Natively Probabilistic Computation

by

Vikash Kumar Mansinghka

S.B., Computer Science, Massachusetts Institute of Technology (2005)
S.B., Mathematics, Massachusetts Institute of Technology (2005)

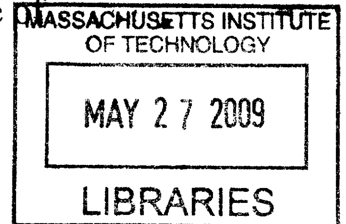
Submitted to the Department of Brain & Cognitive Sciences
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Cognitive Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
MAY 27 2009
LIBRARIES

© Massachusetts Institute of Technology 2009. All rights reserved.

Author ....
Department of Brain & Cognitive Sciences
April 20, 2009

Certified by ....
Joshua B. Tenenbaum
Paul E. Newton Career Development Professor
Thesis Supervisor

Accepted by ....
Earl Miller
Picower Professor of Neuroscience and
Chairman, Department Committee on Graduate Students

ARCHIVES

Natively Probabilistic Computation

by

Vikash Kumar Mansinghka

Submitted to the Department of Brain & Cognitive Sciences
on April 20, 2009, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Cognitive Science

Abstract

I introduce a new set of natively probabilistic computing abstractions, including probabilistic generalizations of Boolean circuits, backtracking search and pure Lisp. I show how these tools let one compactly specify probabilistic generative models, generalize and parallelize widely used sampling algorithms like rejection sampling and Markov chain Monte Carlo, and solve difficult Bayesian inference problems.

I first introduce Church, a probabilistic programming language for describing probabilistic generative processes that induce distributions, which generalizes Lisp, a language for describing deterministic procedures that induce functions. I highlight the ways randomness meshes with the reflectiveness of Lisp to support the representation of structured, uncertain knowledge, including nonparametric Bayesian models from the current literature, programs for decision making under uncertainty, and programs that learn very simple programs from data. I then introduce systematic stochastic search, a recursive algorithm for exact and approximate sampling that generalizes a popular form of backtracking search to the broader setting of stochastic simulation and recovers widely used particle filters as a special case. I use it to solve probabilistic reasoning problems from statistical physics, causal reasoning and stereo vision. Finally, I introduce stochastic digital circuits that model the probability algebra just as traditional Boolean circuits model the Boolean algebra. I show how these circuits can be used to build massively parallel, fault-tolerant machines for sampling and allow one to efficiently run Markov chain Monte Carlo methods on models with hundreds of thousands of variables in real time.

I emphasize the ways in which these ideas fit together into a coherent software and hardware stack for natively probabilistic computing, organized around distributions and samplers rather than deterministic functions. I argue that by building uncertainty and randomness into the foundations of our programming languages and computing machines, we may arrive at ones that are more powerful, flexible and efficient than deterministic designs, and are in better alignment with the needs of computational science, statistics and artificial intelligence.

Thesis Supervisor: Joshua B. Tenenbaum
Title: Paul E. Newton Career Development Professor

"Every reader should ask himself periodically 'Toward what end, toward what end?' but do not ask it too often lest you pass up the fun of programming for the constipation of bittersweet philosophy."

– Alan Perlis, 1922 - 1990

"After growing wildly for years, the field of computing appears to be reaching its infancy."

– John Pierce, 1910 - 2002

Acknowledgments

I am lucky to have many people to thank.

First, my advisor, Josh Tenenbaum. It is probably fair to say I would not have started my PhD without his support, finished it without his guidance, or enjoyed it as much as I did without his friendship. He brings a rare combination of vision, depth and pragmatism to his work that continues to inspire me. I'll even forgive him for calling me a "proposal distribution with high variance" in lecture, mostly because it is true.

Next, my collaborators. Without Eric Jonas, the circuits project would probably never have gotten past half-baked squiggles on a whiteboard, and I also might never have learned that one of the best reasons to work hard is to earn the right to work hard on harder things (or that Haskell programmers sometimes need to be kicked in the monads). Without Noah Goodman, Church would never have been conceived, and my intellectual life as a PhD student would have involved fewer heated arguments and much less depth. I'm very glad we ended up in the same place at the same time. Without Dan Roy, who brings a rigorous mathematical perspective to all his work, the search project would have remained a collection of hacks and (along with several other projects of mine) might never have seen the light of day. Dan and I first started working together while taking Josh's class in 2004, doing problem sets in the Building 66 cluster at 2am. The late hour and impending deadline turned out to be regular features of our collaborations; future students of his should know that his attitude and energy seems to improve as things get closer to the wire.

Beau Cronin and Keith Bonawitz also deserve thanks for their friendship, support and collaboration. Keith's work on the Blaise language for distributed stochastic automata gave me the first hints of the approach in this dissertation, and Beau provided critical support in keeping the main thing the main thing as this process was wrapping up. Had I not met them, this dissertation might have never been written, and I look forward to continuing to work with them in the years to come.

My committee members all mentored me in various ways long before I knew I'd be writing a PhD thesis someday. Hal Abelson's uncompromising lucidity has been inspiring me since I was an undergraduate, and his insistence on asking the question "What is the problem of which your work is a subproblem?" has helped me avoid many local minima. Tom Knight repeatedly encouraged me to break free of the architectural constraints of the conventional microprocessor while cheerfully pointing out the large graveyard filled with the bodies of almost everybody who tries. Tommy Poggio helped straighten out my thinking during my first ill-conceived attempt to go to graduate school and did some of the earliest work that inspired this thesis.

Two influences deserve special recognition. Gerry Sussman has mentored me since I was a freshman. His creativity and perspective on programming, teaching, ethics and physics have shaped my thinking and helped to define the framework of my MIT education. I hope he will forgive me my occasional bouts of zealotry and evangelism and continue to give me crap about it, along with tons of other advice that I'm rarely sensible enough to follow (though I'm grateful each time I do). Ray Olszewski taught an after-school BASIC programming class to a bunch of 8 year olds, including me, turning us loose in a room full of Apple II computers. This experience changed my life, and I will always be grateful for it. A few other teachers I must thank include Patrick Winston, Bruce Cohen, Sharon Lee, Scott Gasparian, Carla Newton, and Mary Laycock.

