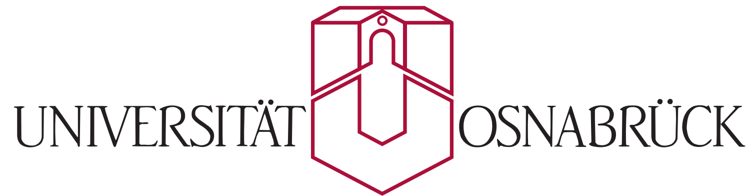


Andreas J. Stuhlmüller

Learning Compositional Representations

A thesis submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science
Department of Cognitive Science
University of Osnabrück

First Supervisor: Noah Goodman, Ph.D.
Second Supervisor: Prof. Dr. Kai-Uwe Kühnberger
Submitted: August 27, 2009

Declaration

I hereby declare that this submission is my own work and that I have not used any other resources or auxiliaries except for those where due acknowledgment has been made in the text.

Osnabrück, August 27, 2009

....
Andreas Stuhlmüller (Matrikel 928118)

Hiermit erkläre ich, dass ich die vorliegende Arbeit eigenständig verfasst habe und keine anderen Quellen und Hilfsmittel als die im Text angegebenen verwendet habe.

Osnabrück, 27. August 2009

....
Andreas Stuhlmüller (Matrikel 928118)

Abstract

How could an agent learn and reason in a complexly structured, stochastic world? This problem is at the center of both artificial intelligence and psychology. One candidate answer to this question is that both learning and reasoning can be explained as probabilistic inference over a language-like hypothesis space for generative models. The goal of this thesis is to describe what makes this approach plausible and to demonstrate the learning of generative models from structured observations in a simple world consisting of stochastic, tree-generating programs. In order to make program inference feasible, I derive a new class of adaptive importance sampling algorithms for probabilistic programs that let us compute the likelihood of structured observations under a given generative model. Using this algorithm in combination with Markov Chain Monte Carlo methods, I quantitatively show that we can reliably infer generative models from observations in our demonstration world.

Contents

1 Introduction **7**

2 Background **13**
- 2.1 Representationalism 13
- 2.2 A Language of Thought 14
- 2.3 Generative Models 16
- 2.4 Probabilistic Inference 18

3 Model **23**

4 Inference **27**
- 4.1 Importance Sampling 29
- 4.2 Adaptive Importance Sampling 37
- 4.3 Markov Chain Monte Carlo 47

5 Setup **53**
- 5.1 A Structured World 53
- 5.2 A Probabilistic Representation Language 55

6 Learning **57**
- 6.1 Method 57
- 6.2 Illustration 59
- 6.3 Quantitative Experiments 62

7 Discussion **69**
- 7.1 Interpretation 69
- 7.2 Related Work 70
- 7.3 Future Research 71

8 Conclusion **75**

Bibliography **78**

Appendix **82**

Chapter 1

Introduction

As part of our project of understanding the human mind, we would like to explain how physical systems can exhibit what we call *learning* and *reasoning*. In order to solve problems that are difficult given our current technologies and knowledge, we would like to build systems that learn and reason. These are the two motivations I have in mind when I talk about an explanation of learning and reasoning, and the success of such an explanation shall be determined by its contribution to these larger endeavors.

I take it as a working hypothesis that any such explanation must be a reductive explanation. What I mean by a *reductive explanation* is an explanation that shows how more elementary building blocks must be combined to create a system that exhibits the properties in question. An explanation of a mental phenomenon like learning and reasoning can be reductive only if it explains intentional terms like *belief* using more basic, well-defined vocabulary.

In the following, I will first illustrate the abilities I want to explain in my current and future work, then outline the approach to understanding learning and reasoning that I take and describe the extent of the present work.

The question I want to answer is this: How could a system, be it a human or a machine, learn and reason in as complex a world as ours? How do we discover and make use of structure in raw sensory data? What we see is not a mess of colored pixels, what we hear is not a rush of uninterpreted sound waves — what we perceive carries meaning. Our perceptions have connotations

and make sense not in isolation but only when placed into a larger model of the world. Phrasing my question in a way that carries more assumptions, I ask: How can a system acquire — *learn* — such a model of the world from sensory data, and how can this model be used to make predictions about how the world works and to judge which statements about the world are true and which are not — to *reason*?

Examples abound. Children learn to recognize and categorize objects. They reason about how objects behave using naive physics and knowledge about causality, both of which might be learned from sensory data. They grasp the concept of a number, learn about social relations, and learn the principles of language understanding and production. Not unlike children, we scientists infer the principles of our fields from experimental data. Using concepts we create to transfer knowledge across situations, we reason about the behavior even of systems we never observe in person. The predictions of both children and scientists fare better than those of any artificial system we have constructed so far, and they do so systematically: information processing that leads to successful learning and reasoning has method. What is this method?

In order to learn more about this method, I will look at simple systems and explore how incremental[1] learning can take place in such systems. For example, it is obvious that the trees in each row of figure 1.1 belong to the same class. What is the method we use to determine that this is the case? The formal basis I build on to answer questions like these will be Bayesian probability. Before I say more about the extent of the current work, I will explain why I have chosen the approach of looking at simple systems.

[1] By "incremental learning", I refer to learning that in some way builds on what has been learned before.

It is an almost tautological truth that we cannot understand complex systems as long as we do not understand simple systems. Therefore, the practical part of this work is limited to learning and reasoning in a simple toy world that we do understand completely. There is another reason for choosing a simple system, and this one is based on the idea of reductive explanation. If simple and complex systems that learn and reason are composed of similar building blocks, composed in a similar manner, then understanding simple systems will bring us a long way towards understanding more complex ones. The constructive counterpart of (iterated) reductive explanation is stratified design. *Stratified*

design denotes the idea that complex systems should be designed as a sequence of layers, where each layer is described by a language appropriate to that level of detail. If understanding a simple system helps to understand more complex ones, then building a simple one may be useful for building more complex ones if the sequence of layers is similar. This approach is fundamental to the practical part of this work, where I first extend the probabilistic programming language Church, then use this language to describe program inference.

Research on learning and reasoning blurs the separation between methodology and content. Not only do we approach the problem by looking at the building blocks that are part of the larger solution, but the idea of using building blocks — the idea of *compositionality* — becomes a building block itself. The more general idea of *incremental learning* will be one of the key features of my approach. Children's learning takes time. For example, when children learn the number words (which tends to start at an age of two), they spend approximately six months in a *one-knower* stage during which they know that *one* means one and if asked to give one item, they give one, otherwise a handful. They then spend about nine months in a *two-knower* stage, about three month in a *three-knower* stage and then usually make the inductive leap and use counting to give you any number of items you request ([Wyn90, Wyn92, PGT]). It is not a far-fetched hypothesis that each step builds on what has been learned before and reuses that which does not need to be changed. There is more than one way to be incremental: Learning can be incremental in the sense that, in the construction of new mental models, existing models can be used either as the basis that is being modified or as a building block. More generally, inference can be incremental in the sense that, over time, a learner can accumulate *procedural knowledge* on how to best learn and reason in certain situations. In my work, I want to formalize both ways of being incremental.

Any attempt at a reductive explanation must be built on a firm basis or it cannot succeed. Words need to have precise meanings and where this is not possible, programs and mathematics need to provide clarity. The foundation my approach builds on is Bayesian probability theory and, more specifically, probabilistic inference in generative models, a notion that I will explain and justify in the background chapters.

Taken together, the approach to learning and reasoning that I consider is the following: Compositional representations that mirror the compositional structure of the world are learned incrementally, with each step building on the components that were learned before. On a computational level, probabilistic inference explains the principles behind both learning and reasoning. These thoughts are not new (as will become clear in the background chapters and in the discussion of related work), but I believe that an exploration of these principles using a simple learner in a interestingly structured toy world will nonetheless prove insightful.

I will now state the extent of the current work using terminology introduced by Marr [Mar82]. He distinguishes three levels that can be used to analyze a computation: The computational level that describes the goal of a computation and the logic behind its strategy, the algorithmic level that describes the representation of input and output and the algorithm itself, and the level of implementation that describes how the computation is realized physically.

There are two sides to this work and, consequently, two sides to its claims. Any claims about the human mind are to be taken as claims on the computational level, i.e. about what the overall strategy is that the human mind uses when it engages in learning and reasoning. When I talk about how artificial systems could learn and reason, my claims are not only computational-, but also algorithmic-level claims: I will describe specific algorithms and in as far as I claim that these may constitute part of systems that learns and reasons about interesting problems, these claims should be evaluated literally, not just as claims about the more general principles behind the algorithms.

In the next chapter, I will describe the philosophical, statistical and computational background that provides the foundation on which the actual model of learning and reasoning will be built, and which sheds light on the scientific context that gave rise to this approach.

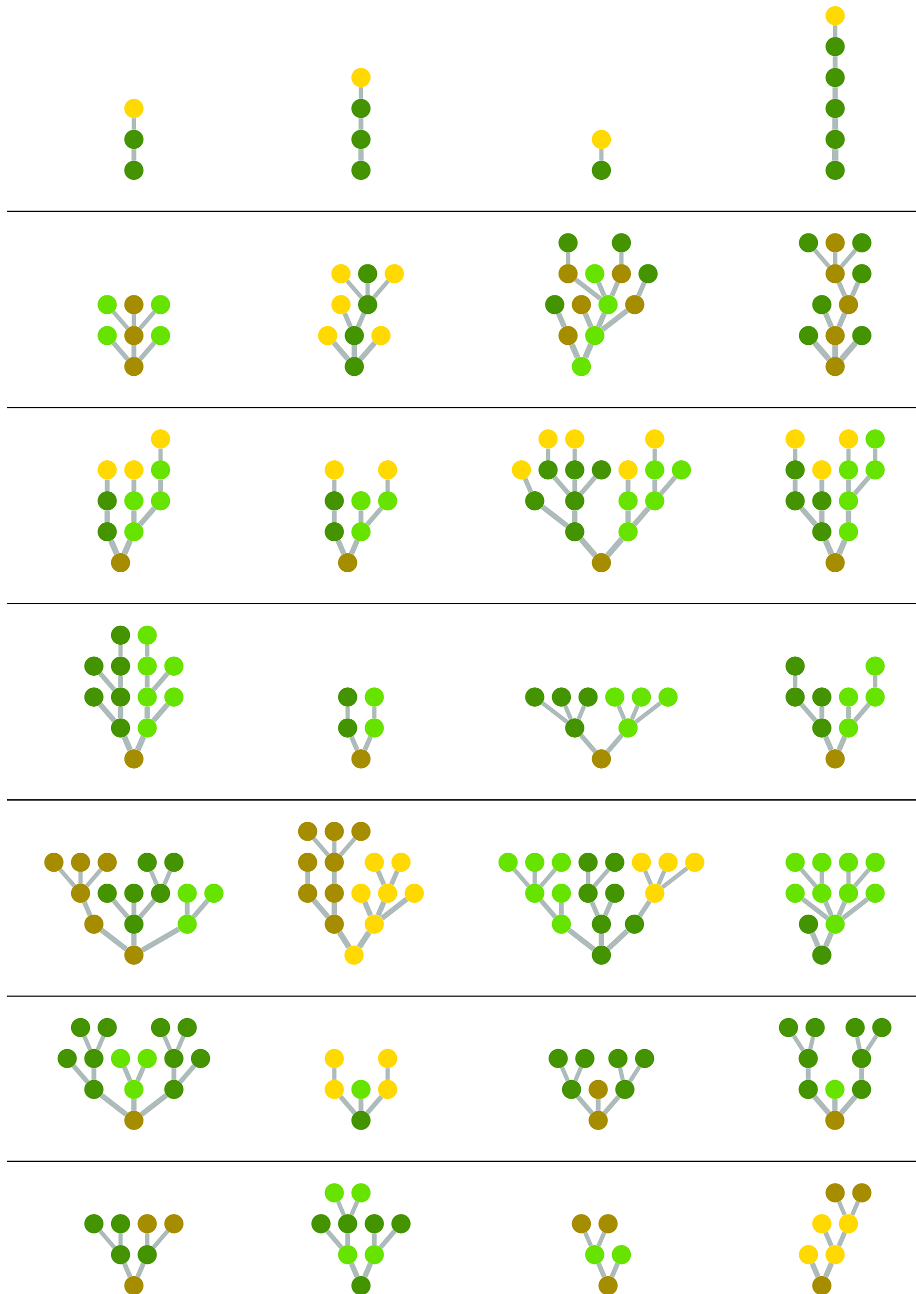


Figure 1.1: Examples for structured observations

Chapter 2

Background

2.1 Representationalism

In his encyclopedic introduction to cognitive science, Paul Thagard [Tha08] suggests that the central hypothesis of the field is "that thinking can best be understood in terms of representational structures in the mind and computational procedures that operate on those structures". Cognitive scientists disagree about the nature of this representation, about the extent to which it is implicit and explicit, and about what exactly "implicit", "explicit" and "representation" mean in the first place. However, the basic idea is clear: In order for a system to systematically derive true statements about an object that is out there in the world and that is not currently perceived, the information that is necessary to derive true statements must be found somewhere within that system[1]. In order to think about something, there must be some mental intermediary of the object of thought. This is what I will call the *representation* of the object.

[1] As a consequence, if we want to find out more about what is represented, we can look at the types of true statements that can be derived systematically in absence of perception. The information that is necessary to make these statements must then be part of the internal representation.

This notion of representation is a very broad one. The reasons for choosing this interpretation are that it includes all the more constrained versions as a subset, that my primary concern is to learn which constraints there are on any thinking system, and that I therefore do not want to prematurely exclude systems that process information in a way that differs from how humans do it. In the next section, we will see that there are much stronger constraints on which kinds of representations are suitable to serve as the objects of thought in worlds like ours.

By *representational medium*, I will refer to a structure that contains representations, that is, information about the external world[2]. We can talk about these structures on different levels of abstraction, and depending on which system we are talking about, we might prefer different views. For example, we might talk about how neurotransmitter levels represent something, i.e. talk about the implementational level, we might talk about how the connection weights in an artificial neural network represent something, i.e. talk about the algorithmic level, or we might talk about how certain data structures represent something, i.e. talk about the computational level. My considerations in the next section will mainly be on a computational level, because the constraints I want to talk about can be expressed most clearly using computational language.

[2]By using the term "external world", I do not want to exclude the agent that has these representations; it may well represent itself.

2.2 A Language of Thought

What do we know about how these representations look like for human beings, and what can we infer about what they should look like in any thinking system? This is the question this section deals with, and in doing so I will follow [Fod76] and [Fod07].

There are at least three properties that any representational medium needs to have in order to entertain complex thoughts about the real world and to use them in reasoning: Semanticity, productivity, and systematicity. I will first describe what each of these refers to, then show how another property, compositionality, explains productivity and systematicity, and argue that it is reasonable to call any representational medium that exhibits these properties a language.

Semanticity means that the medium must have a capacity to describe real and possible states of affairs, that is, it must have a means for truth and reference. If there is no way for the medium to represent what is real or possible, then thinking about what is real and possible cannot be done using this medium.

Productivity means that there must be no in-principle, i.e. computational-level, upper bound on the complexity of a thought that can be expressed in the representational medium, just like there is no in-principle upper bound on the complexity of a sentence in natural language[3]. This captures the intuition that a cognitive agent needs to have the competence to think any one

[3]We can always form new sentences from components:
The boy who likes a girl smiles.
The boy who likes a girl who likes a boy smiles.
The boy who likes a girl who likes a boy who likes a girl smiles.
The boy who likes a girl who likes a boy who likes a girl who likes a boy smiles.
...

of the infinitely many distinct thoughts that it could potentially think. Any in-principle boundary on the complexity of thoughts would give rise to the question what it is that makes thoughts that are only a little more complex suddenly unthinkable.

Systematicity means that the ability to entertain certain thoughts must be intrinsically connected to the ability to entertain certain others. For example, if I can entertain the thought that Spock does not like Italian food, then I must also be able to entertain the thought that Spock likes Italian food, otherwise it is questionable whether I can understand what the negated version of the thought means. Likewise, if I can entertain the thought that Gillian likes wales, I must also be able to entertain the thought that wales like Gillian.

In thinking about what it is that makes a representational medium productive and systematic, our current best (and only) explanation is that there is another property that a representational system suitable for thought must have and that underlies these two phenomena: namely, that the semantics of the representations is (to some extent) compositional.

Compositionality usually denotes the claim that the meaning of a complex expression in a representational medium is (up to a limited number of exceptions) fully determined by its structure and by the meanings of its constituents [Sza08]. However, it is doubtful that compositionality alone can fully explain the meaning of an expression *in use* [Cow]. In many situations, it seems to be the inferential role of an expression (i.e. how the expression is used) that determines its meaning. Therefore, I will take compositionality only to mean that the meaning of a compound expression is determined, *among other things*, by the meanings of its constituents[4].

[4]"The meaning of x" may not be a coherent notion. It may be that we always need to state a context (e.g. "the meaning of x for inference") if we want to make this notion precise.

Systematicity is connected to the concept of *compositionality* through the fact that thoughts that are so related seem to be composed partly out of the same semantic elements.

It is not clear that systematicity and productivity entail compositionality; the inference here is an "inference to the best explanation" that is supported by other phenomena, e.g. the fact that descriptions — compound symbols where each term contributes property specifications to the overall meaning — work the way they do [Fod07]. What we do know is that language is systematic and productive, and that this is the case because sentences have

constituent structure, i.e. sentences are made of smaller units that carry meaning. If thoughts, too, are systematic and productive, it is reasonable to suppose that this is due to their constituent structure — in particular if we do not know of any other explanation.

Taken together, what we require of a representational medium for thought is that thoughts can be combined in a productive way, that complex thoughts have constituents which, among other factors, determine the meaning of the complex (or compound) thought, that the relation between compound and constituent thoughts is a systematic relation, and that thoughts can express both real and possible states of affairs. This is close enough to a description of the properties that both natural and programming languages have for us to call any medium that fulfills these properties a *language of thought*[5].

[5] Although it might be more accurate to talk about a language of computation, since we do not distinguish conscious and unconscious thought [Fod76].

2.3 Generative Models

In the last section, considerations on what thoughts need to be able to express and which principles must hold for what they can express led us to the conclusion that any representational medium for thought must exhibit certain language-like properties. In order to learn more about what such a language of thought must look like, I will now consider how it is used in thinking about the world.

Assume I give you a machine that prints money. There is a paper tray that contains some empty paper on the left, a funnel for liquid color on the right, three rusty iron buttons with labels "$10", "$100", "$1000" on top, and a money tray on the bottom. I explain to you that you first put paper in the paper tray, then press your button of choice, one sheet of paper gets sucked into the machine, you pour color into the funnel, the color is used to print your note onto the sheet of paper, and a few seconds later, the note lands in the money tray on the bottom. If everything works right — I mention that the machine is from the last century and not as reliable as the fancy new money printing machines.

You decide that our economy suffers from deflation and that this is as good an opportunity as any to change that. Your first attempt results in two notes, but each only has color on one side, the other side is plain white. Not yet discouraged, you try again. You follow my instructions, choose the $100 button, pour a bit of

liquid into the funnel, and shortly after, look at your brand new note. It does not look quite right — the green is much lighter than it is supposed to be. "This is my last try," you say, choose $10, notice that the paper tray is empty, add some paper you brought, and what you get is indeed a $10 note, but it is much larger than the notes you know and it looks like the reason for this is that it has a white margin around the actual print. What might have gone wrong in each of these cases?

Which properties must your representation of the machine I just described have such that you can reason — make informed guesses — about what went wrong in each of these cases?

