly, e.g. by applying a program edit.

Moreover, we can use a dependency graph to automatically find ways to simplify a program. A program is more difficult to sample from when the evidence favors improbable paths. One way to aid sampling is to alter the program such that it agrees with the evidence. Using a dependency graph, we can perform a bottom-up search during which we keep track of observe statements and detect statements that depend on the observes. Then, we can modify those statements to agree with the observe statement(s), consequently making the paths we are interested in more easily reachable.

Coarse-to-Fine Sequential Monte Carlo

Stuhlmüller et al. [2015] introduced an algorithm that takes a probabilistic program as input and generates a sequence of approximating distributions, ranging from "coarse" to "fine". In each next step, more details of the state space are resolved. Or, in the opposite direction, elements of the program are "lifted": they are replaced by approximations of themselves. The user must provide a mapping from values to abstractions of those values.

The algorithm explores high-probability regions early on by using heuristic factors. These are factor statements that assign scores to certain traces and are canceled out in some finer level by factor statements with negation of the same score, thereby leaving the final distribution unaltered.

One shortcoming of the algorithm is that it requires a transformation to ensure that random choices do not depend on one another, which comes with a statistical inefficiency.

Incremental Analysis for Probabilistic Programs

Zhang et al. [2017] considers a similar goal to Cusumano-Towner et al. [2018]: to infer the posterior probability distribution of a probabilistic program upon having undergone a small change. They focus on a specific type of modification: changes in probabilistic knowledge, i.e. sample or observe statements. Such changes, however small, invalidate the results of exact inference, and thus inference quickly becomes costly when dealing with several variations of the same probabilistic program.

The algorithm performs a pre-inference step that lets us to reuse part of the existing inference results of a program upon modification while maintaining precision. In this step, dependencies between different data-flow facts are identified through a static dependence analysis. Data-flow facts each consist of possible value assignments to the probabilistic statements of a program. During inference itself, such dependencies are used to update the posterior probability of a program in a sparse manner, when it has undergone a small change in probabilistic knowledge.

The method is shown to improve upon another data-flow-based inference method [Claret et al., 2013], which differs in that it does not make use of any incremental analysis. It remains the question whether the method is competitive with alternative inference methods, including those catered towards continuous distributions. Though the method appears most suitable for discrete probabilistic programs, it does offer support for continuous distributions by approximating them using discrete distributions over a finite set.

3.2 An Alternative Approach

Inference over probabilistic programs becomes difficult when the evidence favors improbable paths, and this problem we hope to alleviate by means of incremental inference. We now turn to a set of solutions that take on a different view: they take a probabilistic program and turn it into a set of simplified programs of fixed support over which reasoning is much simpler. Still, the task of obtaining this set of simplified programs is non-trivial.

Divide, Conquer, and Combine

While universal PPLs allow for expressing probabilistic programs with stochastic support, inference on such programs can be very challenging. An issue that arises is one of locality: a sample with a high probability in one configuration may not have a high probability in another configuration. To solve this issue, Wang et al. [2024] introduced the Divide, Conquer, and Combine (DCC) framework. DCC considers each configuration separately: a probabilistic program is "divided" into several straight line sub-programs, each with fixed support. Inference is performed on each of these separately and the results are combined at the end.

One option to identify the SLPs is to perform static analysis. However, when considering every possible program that can be expressed in a universal PPL, the number of paths may very well become unbounded. Hence, DCC discovers them dynamically in a manner that resembles that of MCMC samplers. The key difference here is that DCC does not immediately reject new SLPs based on a single sample but instead remembers them, thereby avoiding the aforementioned locality issue. This hints at the following trade-off: *exploiting* SLPs with a higher likelihood versus *exploring* SLPs with a lower likelihood that may (or may not) prove to be more promising in subsequent steps.

A consequence of DCC is that by representing each possible path as an SLP, unlikely paths are no longer difficult to sample from. If such paths agree with the evidence, we can sample from them more often by prioritizing their corresponding SLPs. In other words, we won't have to reject as many samples in such cases.

Sound Abstraction and Decomposition of Probabilistic Programs

Holtzen et al. [2018] introduced a sound abstraction over probabilistic programs. A

probabilistic program is automatically transformed into a set of abstractions, where abstractions themselves are simplified programs. Such abstractions are reminiscent of SLPs: they allow us to perform inference in a decomposed manner, over each of the abstractions. By doing so, we can exploit certain conditional independencies that are present in probabilistic programs.

An abstraction is made up of predicates, i.e. any statement about the program that can be true or false, that are assigned to different parts of the original program. Consequently, abstractions are discrete programs, and thus the inference that is performed over them can be done by enumeration.

As for selecting the predicates, these are mostly assumed to be provided by the user. The paper suggests the following heuristic: including every Boolean expression of a program, thereby capturing the behavior of if and observe statements.

The notion of creating abstractions over probabilistic programs can be useful in the context of incremental inference. An abstraction can be viewed as a simplification of a probabilistic program. Moreover, one can consider building the abstraction of only a part of some program and maintaining the rest of the program in its original form. This may no longer result in a fully discrete program, but can help in building a sequence of programs such as described by the Coarse-to-Fine algorithm.

3.3 Computational Tractability

A question left unanswered is the following: what changes must we make to simplify probabilistic programs? This question is tightly linked to another one: what actually makes a probabilistic program "difficult", i.e. when does inference become difficult, or even intractable?

In an attempt to answer these questions, Freer et al. [2010] offered the following insight: sometimes, *adding* stochasticity to a probabilistic program, e.g., by adding uncertainty to some value that is otherwise considered to be a constant, may make inference easier. It can be seen as a way to "relax" the program: we no longer look for the exact assignment of variables that leads to some sought-after outcome, but instead approximate it. The paper reports having had success with this approach though it has not yet been tested rigorously.