My colleagues and friends really made my MIT experience. Although I don't have time or space to acknowledge them all here, and hope I'll have a chance to correct any omissions in person, here are a few. Charles Kemp did the work that made me want to study probabilistic methods in the first place, and let me share an office with him and regularly waste his time. Tom Griffiths taught me how to think about nonparametrics and humored me while I tried my hand at cognitive science. Zane Shelby provided invaluable encouragement and companionship in regular doses; I'm sorry to be leaving Boston just as he finally arrived. Jay Pottharst has been and continues to be a wonderful friend, sharing good times with me and helping me through rough ones (and, incidentally, taught me basically all the math I know). Jon Gonda made me the drink that gave me the hangover that fueled the procrastination that, two conference rejections and two years later, led to the search chapter. Julie Vinogradsky walked with me during the final stages, providing a fresh perspective. I should also acknowledge Ryan Rifkin, Jacob Beal, Cameron Freer, David Wingate, Pat Shafto, Elizabeth Bonawitz, Mike Frank, Ed Vul, Konrad Koerding, Amy Perfors and Yarden Katz.

Finally, I would like to thank Asha Mansinghka, my mother, and Surendra Mansinghka, my father, for the sacrifices they made to give me the opportunity to follow my passions, and for — in the end — making it possible for me to keep "playing on the damned computer".

Chapter 2 is based on joint work with Noah Goodman, Daniel Roy, Keith Bonawitz and Josh Tenenbaum, published in UAI 2008 as "Church: a language for generative models". Chapter 3 is based on joint work with Daniel Roy, Eric Jonas and Josh Tenenbaum, published in AISTATS 2009 as "Exact and Approximate Sampling by Systematic Stochastic Search". Chaper 4 is based on joint work with Eric Jonas and Josh Tenenbaum, published as a CSAIL technical report as "Stochastic Digital Circuits for Probabilistic Inference". The work in this dissertation was partially supported by gifts from the Eli Lilly Corporation and Google, and graduate fellowships from the National Science Foundation and MIT's Lincoln Laboratory.

Contents

1 Introduction **21**

1.1 Distributions generalize functions, and sampling generalizes evaluation 25

1.2 Natively probabilistic computation via stratified design 28

1.3 Contributions 31

1.3.1 Moral: Don't calculate probabilities; sample good guesses. 34

2 Church: a universal language for probabilistic generative models **37**

2.1 Introduction 37

2.1.1 Languages for Knowledge Representation 39

2.1.2 Why Build on Lisp? 41

2.2 A Brief Tour of Church 43

2.2.1 Stochasticity and Call-by-Value 43

2.2.2 Universal Bayesian Inference and Conditional Simulation 48

2.2.3 Expressiveness 55

2.3 Discussion and Directions 63

2.3.1 Inference and Complexity 63

2.3.2 Semantics and First-class Objects 64

2.3.3 Probabilistic programs, integer programs and logic programs 66

2.3.4 Inductive Programming and Programming Style 66

3 Systematic Stochastic Search **69**

3.1 Introduction 69

3.1.1 Sampling can manage multimodality where other inference methods fail . . 70

3.1.2 Systematic vs Local Algorithms, Fixed Points, and Samplers 71

3.2 The Adaptive Sequential Rejection Algorithm 74

3.2.1 Adaptation Stochastically Generalizes Backtracking 76
3.2.2 Sequences of Distributions for Graphical Models 79
3.3 Experiments 80
3.4 Discussion 83
3.5 Appendix 85

4 Stochastic Digital Circuits for Probabilistic Inference **91**
4.1 Introduction 91
4.2 Stochastic Logic Circuits 92
4.2.1 Sampling is linear, while probability evaluation is exponential 95
4.2.2 Probabilistic Completeness and Finite Precision 96
4.2.3 Stochastic Circuit Complexity and Design Patterns 98
4.2.4 Approximate Inference using MCMC and Gibbs processors 100
4.2.5 Implementation via commodity FPGAs 102
4.2.6 Related Work 103
4.3 Performance Estimates 103
4.3.1 Robustness 105
4.4 Discussion and Future Work 107

5 Challenges in Connecting the Layers **111**
5.1 Generating Circuits That Solve Factor Graphs 112
5.2 Probabilistic Compilation and Architecture 115
5.2.1 How should we parallelize recursive processes? 116
5.2.2 How should we represent structured state for MCMC? 118
5.3 Probabilistic Computational Complexity 119

6 Conclusion **125**

List of Figures

1-1 Probabilistic generative modeling involves probabilistically coherent reasoning over distributions on possible worlds and the causal histories that generated them. In this simple example, a causal history consists of 3 independent coin flips, and all worlds are equally likely. Modern probabilistic models regularly involve thousands or millions of stochastic choices with complex dependencies. 22

1-2 Engineering languages, equipped with powerful primitives and closed under expressive means of composition and abstraction, form the basis of our ability to synthesize and analyze complex systems. Many of these languages are related, so that specifications written in one language can be implemented in another. Our stack of software and hardware abstractions for computing consists of these. 29

1-3 The layers of language supporting modern probabilistic AI. Each layer focuses on specifying and evaluating deterministic functions, building up from Boolean functions, to state-update functions for deterministic finite state machines, to programs in functional programming languages, to data structures and functions for computing probabilities. 30

1-4 A stack of abstractions for natively probabilistic computing. The stack builds up from probabilistic circuits, to massively parallel, fault-tolerant stochastic finite state machines, to probabilistic programs. Each layer is based on *distributions* and *samplers*, recovering the corresponding layer in Figure 1-3 as a special case based on functions. 32

1-5 Probabilistic programming languages like Church, which describe stochastic processes or probabilistic generative processes, can be used to represent both uncertain beliefs and useful algorithms in a single notation. Learning and reasoning will involve executing probabilistic programs and learning probabilistic programs from data, using general machinery. For the subset of models described by factor graphs, I show how to solve them using a new, recursive sampling algorithm, as well as generate static-memory stochastic state machines for solving them, that can be implemented via massively parallel stochastic circuits. 36

2-1 Inputs and repeated outputs from a Church machine session showing the interaction of call-by-value semantics with stochasticity. The first example shows interference stemming from binding at lambda application. The second and third examples show how random worlds can be built using binding, where the rules of logic apply to the values of expressions that only involve deterministic procedures and symbols. The fourth example shows what happens when trying to compare the invocation of two coin flips. We argue this is appropriate behavior for a knowledge representation system. `(x)` denotes the result of simulating the (potentially stochastic) process named `x`, where `x` denotes the fixed value of the symbol `x`. . . . 44

2-2 Expressing the Beta-Bernoulli model as a pure thunk valued procedure. To construct a coin, we return a procedure that closes over a single randomly chosen coin weight. Calls to such procedures are *exchangeable*, not independent and identically distributed: the order of the calls doesn't matter, but knowledge of one return value does change our expectations about other return values. However, if we can look inside the closed-over environment, we see the representation in terms of independent, identically distributed draws. 47