First, your representation must contain the relevant information about how the system works, i.e. it must contain information about the *causal structure* of the system. If your representation of the money making machine does not contain the information that the liquid you put into the funnel is used as a print color, then it is hard to figure out that you might need to put more liquid into the machine if the print is too light. Your representation cannot just be blank black box placeholder for the machine. It must be *a model with internal structure.*

Second, you must be able to use this information to think about *how this causal structure gives rise to observations.* If the information on how the machine works is there, e.g. because your representation contains the sequence of letters that made up my instruction sentences, but not in a form that can be used for computation, then your representation will be of little advantage to you. The details of how the machine works must be contained within your representation — your model of the machine — in a way that lets you estimate the sequence of events that lead to a certain outcome. If there is uncertainty in this generative process, then your model needs to define a probability distribution over the different mechanisms that could take place and, as a consequence, over the different outcomes of these mechanisms.

Third, you must be able to *invert* this model, i.e. given an observed outcome, you must be able to use it to infer what the generative process under this model looked like[6]. Such a process often contains hidden variables (or states), and it is often useful to be able to infer these hidden variables, since they explain the correlations and dependencies in the observations [Man09]. In the case of the money making machine, what happens between your

[6]Your model need not accurately mirror reality; given an observation, you need to be able to invert *your* generative model, but what you inferred by inverting it can be mistaken.

input and the output is a hidden variable. If you could guess what exactly caused the failure, you could use your guess to your advantage. If there are multiple ways that could result in the same outcome and you are uncertain about which one actually took place, then your representation needs to take this into account such that you can reason probabilistically. As we will see in subsequent chapters, the problem of inverting a model is the main difficulty posed by learning and reasoning in a complex, stochastic world.

Taken together, if a model mirrors the causal structure of some process in a way that lets us generate observations from the model and that lets us invert the model to infer hidden variables, then we will call it a *generative model*. In our world where there is much uncertainty and possibly true stochasticity, *stochastic generative models* — generative models with mechanisms that contain random choices — will be of particular interest.

We have derived the properties of stochastic generative models by thinking about what a representation needs to look like in order to enable reasoning about complex processes like the money making machine. It is therefore reasonable to assume that any representational medium appropriate for thought in a world with causal structure not unlike our own not only needs to have the language-like features described in the last section, but that it will also need the ability to concisely specify stochastic, generative models of the external world.

2.4 Probabilistic Inference

There are two related questions that need to be answered before we can make use of the notion of a stochastic generative model to explain learning and reasoning: First, how can we make our talk about generative models and, more specifically, about reasoning and learning with such models precise? Second, what are the rules that reasoning must obey in order to systematically arrive at true beliefs? The short answer is that we can formally talk about generative models using probability theory and probabilistic programs, and that the Bayesian interpretation of probability theory is the normative theory for reasoning under uncertainty[7]. The long answer is the remainder of this section.

[7] This is not uncontested, but as explained in this chapter, there are good reasons to believe it.

The laws of Boolean logic provide a normative theory of reasoning when all beliefs are either absolutely true or absolutely false. Boolean logic constrains rational belief and reasoning in two ways [Tal08]: First, enforcing deductive consistency constrains which beliefs can be held together at any one point. Second, the deductive rules of inference constrain admissible changes in belief. For instance, the rule of *modus ponens* requires that from premises P and $P \rightarrow Q$, one infers Q.

If Boolean logic is the normative theory for reasoning under certainty, then the Bayesian probability calculus is the normative theory for reasoning under uncertainty. A central assumption of the Bayesian probability calculus is that one's degree of belief in a proposition x can be represented by a real number $p(x)$ that is between 0 and 1, with 0 meaning "x is completely false" and 1 meaning "x is completely true". For this reason, this view is also called the *subjective interpretation* of probability. Together with the two desiderata that, first, reasoning with these degrees qualitatively corresponds to common sense and that, second, this reasoning is consistent, the rules of the probability calculus are uniquely determined. Only one set of mathematical operations for the manipulation of plausibilities has all these properties [JB03]. Besides this line of reasoning[8], there are others that come to the same conclusion, the most notable being the 'Dutch book' argument. This argument suggests that any violation of the laws of probability leads to bad choices. For example, one might then accept combinations of gambles that each appear fair when taken on their own, but which guarantee a loss when taken together [CTY06]. In summary, the Bayesian probability calculus seems to be the natural extension of the normative laws of Boolean logic to the realm of uncertain reasoning.

[8] This line of reasoning is formalized in the derivation of the Bayesian probability calculus from Cox' axioms.

What do the laws of probability tell us about how to reason and, in particular, about how to reason with generative models?

A useful concept will be the notion of a *hypothesis space*. In a given situation and for a given model, the hypothesis space includes all the hypotheses the model can entertain. Intuitively, if you think about a particular event before you have observed its outcome, then your hypothesis space of possible outcomes includes all the outcomes you can conceive, with some being more likely, some less. Likewise, if you think about a particular process and have observed its outcome but how exactly the process gave rise

to the outcome was hidden from you, then your hypothesis space for the hidden mechanism includes all the ways the process could have conceivably led to that outcome, with some ways being more likely, some less.

An idea central to the probability calculus is the evaluation of conditional probabilities ([CTY06], [TR99]). In many cases we are interested in the probability $p(h_i|d)$, i.e. the degree of belief in a particular hypothesis h_i (about the state of reality) given that we have observed some data d. If we can compute this quantity for different hypotheses h_i, we can compare hypotheses and judge how likely each is given our observations and given our prior knowledge. We could then use this result to choose our actions such that they take into account the different hypotheses to a differing degree. In many cases, we *can* compute this quantity: Bayes' rule (which can be derived from the definition of conditional probability) states how $p(h_i|d)$ can be calculated from quantities that are often known or can be approximated:

$$p(h_i|d) = \frac{p(d|h_i)p(h_i)}{p(d)} \tag{2.1}$$

The *likelihood* $p(d|h_i)$ denotes the probability that we would observe data d if h_i were in fact the true hypothesis. In the following, we will almost always write down our hypotheses about complex processes as stochastic generative models, and the problem of estimating the likelihood of observing data d given some generative model will be of great importance.[9]

[9] In particular, this problem will be the main motivation for introducing a new class of adaptive importance sampling algorithms later on.

The *prior probability* $p(h_i)$ denotes how probable we think it is that h_i is the true hypothesis before we have observed the data d. A particularly useful type of prior are formalizations of Occam's razor: If the notion of simplicity can be formalized, simpler hypotheses can be given an a priori higher probability.

The *posterior probability* $p(h_i|d)$ denotes our belief in h after we have observed data d. In computing this quantity, we have given a formal answer to the problem of inverting a generative model that came up in the last section. Given an observed outcome, we can compute the posterior probability of different generative models and thereby infer what the true generative model is likely to have looked like.[10]

[10] As long as we are only interested in comparing different hypotheses, we do not need to worry about the denominator, $p(d)$, since it is the same for all h_i. It serves to enforce the constraint that the probabilities on the right add up to 1, i.e. define a proper distribution.

If we can formalize our beliefs within the Bayesian probability calculus — that, as we have seen above, can be derived from

relatively weak common sense and consistency assumptions — then Bayes' rule normatively determines how these beliefs need to be updated in light of new data. Before we can implement an agent that can learn and reason with representations of interesting processes while following the normative rules that the probability calculus imposes on inference, there is at least one more thing we need to do: We need to show how we can formally write down *complex* generative models.

Programming languages are our best tools for the formalization of processes that generate values. Therefore, when I formally talk about stochastic generative processes, I will use a probabilistic programming language called *Church*.

Interlude: Church

Church provides a computationally universal[11] notation for stochastic processes and the distributions over return values that they induce ([GMR+08], [Man09]). Using Church, we can simulate (i.e. generate samples from) the generative processes expressed in the language both conditionally and unconditionally. A stochastic generative process is represented by a procedure that makes stochastic choices; by executing the procedure, we simulate from the process and sample a value from the distribution on return values induced by the process. By generalizing functions to distributions and evaluation to sampling, Church contains the pure[12] subset of Scheme, a dialect of Lisp.

[11] Turing completeness: Church programs can compute every function computable by a Turing machine.

[12] pure = side-effect free

In the following, I will first introduce a few concepts related to Church that we will make use of later on, then demonstrate them using a very simple Church program.

A *Church expression* describes a generative process: the meaning of an expression is specified through the primitive procedures `eval`, which samples from the process, and `query`, which samples from the process conditionally. A *Church environment* is a list of pairs, with each pair consisting of a variable symbol and a value [GMR+08]. A Church expression together with an environment defines a probability distribution over return values. We denote this distribution by $\mu_{expr,env}$ and the probability density of a particular return value v by $\mu_{expr,env}(v)$.

Primitive procedures are deterministic functions that are bound in from the underlying Scheme that is used to implement Church

and that are therefore a kind of black box to Church; Church cannot introspect the evaluation process that happens when such a procedure is applied to its arguments. Examples for primitive procedures are the logical operators `and`, `or` and `not`, operators that deal with data structures like `pair` and `list` and arithmetic operators like `+`, `-` and `*`.

Elementary random procedures (erps) are the stochastic equivalents of primitive procedures. As was the case for primitive procedures, Church does not introspect the evaluation process of erps. Each elementary random procedure is associated with a scoring function that returns the probability of a return value given an environment and operand settings. Examples for erps are `flip` (which throws a fair coin or, if given an argument, a weighted coin), `sample-integer`, `beta`, `gamma`, `uniform` and `gaussian` (which draw from the respective distributions).

A *generative history* for a `(expr, env)` pair is a sequence of recursive calls to `eval`, and their return values, made by `(eval expr env)`. A `(expr, env)` pair thus defines a distribution over generative histories. The distribution over return values then results from binning all histories that result in the same return value.

```
(and (flip) (flip))                                    (2.2)
```

In this example, we use the primitive procedure `and` and the elementary random procedure `flip`. The expression `(and (flip) (flip))` describes a generative process: First flip two fair coins, then return `true` if both come up `true`, otherwise return `false`. There are four possible generative histories, one for each combination of return values for the flips. We get the distribution over return values by binning the histories: There are three histories that result in the return value `false`, one that results in `true`, and each history has a probability of .25. Therefore, the distribution on return values has probability .75 for `false` and probability .25 for `true`.

In this section, I have shown a precise way to talk about learning and reasoning with generative models. Probabilistic programs formalize the notion of a generative model and the Bayesian probability calculus imposes normative constraints on inference. In the next chapter, I will connect these ideas to the idea of a representational language of thought and describe on a computational level what this implies for learning and reasoning

Chapter 3

Model

In the last chapter, we have seen that thought requires a representational medium that allows compositional, systematic, productive representations that can represent both real and possible states of affairs. A representational medium for thought is likely to have the ability to express stochastic models with internal structure that explains the causal process that results in observations and that can be used to reason backwards from observations to the hidden causes that are out there in the world. The Bayesian probability calculus provides a normative theory for reasoning and, in particular, for reasoning with generative models.

In this chapter, I will formally tie together these observations to explain, on a computational level, how reasoning and learning can occur in a complex world. This entails that we first formalize the language-like representational medium, then explain learning and reasoning as probabilistic inference, and finally show using probabilistic programming how these two connect.

The representational medium can be formalized as a grammar for probabilistic programs. Every program that can be expressed in this grammar defines a generative model. For the purpose of this chapter, we will assume that this grammar defines infinitely many generative models $m_1, m_2, ...$ and that at least one of these programs captures the causal structure of whatever process we are trying to learn well enough to be useful in predicting future observations[1]. If we want to ensure that this is the case in practice and if we do not know what the true generative process looks like, one solution would be to use a grammar that includes all computable programs.

[1] In Bayesian reasoning, if the prior distribution does not assign nonzero probability to any hypothesis that comes close to the true hypothesis, the posterior will not do so either.

By means of our grammar, we have specified the extent of the hypothesis space. If the grammar includes production probabilities, then we have also specified the prior distribution on hypotheses, i.e. which hypotheses are more likely a priori. For example, if our grammar is a probabilistic context-free grammar, we have induced a sort of Occam's razor prior: Hypotheses that are expressed by shorter programs are considered more likely a priori.

In the background chapter, we have encountered Bayesian inference as a means to make talk about reasoning under uncertainty precise and as a normative theory of this kind of reasoning. However, we have not yet made an attempt to specify what exactly is meant by "learning" and "reasoning" when expressed in this precise language.

Formally, what it means to learn a generative model, i.e. a probabilistic program, from observations $d = (d_1, d_2, ..., d_n)$ is to compute (or estimate) the posterior distribution p over all (infinitely many) generative models $m_1, m_2, ...$ that can be expressed within our language of thought, with the probability of each model being $p(m_i|d) \propto p(m_i)p(d|m_i)$.

Hereby, $p(m_i)$ specifies the prior probability we assign to the generative model m_i and $p(d|m_i)$ specifies the likelihood that this model assigns to the observed data d. After this computation is done, we know for each model m_i how probable we think it is that this one truly generated the data we have observed. When we want to make predictions using what we have learned, we can use either the generative model with the highest posteriori probability[2] or we can average the predictions of *all* generative models with each being weighted according to its posterior probability. The MAP approach is easier to compute, but, in many cases, disregards much of what we could learn from the data and therefore we will strive to use the fully Bayesian approach whenever possible.

[2] This approach is called MAP, Maximum A Posteriori

By reasoning, I denote the process of determining whether a particular proposition x is true in a real or counterfactual world or, equivalently, whether such a world has a particular property. When we want to reason with the posterior distribution p over generative models that we learned from our observations, we again have two possibilities. We can either choose the less precise MAP approach that reasons using the generative model that has the highest posterior probability, or we can average over all models in our hypotheses space, with each model contribution to the overall

conclusion in proportion to its posterior probability. In the first case, we compute $p(x|m_j)$ with $m_j = \arg\max_{m_i} p(m_i|d)$. In the second case, we compute the weighted average $\sum_i p(x|m_i)p(m_i|d)$. This takes care of both deductive and inductive reasoning, with deductive reasoning being only a special case of inductive reasoning.

I will now use the probabilistic programming language Church to formally describe how to connect a language-like representational medium with Bayesian inference such that we get a model that takes some data $d_1, d_2, ..., d_n$ as input and learns how the generating process out there in the world looks like[3]:

[3]The learned models can then be used to reason about the generating process, but this is not shown here.

```
; The true observations are fixed:
(define true-observations
  (list ...))

; The representational medium is specified
; as an arbitrary stochastic procedure from
; which we can sample expressions for
; generative models:
(define sample-expression
  (lambda () ...))

; The main inference loop. Church's query samples
; conditionally from the random world that is its
; first argument, returns what is given as its
; second argument and conditions on the third
; argument evaluating to true:
(query
  ; The generative model 'in the head':
  ((define hyp-expression (sample-expression))
   (define hyp-observations
     (repeat
       (length true-observations)
       (lambda () (eval hyp-expression (get-env))))))
  ; What we want to know:
  hyp-expression
  ; What must be true for any models
  ; we want to sample:
  (equal? hyp-observations true-observations)
  (get-env))
```
(3.1)

Two questions are left open: First, does this work in practice?

Is there a simple model for which we know that this kind of representation learning works? This is the question I want to answer affirmatively in the remainder of this thesis. Second, this looks computationally expensive, even for a simple example – *how* can this work in practice? This is the question on which method to use to compute the posterior distribution, and I am going to tackle it in the next chapter.

Chapter 4

Inference

In the last chapter, we have seen that the problem of finding models that explain our observations well can be modeled as computing the posterior distribution over all generative models conditioned on them generating the data, i.e. $p(m|d)$. When we make predictions, we can then weigh the prediction each individual model makes by how likely each model is under that distribution. One way to look at the task of sampling from the distribution over generative models is to see it as composed of the two subproblems of *model scoring* and *model space exploration*. In these subproblems, the two questions we ask are: First, if we have any given model m and data d, how can we compute $p(m|d)$? Second, how can we move from model to model such that what we get are actual samples from the distribution over models? As a precursor to the two methods described in this chapter, importance sampling and Markov Chain Monte Carlo, I will now look at these two questions in sequence:

Scoring

Given observations and a model, how can we score the model? More formally, what is the posterior probability of a certain model m given data d? Bayes' rule tells us that $p(m|d) \propto p(d|m)p(m)$, which means that we can again decompose our problem, namely into finding $p(m)$, the prior probability of the model, and finding $p(d|m)$, the likelihood of the data under our model. Since what we would like to use our score for is to compare different models, it does not matter that $p(d|m)p(m)$ is only proportional to the true posterior probability and that we do not know the normalization constant that would turn this into the true posterior.

Exploration

How do we explore the space of possible models efficiently? We

express our models in a representation language that exhibits productivity (through compositionality), therefore there are infinitely many possible models. Consequently, enumeration is not an option. If what we want to do is to find the model with the highest posterior score, we are looking at an optimization problem with a discrete (but not finite) set of solutions and thus can employ search and combinatorial optimization techniques. On the other hand, we might want to reason using the full posterior distribution over models. For example, in the case where we have two or more models which are approximately equally likely, we might not want to completely disregard one of the two models but use both in our judgements and weigh their influences accordingly. In this case, what we need to to is to represent the posterior distribution over models in a way that gives us an efficient method of getting at those models that make up the bulk of the probability mass.

In the next two sections, I will present an answer to these questions. I will first describe adaptive importance sampling, an algorithm that — among other uses — can be used to efficiently compute the likelihood of observations given a model, then Markov Chain Monte Carlo, an algorithm that can be used to sample from the posterior distribution over models given that a method is available to compute the likelihood of the observations.

Notation

Sans-serif letters, such as x, y, z, denote one-dimensional random variables unless stated otherwise. Bold sans-serif letters, such as $\boldsymbol{\mathsf{x}}$, $\boldsymbol{\mathsf{y}}$, $\boldsymbol{\mathsf{z}}$, denote multi-dimensional random variables[1]. Letters in regular, serifed fonts denote instantiations of random variables. For example, $\boldsymbol{\mathsf{x}} = \boldsymbol{x}$ means that $\mathsf{x_i} = x_i$ for all one-dimensional components $\mathsf{x_i}$ of the multi-dimensional random variable $\boldsymbol{\mathsf{x}}$. When we want to say that a random variable $\boldsymbol{\mathsf{x}}$ is distributed according to a distribution p, we write $\boldsymbol{\mathsf{x}} \sim p$. By $p_{\boldsymbol{\mathsf{x}}}$, we then denote the probability mass function for $\boldsymbol{\mathsf{x}}$ if $\boldsymbol{\mathsf{x}}$ is discrete, and the probability density function if $\boldsymbol{\mathsf{x}}$ is continuous. By $p_{\boldsymbol{\mathsf{x}}}(\boldsymbol{x})$ we denote the probability mass (or density) of the value $\boldsymbol{x}$ under distribution $p_{\boldsymbol{\mathsf{x}}}$. By $\langle f(\boldsymbol{\mathsf{x}}) \rangle$ we denote the expected value of function f when applied to random variable $\boldsymbol{\mathsf{x}}$. The set of possible values that $\mathsf{x_i}$ can take is denoted by $\Omega_{\mathsf{x_i}}$ and the set of possible values that $\boldsymbol{\mathsf{x}}$ can take is denoted by $\Omega_{\boldsymbol{\mathsf{x}}} = \bigtimes_{i=1}^{n} \Omega_{\mathsf{x_i}}$.

[1] i.e. vectors of random variables

4.1 Importance Sampling

In this section, I describe importance sampling, an inference method that — among other uses — lets us estimate how likely an observed value is *a priori* under some generating distribution even if we cannot sample from this distribution directly.