3.4 Building Tools

It is difficult to construct generally applicable solutions to approximate inference using advanced sampling-based methods and variational inference (VI). When provided with few samples, such solutions often rely on domain knowledge, thereby losing their generality.

Stites et al. [2021] introduced a language for the construction of inference methods by making use of inference combinators. These are operators for constructing

importance samplers that can be adapted for specific problems. The combinators are properly weighted in terms of the density of the program by construction.

Programs can be used as proposals, whereby variables between the proposal and target are reused when possible or otherwise sampled from the prior. To learn proposals, VI methods are used.

Chapter 4

Problem Statement

In this section, we introduce the necessary notation and formally define the problem.

4.1 Preliminaries

A probabilistic program P consists of a set of random choices $(x_1, x_2, \ldots x_n)$, where each x_i is drawn from a probability distribution $p_i(\cdot|\theta_i)$ specified by P. θ_i denotes the parameters of the distribution p_i, which may be fixed constants or depend on previous random choices.

A trace t is a sequence of values $(t_1, t_2, \ldots t_n)$, where each t_i is the concrete value taken by x_i during a single execution of the program. An observation o is a pair (x_i, v_i) that assigns a concrete value to a random choice in P. We restrict ourselves to programs with only one observation.

We define *difficulty* in terms of how many samples are needed for an accurate posterior estimate, and *efficiency* as how accurate the estimates are for a given number of samples.

4.2 Problem Definition

Input. A probabilistic program P paired with an observation o.

We restrict the form of P, together with an observation o, to a set of predefined structural patterns, which we assume contribute to the difficulty of P. We focus on cases where o asserts a value that is unlikely under the prior induced by P, specifically when o depends on a conditional statement and the branch favored by the prior differs from the one favored by the observation. We refer to this specific pattern, namely the conditional statement that causes this disagreement, as a *local difficulty*. We categorize cases based on the structure of the conditional and handle each case accordingly. If P does not match any of the predefined patterns, no modification is applied.

We consider the following cases: (a) conditionals based on a Boolean random variable, (b) conditionals that compare a uniform random variable to a fixed value, and (c) conditionals that compare a normal random variable to a range of values.

Problem. Given a probabilistic program P, the goal is to find a probabilistic program P' such that incremental inference on the pair (P', P) is more efficient than inference on P.

We require the pairs (P', P) to closely resemble each other so that traces from P' can be reused for P by reweighing them, following the work of Cusumano-Towner et al. [2018], who show that efficiency is dependent on the similarity between corresponding random choices in P and P'. For simplicity, we focus on a single iteration, with the understanding that the approach can be applied recursively to produce a sequence of programs.

Chapter 5

Method

5.1 Overview

Following the definition in Chapter 4, the goal of the method is to obtain a simplified version of a given probabilistic program P, such that it can be used for incremental inference. This goal is threefold: given some P, the method should detect difficulties, generate appropriate modifications, and apply those modifications. We implement the method using Julia, making use of the probabilistic programming framework Gen [Cusumano-Towner et al., 2019].

While several factors may contribute to the difficulty of a program, we focus on cases where difficulty arises from observations that disagree with the prior. Specifically, when the disagreement occurs in a local difficulty, that is, a conditional statement in which the prior and the observation favor different branches, thereby contributing to the overall difficulty of a program.

Once a local difficulty is located, the method generates changes that result in a simplified program, i.e., one that is similar but easier to sample from. Specifically, we look for changes that result in the prior of the program being closer to the posterior, or in other words, we look for a program that 'agrees with the evidence' more.

We first describe the process at a conceptual level, and then present the corresponding algorithm in more detail.

5.2 Procedure

The method is based on the insight that the dependency structure of a program can be used to systematically locate its local difficulties. It consists of three main components: locating difficulties, matching them to appropriate changes, and applying these changes. We now walk through each of these steps.

1. **Locating local difficulties:** we locate points in the program where an observation asserts that a random choice should take on a value that is unlikely

according to the prior distribution of the program. To do so, we extract the dependency graph from the program, which Gen builds by default, and perform a bottom-up search.

We locate the conditional statement that the observation depends on. If the branch favored by the observation does not match the one that is most likely according to the prior of the program, we mark the conditional statement as a local difficulty.

We consider the following simple cases of conditional statements in which one branch is strongly favored over the other under the prior:

(a) **Boolean:** statements that condition on a Boolean distribution with probability close to either zero or one, e.g., `bernoulli(0.01)`.

(b) **Uniform:** statements that condition a Uniform distribution with a large support size to equal a specific value, e.g., `uniform(1,1000)` is conditioned to equal 42.

(c) **Normal:** statements that condition a Normal distribution to lie within a narrow range of values, with the difficulty determined by both the width of the range and the location of the range within the distribution, i.e., whether it falls in a high- or low-probability region, e.g., `normal(0,2)` is conditioned to fall between 5 and 6.

Note that these scenarios alone do not cause a program to be difficult; difficulty arises only when these scenarios concur with an observation that favors the branch that is less likely under the prior.

We restrict our focus to a set of input programs in which the branches of the conditional statement differ, e.g., in Listing 5.1, the branches assign probabilities close to zero and one, respectively. These are the cases in which disagreement between the observation and prior has a significant impact on inference.

To handle programs with nested conditional statements, we treat preferences, i.e., a condition paired with the branch preferred by the observation, as observations themselves, and apply the procedure recursively.

Example. We revisit Alice's scenario in Listing 5.1 to illustrate the process. The example corresponds to case (a), as it involves a conditional statement that depends on a Boolean random choice with probability close to 0.