2-3 Mutation is allowable as long as it preserves exchangeability. Using this idiom, we give an alternate representation for the Beta-Bernoulli from Figure 2-2, which operates by sampling from a sequence of predictive distributions, updating sufficient statistics after each call. 48

2-4 Memoization, an exchangeable operation, can be used to construct infinite random objects whose pieces are indexed by the arguments to procedures, such as streams of random bits. This permits delayed evaluation in the probabilistic setting. 49

2-5 A metacircular description of QUERY, which generalizes EVAL to provide lexically-scoped *conditional* simulation of an admissible stochastic process. QUERY takes an expression, an environment and a predicate, and returns a sampled value from the *conditional* distribution on values induced by the expression, given the constraint that the predicate applied to that value is true. 51

2-6 A schematic computation trace, showing the DAG obtained by connecting the tree of Church expressions and subexpressions with the tree of environments during the evaluation of the expression `((lambda (x) (+ x 1)) (flip))`. 55

2-7 A complete computation trace saved using the trace debugging facility of the Blaise implementation of Church. 56

2-8 The Dirichlet Process can be naturally expressed as a higher-order Church procedure. Since its de Finetti or stick-breaking representation is infinite, we use MEM to delay the evaluation of the sticks until needed. 56

2-9 Using the DP as a primitive, we can stochastically generalize memoization, yielding a version where a procedure is re-evaluated (possibly resulting in a new value, if the procedure is stochastic) with probability following the Chinese restaurant process each time it is called. With $\alpha = 0$ we recover deterministic memoization, where the procedure is only evaluated once, and with $\alpha = \infty$ we recover the absence of memoization (but wasting space to store the intermediate values). This idiom helps to explain the widespread popularity of the Dirichlet Process in probabilistic generative modeling, where it often serves the role of "stochastically caching" fragments of a generative process. 57

2-10 Nesting EVAL inside QUERY allows one to express the problem of learning Church programs from data, without having to reinvent the representational machinery of a programming language. This program learns various ways of expressing the value 24, consistent with a particular context-free grammar on program text in symbolic expression form. 61

2-11 Planning as inference — or, more formally, softmax-optimal decision making in a sequential decision problem — is one instance of metareasoning, or reasoning about reasoning. The essential recursion captures the question "how should I act today, given that I will reason and act approximately optimally after taking that action?". 63

3-1 A four node Ising model, and its restriction to the variables x_1 and x_3. 80

3-2 (left 4) Exact samples from a 100-dimensional grid ferromagnetic Ising just below the critical temperature. (right 4) Exact samples from a 100-dimensional grid ferromagnetic Ising just above the critical temperature. 81

3-3 (left) Ferromagnetic (right) Antiferromagnetic. (both) Frequency of acceptance in nonadaptive (blue, lower) and adaptive (red, higher) sequential rejection as a function of coupling strength J. Naive rejection approaches suffer from exponential decay in acceptance probability with dimension across all coupling strengths, while generic MCMC approaches like Gibbs sampling fail to converge when the coupling reaches or exceeds the critical value. Note that adaptive rejection improves the bound on the region of criticality. 82

3-4 Comparison of adaptive (top-red, center) and nonadaptive (top-blue/dashed, bottom) rejection sampling on a frustrated 36-dimensional Ising model with uniform $[-2, 2]$ distributed coupling parameters. (top) Cumulative complete samples over 100,000 iterations. (lower plots) A black dot at row i of column j indicates that on the jth iteration, the algorithm succeed in sampling values for the first i variables. Only a mark in the top row indicates a successful complete sample. While the nonadaptive rejection sampler (bottom) often fails after a few steps, the adaptive sampler (center), quickly adapts past this point and starts rapidly generating samples. 87

3-5 Comparison of adaptive (top-red, center) and nonadaptive (top-blue/dashed, bottom) rejection sampling for posterior inference on a randomly generated medical diagnosis network with 20 diseases and 30 symptoms. The parameters are described in the main text. (top) Cumulative complete samples over 100,000 iterations. (lower plots) show the trajectories of a typical adaptive and non-adaptive run in the same format as Figure 3-4. Here, adaptation is critical, as otherwise the monolithic noisy-OR factors result in very low acceptance probabilities in the presence of explaining away. 88

3-6 Comparison of an aggressively annealed Gibbs sampler (linear temperature schedule from 20 to 1 over 200 steps) to the non-adaptive, importance relaxation of our algorithm. The red circle denotes the mean of three 1-particle runs. The horizontal bars highlight the quality of our result. (a) Gibbs stereo image after sampling work comparable to an entire 1-particle pass of (b) our algorithm. (c) Gibbs stereo image after 140 iterations. 89

4-1 **Combinational stochastic logic.** (a) The combinational Boolean logic abstraction, and one example: the AND gate and its associated truth table. (b) The *combinational stochastic logic* abstraction. On each work cycle, samples are drawn on OUT from $P(\texttt{OUT}|\texttt{IN})$, consuming h random bits on RAND to generate nondeterminism. (c) An AND gate can be viewed as a combinational stochastic logic gate that happens to be deterministic. (d) The conditional probability table and schematic for a Θ gate, which flips a coin whose weight was specified on IN as a binary number (e.g. for IN = 0111, $P(\texttt{OUT} = 1|\texttt{IN})$ = 7/16). Θ gates can be implemented by a comparator that outputs 1 if RAND $\leq$ IN. 93

4-2 **Composition and abstraction laws.** (a) Boolean gates support expressive composition and abstraction laws. The law of composition states that any two Boolean gates f and g with compatible bitwidths can be composed to produce a new Boolean gate h. The law of abstraction states that h can now be used in designing further Boolean circuits without reference to the components f and g that it is composed of. (b) The analogous laws for combinational stochastic logic: one can sample from joint distributions built out of pieces, or view a complex circuit abstractly as a primitive that samples from a marginal distribution. Note that in this case the input entropy is divided among the two internal elements; the abstraction preserves the notion of a single incoming source of randomness, even though internally the elements receive effectively distinct streams. 94