4.1.1 Introduction

Importance sampling is a technique for solving the following problem: We are given a target distribution p_{x} for which we would like to compute or estimate the expectation of a function $f : \Omega_{\mathsf{x}} \to \mathbb{R}$ [Mac03, Gog09]. Examples of properties we can estimate this way are the mean, the variance, and the prior and posterior score of certain values. Ideally, we would always compute the exact value ϕ of the expectation:

$$\phi := \langle f(\mathsf{x}) \rangle = \int_{-\infty}^{+\infty} f(\boldsymbol{x}) p_{\mathsf{x}}(\boldsymbol{x}) d\boldsymbol{x} \tag{4.1}$$

If we do not know the analytical expression for p_{x} or cannot compute the exact expectation for other reasons, we can still estimate the expectation from samples $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, ..., \boldsymbol{x}^{(n)}$ if we know how to sample from the target distribution[2]:

$$\hat{\phi} = \frac{1}{n} \sum_{i=1}^{n} f(x^{(n)}) \tag{4.2}$$

However, let us assume that we do not know how to sample from the target distribution p_{x}, but that we *do* know how to compute the probability of any particular value using p_{x}[3]. In this situation, we can use importance sampling to estimate the expectation.

The main idea behind importance sampling is to use a proposal distribution q_{y} instead of the target distribution p_{x} to generate samples and then, since some values will occur more often than they would if we could sample from p_{x}, and some less often, to reweight each sample when estimating the statistic. Therefore, we would like to choose a proposal distribution that is as close to p_{x} as possible. However, the only true constraint that we must not violate in order for importance sampling to give valid estimates is that q_{y} is a valid importance distribution for p_{x}:

[2] This method of estimation is justified by the laws of large numbers which state that the sample average converges to the expected value.

[3] For almost all interesting inference problems, we cannot easily generate samples from the distribution of interest. Here, the number of values in Ω_{x} is usually very high. By definition, correct samples will mostly consist of $x \in \Omega_{\mathsf{x}}$ for which $p_{\mathsf{x}}(x)$ is high – but how can we know where $p_{\mathsf{x}}(x)$ is high without looking at all values (which might be intractable)? [Mac03]

Definition 4.1 q_{y} *is a valid importance distribution for* p_{x} *if we can sample*[4] *from* q_{y}*, if for all* $\boldsymbol{y} \in \Omega_{\mathsf{y}}$*, we can compute* $q_{\mathsf{y}}(\boldsymbol{y})$ *and if for all* $\boldsymbol{x} \in \Omega_{\mathsf{x}}$*, if* $p_{\mathsf{x}}(\boldsymbol{x}) > 0$*, then* $q_{\mathsf{y}}(\boldsymbol{x}) > 0$*.*

[4]One way to formalize the notion "easy to sample from" is to say that the distribution can be expressed in a product form and, in this form, can be specified in polynomial space (depending on the number of random variables). [Gog09]

We will now derive how to estimate the expectation $\langle f(\mathbf{x}) \rangle$ from samples drawn from q_{y}, first for normalized densities, then for the more general case of unnormalized densities. Given a value $\boldsymbol{y}$ drawn from q_{y}, we define its importance weight as follows:

$$w(\boldsymbol{y}) := \frac{p_{\mathsf{x}}(\boldsymbol{y})}{q_{\mathsf{y}}(\boldsymbol{y})} \tag{4.3}$$

Using this definition, we rewrite the expression for the expectation $\langle f(\mathbf{x}) \rangle$:

$$\phi = \int_{-\infty}^{+\infty} w(\boldsymbol{y}) f(\boldsymbol{y}) q_{\mathsf{y}}(\boldsymbol{y}) d\boldsymbol{y} \tag{4.4}$$

We can now estimate the expectation ϕ by using samples $\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)}, ..., \boldsymbol{y}^{(n)}$ from the proposal distribution q_{y} adjusted by their importance weights[5]:

[5]This is justified by the fact that formula 4.4 denotes an expectation and by the laws of large numbers, as before.

$$\hat{\phi} = \frac{1}{n} \sum_{i=1}^{n} w(\boldsymbol{y}^{(i)}) f(\boldsymbol{y}^{(i)}) \tag{4.5}$$

What if we know p_{x} and q_{y} only up to a multiplicative constant[6]? It turns out that we can adjust our estimator for this case [Ber]. By p^*_{x}, q^*_{y}, and w^*, we denote the unnormalized versions of p_{x}, q_{y}, and w. z_p and z_y denote the normalization constants. We start out by rewriting our estimator 4.5 using this terminology:

[6]For Bayesian inference, this is a common situation since the computationally most difficult part is to compute the normalizing sum in the denominator:

$$p(h|d) = \frac{p(d|h)p(h)}{\sum_i p(d|h_i)p(h_i)}$$

$$\hat{\phi} = \frac{1}{n} \sum_{i=1}^{n} \frac{p_{\mathsf{x}}(\boldsymbol{y}^{(i)})}{q_{\mathsf{y}}(\boldsymbol{y}^{(i)})} f(\boldsymbol{y}^{(i)}) \tag{4.6}$$

$$= \frac{1}{n} \sum_{i=1}^{n} \frac{p^*_{\mathsf{x}}(\boldsymbol{y}^{(i)})}{q^*_{\mathsf{y}}(\boldsymbol{y}^{(i)})} \frac{z_q}{z_p} f(\boldsymbol{y}^{(i)}) \tag{4.7}$$

$$= \frac{\frac{1}{n} \sum_{i=1}^{n} \frac{p^*_{\mathsf{x}}(\boldsymbol{y}^{(i)})}{q^*_{\mathsf{y}}(\boldsymbol{y}^{(i)})} f(\boldsymbol{y}^{(i)})}{z_p \big/ z_q} \tag{4.8}$$

All that is left to do is to show that we can estimate the denominator from samples drawn from q_{y}:

$$
\begin{aligned}
\frac{z_p}{z_q} &= \frac{\int_{-\infty}^{+\infty} p_{\mathbf{x}}^*(\boldsymbol{x}) d\boldsymbol{x}}{z_q} && (4.9) \\
&= \int_{-\infty}^{+\infty} \frac{p_{\mathbf{x}}^*(\boldsymbol{x})}{q_{\mathbf{y}}^*(\boldsymbol{x})} \frac{q_{\mathbf{y}}^*(\boldsymbol{x})}{z_q} d\boldsymbol{x} && (4.10) \\
&= \int_{-\infty}^{+\infty} w^*(\boldsymbol{x}) q_{\mathbf{y}}(\boldsymbol{x}) d\boldsymbol{x} && (4.11) \\
&\approx \sum_{i=1}^{n} w^*(\boldsymbol{y}^{(i)}) && (4.12)
\end{aligned}
$$

We can now estimate expectations using importance sampling even without normalized densities:

$$
\hat{\phi} = \frac{\sum_{i=1}^{n} w^*(\boldsymbol{y}^{(i)}) f(\boldsymbol{y}^{(i)})}{\sum_{i=1}^{n} w^*(\boldsymbol{y}^{(i)})} \tag{4.13}
$$

In summary: If we have some valid importance distribution and if we can compute the unnormalized score of the samples under the posterior, then we can use importance sampling to estimate arbitrary expectations for the posterior distribution.

4.1.2 Probabilistic Programs

We will now describe how to apply importance sampling to probabilistic programs[7]. We first state what we want to accomplish illustrated by an example, then describe a simple algorithm that accomplishes it, followed by an algorithm that accomplishes the same goal much more efficiently.

[7]This section require some understanding of how the Church evaluator works. For background on metacircular evaluation in Scheme, see [ASS]; for Church, see [GMR+08].

We presented importance sampling as a means to estimate a expectation $f : \Omega_{\mathtt{x}} \rightarrow \mathbb{R}$ for a target distribution $p_{\mathtt{x}}$. Now, let our target distribution be the conditional distribution defined by the following program constrained to the return value `(pair true *)`[8]:

[8]By `*`, we denote the wildcard, i.e. an unconstrained return value.

```
((lambda (x y) (pair (and x y) x))
 (flip 0.01)
 (flip 0.01))
```
(4.14)

If we repeatedly run this program without constraint, the most sequence of samples we get is likely to start with `(pair false false) (pair false false) (pair false false)`. The number of samples that fulfill the constraint `(pair true *)` is likely to be very small. As a consequence, if we want to efficiently sample from this distribution, running the program forward and rejecting all samples that are no good is very inefficient.

Let us assume that we are not interested in the samples themselves, but that we want to compute some expectation for this target distribution. In other words, we want to compute a function of the sampled values from this program *but* we are interested only in those values where the first value of the returned pair is equal to `true`. As was the case in the importance sampling setup described above, it is not clear how to sample from the target distribution I just described. However, if we could sample from a different distribution and reweight each sample according to the ratio of true score and actual score, then we could estimate expectation such that the result is *as if* we could sample from the true distribution.

A starting point for this idea is to take samples from the unconstrained program and then reweighting the samples according to the ratio of constrained program score to unconstrained score. This approach — sampling from the prior and reweighting accordingly — is called *likelihood weighting*. In order to reweight the samples, we need to know the probabilities of return values

under the (constrained) target distribution up to a multiplicative constant. Since all samples that do not fulfill the constraint have probability 0 and since we do not need to renormalize, we can directly use the probabilities we get from evaluating the unconstrained program, which means that all weights end up being either 0 or 1 and that all we do in this case is to reject samples that do not fulfill the constraint. This has the same problems naïve rejection sampling has, namely that we cannot use many of the samples we take and thus waste computation time. In our example, we discard all samples where not both of the two `(flip)`s happen to return `true`. For programs where most of the values sampled from the prior distribution have probability 0 in the posterior distribution, we need better proposal distributions than the prior.

In order to devise a better proposal distribution, let us go back to program 4.14 and go through the sequence of evaluation steps that would need to happen if we wanted to sample from the target distribution where all values fulfill the constraint `(pair true *)`:

1. The first step in the evaluation is the application of the operator with expression `(lambda (x y) (pair (and x y) x))` to the two operands with expressions `(flip)` and `(flip)`. Like Scheme, Church[9] is an applicative-order language. This means that at procedure applications, the operands are evaluated before the operator is applied. Before we have started to apply the operator to the operands, we do not know how to constrain the operands such that we get the return value `(pair true *)` out of the application. Therefore, we need to *delay the evaluation* of the operands until they are required.

2. Next in sequence, we evaluate the procedure body of `(lambda (x y) <body>)` with `x` bound to the delayed value for the first `(flip)`, `y` bound to the delayed value for the second `(flip)`, and the constraint `(pair true *)` on the whole expression. This means that we evaluate `(pair x (and x y))`, an application of the primitive procedure `pair`. What we need to do in order to sample from the distribution where all values match the pattern `(pair true *)` is to *invert the primitive procedure* `pair` to find out which operand values result in the return value that matches this pattern. Such an inversion would tell us that the first operand can be any

[9] More precisely: MIT-Church without `importance-eval` is applicative-order.

value and that the second operand must be `true`. With knowledge about how the operands must look like, we can go on evaluating the operands.

3. We evaluate the first operand of `(pair (and x y) x)`, the expression `(and x y)`, with constraint `true`, notice that it is a primitive procedure application and therefore we need to *invert the procedure*. Since the only operand values for which `and` can return `true` are `true` for both operands, we know what to do: We need to evaluate both operands with constraint true. Both operands are variable lookups, and for both operands the value has been delayed. We force both operands with the constraint `true`, and in doing so apply the elementary random procedure `(flip)` with constraint `true` twice. If we want to sample from the target distribution for the overall expression, we know that the two applications both need to return true, therefore we *set the elementary random procedures* to `true`. Having evaluated both operands, we now apply the primitive procedure `and` which results in the desired value `true`.

4. We evaluate the second operand of `(pair (and x y) x)`, the expression `x`, notice that it is a variable lookup which has already been forced and therefore look up and return its value `true`.

5. Having evaluated both operands for `(pair (and x y) x)`, we apply `pair` to the returned values `true` and `true` which results in the overall result `(pair true true)` — a value that matches our constraint `(pair true *)`!

We will now review the three ways in which we had to modify the evaluation of the program in order to get a sample that fulfills the constraint, i.e. a sample that does not have probability 0 under the posterior distribution. In particular, we will show how these modifications fulfill the criteria we have posed for a valid importance sampler in the last section: That we can always compute the probability of a return value under the proposal distribution and that, if the probability under the posterior distribution is greater than 0, the probability under the proposal distribution is greater than 0, too.

Setting of elementary random procedures
If we encounter an elementary random procedure application like `(flip)` and know that our target distribution has a certain return value for this application, e.g. `true`, we set the application to the desired value. We know both the posterior score[10] and the proposal score, whereby the latter is 1 if we can successfully set the value. If we cannot set the value because the elementary random procedure assigns probability 0 to it (and therefore need to assign a proposal score of 0), we know that this is also the case under the posterior distribution. Since only these two cases are possible, both criteria are fulfilled.

[10]Given an arbitrary value, an elementary random procedure can return its likelihood.

Inversion of primitive procedures
At applications of primitive procedures like `pair` and `and` where we have a constraint on the return value, we invert the procedure to find out which operand values can result in the desired return value. For each primitive procedure, we therefore store an inverse procedure that, given a return value, returns a list of n possible operands. We uniformly draw one operand choice and go on with the operand constraints being equal to the chosen operand values[11]. This adds a factor of $\frac{1}{n}$ to the proposal score for this sample. Since we know that our inverses contain all possible operand values that could lead to a certain return value, we know that we are not excluding any value that has a probability greater than 0 under the posterior.

[11]Whereas one solution is to randomly choose one of these inverses and then constrain the operands to have the desired values, this will result in a proposal distribution that is not as close to the prior distribution as we might wish. We can use *sequential inverses* as a remedy: We first choose a sequence on the operands, then evaluate in sequence all operands as long as there are inverses that can match any operand choice. When this is no longer the case, we pick an inverse at random that matches our initial operands and force the remaining operands to the desired inverse values.

Delayed evaluation
Instead of evaluating operands before applying compound procedures, we delay the evaluation until the value is actually needed. Delaying the evaluation allows us to push constraints further down the evaluation than would be possible otherwise, where we would have to evaluate the operands of compound procedures without constraints which could result in many samples with probability 0 under the posterior distribution.

What connects these three ideas is that we push the available evidence — the constraint — down the evaluation branches in order to sample from a distribution that is close to the posterior and in particular does not return samples that have probability 0 under the posterior, since these samples are useless for estimating the expectation we care about. In the next section, we will see how we can do even better by adapting the proposal distribution step

by step such that even in those cases where pushing the evidence was not sufficient to avoid a sample with probability 0 under the posterior, we can learn to avoid making the same mistake in future samples.

4.2 Adaptive Importance Sampling

In this section, I describe a version of importance sampling that improves over time. By learning from the samples it takes something about the distribution it is sampling from, this method can incrementally achieve lower rates of rejected and low-probability samples and thus it can result in more accurate estimates.

4.2.1 Introduction

Adaptive importance sampling denotes a version of importance sampling where we do not take all our samples from the same proposal distribution q_{y}, but instead sample from a sequence of distributions $q_{\mathsf{y}}^{(1)}$, $q_{\mathsf{y}}^{(2)}$, ..., $q_{\mathsf{y}}^{(n)}$. This is of particular interest if each sample gives us information about how the posterior distribution looks like, such that we can improve our proposal distribution $q_{\mathsf{y}}^{(i)}$ in the following step.

According to [RC04], it is possible to keep the original importance weights in this setup and still produce an estimate that converges to the true estimate. This means that our estimator is still

$$\hat{\phi} = \frac{\sum_{i=1}^{n} w(\boldsymbol{y}^{(i)}) f(\boldsymbol{y}^{(i)})}{\sum_{i=1}^{n} w(\boldsymbol{y}^{(i)})} \tag{4.15}$$

with the weights being defined as follows:

$$w(\boldsymbol{y}) := \frac{p_{\mathsf{x}}(\boldsymbol{y})}{q_{\mathsf{y}}^{(i)}(\boldsymbol{y})} \tag{4.16}$$

Here, $q_{\mathsf{y}}^{(i)}$ refers to the proposal distribution that was used to generate sample $\boldsymbol{y}$.

However, there is reason to look into modified estimators in order to arrive at more stable estimates [CMMR09]. In particular, if the (unnormalized) weights from one proposal distribution are very large, they will dominate the other samples in the final approximation, no matter how efficient the other proposal distributions are.

4.2.2 Probabilistic Programs

I will now describe how to apply adaptive importance sampling to probabilistic programs. We have seen that the goal of adaptation is to change the distribution we are sampling from such that we get samples that are distributed more like those from the posterior and less like those from the initial, possibly inefficient importance distribution. We also know that we must always maintain a valid importance distribution to keep the guarantee that our estimated expectations approximate the true value. In the following, I present a framework for adaptive importance sampling with probabilistic programs that can accommodate different adaptation policies and, at the same time, makes it easy to ensure that we maintain a valid importance distribution.

I will proceed as follows: First, I define what I mean by a *valid adaptation policy* and shortly review the programming language constructs that will play a central role in our adaptation policies for probabilistic programs. I then formally characterize the class of adaptation policies that are possible within this framework. I define the concept of an *importance table* and, using this concept, characterize a class of *valid* adaptation policies for probabilistic programs and prove that any policy within this class maintains a valid importance distribution.

Definition 4.2 *π is a valid adaptation policy if, given a valid importance distribution $q_{\mathbf{y}}^{(t)}$ for the target distribution $p_{\mathbf{x}}$ and some information i, $\pi(q_{\mathbf{y}}^{(t)}, i)$ returns a new valid importance distribution $q_{\mathbf{y}}^{(t+1)}$.*

We will see that, in our framework, adaptation always happens when functions are applied. Therefore, the two language constructs that are at the center of adaptation are both functions: primitive procedures and elementary random procedures. We have seen both constructs in our introduction to Church. Primitive procedures are deterministic functions like `and`, `or`, and `pair`. For many primitive procedures, inverses exist. This means that a primitive procedure `p` is associated with another primitive procedure `inv-p` that, given a desired return value, returns the list of operand assignments that would result in this return value if `p` was applied to any of the assignments. Elementary random procedures are stochastic functions like `flip`, `sample-integer` and `gaussian`; they do not have inverses, but they do have a scoring

function that returns the probability of a return value given an environment and operand settings.

I will now characterize the class of adaptation policies that are part of the framework I am about to describe. As first introduced in the section on importance sampling for probabilistic programs, we are still in a setup where we try to push information about the target value of the evaluation as far down the evaluation tree as possible. This means that at each evaluation, we have information on what the target value of this evaluation should be, be it a fixed value or the wildcard setting that does not constrain the return value.

One way of looking at any adaptation policy is to see it as separated into information collection (what information on how the posterior looks like is stored during the evaluation process), information propagation (what additional information is deduced from what is collected), and information usage (how the stored information is used to change the importance distribution). I will first introduce the data structure that is used to store information, the importance table, then describe when and how information is stored, propagated, and used.

The importance table is a hash table that uses `(expr, r-env)` pairs as keys and association lists[12] $((v_1, w_1), (v_2, w_2), ..., (v_n, w_n))$ as values. In the introduction to Church, we have seen that an expression together with an environment defines a distribution over return values. This distribution does not depend on the complete environment, but only on part of it (i.e. those variables within the environment that are actually used in the expression). These distributions are what we will store information on; in particular, we will store information on what values these distribution can return and how probable the values are. This is why the hash table keys each consist of an expression and an relevant environment, and this is why the hash table values are association lists of return values associated with weights[13].

By $\omega_{expr,env}(v)$, I denote the function that looks up the importance table entry for the pair `(expr, r-env)` and returns the weight that is associated with the value v in the association list for this pair. If no entry is found in the importance table for this `(expr, r-env, val)` triple, then ω returns a default score d as defined by the active adaptation policy[14].