Listing 5.1: Alice's example

```
illness = bernoulli(0.01)
sneeze = illness ? bernoulli(0.9) : bernoulli(0.01)
observe(sneeze = true)
return illness
```

In Gen, observations are specified outside the model, so the search begins at the random choice `sneeze`, which is the observed variable and depends on a conditional statement. The observation favors the true branch, i.e., for the condition `illness` to be true, because a Boolean distribution with probability 0.9 is far more likely to produce `sneeze = true` than one with probability 0.01. However, under the prior distribution, `illness` is most likely assigned false (probability 0.01), so the prior favors the false branch. Therefore, the observation and prior disagree, and the program contains a local difficulty.

2. **Identifying changes:** we match the detected local difficulty to an appropriate change. The change is designed to increase the prior probability of the branch that is favored by the observation. We again identify different cases to handle the generation of this change.

   Rather than reasoning mathematically about the appropriate magnitude of change, we study the effect of a fixed change empirically across programs of varying local difficulties. As mentioned in Chapter 4, the simplified program should still closely resemble the original program, so the fixed change is kept small.

   (a) **Boolean:** we tune the probability of the conditioned Boolean variable, i.e., probabilities close to zero are increased, while those close to one are decreased.

   We do so by means of a simple rule: $p{*}10$ for very low probabilities and $1-(1-p)*10$ for very high probabilities, e.g., `bernoulli(0.01)` becomes `bernoulli(0.1)`.

   (b) **Uniform:** we tighten the bounds of the conditioned Uniform distribution, by dividing both the outer bounds and the conditioned value by the same amount, effectively creating a binned version of the distribution.

   We choose to divide by 10, which corresponds to using bins of size 10, e.g., `uniform(1,1000)` becomes `uniform(1,100)` and is conditioned on 4 instead of 42.

   (c) **Normal:** we approximate the conditioned Normal distribution with a Categorical distribution, where each bin is weighted according to the probability mass within that bin.

   We choose the width of the bins based on standard deviations, such that the n_{th} bin consists of all points within n standard deviations from the mean, e.g., `normal(0,2)` becomes `categorical(0.68, 0.27,0.05)` and is conditioned on 3, since the range [5, 6] lies within the outer region of the distribution.

   In Listing 5.1, this corresponds to replacing `illness = bernoulli(0.01)` with `illness = bernoulli(0.1)`.

3. **Applying changes:** to apply the generated change, we manipulate the program's structure: using pattern matching on its syntax tree, we locate the nodes that need to be modified and replace them with their updated versions. The resulting program is the simplified program.

5.3 Algorithm

Algorithm 1 shows the algorithm in detail, focusing on only the Boolean case for clarity. We expand on the three steps, showing how they are implemented in the algorithm. An implementation of the algorithm is publicly available at DOI: 10.5281/zenodo.18340865.

1. **Locating local difficulties:** for each current node, the algorithm checks if it matches the pattern of a local difficulty. First, we determine whether the node corresponds to the observed variable and whether it contains a conditional statement (line 2). If not, the search continues in the parent nodes (lines 20-21). If both conditions are met, we then check whether the observation and prior favor the same branch. This check is twofold: we first identify which branch is more likely to agree with the observation (lines 8, 14), and then check whether that branch is unlikely under the prior (lines 9, 15). If both criteria hold, the conditional statement is marked as a local difficulty, and the algorithm proceeds to the next stage (lines 10, 16). Otherwise, we treat the preference indicated by the observation, i.e., the variable inside the condition and the value it must take on to result in the branch favored by the observation, as a new observation and apply the procedure recursively (lines 11-13, 17-19). This recursive process allows the algorithm to deal with nested conditional statements.

   Note that while there is no theoretical threshold for when a program becomes difficult, we impose a practical cap on the degree of difficulty induced by local difficulties so that the changes in the subsequent step can be applied. Specifically, this cap applies to the conditioned variable, e.g., how extreme its probability is in the Boolean case (lines 9, 15). Within this cap, programs still range from non-difficult to difficult, and we empirically show how various input programs differ in difficulty.

2. **Identifying changes:** we apply the heuristics as described in 5.2 (lines 30-33).

3. **Applying changes:** the final step applies the computed change to the program via metaprogramming and is omitted in Algorithm 1, as it is primarily an implementation detail.

Algorithm 1: Locating Difficulty and Generating Change

```
1  Function findDifficulty(cur, obs):
2      if name (cur) = name (obs) and containsCond (cur) then
3          cond, leftBranch, rightBranch ← args (cur);
4          (nameCond, boolCond) ← parseCond (cond);
5          p ← lookup (nameCond);
6          leftDiff ← |value (obs) − leftBranch|;
7          rightDiff ← |value (obs) − rightBranch|;
8          if leftDiff < rightDiff then
9              if isCloseToZero(p) and boolCond then
10                 return generateChange(p);
11             else
12                 newObs ← (nameCond, boolCond));
13                 return findDifficulty(cur, newObs));
14         else if rightDiff < leftDiff then
15             if isCloseToOne(p) and not boolCond then
16                 return generateChange(p);
17             else
18                 newObs ← (nameCond, boolCond));
19                 return findDifficulty(cur, newObs));
20     for parent ← parents (cur) do
21         change ← findDifficulty(parent, obs);
22         if change ≠ nothing then
23             return change
24     return nothing
25 Function parseCond(cond):
26     match cond;
27         case name ⇒ (name, true);
28         case (negOp, name) ⇒ (name, false);
29 Function generateChange(cond):
30     if isCloseToZero(p) then
31         p_new ← p · 10;
32     if isCloseToOne(p) then
33         p_new ← 1 − (1 − p) · 10;
34     return p_new;
```

Chapter 6

Results

We introduce a set of simple input programs that we extend to show the accuracy of the method on sets of programs with varying degrees of difficulty. Listing 6.1 shows a simple input program containing a Bernoulli distribution with a very small probability, which we vary. Listing 6.2 contains a Uniform distribution instead, of which we vary the support. Lastly, Listing 6.3 contains a Normal distribution, and we vary the size of the range that is conditioned on. We examine the accuracy of the algorithm by showing how it behaves as the difficulty of these programs changes.