4-3 **Example stochastic circuit designs.** (a) and (b) show two circuits for sampling from a binomial distribution on n flips of a coin of weight p, consuming nh total bits of entropy. (a) shows a circuit where the coins are flipped in parallel and then summed, costing $O(n)$ space and $O(log(n))$ time per sample. (b) shows a serial ciruit for the same problem, using $O(log(n))$ space (for the accumulator) and $O(n)$ time. Clocked registers are shown as units with inputs and outputs labeled D and Q. (c) shows a stochastic finite state machine (or finite-state Markov chain), with a state register connected to a combinational state transition block. 99

4-4 **Designs for Gibbs samplers.** (a) shows a schematic Gibbs pipline, outlining the operations needed to numerically sample a single arbitrary-size variable. (b) shows a condensed implementation of a Gibbsable binary variable unit, which internally stores both a current state setting and a precomputed CPT lookup table (LUT), and runs in 3 clock cycles. (c) and (d) show colored MRFs, where all variables of each color can be sampled in parallel. (e) shows a distributed circuit that implements Gibbs sampling on the MRF in (c) using the Gibbs units from (b). 101

4-5 **FPGA price/performance estimates.** (a) shows an example synthesized FPGA layout for Gibbs sampling on a 9x9 lattice. (b) compares the price/performance ratio of a stochastic circuit to an optimistic estimate for conventional CPUs, in billions of single-site samples per second per dollar, for a binary square-lattice MRF. (c) shows a zoomed-in comparison. (d) shows price/performance results for stereovision. Approximately 10 billion samples per second are needed for real-time (24 FPS) performance at moderate (320x240 pixels) resolution. 104

4-6 (Top left) The left camera image in the Tsukuba sequence from the Middlebury Stereo Vision database. (Bottom left) The ground truth disparity map. (Top right) The solution found by an annealed Gibbs sampler run at floating point precision after 400 iterations. (Bottom right) The solution found by simulation where probabilities were truncated after calculation but before sampling to 8 bits (which we expect is roughly comparable to full computation on our arithmetic pipeline with 10 to 15-bit fixed point). Note that quality is only slightly degraded. 109

4-7 **Robustness to faults.** (a) shows a butterfly-structured binary Markov random field with attractive potentials. (b) shows robustness results to transient single site faults (assessed by deviation of the equilibrium distribution of a Gibbs sampler circuit from the target). 110

5-1 An overview of the compiler we have implemented to simplify the generation of stochastic digital circuits to simulate from factor graphs. Graph coloring is used to automatically identify fine-grained conditional independencies in the model. These independencies are used to automatically generate a graphical description of a parallel Gibbs sampler in the State-Density-Kernel language, serving as a block diagram level design for the digital circuit. A combination of software support and hand digital design (e.g. for implementing potentials) is needed for the final translation to executable digital circuit. 113

5-2 A screen capture of a Python fragment invoking our factor graph inference compiler. The displayed fragment generates a factor graph, identifies opportunities for exploitable parallelism, and generates an SDK description of a stochastic FSM for sampling from the distribution induced by the factor graph. The code fragment includes boilerplate to generate native code using LLVM for efficient, bit-accurate circuit simulation. 114

5-3 A State-Density-Kernel graph automatically produced by our compiler for performing parallel Gibbs sampling in a 2x2 Ising lattice. The density and kernel structure make explicit the full conditional independencies in the underlying factor graph for maximally parallel execution. 115

5-4 A synthesized FPGA layout for a circuit for a 9x9 Ising lattice, consistent with a 9x9 design (as SDK) produced by our compiler. This circuit assumes random bitstreams are provided from off the FPGA. 122

5-5 An ideal compiler for stochastic computation would involve automatic, universal conditional simulation (to remove all queries) and automatic identification and negotiation of time/space tradeoffs informed by the architectural constraints of the available machine. 123

List of Tables

2.1 An overview of some of the mathematical and computational formalisms for uncertain knowledge representation. Modern systems have come in three generations, using graph theory, logic, and programming languages to manage structure and probability to manage uncertainty. 39

3.1 Systematic algorithms build up complete solutions out of sequences of partial pieces, while local algorithms attempt to iteratively improve structurally complete candidate solutions. The systematic/local distinction cross-cuts many fundamental algorithmic problems. 72

Chapter 1

Introduction

> "For over two millennia, Aristotle's logic has ruled over the thinking of western intellectuals. All precise theories, all scientific models, even models of the process of thinking itself, have in principle conformed to the straight-jacket of logic. But from its shady beginnings devising gambling strategies and counting corpses in medieval London, probability theory and statistical inference now emerge as better foundations for scientific models, especially those of the process of thinking and as essential ingredients of theoretical mathematics, even the foundations of mathematics itself. We propose that this sea change in our perspective will affect virtually all of mathematics in the next century."
>
> — David Mumford, *The Dawning of the Age of Stochasticity*

We would like to build computing machines that can interpret and learn from their sensory experience, act effectively in real time, and — ultimately — design and program themselves. We would like to use these machines as models to help us understand our own minds and brains. We would also like to use computers to build and evaluate models of the world that can help us interpret our data and make more rational decisions.

Over the last 15 years, there has been a flowering of work on these problems, centered on probabilistically coherent or "Bayesian" reasoning (36; 46; 65; 26; 10). A Bayesian reasoner — be it an artificial agent, a program written to aid a human modeler, or a computational model of human learning and reasoning — represents its beliefs via a probability distribution on possible states of the world, including all quantities of potential interest that are not directly observable. This distribution assigns a nonnegative degree of belief to each possible world, such that the total amount of belief sums to 1. A Bayesian reasoner then updates its beliefs in light of evidence according