In our framework, information collection happens at the eval-

[12] An association list is a list of pairs. The first element of the pair is called the key, the rest of the pair is called the datum.

[13] There are adaptation policies for which it is useful to store more than a single real number. For example, one might want to store a list of importance weights for each `(expr, r-env, val)` triple. I limit my analysis to those that store a single real-numbered weight

[14] We might want to use adaptation policies that specify more about the weight lookup function than just the default value. For example, for continuous values it could be useful to take into account close neighbors of the value that is looked up. I will not talk about these kinds of extensions here.

uation of erp applications, whereas information propagation and usage happen when primitive procedures are applied. In the following, I will first explain these three policy elements, then describe more formally what our class of adaptation policies looks like:

Collecting Information
After the application of an elementary random procedure with target value v (which may be a wildcard), we have new information: We now have another pair of (unnormalized) posterior score and (unnormalized) importance score for this return value in the context of a particular expression and environment. At this point, we use a function f that is supplied by the policy to compute the new weight of this target value for the `(expr, env)` pair that we just evaluated:

$$w = f(\boldsymbol{p}, \boldsymbol{q}) \tag{4.17}$$
$$= f((p_1, ..., p_n), (q_1, ..., q_n)) \tag{4.18}$$

Here, p_n is the score of the return value under the target distribution and q_n is the score of the return value under the importance distribution. Since adaptation policies might take into account earlier evaluations of `(expr, env)` that resulted in v, we not only allow the function that computes the weight to depend on the latest scores, but on all previous scores. The weight w that is returned by f must be a real number between 0 and 1. We then add `(v, w)` to the association list that is stored in the importance table for the current `(expr, r-env)` pair.

Propagating Information
At the application of a primitive procedure with a fixed target value v, we propagate information from the operand expressions to the overall expression. We first get a list of all inverses $(inv_1, ..., inv_n)$ that lead to this target value, with each inverse inv_i consisting of a list of operand values $(op_1, ..., op_m)$. Using these values and the importance table, we compute the overall weight of the `(expr, env)` pair that we are currently evaluating as follows:

$$w = \sum_{i=1}^{n} \prod_{j=1}^{m} \omega_{expr_j, env}(inv_{i,j}) \tag{4.19}$$

We then add `(v, w)` to the association list that is stored in the importance table for the current `(expr, r-env)` pair.

This propagation policy assumes that the operand values are independent, i.e. that evaluating one operand does not affect the evaluation of other operands. When we use delayed evaluation, this is no longer true. For example, when we evaluate `((lambda (x) (pair x x)) (flip))` constrained to the return value `(pair true false)` with the operand `(flip)` delayed, this policy cannot learn that the pair application cannot return `(pair true false)`.

This is not a problem for the current project for two reasons. First, it does not result in an invalid propagation mechanism, only in a less efficient one. When we choose the wrong inverse, the resulting sample gets rejected and does not factor into expectation computations. Second, when we later apply our methods to learn programs, we limit ourselves to simple programs without lambdas (since the required methods to switch between programs without lambdas and those with lambdas are not implemented yet; I will explain this in more detail later on).

Using Information

Our goal is to sample each inverse with a probability proportional to its total posterior score, i.e. we want to use each operand assignment as frequently as it would be used under the posterior distribution. Therefore, it is at the application of primitive procedures with a fixed target value v that we allow our adaptation policies to use stored information. We look up all inverses $(inv_1, ..., inv_n)$ and thus know the operand settings $(op_1, ..., op_m)$ for each inverse. We use this to compute a weight w_i for each inverse:

$$w_i = \prod_{j=1}^{m} \omega_{expr_j, env}(inv_{i,j}) \tag{4.20}$$

We then sample the inverse that we are going to use from the distribution over inverses where each inverse has a probability proportional to its weight. Thus, an adaptation policy can change the relative frequency of different inverses and, ideally, change it

in a way that makes the inverse usage close to that under the posterior distribution.

We can now say more precisely what an adaptation policy consists of:

Definition 4.3 *Within our framework, an adaptation policy* π *is a tuple* (f, d)*, where* f *is a function* $(\boldsymbol{P}, \boldsymbol{Q}) \rightarrow [0, 1]$ *and* d *is a default weight that is used when no weight is stored in the importance table for an* `(expr, r-env, v)` *triple. Both* $\boldsymbol{p} \in \boldsymbol{P}$ *and* $\boldsymbol{q} \in \boldsymbol{Q}$ *are unnormalized probabilities.*

What remains to be shown is under which conditions such a policy is valid, i.e. under which conditions the policy creates a new *valid* importance distribution if applied to a valid importance distribution.

Theorem 4.1 π *is a valid adaptation policy if* π *is a policy* (f, d) *with the property that* $f(\boldsymbol{p}, \boldsymbol{q}) > 0$ *if any* $p_i > 0$*, and that* $d > 0$*.*

Before we can prove that this statement is true, we need to review the concept of a generative history and to introduce concept of a valid importance table.

In our introduction to Church, we have seen that a generative history for a `(expr, env)` pair is a sequence of recursive calls to `eval`, and their return values, and that a `(expr, env)` pair thus defines a distribution over generative histories. This gives us another way to think of inverses: When we invert a primitive procedure to get all the operand settings that can result in a return value v, we effectively partition the space of generative histories for the expression that contains the primitive procedure application by the different ways to get the desired value v.[15]

[15] In our analysis, we assume that inverses are complete, i.e. contain *at least* all operand settings that can lead to a return value v.

Definition 4.4 *A valid importance table is an importance table where* `(v, 0)` *is stored in the association list for* `(expr, r-env)` *only if no generative history we get from evaluating* `expr` *in* `r-env` *results in value* v*.*

Proof Structure

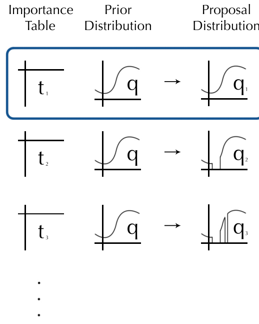
Let μ be an adaptation policy with $f(\boldsymbol{p}, \boldsymbol{q}) > 0$ whenever any $p_i > 0$, and $d > 0$. The proof that shows that μ is a valid adaptation policy will proceed as follows:

1. If we have a valid importance table t and a valid proposal distribution q, then μ constructs a new *valid* proposal distribution q^+ using the information in the importance table.

2. The importance table t_{i+1} that μ constructs from a valid importance table t by information collection and propagation is a valid importance table.

$\therefore$ If we start out with a valid importance table t_i and a valid proposal distribution q_i, μ constructs a new valid proposal distribution q_{i+1} via importance table t_{i+1}. Therefore, μ is a valid adaptation policy.

Note that the empty importance table is a valid importance table, that we can therefore always start with a valid importance table and that we need not make the availability of such a table part of the requirements of a valid adaptation policy as described in theorem 4.1. Similarly, note that — as seen in likelihood weighting — that the prior distribution for a program is a valid importance distribution.



Proof, Part 1: Constructing Valid Proposal Distributions

Assume that we have a valid proposal distribution q for a certain program, i.e. a distribution over generative histories where each history that is possible under the posterior distribution p is also possible under q. Further assume that we have a valid importance table t.

We will now show that the adaptation policy μ uses the importance table t to construct an adapted proposal distribution q^+ that is *valid*:

1. If an `(expr, env)` pair is in the importance table t with a value v and a weight 0, then no generative history we get from evaluating `expr` in `env` results in value v.

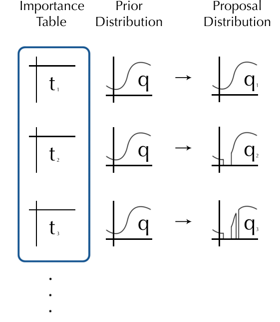
2. If there is no generative history for a `(expr, env)` pair that returns v, then there is no generative history in the posterior distribution over histories that includes a subhistory where `(expr, env)` evaluates to v.

3. If h is a generative history that is impossible under the posterior, and if q^+ is equal to valid q except for the fact that h is excluded from the set of possible generative histories, then q^+ is a valid proposal distribution.

∴ If we have a valid importance table t and a valid proposal distribution q, then μ constructs a new valid proposal distribution q^+.

Statement 1 is true by the definition of “valid importance table”, statement 2 follows from subset reasoning and statement 3 follows from the definition of “valid proposal distribution”.

Proof, Part 2: Constructing Valid Importance Tables



Assume that we have a valid importance table t_i. We will now show that the adaptation policy μ uses the information it collects and propagates to construct an importance table t_{i+1} that is again valid.

1. The importance table t_{i+1} is equal to t_i except for updates two adaptation rules.
2. Rule 1: Information collection at elementary random procedures:
   - i As a consequence of the rules for the weight computation function f, we only update t_{i+1} with `(expr, env)`, value v, weight 0 if an erp application returns the score 0 for v.
   - ii If the erp application returns the score 0 for v in the current environment, we know that there is no generative history for `(expr, env)` that results in v (since this is what a score of 0 means).
   - ∴ Adaptation at elementary random procedures preserves importance table validity.
3. Rule 2: Information propagation at primitive procedures:
   - i As a consequence of the sum-product propagation rule, we only update t_{i+1} with `(expr, env)`, value v, weight 0 if t_i tells us that each possible combinations of operand values that could lead to v (as determined by the appropriate inverse procedure) has weight 0 in the current environment. If this is the case, then, since t_i is valid, no combination of operand settings has a generative history under the posterior.
   - ii If all operand assignments that could lead to the return value v are impossible under the posterior distribution, then we know that the return value itself is impossible under the posterior distribution for `(expr, env)`, i.e. no generative history for `(expr, env)` exists that results in v.
   - ∴ Adaptation at primitive procedures preserves importance table validity.

∴ The importance table t_{i+1} is valid.

4.2.3 A Constraint Propagation Policy

Having established a class of valid adaptation policies, it is now easy to describe the constraint propagation policy that we will use to efficiently score observations from tree-generating programs. I start by introducing some terminology, then describe the actual policy.

If a value v is never returned by the distribution induced by a given `(expr, r-env)` and we are evaluating this pair with constraint v, then we call this situation a *dead-end*. We call the recorded fact that this `(expr, r-env)` pair cannot return this value a *nogood* (both based on terminology used in the SAT[16]-community). Using this terminology, the goal of our constraint propagation policy is to store knowledge about the return values of `(expr, r-env)` pairs and to use this knowledge to avoid dead-ends.

[16] SAT: The Boolean satisfiability problem

This is a formal description of such a policy:

$$f(\boldsymbol{p}, \boldsymbol{q}) = \begin{cases} 1 & \text{if any } p_i > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.21}$$

$$d = 1 \tag{4.22}$$

Since this policy fulfills the requirements that that $f(\boldsymbol{p}, \boldsymbol{q}) > 0$ if any $p_i > 0$, and that $d > 0$, it is a valid policy.

In the last sections, we have derived a class of importance sampling algorithms for probabilistic programs that can be used to efficiently compute expectations. In particular, we have specified one algorithm in this class, a constraint propagation algorithm, that propagates information about which values cannot be returned by elementary random procedures up at applications and thus avoids ahead of time the choosing inverses that contain these values. Using this algorithm, we will be able to compute the likelihood of observations under tree-generating programs by quickly excluding all those configurations of the program from our considerations that could not have given rise to the observations.

4.3 Markov Chain Monte Carlo

Before I introduce the practical setup we use to demonstrate structure learning, I will shortly describe the second class of inference methods we will employ, Markov Chain Monte Carlo (MCMC), and how they are used in Church for conditional sampling.

4.3.1 Introduction

In contrast to importance sampling methods, we are now not only interested in computing the expectation of some function applied to samples from a complex probability distribution, but we want to (approximately) sample from the distribution[17]. I will first introduce some background concepts, then explain the idea behind MCMC.

[17] Typically, it is very difficult to sample from the distribution directly because the state space is large and we do not know which parts of the state space contain high-probability values. MCMC methods are used to overcome this difficulty.

A Monte Carlo algorithm is an algorithm that arrives at its result by repeatedly sampling from a probability distribution.

A Markov chain is a sequence of random variables where each variable is independent of all of its predecessors given its immediate predecessor. Under certain conditions[18], this chain converges to a stationary distribution as its length increases, i.e. the probability that a variable in the chain takes on a certain value converges to a fixed quantity [GY06]. Hereby, it does not matter from which state we start, given that we wait long enough for the chain to converge.

[18] Convergence of a Markov chain with transition kernel K to the distribution p requires two things: First, p is an invariant (or stationary) distribution for K, i.e. $p = pK$. Second, K must be ergodic. A kernel is ergodic if it is irreducible (any state can be reached from any other state) and aperiodic (the stochastic walk does not get stuck in cycles). [Bon]

The main idea behind MCMC methods is to construct a Markov chain that has as its stationary (or long-run) distribution the distribution we want to sample from and that is cheap to compute. Then, by recording the sequence of states we visit when simulating the Markov Chain, we accumulate (correlated) samples from the target distribution. If we want to sample according to the probability under the distribution of interest, this stochastic walk through state space must be such that we spend most of the time producing samples from high-probability regions.

4.3.2 The Metropolis-Hastings Algorithm

The Metropolis-Hastings (MH) algorithm is the most popular method to construct a Markov Chain that has the properties required to converge to the target distribution, and most other

MCMC algorithms can be seen as special cases or extensions of this method [AdFD03].

The main ingredient for MH is a proposal distribution that specifies how to sample a candidate next state given the current state of the Markov chain. Starting from this proposal distribution, MH constructs an acceptance function. Together, proposal distribution and acceptance function specify a Markov chain that has the desired target distribution as its stationary distribution.

Just as was the case for importance sampling, the choice of the proposal distribution is crucial: If the probability is very low that we propose a sequence of steps that lead to a region that has high probability under our target distribution, then it can take a long time until the chain converges and, consequently, until we get samples that accurately represent the target distribution.

The acceptance function gives the probability of accepting this candidate as the actual next state. If the candidate is rejected, the current state becomes the next state. If the target distribution is p and the proposal distribution q, then the acceptance probability of going from current state s to candidate state s^* is given by $A(s, s^*) = \min\{1, \frac{p(s^*)q(s|s^*)}{p(s)q(s^*|s)}\}$. The normalization constant for p and q cancels, therefore all we need to know about the target distribution is some quantity proportional to the probability of a value under this distribution for all values.

4.3.3 MCMC and MH in Church

We have already seen that the probabilistic programming language Church contains a language construct for sampling from conditional distributions, `query`. In the MIT-Church implementation, `mh-query` samples conditionally by using the Metropolis-Hastings algorithm. In light of the fact that this is currently the only universal inference algorithm that works reasonably well for program learning[19], I will now give an overview on *how* the universal `mh-query` works.

[19] The question whether adaptive importance/rejection sampling on its own is an useful method for program learning is still open.

Before we start, I will give a simple example that illustrates how `mh-query` is used in Church. Assume I have flipped an unfair coin ten times and saved the resulting sequence of heads (encoded by `true`) and tails (encoded by `false`) into a variable

`true-coin-flips`:

```
(define true-coin-flips
  (list false false true true false
        false false true false false))
```
(4.23)

Now I would like to infer how biased the coin was – the weight of the coin – but without thinking about a special-purpose way to do inference. What I would like to do is to let Church construct a Markov chain such that visiting the states of this chain is (in the limit) equivalent to sampling from a distribution over weights conditioned on coin flipping returning the results as specified in `true-coin-flips`. Here is how to write this down in Church:

```
(repeated-mh-query 10
  ; Random world:
  '((define hyp-coin-weight (uniform 0 1))
    (define hyp-coin-flips
      (repeat 10 (lambda () (flip hyp-coin-weight)))))
  ; What we want to know:
  'hyp-coin-weight
  ; Condition:
  '(equal? hyp-coin-flips true-coin-flips)
  (get-env))
```
(4.24)

Using `(repeated-mh-query 10 ...)`, we first instruct Church to take ten samples from the conditional distribution we are about to describe. Then we write down a generative model for our observations, the results of the coin flipping. The query part contains the one variable of the generative model that we are actually interested in, i.e. that we would like to sample: the hypothetical coin weight that is inferred by conditioning the observations generated by the generative model to equal the true observations. Running this results in an output that looks like this:

```
(0.106 0.335 0.174 0.443 0.381
 0.584 0.416 0.312 0.467 0.529)
```
(4.25)

What we got are ten approximate samples of the coin weight in the generative model conditioned on our generative model resulting in the actual observations.

Having seen how `mh-query` can be used to conditionally sample from generative models, I will now relate this to the previous sections by explaining how the Markov chain looks like that

Church constructs in order to compute samples. In order to do so, I will answer three questions (following [GMR+08], [Man09]): First, what is the state space of the Markov chain? Second, what is the initial state? And third, what is the proposal distribution?

Intuitively, one could imagine that the state space is exactly the space of possible return values, since at each step of the Metropolis-Hastings algorithm, we want to sample one value. For example, for the query above that returns coin weights, this would be the interval $[0, 1]$ of real numbers. In fact, the *state space* of the Markov chain that `mh-query` constructs is the space of *computation traces* of the Church program defined by the query, i.e. not only the space of return values, but the space of return values augmented by information on how the return values were constructed[20]. Each state contains the dependency structure of the computation and, in particular, contains information on how the return value depends on the stochastic choices that were made during the evaluation.

[20] More formally, the computation trace is a directed acyclic graph composed of two directed trees, one tree for the environment structure of the program, and one tree for the evaluation structure, where an evaluation node points to a subnode for each recursive call to `eval`.

We get the initial state, i.e. the initial trace, by executing the query expression and recording the recursive calls to `eval` and the environment structure along the way[21].

[21] Depending on the program, we may have to force it into a state where the predicate is true, and depending on the program, this may be difficult without a method similar to the constraint propagation policy of our adaptive importance sampler.

Knowing what the state space is (the space of all computation traces that lead to return values for which the condition predicate is true) and what our initial state is (a single trace that we get by evaluating the query program), all that is left to specify to make this a complete Markov chain is how we get from state to state, i.e. how we propose new candidate states.

In order to create a new trace, we first randomly select[22] a place in the current trace where a random choice was made and then re-evaluate the choice. In order to keep the trace consistent, i.e. a valid evaluation history, we then propagate the changes in the values of subexpressions and bound symbols along the trace. The Metropolis-Hastings rule then tells us whether to accept the candidate trace or whether to reject and keep the old trace; if the condition predicate is false, we always reject.

[22] How well MCMC performs depends strongly on how good the proposed states are, i.e. how quickly they explore the high-probability regions of the state space. Using adaptation to construct better MCMC proposals is an issue that deserves future research.

Taken together, universal conditional sampling in Church is implemented using a stochastic walk over computation traces that has the distribution of interest as its long-run distribution.

Now that we have seen two inference methods that can be used to score generative models given some observations and to explore the space of generative models, we can describe the circumstances

under which we will apply these methods to show how learning generative models with language-like representations can work.

Chapter 5

Setup

The general setup for our demonstration of inductive learning will be as follows: A structured world generates observations and the task of the learner is to construct a generative model of the world that closely mimics the actual world and that therefore can be used in reasoning about the world. I have described both the general idea and and a specific algorithm for learning such models in previous chapters; now I want to demonstrate how this turns out in practice. This requires that we determine what the world should look like to make the demonstration meaningful and what the representation language should be.

5.1 A Structured World

I start by describing the four desiderata for a world in which a suitable demonstration of the kind of representation learning I have described so far can take place.