Moreover, we consider the effect on extensions of these programs, which contain nested conditional statements. First, we show the effect of increasing the *conditional nesting depth*, i.e., the number of levels of nested conditional statements, while keeping the local difficulty (i.e., a conditional statement in which the prior and the observation favor different branches) at the outermost level. Next, we show the effect of varying the *conditional nesting level*, i.e., the position of the local difficulty within the nesting.

Note that for each of these, we also vary the probability of `sneeze` in the false branch. For programs in which the condition on `illness` is less likely to be met, we decrease the probability of `sneeze` in the false branch proportionally. If we don't, then in cases where the condition is very unlikely to be met, the false branch will be more likely to result in `sneeze = true` than the true branch, despite its low probability. This results in a lesser increase in the overall difficulty of the program, thereby preventing us from testing the algorithm on more extreme levels of difficulty.

Listing 6.1: Boolean distribution with a probability of order 10^{-3}

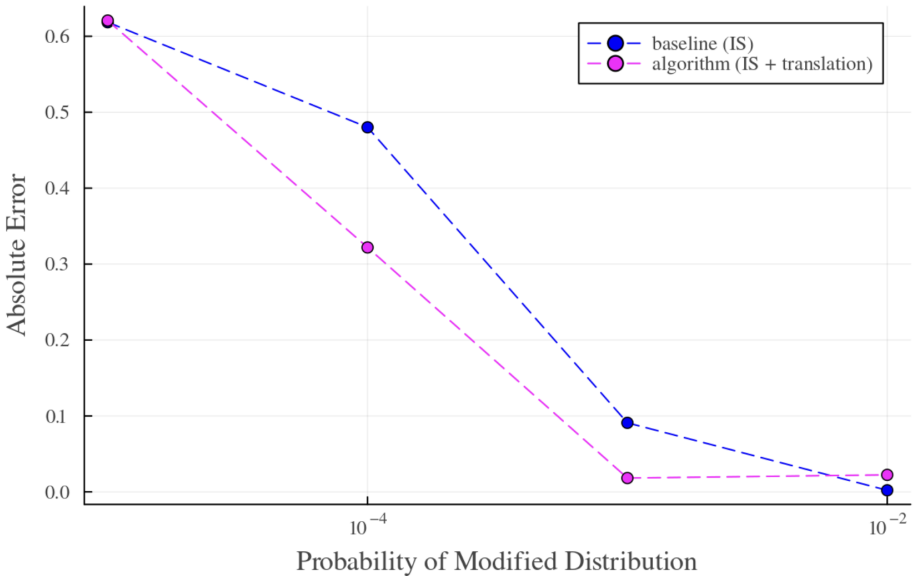
```
illness = bernoulli(0.02)
sneeze = illness ? bernoulli(0.9) : bernoulli(0.01)
observe(sneeze = true)
return illness
```

Listing 6.2: Uniform distribution with a support size of 500

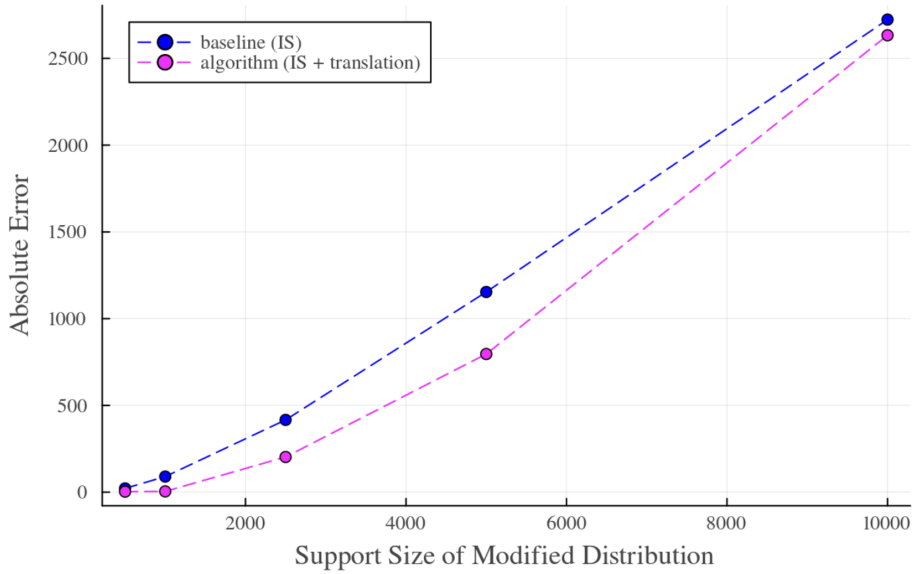
```
illness = uniform(1,500)
sneeze = illness == 50 ? bernoulli(0.9) : bernoulli(0.001)
observe(sneeze = true)
return illness
```