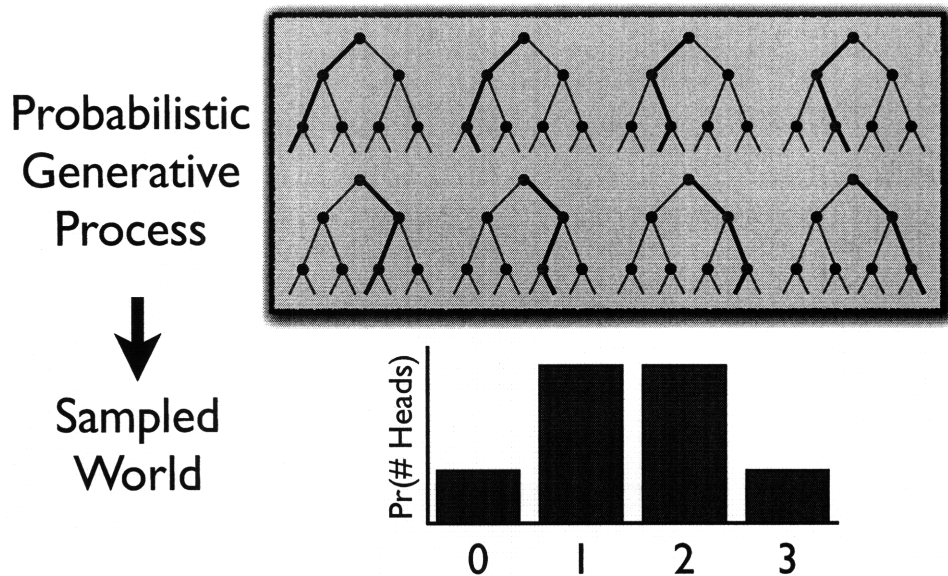


Figure 1-1: Probabilistic generative modeling involves probabilistically coherent reasoning over distributions on possible worlds and the causal histories that generated them. In this simple example, a causal history consists of 3 independent coin flips, and all worlds are equally likely. Modern probabilistic models regularly involve thousands or millions of stochastic choices with complex dependencies.

to the rules of the probability algebra: observed data is taken into account, or conditioned on, by renormalizing (i.e. summing and then dividing) the belief distribution over all worlds consistent with the observations. This renormalization results in a new probability distribution that represents the reasoner's degree of belief in each world after the evidence was observed.

Often the distribution is thought of in terms of a *probabilistic generative process* or "causal history": a sequence of probabilistic choices which produces a particular world, where probability is introduced wherever the modeler has uncertainty about the outcome of a choice. Figure 1-1 shows a toy example, where the world is composed of three independent flips of a fair coin; typical worlds in real examples are far more complex. Probabilistic learning and reasoning involves "inverting" this process using the probability algebra to identify the distribution on choices implied by the observation of a particular condition or overall outcome. These generative models routinely involve rich systems of latent variables that induce complex correlations and dependencies between observable quantities, rarely involving the strong independence, Gaussianity or steady-state assumptions common to classical stochastic models.

This simple mathematical setup is a consistent generalization of deductive Boolean logic to

the case of uncertain knowledge and plausible argument (36). It has been used to fruitfully attack a range of problems in the computational sciences, artificial intelligence, cognitive science and statistics. In each domain, probability is used to bring rich knowledge structures in quantitative contact with noisy, incomplete and often qualitative data, and make predictions about the future with calibrated uncertainty.

Its popularity reflects a growing awareness of its conceptual and technical advantages, such as the ability to construct large probability models out of pieces, and the automatic Occam's Razor that protects fully Bayesian inductive learners — which reason probabilistically about latent choices that are re-used in future events — from overfitting (46). It also reflects the development of computational tools for representing probability models and solving inference problems. The most important of these are probabilistic graphical models (57; 42), which use ideas from graph theory to capture symmetries that simplify the representation of probability distributions on finite worlds, and inference algorithms based on variational methods from computational physics and operations research (86).

However, it has proved very difficult to scale probabilistically coherent methods to either rich structures of knowledge — such as distributions over graphs, trees, grammars, worlds with hidden objects, and arbitrary programs — or to large volumes of data. These limitations of expressiveness and efficiency are substantial. Fully Bayesian learning and reasoning is usually limited to small worlds with tens or hundreds of latent variables, datasets with hundreds or thousands of datapoints, and offline (as opposed to realtime) processing. To understand why, consider that a world comprised of 100 binary variables — just enough to segment a 10x10 image — has 2^{100} distinct states. To build a model of that toy world, each of those states must be assigned a definite probability. Very simple problems in phylogenetics and segmentation grow even quicker: there are over 6 billion binary trees with 20 nodes, and over 10 billion ways to assign only 16 objects into distinct categories. This state-space explosion immediately leads to two challenges:

1. **It seems hard to write down probability models, especially over richly structured worlds.**

   Specifying rich probability models boils down to specifying probability distributions over very large spaces. The main challenges lie in writing down distributions on worlds without needing to directly specify exponentially or infinitely many real numbers, using languages that support the reuse and recombination of probability distributions into ever larger structures.

   Graphical models mitigate this problem somewhat for finite worlds: they allow one to build

up functions that compute the probability of a potentially large set of variables out of sub-functions that depend only on subsets of the variables (86). However, they lack many of the means of combination and abstraction that are supported by richer formal systems, like programming languages, and are especially difficult to apply to worlds that are infinite or involve recursive structures like trees and grammars. As a result, modelers working in these settings exchange models using an informal mix of graphical models, mathematical notation and natural language, impeding reuse and necessitating the development of a large number of one-off inference schemes. This lack of expressiveness has restricted the complexity of models people have been able to build, especially in situations involving metareasoning and inductive learning of structured representations. Graphical models also suffer from the problem that conditioning on data frequently destroys the decomposition of the joint distribution that a graphical model is based on. Put differently, they are not closed under reasoning, which can make it hard to use them as a representation for the beliefs of an agent which must reason repeatedly over time.

2. **Accurate probabilistic inference seems computationally intractable.**

   Performing exact probabilistic inference based on functions that compute probabilities seems to require intractable summations and produce exponentially or infinitely large objects. For example, Bayes' Rule tells us how to go from a model and some data to posterior beliefs:

$$P(H|D) = \frac{P(H)P(D|H)}{P(D)} = \frac{P(H,D)}{\sum_H P(H,D)}$$

   Unfortunately, evaluating the denominator requires computing a sum over exponentially many hypotheses, and even if one could evaluate the probability function $f(H) = P(H|D)$, it remains computationally difficult to find high scoring hypotheses under f or integrate f to make predictions. This intractability stems from the size of the domain. Graphical models and variational methods attempt to mitigate these problems by focusing on finding an H which maximizes $P(H|D)$ and on computing marginal distributions which characterize the average variability of a distribution around its mode, ignoring correlations between variables. However, these problems are still often computationally intractable and require substantial auxiliary approximations (86). They also tend to obscure multimodality. For example, consider a distribution on 100 binary variables which labels them as either all true or all false. A naive mean field (87) or belief propagation (42) approach will confuse that distribution —

which allows just 2 possibilities — with one that says "each variable is independently either true or false", allowing all 2^{100} with equal probability.

Monte Carlo approximations, where a distribution is represented by a list of samples from it, are often easier to apply to problems of probabilistic inference (2; 54; 16), and are often used for structured problems. Unfortunately, they are often perceived as both unreliable and slow, requiring tens of thousands of costly iterations to converge.