First and foremost, the world must contain interesting structure. What I want to demonstrate is the incremental learning of structured representations. If the world itself does not contain interesting structure, accurate representations of the world will not contain structure either. What makes structure interesting? More precisely: What makes us say that a program that generates observations contains interesting structure? Most fundamentally, the program must be shorter than the variety of observations it generates: There must be some way in which it *compresses* the data it generates, some way in which it *abstracts* from the data. At

the same time, the generating program itself must contain at least some information: If the generating program can be expressed too concisely, e.g. as is the case for observations that consist of a long sequence of only 0s, we would not call it interestingly structured either.

Second, the observations must be such that it is possible to recover the underlying structure to an interesting extent. This entails two requirements, one based on in-principle considerations, one based on considerations of computational efficiency. Since the efficiency requirement is better seen as a constraint on the learner and its representation language and inductive bias, we are going to use than on the actual world, I will deal with it in the next section. However, what is relevant here is that the observation must contain sufficient information to probabilistically identify the correct hypothesis (or some small set of likely hypotheses) in principle. If an observation contains only one bit, it can, *on average*, eliminate at most half the probability mass.

Third, the process that generates observations from hidden structure must be stochastic. If the process were deterministic, the same hidden structure would always result in the same observations, we would make one observation and then search through our representational medium, looking for the shortest program that results in the observed value. On the other hand, if the process is stochastic, then making additional observations can result in additional information about the structure of the generative process. For example, it can help us to probabilistically distinguish the generative process `(lambda (a a) (flip))` from `(lambda () (flip) (flip))`. This tells us something else: The stochasticity of the process must not be limited to noise that is added after the observations have been generated, not just *noise at the bottom*, but stochasticity that is added within the generative process. Otherwise, we cannot gain additional information about the internal structure of the generative process from additional observations.

Fourth, the world must be simple. We are going to use the probabilistic programming language Church, therefore simple is defined relative to this language: The longer the shortest complete description we can give for a world, the more complex it is. Our goal is to have worlds that are simple by this measure and that nonetheless satisfy the criteria given above.

Since Church is based on Scheme, its fundamental data structure are lists[1]. One natural choice both for the internal structure of the generative processes that define our world and for the observations is to make them lists of lists – trees, in other words. If the world consists of tree-generating programs with stochastic choices and abstraction, and the observations consist of the trees generated by these programs, then all of the requirements we named above are satisfied. In particular, this choice has the desirable property that the structure of the observations mirrors the structure of the generative process to some extent, thereby providing information that helps to infer from observations what the underlying process looks like.

[1] Pairs, actually. Lists are built out of pairs.

Formally, the syntax for tree-generating programs looks like this:

```
tree ::= branch
branch ::= '((lambda (' var ') ' branch ') ' branch '))' |
           '(if (flip) ' branch branch ')' |
           '(list ' label branch* ')' |
           var
var ::= symbol
label ::= '(sample-label)' | '(quote ' symbol ')'
symbol ::= ['A'-'Z' 'a'-'z'].*
```

(5.1)

This defines which kinds of worlds we are going to consider. Intuitively, each world is a program drawn from this grammar. If we put weights on the productions, we can actually draw worlds and what we have defined here is a distribution on worlds.

5.2 A Probabilistic Representation Language

What does a representation language look like that can efficiently represent worlds like the one described above? And what should the prior distribution on worlds look like that the learner in our demonstration will use?

Clearly, the ideal representation language can represent exactly all possible worlds and no impossible worlds; it is equal to the language defined by the grammar for tree-generating programs that we specified in the last section. Equally clearly, the best prior distribution over worlds assigns exactly as much probability mass

to each world as is the case for the true distribution from which worlds are drawn. What I would like to show is how reasoning and learning can take place given that we have a accurate language to formally write down models of the world and given that we have a reasonable prior distribution over what the world is like (e.g. that simpler worlds are more likely). Therefore, I will use the ideal representation language and prior for our learner.

In the last section, I mentioned that the observations must not only allow us to approximately recover the true underlying structure from an information-theoretic point of view, but they must allow us to do so in a computationally efficient way. This means that in order to find the best hypothesis, it must not be necessary to look at all hypotheses. The hypothesis space must be structured such that from looking at the observations and at some hypotheses, we get information on which hypotheses to examine next. This is an instance of *partial credit*.

For tree-generating programs, partial credit can be achieved by augmenting the representation language of the learner such that each tree-generating program expressed in this language can, with some small probability, generate an arbitrary tree. In particular, we can achieve this by first specifying a tree-generating program that can generate every tree and then allowing at each node that the generating process switches with some small probability from the "real" program to the program that can generate any tree. Thus, in inference, common root structure is used as far as possible, but if the desired observed tree could otherwise not be generated from the current hypothesis, the all-including small-probability process is used. Thereby, the probability of an arbitrary observed tree being generated from some hypothesis tree-generating process is not automatically 0 if the hypothesis only gets the observation partially right.

In this chapter, I have described the setup of the demonstration we use to show how learning of generative models can work. We have chosen a world that consists of simple, stochastic, tree-generating programs with the observations being trees with colored nodes and, for the learner, a matching representation language an bias. In the next chapter, I will show what the actual implementation of the learner looks like and how it fares in this world.

Chapter 6

Learning

How does an example of concept induction in a simple world look like? This is the question I want to answer in this chapter. In the last chapter, we have seen a toy world that consists of structured, tree-generating programs. The learner — who has the job to infer the true structure of the world from observations — uses a representation language with primitives and inductive bias both matched to the tree world. We have already seen a formalization of the problem of learning and reasoning in a structured world, and we have also seen two inference mechanisms that, in combination, can be used to learn representations from structured observations. In this chapter, I will show what this kind of structure learning looks like for the tree world.

6.1 Method

In the chapter on inference, we have seen that the problem of finding a good model for one's observations can be decomposed into the subproblems of model scoring and model space exploration. Subsequently, we have looked at adaptive importance sampling, an inference algorithm that can be used to efficiently score models, and Markov Chain Monte Carlo, an inference algorithm that can be used sample from the posterior distribution over models.

As was the case in the model chapter, it is still our goal to sample from the distribution over hypothetical model expressions conditioned on the observations generated from the hypothetical model being equal to the true observations. The more likely our

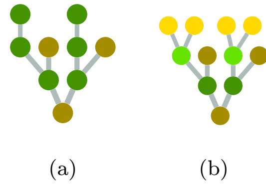
hypothetical model is to have generated the observations, the more probable it needs to be that we sample this model. At the same time, not only the likelihood of the observations should factor into the total score of a model, but also the random choices that were made when the model was generated: The more random choices were made, the less likely the model is a priori, the less probable we want it to be under the posterior.

We achieve this goal by writing down a generative model that first generates an expression and, subsequently, scores the true observations under the program defined by this expression. Using Church's universal `mh-query`, we condition the output on the estimated score, thus making those models most probable that best account for the observations. At the same time, the randomness that went into generating the expression factors into the posterior distribution through the generative model: the MH algorithm is less likely to go sample states that have lower prior probability, which is the case for more complex expressions.

For each observed tree, we estimate the likelihood that the program defined by the expression `hyp-expression` returns this tree using our adaptive importance sampler. Then, the total likelihood of the observations under the current hypothesis is the product of the individual observation likelihoods. Since we are not only interested in sampling expressions for hypothetical models of the observed data, but also in the estimated likelihood of the data under each hypothetical model, we query on both, i.e. let our samples contain pairs of model expression and likelihood of observations.

In code:

```
(mh-query
  ; The generative model 'in the head':
  ((define hyp-expression (sample-expression))
   (define observation-likelihoods
     (map (lambda (tree)
          (ais-estimate-prior hyp-expression tree))
       observed-trees))                                    (6.1)
   (define logscore (sum observation-likelihoods)))
  ; What we want to know:
  (pair hyp-expression logscore)
  ; Condition:
  (log-flip logscore)
  (get-env))
```



(a) (b)

Figure 6.1: Observed trees for which we want to find a generators.

6.2 Illustration

In the following, I will illustrate a sequence of programs that step by step have a higher estimated likelihood for the observation pictured in figure 6.1. This is for two reasons: First, this shows that computing the likelihood of the observations by adaptive importance and using our tree grammar results in a very intuitive sequence of likelihoods in the following sense: Programs for which the likelihood of the observations is rated as high are actually those programs that intuitively seem likely to have generated the observations. The second reason for showing this sequence of programs is to give an idea of the way `mh-query` would sample hypothetical programs given observations.

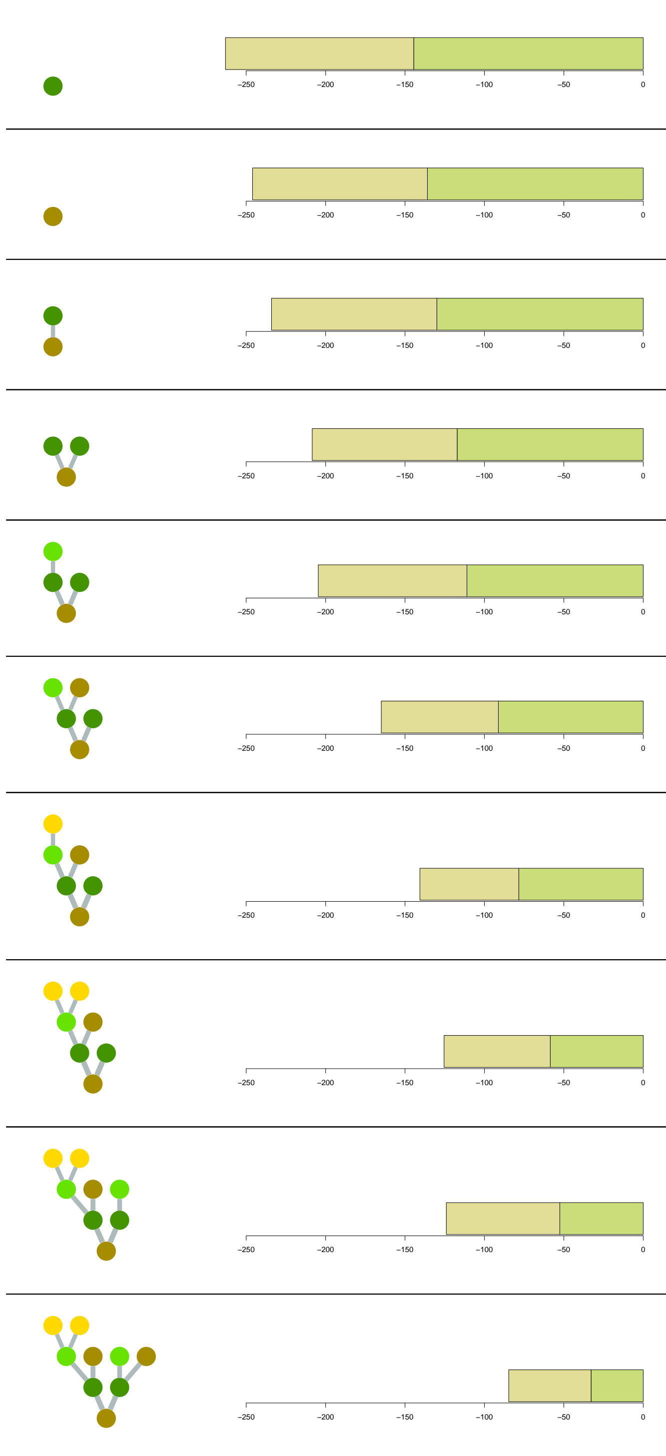
In contrast to the following sequence, a sequence of MH samples usually looks less like the result of a hill climbing algorithm than the following illustration[1]. However, starting from an expression for which the observations have low likelihood, the step-by-step construction of an expression with higher likelihood looks very similar if done by MH sampling.

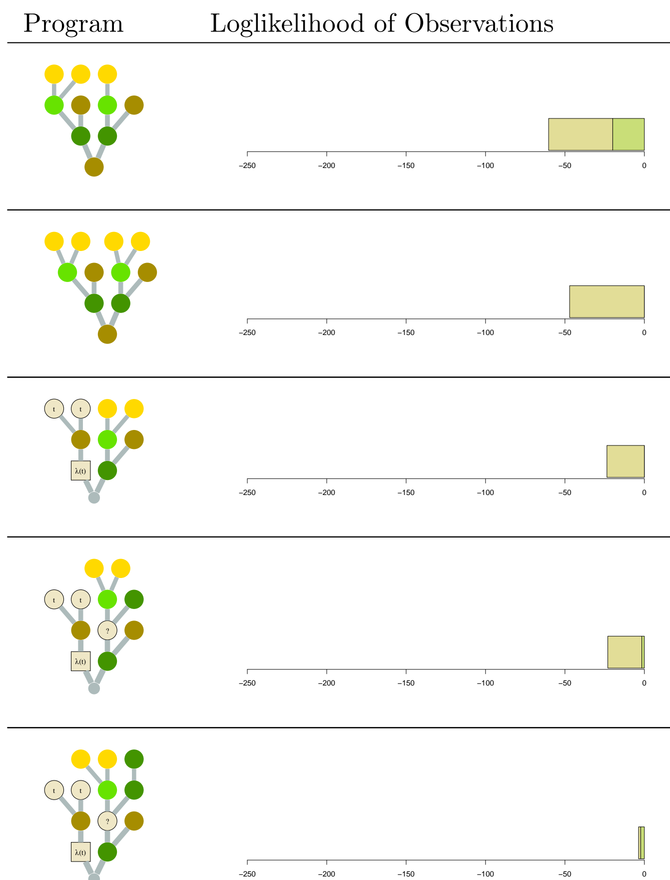
[1] Since MH accepts samples proportional to their posterior score, going from a sample with a high score to a lower-scoring sample is perfectly possible — it only happens proportionally less often than going to another high-scoring sample.

On the left of the following illustration, the program is presented in a way that exposes its structure. Programs are not identical to trees; as we have seen in the previous chapter, every program can give rise to any tree in order to allow the computation of a meaningful partial-credit carrying likelihood for all observations. However, for a program like `(node 'a)`, all trees but the one that carries only a single node of the appropriate color have very low probability.

On the right, the estimated loglikelihood of the observations is shown, with the likelihood of observation 1 ◻ making up the left side of the bar and that of observation 2 ◻ the right side. The smaller the bar is for a program, the larger is the loglikelihod of the observations under this program.

Program Loglikelihood of Observations





In code, the final program looks like this:

```
((lambda (t) (node 'a t t))
 (node 'b
   (if (flip)
       (node 'c (node 'd) (node 'd))
       (node 'b (node 'b)))
   (node 'a)))
```
(6.2)

Lambda abstraction captures the duplication of tree structure in a concise way and the stochastic choice `(if (flip) ...)` takes care of partial variation in the observations.

There are two reasons why our learner will prefer the program using lambda abstraction to an analogous program without abstraction, and only the first one is captured in the plots above: The observed data is more likely under this program since fewer random choices have to be made to generate the data. By du-

plicating a branch using abstraction, all the randomness in this branch factors only once into the likelihood computation instead of twice. The other reason is that the program itself is shorter and thus more likely a priori, since fewer random choices have to be made in generating this program from the distribution over programs.

Having seen qualitatively how a program can be constructed step by step by looking at which changes improve the likelihood of the observations that this program is supposed to give rise to, we will now take a more quantitative approach and look at how the Metropolis-Hastings inference algorithm (code 6.1) performs when run on randomly generated expressions and observations.

6.3 Quantitative Experiments

The goal of this section is to show that learning tree-generating programs from observations reliably works if we use the methods described in the previous chapters. I will first describe the learning setup including which statistics I collect, then show some of the results (with all of them being in the appendix) and some aggregate statistics. Finally, I discuss what we can and cannot conclude from the data we have collected.

Before we can design our demonstration, we need to define our measure of success: What does it mean that learning generating programs for observations *works*? One option for such a measure is the successful recovery of the program that originally generated the observations — the actual structure out there in the world. If a learner is given a few observations and then systematically assigns high probability to the true generator, then it is clearly a successful learner. However, since we only get to look at a few observations, the true structure may not always be the best explanation. The best explanation of an observation is the explanation that, across all situations that could give rise to this observation and weighted by how often the situations occur, most accurately reflects the true distribution over situations.

In the sequence of programs that was illustrated in the last section, the best explanation was one that used lambda abstraction to account for recurring structure. In contrast, our quantitative experiments will not include lambda abstraction, neither in the true programs that generate observations nor in the grammar that

is available for inference. The reason for this constraint is that, in order to efficient MCMC inference over the space of programs that includes lambda abstraction, there must be proposals that allow the algorithm to switch between a program without lambda abstraction to a program with lambda abstraction and vice versa. Moreover, in order to stochastically walk through the space of programs that includes lambda abstraction in a way that results in efficient inference, there must not only be a way to construct programs with lambdas, but these states must be reachable from similar programs that do not use lambdas. For example, there must be a way to go from

```
(node 'b
  (node 'a (node 'b) (node 'c))
  (node 'a (node 'b) (node 'c)))
```
(6.3)

to

```
((lambda (x) (node 'b x x))
 (node 'a (node 'b) (node 'c))
```
(6.4)

and back[2]. If the programs with lambda abstraction are not reachable from similar programs without lambdas, the inference algorithm will not efficiently explore the different modes of the distribution over programs and instead mostly be stuck in one or the other — it will not "mix well". Lambda learning will allow us to infer more interesting and varied programs from observed data, but its implementation remains future work.

[2]This type of proposal has been proposed by Noah Goodman, who named it "inverse inlining".

The basic idea of the setup we will use to quantitatively demonstrate program learning is to randomly generate programs and then test how well our inference algorithm performs on observations generated from these programs, measured by how likely the observations are under the inferred programs compared to the true generating programs. In sequence:

1. We first sample a random tree-generating program from the probabilistic grammar that defines which tree worlds are possible.
2. By running this program, we sample structured observations, i.e. trees.
3. We estimate the likelihood of these observations under the true generating program.

4. We run the inference algorithm (MCMC with AIS scoring) on these observations, thus sampling programs that could have generated the observations, together with the estimated likelihood of the observations under each program.

5. We plot the true likelihood of the observations together with the estimated likelihood under the programs that were inferred from the observations.

In total, I have run the sequence described above 79 times, thus creating 79 tree-generating programs, 79 worlds. From each of these, 5 observations have been sampled. Using these observations, the inference algorithm was run for 2000 MH steps, i.e. 2000 programs[3] were learned in each of the 79 worlds.

[3] together with the likelihood of the observations under these programs

Figure 6.2 shows the result of the first twelve runs. The estimated true likelihood is shown with a green line, the likelihood of the observations under each of the 2000 sampled programs is shown in black. The plots show that the MCMC algorithm starts out with a program that does not explain the observations very well, but after a short time samples from a region of the state space where most of the programs do explain the observations well. For a complete listing of the results, including information on what the generating programs were, see the appendix.

Figure 6.3 summarizes these results by showing how the (estimated) likelihood of the observations under the true generation program relates to the highest (estimated) likelihood for all learned programs. As illustrated by the fact that the best learned likelihood is often as high as or higher than the true likelihood (i.e. many points are on or above the middle line in the rescored figure), the learned programs do well at explaining the observed data.

The fact that there are several programs which do not explain the observations well can be explained by looking at the list of experimental results and programs in the appendix. Those observations that were not explained well within 2000 MH samples are almost always those generated by complex programs, and it is reasonable to suggest that running the inference engine for a longer amount of time would result in programs being learned for these observations, too. It is nonetheless worthwhile to ask how the inference algorithm scales with the length of the smallest programs

that can explain the data well; this task, however, is beyond the goals of this thesis.