Listing 6.3: Normal distribution with a range that is conditioned on of size 1

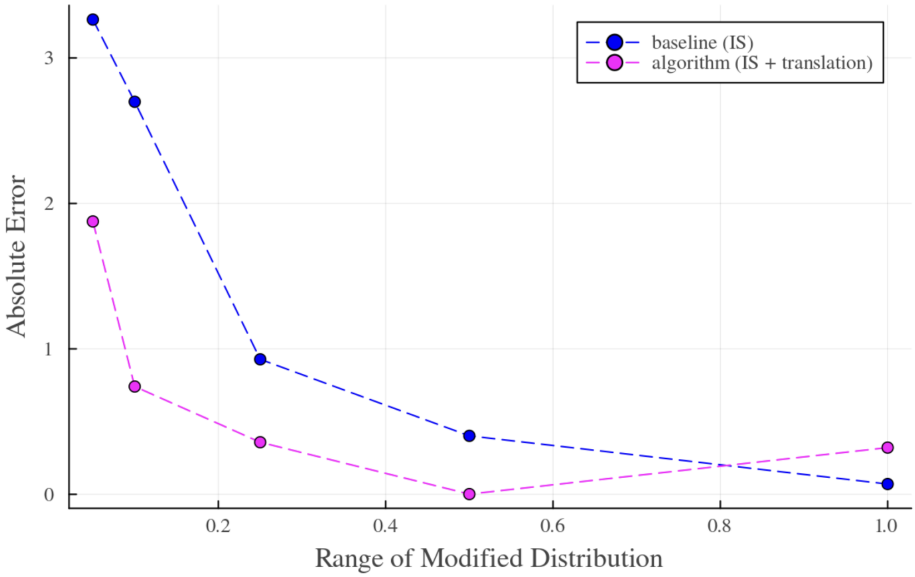
```
illness = normal(0,2)
sneeze = (5 < illness < 6) ?
                    bernoulli(0.9) : bernoulli(0.001)
observe(sneeze = true)
return illness
```



(a) Modifying the Boolean distribution. Difficulty decreases as the probability of the modified distribution increases.

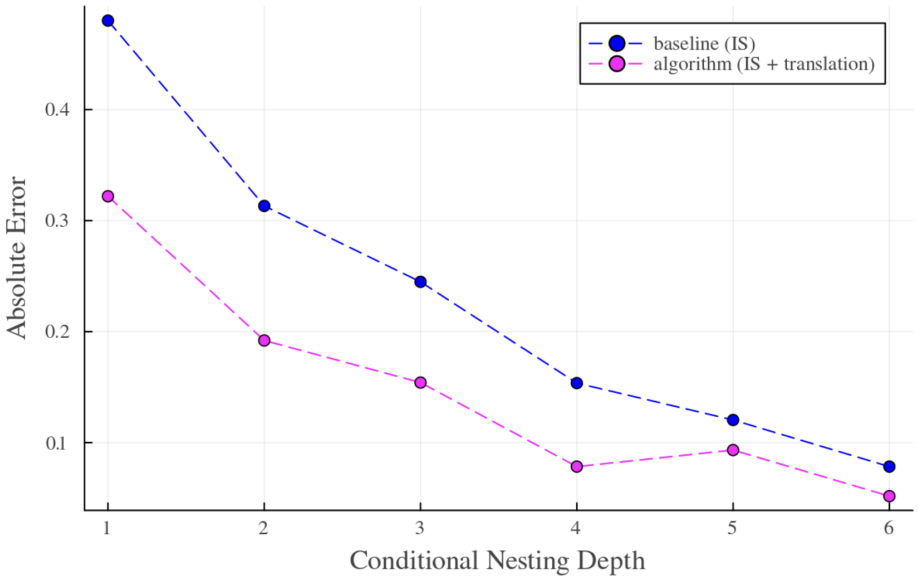


(b) Modifying the Uniform distribution. Difficulty increases as the support size of the modified distribution increases.

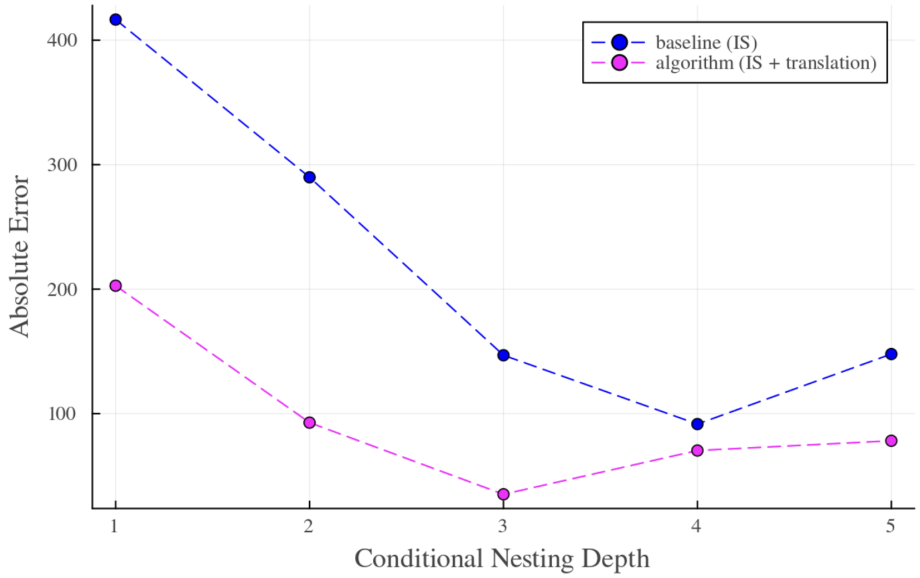


(c) Modifying the Normal distribution. Difficulty decreases as the range that is conditioned on increases.