In this dissertation, I approach these problems of representational expressiveness and inference efficiency from below, arguing that they reflect a basic mismatch between the mathematics of probability and our fundamentally deterministic and deductive view of computation. Rather than represent distributions in terms of functions that calculate probabilities, as in graphical models and variational methods, I represent distributions using procedures that generate samples from them. Based on this idea, I introduce a new set of natively probabilistic abstractions for building and programming computing engines, including probabilistic generalizations of Boolean circuits, backtracking search and pure Lisp, and lay out a program for developing them into a coherent stack for natively probabilistic computation.

1.1 Distributions generalize functions, and sampling generalizes evaluation

The most basic elements of digital computers, Boolean gates, correspond to elementary Boolean functions that compute the truth functions for logical connectives in the Boolean algebra. Since Turing, our theories of computation have defined computers as machines that evaluate deterministic functions — things that produce definite outputs for given inputs — by a series of deterministic steps. Our assessments of algorithmic complexity attempt to measure the resources needed for binary, all-or-nothing decisions like Boolean satisfiability or graph colorability. Even our programming languages center on deterministic functions and logical deduction. Lisp, one of the oldest, was introduced to define recursive functions. Languages like Haskell generally restrict programmers to define pure functions in an algebraic style and require that the type structure of the program be provably consistent before the program can be executed. The sloganized version of the Curry-Howard isomorphism — that "proofs are programs and programs are proofs" — is perhaps the most extreme version of the view that deduction and deterministic computation are one and the

same. Although randomized algorithms are widely used in practice, they are typically presented as strategies for approximately evaluating deterministic functions, for example by feeding the output of a sampler into a Monte Carlo estimator.

Given this deterministic orientation, it might seem particularly natural to represent a probability distribution in terms of a function that evaluates the probability of a state. However, from a mathematical perspective, we know that probability distributions contain functions as a special case: any function can be written as a probability kernel which is a delta distribution for each input – for each input x the kernel puts probability 1 on $f(x)$ and probability 0 everywhere else. Put differently, there are many more conditional distributions inducing stochastic mappings from some set X to some set Y than there are functions from X to Y. The functions simply correspond to the case where the mapping is 1 to 1. Of course, we normally circumvent this by using functions that calculate probabilities, which have a different type signature: a conditional distribution becomes a function from pairs (X, Y) to $[0, 1] \in \mathcal{R}$ that evaluates the probability of a given input-output pairing. This seemingly basic choice also lies at the heart of the measure theoretic approach to probability, where we work with functions mapping subsets of a space of possible worlds — i.e. predicates — to probabilities. However, it causes considerable difficulties when one wants to build big distributions from pieces or generate samples from even small distributions efficiently.

First, if what I have is a distribution or density represented as a probability function p from (X, Y) to $[0, 1]$, I need to develop independent methods to simulate from it, which rapidly grow complex as enumeration rapidly grows intractable. Second, despite the fact that conditional distributions can be arbitrarily composed, the output of such a function, a probability value, cannot be used as the input to another one, which expects an input-output pair. Instead, I have to compute a new probability function by some means that computes the probability function for the composition. This mismatch makes setting up probabilistic recursions especially cumbersome. Third, the complexity of compositions appears to behave badly. If I compose two functions f and g, the complexity of evaluating $g(f(x))$ is just the sum of the complexities of evaluating f and g. We would like to be able to compose distributions in the same way. However, if I have two conditional probability distributions $B|A$ and $C|B$, I have that their combined or "marginal" probability function is $p(C|A) = \sum_B P(C, B|A)$. The complexity of chaining conditional distributions together in terms of their probability functions grows exponentially rather than linearly. Thus both specifying and working with structures with even a small number of pieces rapidly becomes problematic.

The starting point for this dissertation is to make a different choice. Instead of representing a probability distribution in terms of a deterministic procedure that evaluates the probability of pos-

sible outcomes, I will represent probability distributions in terms of *probabilistic procedures that generate samples* from them. A "computation" will involve *simulating* one of these probabilistic procedures to generate a sample. Repeating a computation will, in general, yield a different result — but the *histogram* or table of long-run frequencies of the recorded outputs of a computation will converge to its underlying distribution[1]. This approach allows one to arbitrarily compose samplers, constructing large stochastic systems out of pieces, while preserving linearity of complexity. Furthermore, this approach contains the deterministic skeleton of computing, where a "computation" involves evaluating a deterministic function to obtain a fixed value, as a special case where the probabilistic procedure happens to consume no randomness. Throughout, I will view the specification of samplers and the efficient simulation from complex probability distributions, possibly obtained from other distributions by chains of conditioning, as the main goal[2].

Functions and deterministic procedures are used all throughout computing, from Boolean circuits through state machine updates to algorithms and programming languages. At each of these levels, I will ask "what set of distributions and samplers contains this set of functions and deterministic procedures as a well-defined special case, preserving its key properties?" *My thesis is that it is both feasible and useful to generalize basic computing abstractions in this way, yielding a conceptually coherent hardware and software stack for natively probabilistic computing*. By building uncertain knowledge and random processes into the foundations, systematically generalizing functions to distributions and procedures for evaluation to samplers at every level, we may obtain machinery that is more powerful, flexible and efficient than traditional designs. We may also find the resulting natively probabilistic machines more suitable for uncertain reasoning and inductive learning, and thus arrive at languages, algorithms and machines that are in better alignment with the needs of modern computational science, statistics and artificial intelligence.

[1]This procedural view of distributions has the flavor of frequentism — we will interact with uncertain beliefs through procedures that precisely model them random experiments. We will see, however, that this is a particularly natural way to build computers that are good at Bayesian reasoning.

[2]Contrast this view with the traditional one, where randomized methods are mainly used to approximate functions. For example, the Monte Carlo method is often seen as a way to approximate the value of of high dimensional integrals, such as the expectation of functions with respect to probability distributions. The main object of computation, however, is taken to be evaluating functions, for example to solve satisfiability problems, sort lists, find maxima or minima of functions, or compute sums.

1.2 Natively probabilistic computation via stratified design

The task of generalizing deterministic computing abstractions to the probabilistic setting immediately raises three, potentially thorny issues:

1. Every deterministic function can be obtained as the deterministic limit of many different sequences of probability distributions. How should we choose among them?

2. How can we preserve the combinatorial expressiveness of computing technology as we make our generalizations, so that we can build rich probabilistic structures out of reusable pieces?

3. How can we ensure compatibility between the different layers of abstraction in computing, so that we can define an object at one level — for example, a distribution to be sampled from — and be confident that implementation in terms of the lower levels — for example, an algorithm to sample from it, and a circuit implementing that algorithm — will be possible?