In the last sections, we have seen the tree world as a practical example of how programs can be learned from observations. We started by looking at the (fully general) basic idea of using a combination of Markov Chain Monte Carlo methods to explore the space of possible programs together with adaptive importance sampling to score how likely any given observation is under a given program. We have then first given an illustration of how small changes to a program can lead to a sequential increase in the likelihood of the observation, thus allowing for a smooth trajectory from programs that do not explain the observations well to those that do. Finally, I have argued that this works reliably in the tree domain by showing how explanations can be learned for observations created from randomly generated programs.

In the remainder of this thesis, I will discuss what the results in this chapter mean and what the adaptive importance sampling algorithm can be used for more generally, which related work there is and what future research looks most promising. Finally, I will conclude by reviewing the content of this thesis by summing up the main points from each chapter.

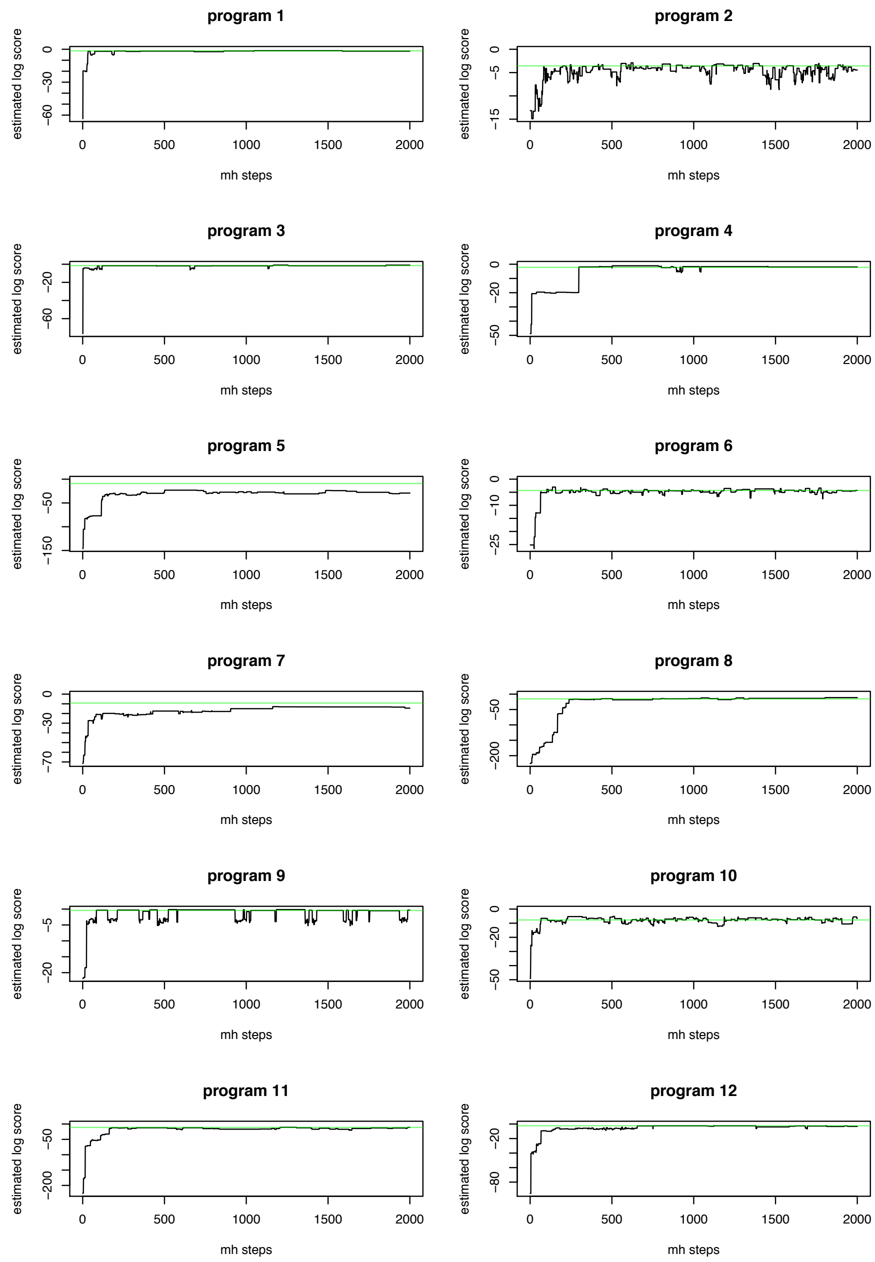


Figure 6.2: MCMC results for a few programs. For the complete statistics, see the appendix.

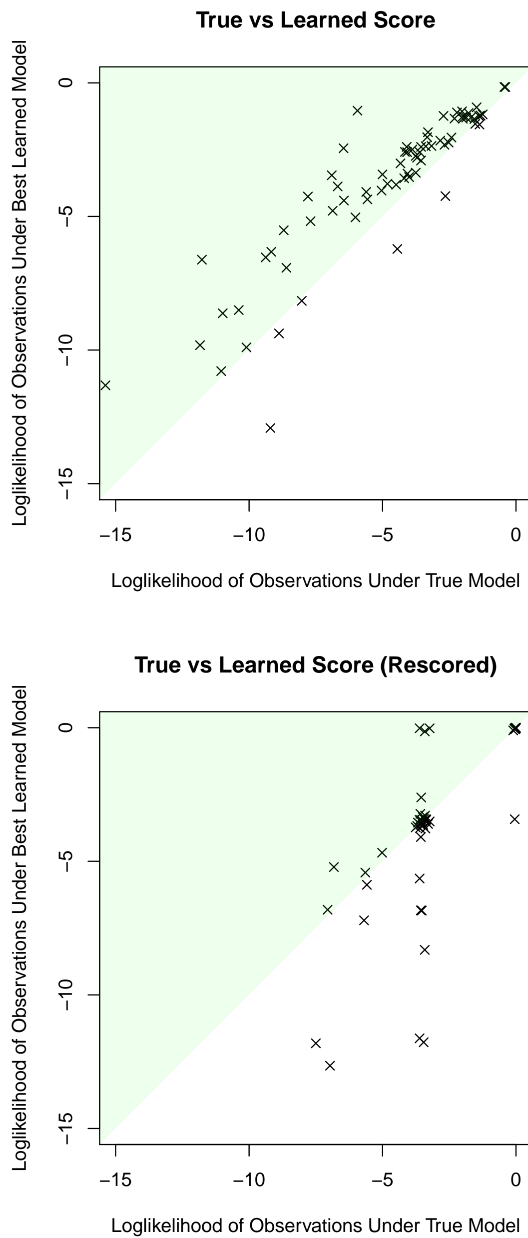


Figure 6.3: MCMC results summarized. Rescoring the sampled programs with 300 importance samples instead of the 30 that were used at each MH step exposes cluster structure (programs with one flip, two flips) and removes the selection bias present in the first figure (with high estimation variance in the scores, selecting the learned program with the best score results in scores that are only accidentally high).

Chapter 7

Discussion

7.1 Interpretation

Given that most chapters already discuss their respective findings, I will only discuss two questions here that strike me as particularly important. First, what do the results in the chapter on learning actually show? Second, to what extent is the class of adaptive importance sampling algorithms that we have derived generally useful rather than problem-specific?

Starting with the results in the chapter on learning in a simple toy world: What do they show beyond the immediate result that *in this world* generative models can be recovered from observations? One generalization of this result is that, if you can efficiently estimate the likelihood of observations under any given generative model, and if you can explore the space of generative models efficiently, then you can find programs that explain the observations well. We used two main ingredients to make these two operations efficient: Adaptive importance sampling to score models and a representation language that expresses models such that they can generate any observation with nonzero, graded probability. Lessons one might learn are that sampling algorithms should use any information about the posterior distribution to generate better samples (here: evidence pushing, adaptation) and samples should provide information on where to look for better samples (partial credit).

To what extent is the class of adaptive importance sampling algorithms that we have derived generally useful rather than problem-

specific? In its current form, the information propagation rule of the algorithm assumes that the operands are evaluated independently, an assumption that clashes with the idea of delayed evaluation that we used to push the evidence through compound procedure applications. As a consequence, the current algorithm suffers from inefficiencies when applied to programs containing compound procedures and is best seen as a preliminary version of the final algorithm. However, there is a natural extension we call "sequential inverses" that will allow us to combine delayed evaluation and adaptation at primitive procedures. Once the information propagation rule is extended to its more general form, this algorithm can be used to sample solutions to logical satisfiability problems, to parse the yield from grammars, even from those that do not fall inside the class of context-free or context-sensitive grammars, and to solve similar problems where large parts of the (prior) state space are impossible under the posterior distribution. A closer examination of the capabilities of the class of algorithms will only be possible once the proposed extension has been completed.

7.2 Related Work

An important part of the related work has already been mentioned in the chapter describing the philosophical, statistical and computational background of our approach. In particular, this includes the idea of a language of thought (due to Fodor [Fod76]) and the Bayesian approach to probabilistic inference (promoted by Jaynes, among others [JB03]). However, there are two research programs that have been relevant to this project, but not sufficiently exposed so far. The first is the psychological research program built on the exploration of a hypothesized probabilistic language of thought, the second is the prior research on adaptive importance sampling algorithms.

The *Probabilistic Language of Thought* research program headed by Noah Goodman provides the scientific context that gave rise to this work, and most of the ideas that are part of this work are ideas developed within this research program. In [GTFG08], a Bayesian model of concept learning is presented and tested using a grammatically structured hypothesis space. One way of looking at the current work is to see it as a generative and more com-

putational version of this discriminative, more psychological, and more empirical work. [KGT08] makes use of a compositional language of thought to explain the learning of intuitive theories on a computational level. This research program also gave rise to the probabilistic programming language Church [GMR+08] that was fundamental to the implementation of our project. The next bigger context that subsumes both this work and the probabilistic language of thought program is the Bayesian, computational approach to concept learning and to cognitive science in general [TR99].

There are a few inference algorithms that are in the same spirit as our adaptive importance sampler for probabilistic programs. The most similar algorithm known to me is SampleSearch [GD07] which, in contrast to our algorithm, is based on backtracking instead of adaptation and which is only asymptotically unbiased, whereas our algorithm results in unbiased estimates. In addition, like AIS-BN [CD00] and Ortiz and Kaelbling's adaptive importance sampler [OK00], these algorithms are written for graphical models and do not make use of the more intricate structure of probabilistic programs. Systematic Stochastic Search [MRJT] is a sequential rejection sampler that has the same property. Whereas our algorithm conditions only on the relevant environment of an expression when learning how to sample from it, this algorithm conditions on the complete prior evaluation history and thus does not generalize from its adaptations. Pfeffer [Pfe07] presents an importance sampling algorithm for probabilistic programs, which makes use of delayed evaluation and of checking ahead of time what values an expression can return. However, in contrast to our algorithm, this algorithm is not adaptive.

7.3 Future Research

Due to the broad nature of the approach to learning and reasoning that we took, and due to the broad nature of the topic itself, there are many directions future research could take. I will limit myself to sketching those that are both exciting and realizable in the near term.

Instead of having the learner passively infer the structure of the world from observations that cannot be influenced by the learner, we could consider an active setting. Here, an agent interacts with

the environment in a goal-directed way; what and how to learn might then not be based on considerations of epistemic accuracy, but instead on utilitarian considerations. We might look at setups similar to those used in reinforcement learning, but consider structured generative models as more interesting, useful and realistic representations for world model, goals, and actions than the representations that are commonly used.

The programs that were learned in the demonstration are composed of more elementary parts, and the learning happens in a stepwise fashion, but there is an even more interesting element to the idea of compositionality that we did not make use of: The idea of reusing programs that were itself learned, an idea that enables a learner to gain expressive power with each new concept in its knowledge repository. This idea is tied to a number of proposals for future research, the overarching theme being the idea of learning *systems of concepts*. Technically, one might achieve this by allowing *inverse inlining* proposals, i.e. proposals that move from programs without lambda abstraction to programs with lambda abstraction. Both sequential learning and simultaneous learning are interesting from a computational and from a psychological perspective. In sequential learning tasks, concepts that are learned earlier on are reused in the compositional structure of subsequent concepts if this allows the world model to be expressed more concisely. This makes the ordering of the learning data significant. A similar idea is to make the world itself change such that the learner step by step accumulates evidence that his current theory is no longer accurate until a theory revision is justified. In simultaneous learning, multiple concepts need to be extracted from observations at once, and which conceptual properties make this easy, which make it hard, and how exactly this can be done by either humans or machines is a question that future research needs to answer.

I have presented probabilistic inference over structured generative models as a computational theory of both learning and reasoning, but in the actual demonstration, I have only shown learning. Clearly, accurate beliefs are of very limited use if one cannot use them to reason about the world, and a demonstration of a computational model of reasoning, in particular if it combines inductive and deductive reasoning in an elegant way, will be an important task for future research. Exploiting the fact

that what our model of learning infers is a richly structured generative model, a natural extension to reasoning is the following: Conditional sampling from the generative world model allows the learner to make inductive predictions about any property of the world that can be expressed within the representation language. By giving no special treatment to deduction, but instead taking it to be just a limiting case of induction, approximate deductive reasoning becomes feasible within the framework of probabilistic inference. How productive this view on deduction is, both from a psychological and from a machine learning perspective, is an open question.

How could we test whether the characterization of concept learning as program induction is an adequate description of what humans do? Building on the research that has already been done on modeling concept learning as Bayesian inference over a language-like hypotheses space [GTFG08], we might consider a sequence of experiments to test in how far a hypothesis space defined by a language for generative model predicts human performance. Starting with the tree visualizations developed in this thesis, we can compare the generalization performance first for tree-generating programs without lambda abstraction, then for programs that use lambda abstraction to enable reuse of branches and finally for those that use higher-order functions. One particularly interesting test would be to determine whether a language based on generative models makes better predictions in this task than a discriminative classifier based on surface features.

Simple versions of problems are often amenable to methods that cannot be used to solve more complex and more data-intensive versions. If we want to infer more complex programs from observations, what needs to change to make our approach scale?

If the same basic model of partitioning the problem into exploration and scoring is maintained, then we need to make sure that both are efficient for larger programs. For the exploration part, if we keep MCMC as the query implementation, then efficiency means that we need to explore the high-probability regions of our hypothesis space quickly, which, for large hypotheses spaces, depends on making good proposals. Here, one idea would be to randomly choose any node in the program trace (not necessarily an erp) and then to use adaptive importance sampling to regenerate a trace that fulfills the constraint at this node. However, in

how far this can be used to make efficient proposals for program learning is an empirical question, and we might find out that a yet unknown alternative to MCMC results in more efficient inference. For the scoring part, if we keep the adaptive importance sampler, the efficiency will depend on the exact features used by the program to be learned (e.g. random operators might result in difficulties), on the number of likely parses per observation and on the state of the algorithm (e.g. on whether soft adaptation is implemented and if yes, what its convergence properties are). There is no in-principle reason why scoring would not work efficiently for larger programs, however the exact scaling properties remain the realm of future work.

More generally, exploiting incremental learning might be crucial to learning complex programs. If abstraction can be used to first learn simple programs that can be combined into more complex programs later on, then the fact that jumps between different complex programs require only small variations in the combinations of constituent programs might enable more efficient inference. The other side of incremental learning is what could be called "learning to learn": By using the adaptive importance sampler for a long time, the importance table will accumulate useful information on which values are how likely under which expressions and in which environments. A large importance table that persists across program runs might help to more quickly learn even complex programs. Due to the easy way to merge a large number of weighted samples from different distributions into one estimate, adaptive importance sampling is very easy to parallelize across computers. Such a parallelization, possibly together with a shared importance table, might help to solve the problem of learning complex programs if the algorithms used are not in the wrong complexity class for this task. This, too, is an interesting question for future research.

Chapter 8

Conclusion

We started out with the question how an agent could learn and reason in a structured, probabilistic world. In the following, I will review the progress that we have made, first by giving an overview, then by explaining more precisely what has been achieved in each of the chapters.

After first philosophical considerations on what the properties of a medium of thought must look like, we presented a formalization of learning and reasoning as probabilistic inference. In order to give a practical example of this formalization for a simple but interesting world, we first described two methods of inference, one of them being a new class of importance sampling algorithms for probabilistic programs. After describing the desiderata for a simple example of learning programs from observations, we first illustrated what learning looks like in this example, then quantitatively showed that inferring programs from observations reliably works. I will now describe these steps one by one.

As a first step, we considered what an informational substrate of thought might look like. If an agent systematically derives true statements about an object that is out there in the world and that is no currently perceived, then the information that is necessary to derive these statements must be found somewhere within that agent: The object must have a representation. The informational structure that is used to store representations must have certain properties: Representations must be able to be combined in a productive way, the meaning of compound representations must (among other factors) be determined by the constituents, the relation between compounds and constituents must be a sys-

tematic one, and both real and possible states of affairs must be expressible in this medium. One kind of representation that is particularly useful are (stochastic) generative models, i.e. models that mirror the causal structure of some process in a way that lets us generate observations from the model and that lets us invert the model to infer hidden properties of the process.

Having established the properties of mental representations, the next step in our process of reductive explanation was to find a way to formally talk about these kinds of representations and to give a formal account of what it means to learn and reason with these kinds of representations. Probability theory is our best tool to formally talk about uncertainty, and programming languages are our best tools for the formalization of processes that generate values, therefore, I have used the probabilistic programming language Church to formally talk about stochastic generative processes. If, using this language, we can formalize beliefs within the Bayesian probability calculus, which itself can be derived from relatively weak common sense and consistency assumptions, then Bayes' rule normatively determines how these beliefs need to be updated in light of new data. On a computational level, both learning (formalized as inferring a model from observations) and reasoning (formalized as determining whether a proposition is true in a given model) can then be interpreted as probabilistic inference.

In order to move towards a practical demonstration of learning generative models, we needed to solve the problem of how to do inference over generative models. We have decomposed the problem of learning these models into two parts: First, given observations and a model, how can we score the model? Second, how do we explore the space of possible models efficiently? In order to solve the first question, we have derived a new class of adaptive importance algorithms for probabilistic programs that rely on two principles: When evaluating a program, push the evidence as far down the evaluation as possible, thus sampling from a distribution that is close to the posterior distribution $p(model|data)$. After each evaluation, adapt the distribution we are sampling from, using any information we have learned during the evaluation about what values can and cannot be returned by subexpressions. The second inference method we use is called Markov Chain Monte Carlo; here, the main idea is to construct a Markov chain that

has as its long-run distribution the distribution we want to sample from. Taken together, these two inference methods let us infer programs from observations.

The setup of the actual learning demonstration was as follows: A structured world generates observations, and the task of the learner is to construct a generative model of the world that closely mimics the actual world. Of our demo world, we required that it contains interesting structure, that this structure can be (approximately) recovered from observations, and that the world is stochastic and simple. The world that we chose is one that consists of a stochastic tree-generating program, with the observations being trees with colored nodes. For the learner, we chose a representation language that offers an inductive bias over worlds that closely matches that of the true world-generating process and that can be used with partial credit, i.e. that, when we compute the likelihood of observations under a model that is almost right, this likelihood is higher than for a model that is completely wrong.

In our demonstration of structure learning for the tree world, I have first described how to combine Markov Chain Monte Carlo with adaptive importance sampling such that we can explore the space of possible programs. I have then demonstrated how small improvements to a program can lead to a sequential increase in the likelihood of the observations, thus allowing for a smooth trajectory from programs that do not explain the observations well to those that do. Finally, I have argued that this works reliably in the tree domain by showing how explanations can be learned for observations created from randomly generated programs.

There is a variety of directions one could take to further extend this work. On the learning side, the most notable are an extension to learning with lambda abstraction, learning in an active setting, and learning systems of concepts. On the algorithmic side, more intelligent procedure inversion and parallelizing the adaptive importance sampler promise the most immediate benefits. The framework of learning and reasoning with generative models expressed in a probabilistic language of thought promises to offer solutions to a large number of open questions in psychology and artificial intelligence, and one goal of this work has been to illustrate this possibility using the simplest example that could do.