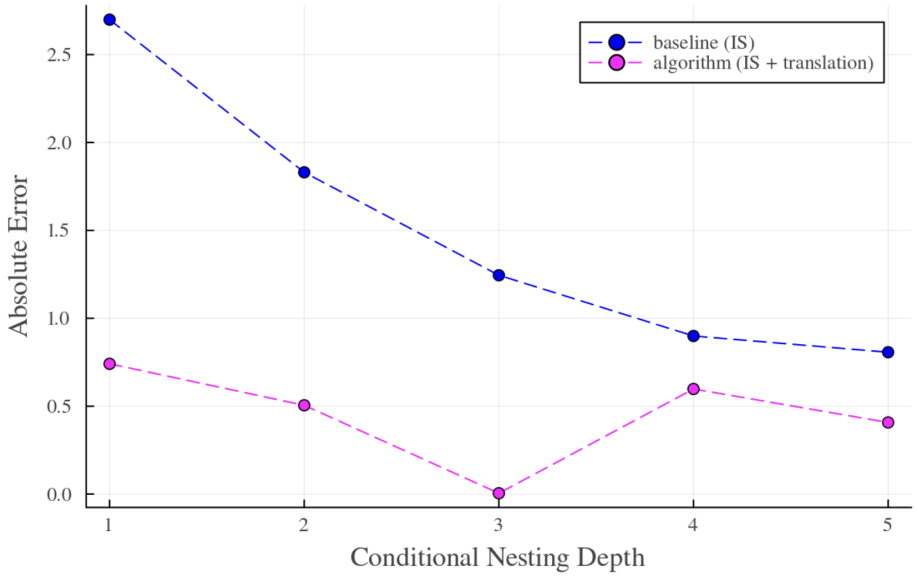
Figure 6.1: Comparison of the algorithm against baseline on programs with a single conditional statement for various degrees of difficulty, using 1000 samples.



(a) Modifying the Bernoulli distribution, with a probability of 0.0002.



(b) Modifying the Uniform distribution, with a support size of 2500.



(c) Modifying the Normal distribution, conditioning on a range of 0.1.

Figure 6.2: Comparison of the algorithm against baseline on programs where we fix the parameters of the local difficulty but vary the number of nested conditional statements, using 1000 samples.

The algorithm can be used in combination with any existing sampling method. We compare the algorithm, used in combination with importance sampling, against a baseline implementation that only uses importance sampling.

6.1 Simple Programs

Figure 6.1a shows the accuracy of the algorithm and baseline methods on the problem in Listing 6.1. We vary the probability of the Bernoulli distribution to show the change in accuracy as the degree of difficulty of input programs varies. In this case, conditioning on a Bernoulli distribution with a smaller probability, when observing the variable to be true, is considered more difficult. We see that for very small probabilities, the methods behave equally poorly, while for larger probabilities, both are able to closely approximate the true answer. For values in between, the algorithm outperforms the baseline. For the largest probability, i.e., that of order $10^{-}2$, the algorithm appears to have a small bias when compared against the baseline.

Figure 6.1b shows the accuracy of the methods on Listing 6.2. We vary the support size of the Uniform distribution to show how the accuracy changes as the difficulty of the input program varies. Here, a larger distribution size is considered more difficult. We see a similar trend to that observed in Figure 6.1a. Both the baseline and the algorithm steadily worsen in terms of accuracy as the program

becomes more difficult. The algorithm outperforms the baseline for intermediate values, while for very small or large support sizes, the methods behave similarly.

Similarly, Figure 6.1c shows the accuracy of the methods on Listing 6.3. We vary the range of values on which the Normal distribution is conditioned, and a narrower range is considered more difficult. We again observe a similar trend. For most ranges, the algorithm outperforms the baseline, and the difference in absolute error is largest for intermediate values. For smaller ranges, the methods do not converge, but their curves suggest eventual convergence. For a larger range, i.e., one of size 1, we see that the algorithm is slightly outperformed by the baseline, similar to Figure 6.1a, suggesting the algorithm has a slight bias.

6.2 Increasing the Conditional Nesting Depth

In Figure 6.2, we again show the accuracy of the methods on the programs in Listing 6.1, Listing 6.2, and Listing 6.3, this time varying the overall difficulty of programs by extending program structures. We do so by adding additional sample statements, whose parameters are always conditioned on the previous sample statement via a conditional statement, thereby increasing the programs' conditional depth. The observation always concerns the innermost sample statement in the dependency chain. Note that the added conditional statements do not introduce new local difficulties, i.e., their prior agrees with the observation, and only influence the overall difficulty through increasing the conditional depth.

For the most part, the overall difficulty appears to decrease as the number of if-statements increases. Moreover, we observe that the algorithm consistently outperforms the baseline. Figure 6.2a shows that, in the Bernoulli case, the baseline consistently decreases as the conditional nesting depth increases. The algorithm follows a similar trend, although there is a slight deviation at a depth of 5, and the differences between the two methods lessen as the depth increases, suggesting convergence. Figure 6.2b shows similar patterns for the Uniform case. Both methods largely improve as the conditional nesting depth increases and appear to converge toward one another, though deviations occur at higher depths. Finally, in the Normal case shown in Figure 6.2c, the baseline steadily decreases as the depth increases, and while the two methods tend to approach each other, the algorithm shows some deviations from the general trend.

6.3 Varying the Conditional Nesting Level

Building on top of the programs with multiple if-statements, we now vary the conditional nesting level. Rather than always placing the local difficulty at the outermost level, we vary where it occurs in the dependency chain. As a result, the variable we're interested in, `illness`, is no longer the source of difficulty for obtaining its posterior. Nevertheless, the posterior is still influenced by the variables

that depend on it, since the observation is conditioned on one of them, namely the innermost variable.