I leave it to the reader to judge how natural, expressive, coherent and useful the generalizations I present ultimately are. However, it is worth reflecting on the criteria that inform my approach.

The structure of deterministic computers, and in particular their *stratified design*, provides the key constraint that guided the generalizations I sought. A stratified approach to synthesizing complex systems involves building layers of language, each of which serves as an implementation language for the layer above. To be a language in this sense, a formal system must:

1. Provide powerful primitives, which serve as the initial building blocks of statements in the language.

2. Be closed under expressive means of combination, which allow compound statements to be constructed by combining smaller statements.

3. Provide means of abstraction, by which statements can be referred to without reference to their pieces, and objects can be created which are indistinguishable from built-in primitives.

Examples of such languages are ubiquitous in engineering; Table 1-2 lists a few examples, drawn from signal processing, analog and digital electronics, and computer programming. A critical requirement for a stratified design is that one be able to both specify a complex system in terms of pieces and then implement that system in terms of implementations of the pieces. As a consequence, if I add something to the specification of a system, I can simply add a corresponding piece

Language	Primitives	Means of Composition and Abstraction	Relations
Linear Time-Invariant Systems	Low-pass, high-pass filters; impulse responses	Serial composition as convolution, summation; black box abstraction	Used to implement signal processing operations; implemented in analog electronics and matrix algebra
Lumped-parameter circuits	Resistors, capacitors, inductors, transistors	Connection of device terminals; Thevenin and Norton equivalence	Used to implement LTI and digital systems, and to model electromagnetic, thermal and mechanical systems; implemented physically, hiding wave propagation
Combinational Boolean Logic	AND, OR, and NOT gates	Composition and abstraction of Boolean functions	Used to implement discrete, deterministic functions; implemented using lumped-parameter electronics
Synchronous Digital Electronics	Registered logic blocks, combinational blocks, and finite state machines	Discrete finite state machine composition and abstraction, inheriting from transition updates	Used to implement digital computers; implemented using combinational Boolean logic and lumped-parameter electronics
Lisp	Literal values, primitive procedures, special forms	Compound expressions; lambda abstraction	Used to implement processes that transform structured data; implementable directly in terms of synchronous digital electronics

Figure 1-2: Engineering languages, equipped with powerful primitives and closed under expressive means of composition and abstraction, form the basis of our ability to synthesize and analyze complex systems. Many of these languages are related, so that specifications written in one language can be implemented in another. Our stack of software and hardware abstractions for computing consists of these.

to the implementation, without worrying about all possible pairwise interactions among the parts. This enables a powerful separation of concerns; the disciplines of programming languages and semiconductor physics cooperate productively without their practitioners ever interacting directly.

Accordingly, the way we will look for good probabilistic generalizations is by insisting that the probabilistic objects we construct satisfy two constraints:

1. They should preserve the composition and abstraction laws of their deterministic counterparts, in both the deterministic limit and in the broader, probabilistic setting. This ensures we can build large structures out of reusable pieces.

2. Where appropriate, they should be implementable in terms of a lower layer of language, and be useful for implementing higher-level layers, in ways that correspond with their deterministic counterparts. This ensures the levels of language interoperate, collectively bridging the gaps between knowledge structures, programs, computing machines, and networks of gates.

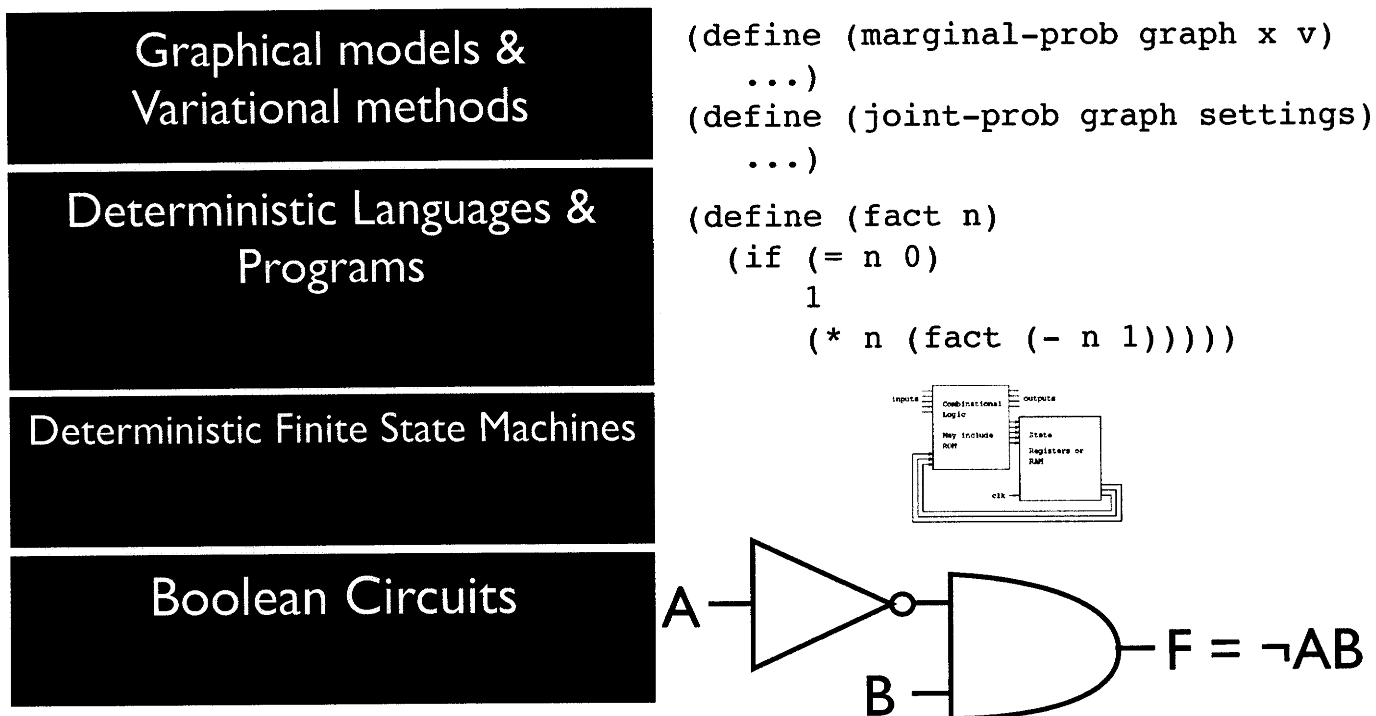


Figure 1-3: The layers of language supporting modern probabilistic AI. Each layer focuses on specifying and evaluating deterministic functions, building up from Boolean functions, to state-update functions for deterministic finite state machines, to programs in functional programming languages, to data structures and functions for computing probabilities.