Bibliography

[AdFD03] Christophe Andrieu, Nando de Freitas, and Arnaud Doucet. An introduction to mcmc for machine learning. Mar 2003.

[ASS] Harold Abelson, Julie Sussman, and Jerry Sussman. Structure and interpretation of computer programs.

[Ber] Kasper K Berthelsen. Importance sampling for unnormalized densities.

[Bon] KA Bonawitz. Composable probabilistic inference with blaise.

[CD00] J Cheng and MJ Druzdzel. Ais-bn: An adaptive importance sampling algorithm for evidential reasoning in large bayesian networks. *Journal of Artificial Intelligence Research*, 2000.

[CMMR09] Jean-Marie Cornuet, Jean-Michel Marin, Antonietta Mira, and Christian Robert. Adaptive multiple importance sampling. *arXiv*, stat.CO, Jan 2009.

[Cow] J Cowley. Compositionality in inferential-role semantics.

[CTY06] Nick Chater, Joshua B Tenenbaum, and Alan Yuille. Probabilistic models of cognition: Conceptual foundations. pages 1–5, Oct 2006.

[Fod76] Jerry A Fodor. The language of thought. page 214, Jan 1976.

[Fod07] Jerry A Fodor. Concepts - where cognitive science went wrong. pages 1–186, Dec 2007.

[GD07] V Gogate and R Dechter. Samplesearch: A scheme that searches for consistent samples. *AISTATS 2007*, Jan 2007.

[GMR+08] Noah D Goodman, Vikash Mansinghka, Daniel M Roy, KA Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. *Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence*, pages 220–229, 2008.

[Gog09] Vibhav Gogate. Sampling algorithms for probabilistic graphical models with determinism. 2009.

[GTFG08] Noah D Goodman, Joshua B Tenenbaum, Jacob Feldman, and Thomas L Griffiths. A rational analysis of rule-based concept learning. *Cognitive Science*, 32(1):108–154, 2008.

[GY06] T Griffiths and A Yuille. Technical introduction: A primer on probabilistic inference. *Department of Statistics*, Jan 2006.

[JB03] ET Jaynes and GL Bretthorst. *Probability theory: the logic of science*. 2003.

[KGT08] Charles Kemp, Noah D Goodman, and Joshua B Tenenbaum. Theory acquisition and the language of thought. *Proceedings of Thirtieth Annual Meeting of the Cognitive Science Society*, 2008.

[Mac03] David MacKay. *Information theory, inference and learning algorithms*. 2003.

[Man09] Vikash Mansinghka. Natively probabilistic computing. pages 1–129, Mar 2009. Notes.

[Mar82] D Marr. Vision: A computational investigation into the human representation and processing of visual information. *Henry Holt and Co., Inc. New York, NY, USA*, Jan 1982.

[MRJT] Vikash Mansinghka, Daniel Roy, Eric Jonas, and Joshua B Tenenbaum. Exact and approximate sampling by systematic stochastic search.

[OK00] LE Ortiz and LP Kaelbling. Adaptive importance sampling for estimation in structured domains. *Uncertainty in artificial intelligence*, pages 446–454, 2000.

[Pfe07] Avi Pfeffer. A general importance sampling algorithm for probabilistic programs. 2007.

[PGT] Steven T. Piantadosi, Noah D Goodman, and Joshua B Tenenbaum. A formal model of number word acquisition (under revision).

[RC04] Christian P. Robert and George Casella. Monte carlo statistical methods. page 645, Jan 2004.

[Sza08] Zoltán Gendler Szabó. Compositionality. 2008.

[Tal08] William Talbott. Bayesian epistemology. 2008.

[Tha08] Paul Thagard. Cognitive science. 2008.

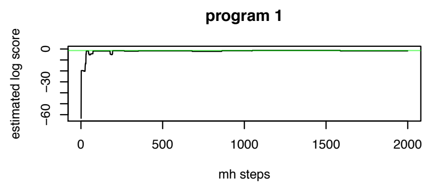
[TR99] Joshua B Tenenbaum and WA Richards. A bayesian framework for concept learning. *Department of Artificial Intelligence, Edinburgh University*, 1999.

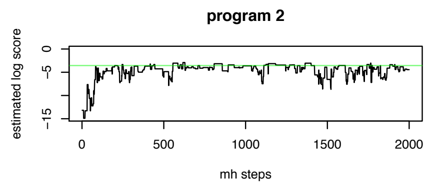
[Wyn90] K Wynn. Children's understanding of counting. *Cognition*, Jan 1990.

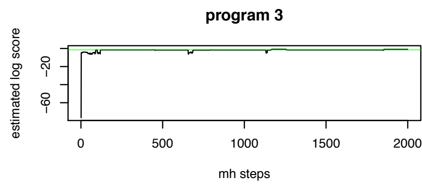
[Wyn92] K Wynn. Addition and subtraction by human infants. *nature.com*, Jan 1992.

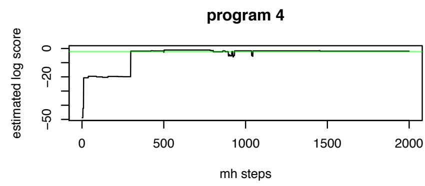
Appendix

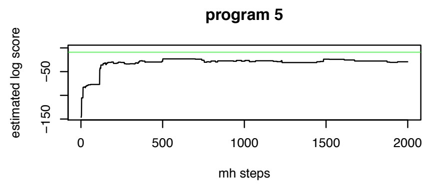
Experimental Results

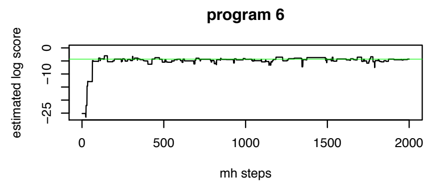


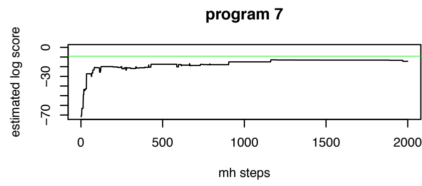


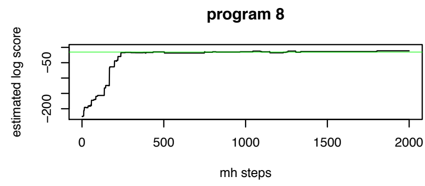


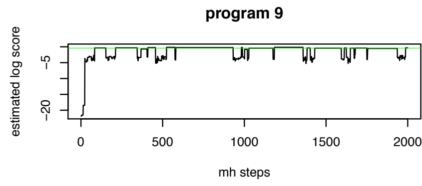


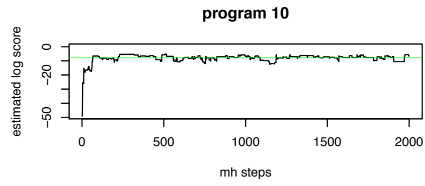


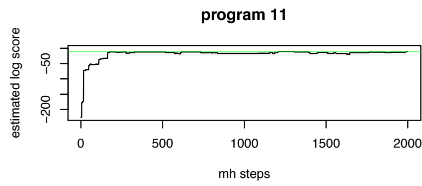


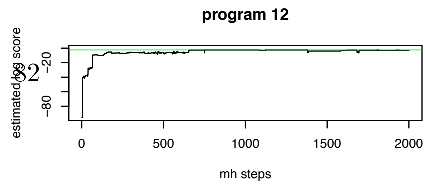


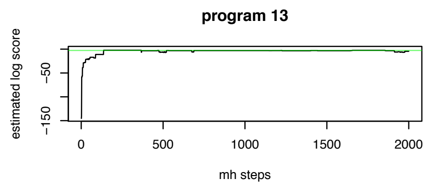


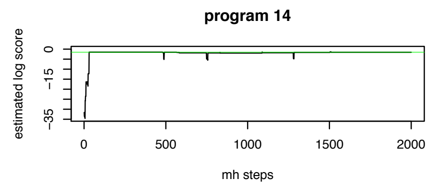


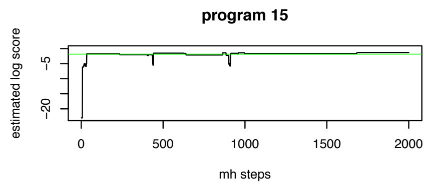


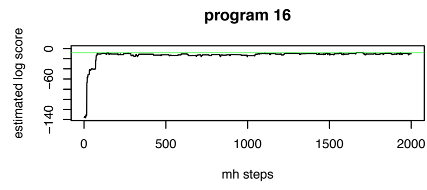


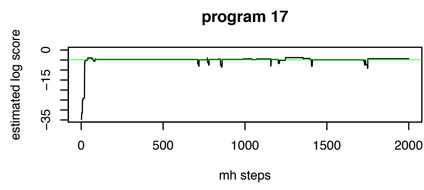


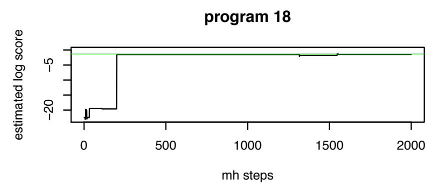

program 13
estimated log score
-150
-50
0
500
1000
1500
2000
mh steps

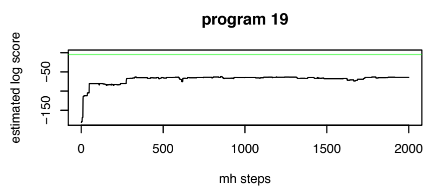

program 14
estimated log score
-35
-15
0
0
500
1000
1500
2000
mh steps

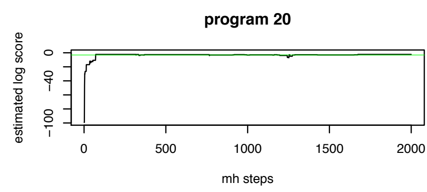

program 15
estimated log score
-20
-5
0
500
1000
1500
2000
mh steps

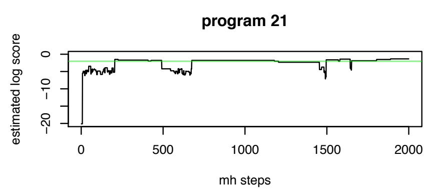

program 16
estimated log score
-140
-60
0
0
500
1000
1500
2000
mh steps

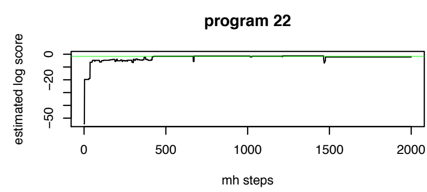

program 17
estimated log score
-35
-15
0
0
500
1000
1500
2000
mh steps

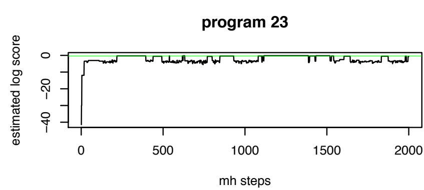

program 18
estimated log score
-20
-5
0
500
1000
1500
2000
mh steps

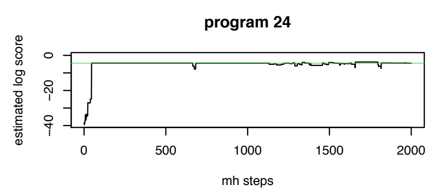

program 19
estimated log score
-150
-50
0
500
1000
1500
2000
mh steps

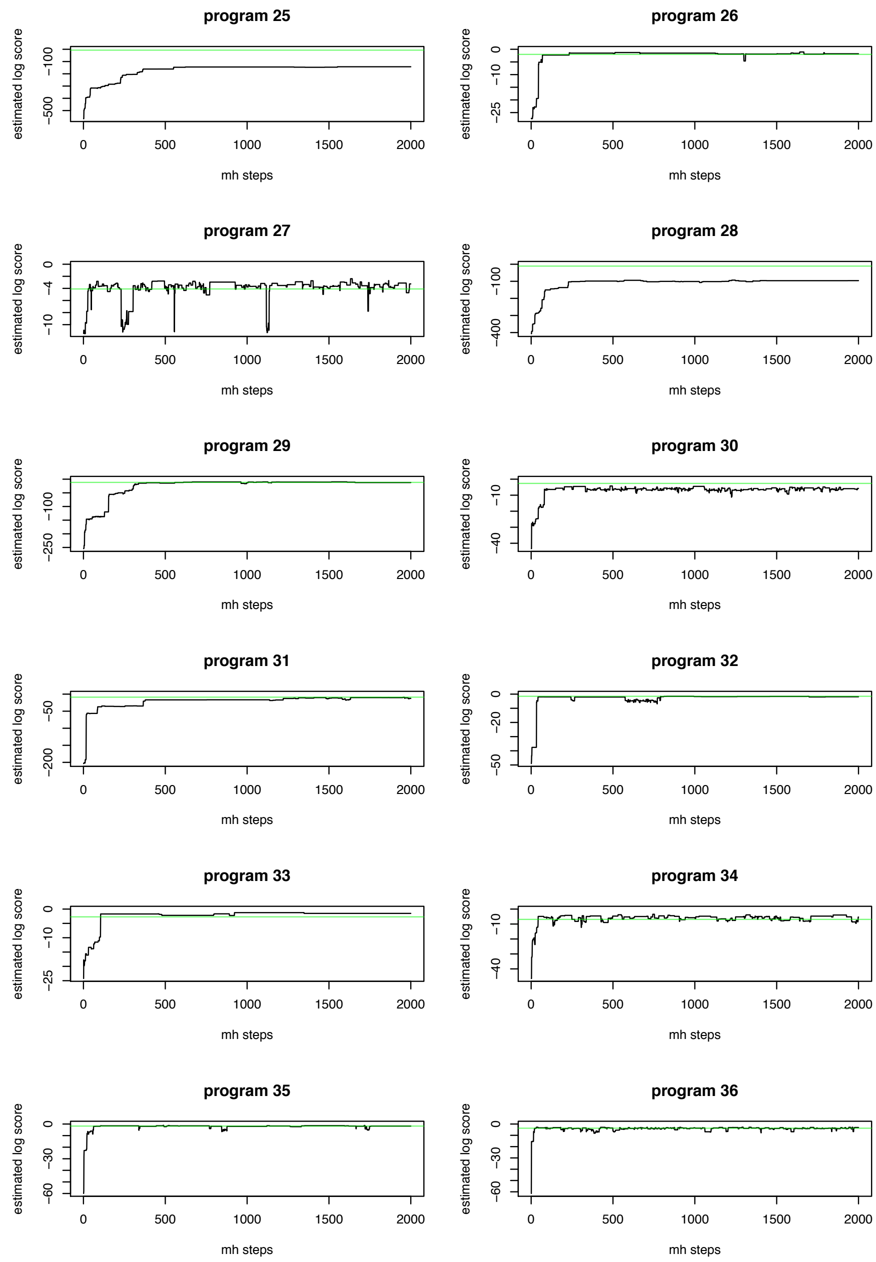

program 20
estimated log score
-100
-40
0
0
500
1000
1500
2000
mh steps

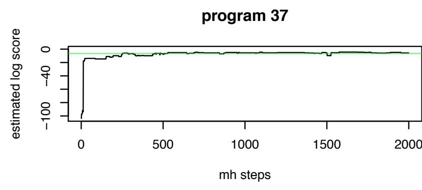

program 21
estimated log score
-20
-10
0
0
500
1000
1500
2000
mh steps

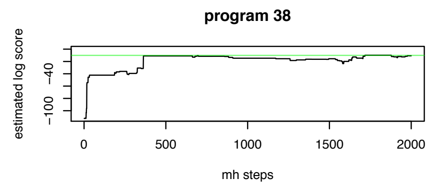

program 22
estimated log score
-50
-20
0
0
500
1000
1500
2000
mh steps

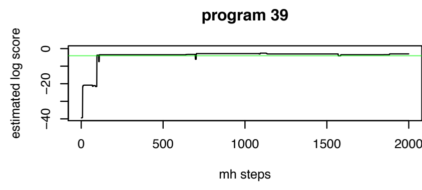

program 23
estimated log score
-40
-20
0
0
500
1000
1500
2000
mh steps

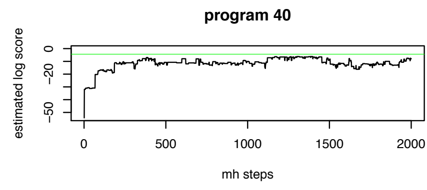

program 24
estimated log score
-40
-20
0
0
500
1000
1500
2000
mh steps

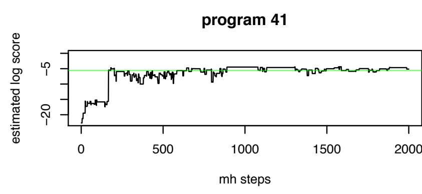

program 25
program 26
program 27
program 28
program 29
program 30
program 31
program 32
program 33
program 34
program 35
program 36
estimated log score
mh steps
0
500
1000
1500
2000