Table 6.1 shows a comparison of the algorithm against the baseline for the Uniform case, varying both the conditional nesting depth and the conditional nesting level at which the local difficulty occurs. The baseline shows the following trend: for higher conditional nesting levels, i.e., those where the local difficulty is more deeply nested, the error increases. Moreover, programs whose local difficulty occurs at the same relative position within the conditional chain appear to have more similar errors than those with the same absolute nesting level. For example, programs in which the local difficulty appears at the maximum depth show similar errors, even when their maximum depth differs. The algorithm appears more robust as it consistently outperforms the baseline across all depths and levels. The errors of the algorithm remain low throughout and do not exhibit any clear patterns.

		Conditional Nesting Level									
		First		Second		Third		Fourth		Fifth	
		A	B	A	B	A	B	A	B	A	B
Max Depth	1	0.08	0.17	-	-	-	-	-	-	-	-
	2	0.04	0.12	0.03	0.24	-	-	-	-	-	-
	3	0.01	0.06	0.07	0.15	0.04	0.21	-	-	-	-
	4	0.03	0.04	0.09	0.11	0.04	0.16	0.08	0.21	-	-
	5	0.03	0.06	0.04	0.09	0.09	0.11	0.04	0.13	0.05	0.24

Table 6.1: Comparison of normalized absolute errors of the algorithm (A) versus the baseline (B) for programs in which we vary both the maximum conditional nesting depth and the position within the nesting at which the local difficulty occurs, which involves a Uniform distribution with support size of 2500, using 1000 samples.

Chapter 7

Discussion

This section aims to provide intuition for the three experiments introduced in Chapter 6: the effects of varying the parameters of a local difficulty (i.e., a conditional for which the observation prefers the unlikelier path under the prior), varying the conditional nesting level of that local difficulty, and varying the overall conditional nesting depth of the programs. The experiments illustrate which program structures contribute to the overall program difficulty and which changes are effective when used in combination with the incremental inference framework.

Note, however, that the programs considered do not capture the full complexity of real-world examples. They are constructed to contain structural patterns that increase their complexity, which the method can detect and modify through pattern-matching. The assumption is that such patterns may also appear and contribute to complexity in real-world programs, albeit in less easily identifiable forms. The range of structural patterns studied is limited to those we were able to systematically construct and analyze, and is by no means exhaustive.

7.1 Effect of the Degree of Difficulty

Figure 6.1 shows that both the algorithm and the baseline decrease in accuracy for more difficult programs, i.e., those for which the prior and posterior distributions diverge. This is in line with our assumption that programs in which the prior distribution disagrees with the evidence are more difficult. We see that the algorithm outperforms the baseline across a range of difficulty levels for problems involving a local difficulty, where the differences in overall difficulty are obtained by varying the parameters of the local difficulty. This suggests that our method of simplifying programs for use with incremental inference is effective.

We observe that as the problems become either very easy or very difficult, the accuracy of the algorithm and baseline converge. In easy cases, the baseline already achieves a low error, leaving little room for improvement. In more difficult cases, the simplification appears insufficient to overcome the difference between prior and posterior, that is, the simplified program may still be too difficult to sample

from to achieve any improvement over the baseline. We conclude that the impact of a fixed amount of change depends on program difficulty.

Rather than learning the optimal change for each input program, the result shows which types of changes may be effective and, for each change, for which programs they are most effective. Taken together, this provides guidance for answering the inverse question, namely which change should be applied to a given program to optimize its effect.

7.2 Effect of the Conditional Nesting Depth

In Figure 6.2, we observed that increasing the conditional nesting depth generally reduces the overall difficulty of programs, and that in easier programs, the algorithm and baseline tend to converge toward each other. For easier problems, the baseline achieves very low errors by itself, and therefore any improvements that the algorithm offers have much less impact than for more difficult problems.

One may expect additional structure, such as an increased conditional depth, to increase the overall difficulty of programs, but if the added conditional does not impose difficulty by itself, i.e., if it is not a local difficulty, this is not necessarily the case. We illustrate this phenomenon with a simple example.

In Listing 7.1, two of the possible paths agree with the observation, namely those where `illness` is either true or false and `sneeze` is true. In Listing 7.2, each of these paths becomes slightly less probable according to the prior, but additional paths now also agree with the observation. The probability mass of the paths for which the observation is true increases, making the prior closer to the posterior. As a result, the extended program is slightly easier to sample from, and each time we add a conditional, this effect grows.

Note that with different parameters, sampling could instead become harder, i.e., different probabilities for the true and false branches of each added conditional would distribute probability mass differently. The result is limited to the specific parameters tested and does not show how the method behaves for programs in which added structure increases difficulty rather than decreases it.

Listing 7.1: Boolean example

```
illness = bernoulli(0.02)
sneeze = illness ? bernoulli(0.9) : bernoulli(0.01)
observe(sneeze = true)
return illness
```

Listing 7.2: Extended Boolean example

```
illness = bernoulli(0.02)
sneeze = illness ? bernoulli(0.9) : bernoulli(0.01)
fever = sneeze ? bernoulli(0.9) : bernoulli(0.01)
```

```
observe(fever = true)
return illness
```

7.3 Effect of the Conditional Nesting Level

Table 6.1 shows that while the accuracy of the baseline varies across varying conditional nesting levels, the algorithm consistently achieves a low error for the given program. The baseline performs worse on programs where the local difficulty is placed closer to the maximum conditional nesting depth of the program.

We can understand this result as follows. The variables that occur after the local difficulty act similarly to those in Figure 6.2. By increasing the conditional depth, the existing paths that agree with the observation decrease in probability, but the overall probability mass of paths for which the observation is true increases. For the specific parameters used, the gap between prior and posterior decreases as a result, thereby decreasing the overall difficulty of programs.

Variables occurring before the local difficulty in the dependence chain (the outermost ones) have little impact on the program's overall difficulty. After the local difficulty, traces are already low-probability under the prior, so small changes in downstream variables have a larger effect. Upstream variables, on the other hand, tend to result in partial traces that are already likely under both the prior and posterior, so small changes, such as adding an extra variable that largely agrees with the observation, have much less impact.