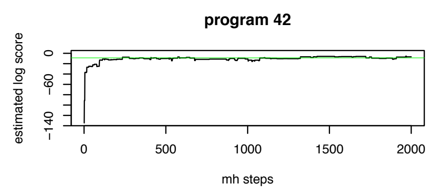

program 37
estimated log score
0
−40
−100
0 500 1000 1500 2000
mh steps

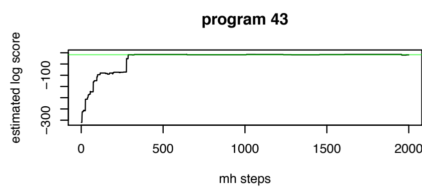

program 38
estimated log score
−40
−100
0 500 1000 1500 2000
mh steps

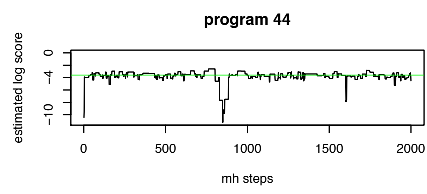

program 39
estimated log score
0
−20
−40
0 500 1000 1500 2000
mh steps

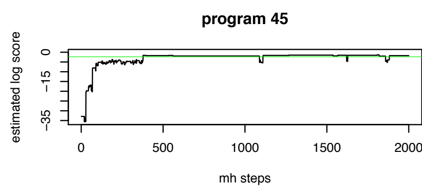

program 40
estimated log score
0
−20
−50
0 500 1000 1500 2000
mh steps

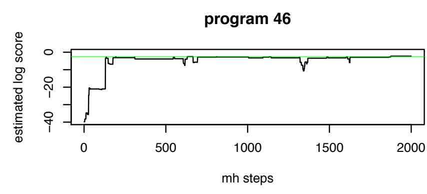

program 41
estimated log score
−5
−20
0 500 1000 1500 2000
mh steps

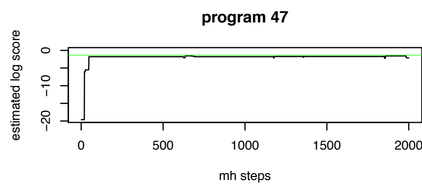

program 42
estimated log score
0
−60
−140
0 500 1000 1500 2000
mh steps

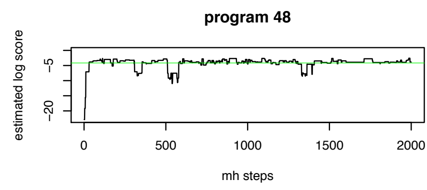

program 43
estimated log score
−100
−300
0 500 1000 1500 2000
mh steps

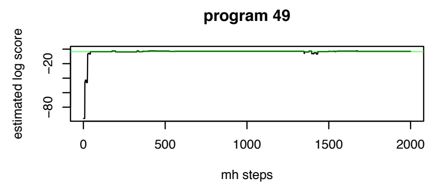
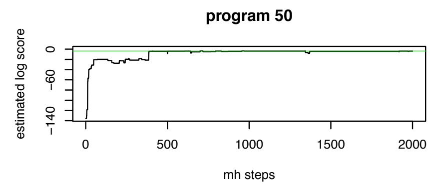
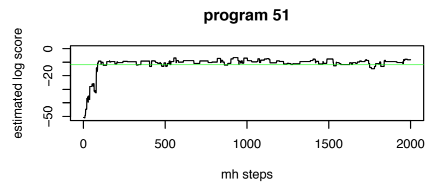
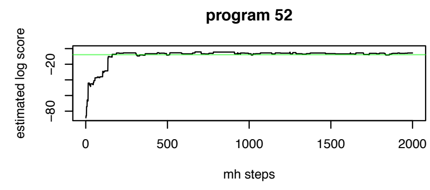
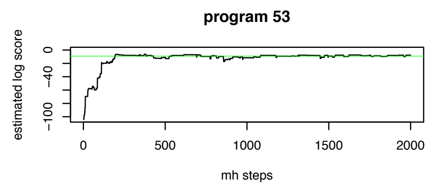
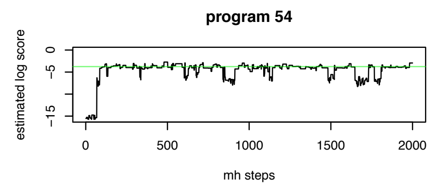
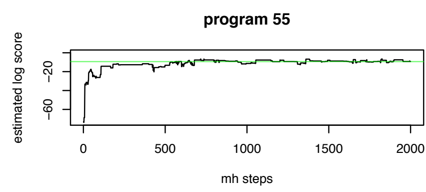
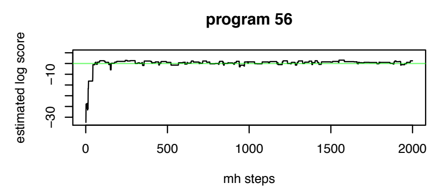
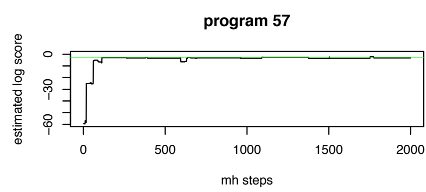
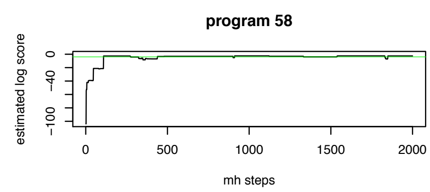
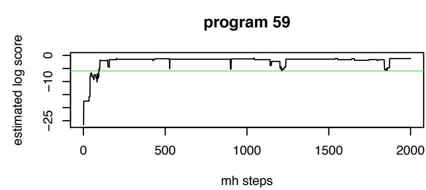
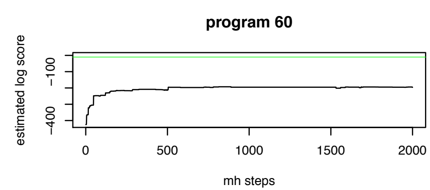

program 44
estimated log score
0
−4
−10
0 500 1000 1500 2000
mh steps

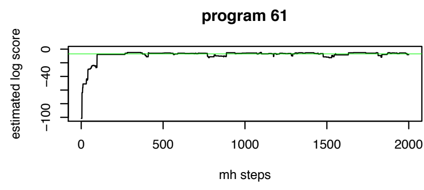

program 45
estimated log score
0
−15
−35
0 500 1000 1500 2000
mh steps

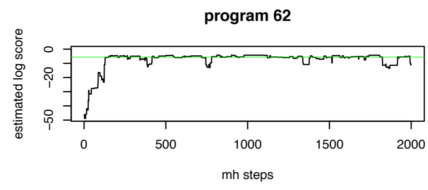

program 46
estimated log score
0
−20
−40
0 500 1000 1500 2000
mh steps

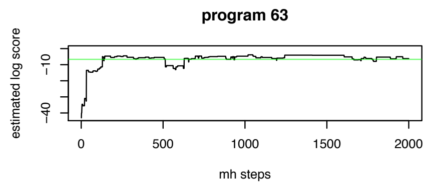

program 47
estimated log score
0
−10
−20
0 500 1000 1500 2000
mh steps

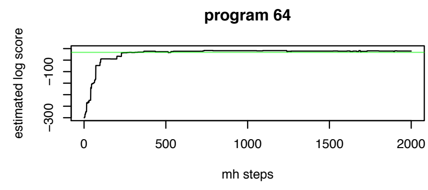

program 48
estimated log score
−5
−20
0 500 1000 1500 2000
mh steps

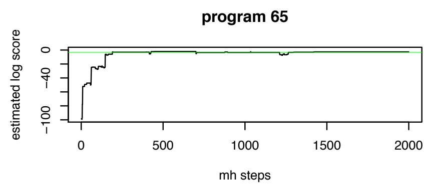

program 49
estimated log score
-20
-80
0
500
1000
1500
2000
mh steps

program 50
estimated log score
0
-60
-140
0
500
1000
1500
2000
mh steps

program 51
estimated log score
0
-20
-50
0
500
1000
1500
2000
mh steps

program 52
estimated log score
-20
-80
0
500
1000
1500
2000
mh steps

program 53
estimated log score
0
-40
-100
0
500
1000
1500
2000
mh steps

program 54
estimated log score
0
-5
-15
0
500
1000
1500
2000
mh steps

program 55
estimated log score
-20
-60
0
500
1000
1500
2000
mh steps

program 56
estimated log score
-10
-30
0
500
1000
1500
2000
mh steps

program 57
estimated log score
0
-30
-60
0
500
1000
1500
2000
mh steps

program 58
estimated log score
0
-40
-100
0
500
1000
1500
2000
mh steps

program 59
estimated log score
0
-10
-25
0
500
1000
1500
2000
mh steps

program 60
estimated log score
-100
-400
0
500
1000
1500
2000
mh steps

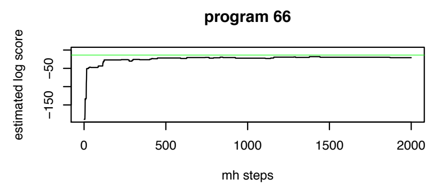

program 61
estimated log score
0
−40
−100
0
500
1000
1500
2000
mh steps

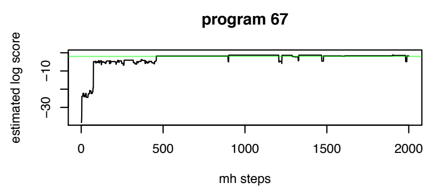

program 62
estimated log score
0
−20
−50
0
500
1000
1500
2000
mh steps

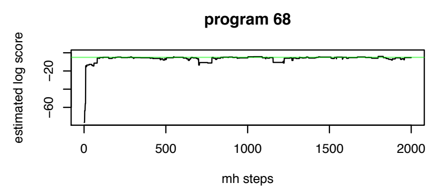

program 63
estimated log score
−10
−40
0
500
1000
1500
2000
mh steps

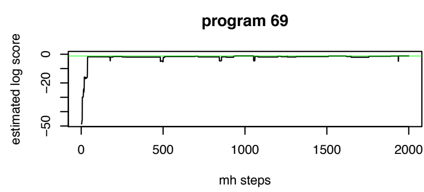

program 64
estimated log score
−100
−300
0
500
1000
1500
2000
mh steps

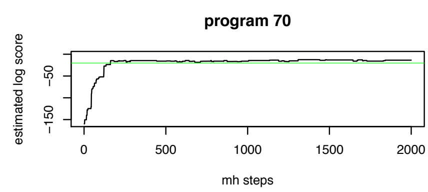

program 65
estimated log score
0
−40
−100
0
500
1000
1500
2000
mh steps

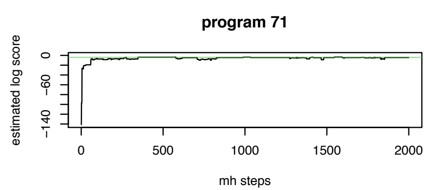

program 66
estimated log score
−50
−150
0
500
1000
1500
2000
mh steps

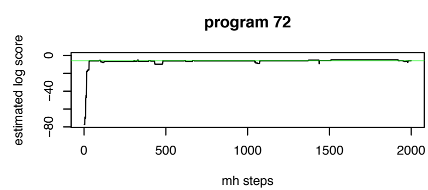

program 67
estimated log score
−10
−30
0
500
1000
1500
2000
mh steps

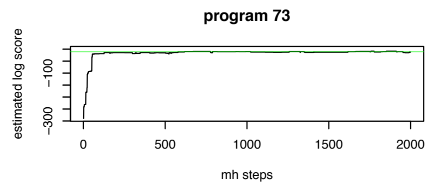

program 68
estimated log score
−20
−60
0
500
1000
1500
2000
mh steps

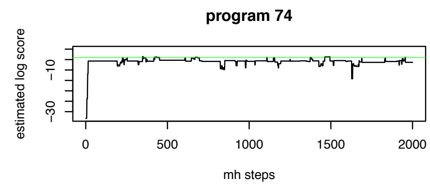

program 69
estimated log score
0
−20
−50
0
500
1000
1500
2000
mh steps

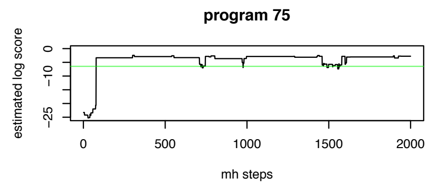

program 70
estimated log score
−50
−150
0
500
1000
1500
2000
mh steps

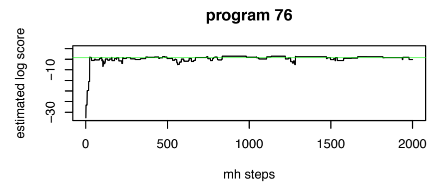

program 71
estimated log score
0
−60
−140
0
500
1000
1500
2000
mh steps

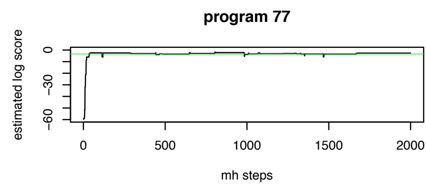

program 72
estimated log score
0
−40
−80
0
500
1000
1500
2000
mh steps

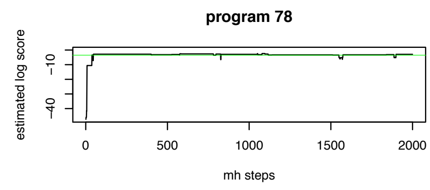

program 73
estimated log score
mh steps

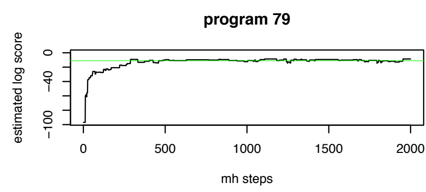

program 74
estimated log score
mh steps


program 75
estimated log score
mh steps


program 76
estimated log score
mh steps


program 77
estimated log score
mh steps


program 78
estimated log score
mh steps


program 79
estimated log score
mh steps

Sampled Programs

#	Program
1	(lambda () (node ’a (node ’a)))
2	(lambda () (if (flip) (node ’d (node ’b)) (node ’a)))
3	(lambda () (node ’a (node ’b)))
4	(lambda () (node ’a (node ’d)))
5	(lambda () (if (flip) (node ’a (node ’c (node ’d (node ’c)) (node ’d))) (node ’b (node ’c) (if (flip) (node ’b (if (flip) (node ’b) (node ’c))) (node ’b)))))
6	(lambda () (if (flip) (node ’a) (node ’d (node ’a))))
7	(lambda () (if (flip) (node ’a (if (flip) (node ’b) (node ’c))) (node ’a (node ’d (node ’c (if (flip) (node ’a) (node ’a)) (node ’b))))))
8	(lambda () (node ’b (if (flip) (node ’b) (node ’c)) (node ’c) (if (flip) (node ’d) (node ’b)) (node ’a) (node ’d))) a) (d)))
9	(lambda () (if (flip) (node ’c) (node ’c)))
10	(lambda () (if (flip) (node ’a (if (flip) (node ’a) (node ’b))) (node ’b)))
11	(lambda () (node ’c (node ’c) (if (flip) (node ’a) (node ’c)) (if (flip) (node ’c) (node ’a)) (node ’b)))
12	(lambda () (node ’a (node ’d) (if (flip) (node ’d) (node ’d))))
13	(lambda () (node ’d (node ’c) (if (flip) (node ’a) (node ’a))))
14	(lambda () (node ’c (node ’b))) b)) (c (b)))
15	(lambda () (node ’a (node ’b))) b)) (a (b)))
16	(lambda () (node ’a (if (flip) (node ’b) (node ’b)) (if (flip) (node ’b) (node ’c)) (node ’b)))
17	(lambda () (node ’b (if (flip) (node ’a) (node ’d))))
18	(lambda () (node ’c (node ’b))) b)) (c (b)))
19	(lambda () (if (flip) (node ’d (node ’c) (if (flip) (node ’b) (node ’d)) (node ’d) (node ’b (node ’d)) (node ’c) (node ’c)) (node ’a)))
20	(lambda () (node ’a (node ’a) (node ’c))) (a) (c)) (a (a) (c)))
21	(lambda () (node ’b (node ’b))) b)) (b (b)))
22	(lambda () (node ’d (node ’d))) d)) (d (d)))
23	(lambda () (if (flip) (node ’d) (node ’d)))
24	(lambda () (node ’b (if (flip) (node ’c) (node ’b))))
25	(lambda () (node ’b (node ’a) (node ’b (if (flip) (node ’b) (node ’c (node ’a))) (node ’b) (node ’d) (if (flip) (node ’a) (node ’c))) (node ’a) (node ’a))) (b (a) (b (c (a)) (b) (d) (a)) (a) (a)))
26	(lambda () (node ’b (node ’c))) c)) (b (c)))
27	(lambda () (if (flip) (node ’b) (node ’d)))
28	(lambda () (node ’c (node ’b (node ’b (if (flip) (node ’a) (node ’c)))) (if (flip) (node ’d) (node ’b)) (node ’a) (node ’a (if (flip) (node ’b) (node ’c)) (node ’d))))
29	(lambda () (node ’c (node ’d (node ’d)) (node ’b (if (flip) (node ’c) (node ’a (node ’b))) (if (flip) (node ’a) (node ’a)))))
30	(lambda () (node ’b (node ’a) (node ’b)))
31	(lambda () (node ’c (node ’c (node ’a) (node ’c) (if (flip) (node ’d) (node ’b)))))
32	(lambda () (node ’a (if (flip) (node ’a) (node ’a))))
33	(lambda () (node ’a (node ’d))) d))
34	(lambda () (node ’b (if (flip) (node ’b (node ’d)) (node ’a))))
35	(lambda () (node ’b (node ’c)))
36	(lambda () (if (flip) (node ’a) (node ’d)))
37	(lambda () (if (flip) (node ’b (node ’a)) (node ’b (node ’a (node ’a)))))
38	(lambda () (node ’a (node ’d (if (flip) (node ’d (if (flip) (node ’d (node ’c)) (node ’c))) (node ’c)))))
39	(lambda () (node ’a (node ’d (node ’d))))
40	(lambda () (node ’a (if (flip) (node ’b (node ’d)) (node ’b))))
41	(lambda () (if (flip) (node ’c) (node ’b (node ’a) (node ’b))))
42	(lambda () (if (flip) (node ’a) (node ’d (node ’c) (node ’d (node ’b)))))
43	(lambda () (node ’a (node ’b (node ’c)) (node ’d) (node ’d) (node ’c) (node ’b)))
44	(lambda () (if (flip) (node ’c) (node ’b)))
45	(lambda () (node ’c (node ’d))) d))

#	Program
46	(lambda () (node 'b (node 'c) (node 'b)))
47	(lambda () (node 'b (node 'c)))
48	(lambda () (if (flip) (node 'd) (node 'b)))
49	(lambda () (node 'b (node 'd) (node 'c)))
50	(lambda () (node 'd (node 'd) (node 'c) (node 'b)))
51	(lambda () (node 'b (node 'c) (if (flip) (node 'b) (node 'd (if (flip) (node 'a) (node 'c))))))
52	(lambda () (node 'a (if (flip) (node 'd (node 'a) (node 'd)) (node 'a))))
53	(lambda () (node 'd (node 'd (node 'a)) (if (flip) (node 'd) (node 'a (node 'd) (node 'b)))))
54	(lambda () (if (flip) (node 'a) (node 'c)))
55	(lambda () (if (flip) (node 'a (node 'c)) (node 'd (node 'd) (if (flip) (node 'd) (node 'a)) (if (flip) (node 'b) (node 'c)))))
56	(lambda () (if (flip) (node 'd (node 'c)) (node 'a)))
57	(lambda () (node 'c (node 'd) (node 'd)))
58	(lambda () (node 'd (node 'b (node 'b))))
59	(lambda () (if (flip) (node 'a (node 'a)) (node 'd)))
60	(lambda () (node 'd (if (flip) (node 'c) (node 'd)) (node 'b) (node 'b) (node 'd) (if (flip) (node 'c) (node 'a)) (if (flip) (node 'a) (node 'a (node 'a))) (node 'a)))
61	(lambda () (node 'c (node 'd (if (flip) (node 'd) (node 'b)))))
62	(lambda () (if (flip) (node 'd) (node 'd (node 'a) (node 'd))))
63	(lambda () (node 'c (if (flip) (node 'b) (node 'a (node 'a)))))
64	(lambda () (node 'd (node 'b (node 'c)) (node 'c (node 'b) (node 'd)) (if (flip) (node 'a) (node 'd (node 'd))) (node 'c)))
65	(lambda () (node 'c (node 'c) (node 'b)))
66	(lambda () (if (flip) (node 'd (node 'c (if (flip) (node 'a) (node 'd)) (node 'c (node 'c))) (node 'd)) (node 'a (if (flip) (node 'd) (node 'b (if (flip) (node 'b) (node 'b)))))))
67	(lambda () (node 'd (node 'd)))
68	(lambda () (if (flip) (node 'b) (node 'b (node 'a) (node 'b))))
69	(lambda () (node 'c (if (flip) (node 'c) (node 'c))))
70	(lambda () (node 'b (if (flip) (node 'a (node 'a)) (node 'a)) (node 'd) (if (flip) (node 'c (node 'c (node 'c))) (node 'c))))
71	(lambda () (node 'b (node 'b (node 'a) (node 'd))))
72	(lambda () (node 'b (node 'a (if (flip) (node 'd) (node 'c)))))
73	(lambda () (node 'b (node 'c) (node 'd (node 'b) (if (flip) (node 'b) (node 'c)) (node 'b))))
74	(lambda () (if (flip) (node 'b (node 'c)) (node 'b)))
75	(lambda () (node 'd (if (flip) (node 'd) (node 'a (node 'b) (node 'a))) (node 'd)))
76	(lambda () (if (flip) (node 'd) (node 'b (node 'c))))
77	(lambda () (node 'c (node 'c) (node 'd)))
78	(lambda () (node 'c (node 'a) (node 'd)))
79	(lambda () (node 'b (if (flip) (node 'c (if (flip) (node 'a) (node 'd))) (node 'a (node 'c) (if (flip) (node 'd) (node 'd))))))