Similar to Figure 6.2, the result is limited to the specific parameters used. With different probabilities for the variables before or after the local difficulty, the distribution of probability mass could shift, and the observed correlation between conditional nesting level and overall program difficulty might be reversed.

Chapter 8

Conclusions and Future Work

8.1 Future Work

Adaptive Changes

Our method applies changes constructed according to predefined rules. A future direction would be to learn heuristics that better approximate the ideal change for each program, thereby maximizing improvement. One might expect that a good heuristic simply applies changes of higher magnitudes to more difficult problems, but this is not necessarily the case, since the original program and its simplification must be similar for incremental inference to be effective [Cusumano-Towner et al., 2018]. Therefore, learning such a heuristic may involve balancing the magnitude of changes against the need to preserve similarity.

One possible approach is to start off by taking the changes we have introduced and fine-tuning them to each of a set of programs, that is, learning the parameters for which the change is most efficient. One may then search for patterns between the programs and their corresponding 'optimal' change, and use this information to construct a more general heuristic.

Real-World Applicability

While our method identifies sources of difficulty that contribute to overall program difficulty, it currently operates on a very limited set of patterns.

One future direction is to extend this set of patterns such that the set of input programs that contain these patterns more closely resembles those found in practice. For example, one may consider programs that consist of multiple local difficulties, and consider applying changes to each of them, or choosing to only apply the most impactful one to ensure similarity between the input program and its simplification. One may also consider programs that consist of multiple observations, each of which may play a role in a distinct or overlapping local difficulty. Lastly, one may consider different sources of difficulty altogether, that is, ones that do not follow the strict pattern of the local difficulties we have introduced.

Another direction is to study real-world problems to investigate whether and how these difficulties show up, even if not in the exact form our method currently detects. However, in real-world programs, difficulty is likely to arise from interactions between multiple parts of a program. A future direction is therefore to develop a method that detects these more global sources of difficulty and design appropriate changes to address them.

8.2 Conclusions

We implemented a method that uses incremental inference to more efficiently sample from complex probabilistic programs. The method automates this process by detecting difficult parts of a program, generating appropriate changes, and applying these changes to produce a simplified version of the program. The simplified and original programs then form a pair to which incremental inference is then applied.

We identified a category of program patterns, defined by both the structural form of programs and their parameters, that make a probabilistic program harder to sample from and devised an algorithm that detects these patterns via a bottom-up search of the program's dependency graph. For each pattern, we designed a corresponding heuristic that computes an appropriate change using parameters specific to the given program. These changes are then applied automatically by means of metaprogramming.

For a specific set of input programs, we showed that applying incremental inference to the resulting program pairs improved efficiency compared to a baseline sampling method. Although this set of input programs is limited, the programs are deliberately constructed to induce inference complexity in structured ways. We suggest that these findings can inform improvements to inference on a broader class of probabilistic programs.

Bibliography

G. Claret, S. Rajamani, A. Nori, A. Gordon, and J. Borgström. Bayesian inference using data flow analysis. pages 92–102, 08 2013. doi: 10.1145/2491411.2491423.

M. Cusumano-Towner, B. Bichsel, T. Gehr, M. Vechev, and V. K. Mansinghka. Incremental inference for probabilistic programs. *SIGPLAN Not.*, 53(4):571–585, jun 2018. ISSN 0362-1340. doi: 10.1145/3296979.3192399. URL https://doi.org/10.1145/3296979.3192399.

M. F. Cusumano-Towner, F. A. Saad, A. K. Lew, and V. K. Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 221–236, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314642. URL https://doi.org/10.1145/3314221.3314642.

N. Fillion and R. M. Corless. Perturbation theory, 2022. URL https://arxiv.org/abs/2212.07380.

C. E. Freer, V. K. Mansinghka, and D. M. Roy. When are probabilistic programs probably computationally tractable?, 2010. URL http://danroy.org/papers/FreerManRoy-NIPSMC-2010.pdf.

S. Holtzen, G. Van den Broeck, and T. Millstein. Sound abstraction and decomposition of probabilistic programs. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1999–2008. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/holtzen18a.html.

X. Li, F. Chadwick, and B. Swallow. Advances in approximate bayesian inference for models in epidemiology. *Epidemics*, 53:100855, 2025.

S. Stites, H. Zimmermann, H. Wu, E. Sennesh, and J.-W. van de Meent. Learning proposals for probabilistic programs with inference combinators, 2021. URL https://arxiv.org/abs/2103.00668.

A. Stuhlmüller, R. X. D. Hawkins, N. Siddharth, and N. D. Goodman. Coarse-to-fine sequential monte carlo for probabilistic programs. *CoRR*, abs/1509.02962, 2015. URL http://arxiv.org/abs/1509.02962.

J.-W. van de Meent, B. Paige, H. Yang, and F. Wood. An introduction to probabilistic programming, 2021. URL https://arxiv.org/abs/1809.10756.

E. van Krieken, T. Thanapalasingam, J. M. Tomczak, F. van Harmelen, and A. ten Teije. A-nesi: A scalable approximate method for probabilistic neurosymbolic inference, 2023. URL https://arxiv.org/abs/2212.12393.

W. Wang, L. Ding, M. Zeng, X. Zhou, L. Shen, Y. Luo, and D. Tao. Divide, conquer and combine: A training-free framework for high-resolution image perception in multimodal large language models, 2024. URL https://arxiv.org/abs/2408.15556.

J. Zhang, Y. Sui, and J. Xue. Incremental analysis for probabilistic programs. 2017. doi: 10.1007/978-3-319-66706-5_22.