This approach has repeatedly yielded design constraints that seem to be responsible for many of the pleasant properties of the abstractions I will introduce. We have already seen the first effect of this constraint. We will be working with samplers, not probability functions, because samplers can be recursively composed the way that functions can, and probabilistic procedures that implement samplers can be cheaply composed the way that deterministic procedures for evaluating functions can (but probability functions can't).

The power and expressiveness of modern computing — as well as the means by which enormous complexity is hidden from the users of each level — can be appreciated by understanding the ways in which deterministic programming and digital computers embody the principles of stratified design. Identifying some of the key layers of language that have composable implementations,

we can construct an (admittedly simplified[3]) picture of the stack, shown in Figure 1-3. At the lowest level, we construct our computers out of stateless Boolean circuits, implemented in terms of transistor-based Boolean gates. We assemble these circuits into deterministic finite state machines, arranged in structures especially appropriate for high-precision floating point arithmetic and for the execution of sequential programs. We program these state machines in a variety of languages, focusing on the specification and evaluation of complex, deterministic functions involved in numerical linear algebra and combinatorial optimization. We sometimes link these functions together to build engines for statistical and logical reasoning, and build our models and write AI programs in special-purpose declarative languages that these engines interpret.

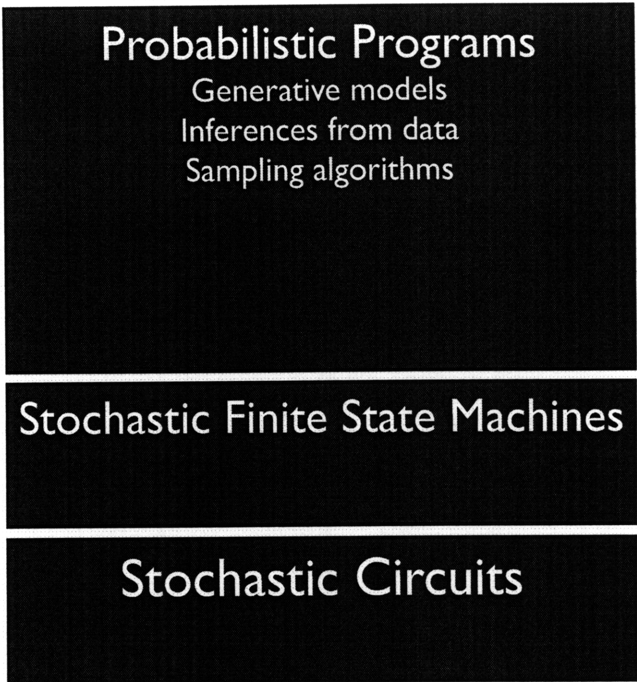
The stack of abstractions I propose is shown in Figure 1-4. At the lowest level, we will define our machines in terms of stochastic digital circuits that naturally produce samples from probability distributions rather than evaluate Boolean functions. We will wire these circuits into stochastic finite state machines — or Markov chains — whose natural convergence dynamics carries out useful sampling algorithms. Rather than perform any particular sequence of instructions precisely, these machines will reliably produce reasonable outputs in appropriate proportions. We will adopt reconfigurable computing, where reprogramming a machine feels much like rewiring a circuit, so we can reflect the conditional independencies of each individual probabilistic program we running directly in the structure of our machines and leverage massive parallelism. At the top, our programming langauges will allow us to define probabilistic procedures for generating samples from recursively defined distributions, supporting both algorithm design and model-building in the same notation. Both the inputs to and the outputs of learning and reasoning will be probabilistic programs, along with the learning and reasoning algorithms themselves.

1.3 Contributions

My main contribution is to lay out a program of research aimed at developing natively probabilistic computers, built around distributions and samplers rather than functions and evaluators. The specific technical contributions I make in support of this research program include:

1. Church, a probabilistic programming language for describing probabilistic generative pro-

[3]Software running directly on the Scheme Chip probably came the closest to conforming to this stack, which omits many the complexities associated with the economics of computing (including operating systems and virtualization in both software and hardware). Instead, we are focusing only on the layers that are conceptually necessary for computation in support of data analysis and intelligent behavior.




```
(query (list (flip) (flip) (flip))
       (lambda (flips)
          (= (sum flips) 2)))
```

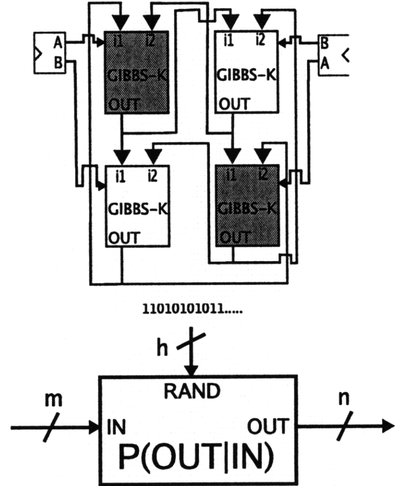


Figure 1-4: A stack of abstractions for natively probabilistic computing. The stack builds up from probabilistic circuits, to massively parallel, fault-tolerant stochastic finite state machines, to probabilistic programs. Each layer is based on *distributions* and *samplers*, recovering the corresponding layer in Figure 1-3 as a special case based on functions.

cesses that induce distributions. Church generalizes Lisp, a language for describing deterministic procedures that induce functions. I highlight some of the ways stochasticity meshes with the reflectiveness of Lisp to support the representation of structured, uncertain knowledge, including nonparametric Bayesian models from the current literature, programs for decision making under uncertainty, and programs that learn very simple programs from data.

2. Systematic stochastic search, a recursive algorithm for exact and approximate sampling that generalizes a popular form of backtracking search to the broader setting of stochastic simulation and recovers widely used particle filters as a special case. I use it to solve probabilistic reasoning problems from statistical physics, causal reasoning and stereo vision.

3. Stochastic digital circuits that are universal for the probability algebra just as traditional Boolean circuits are universal for the Boolean algebra. I show how these circuits can be used to build massively parallel, low precision, fault-tolerant machines for sampling and allow

one to efficiently run Markov chain Monte Carlo methods like Gibbs sampling on models with hundreds of thousands of variables in real time.

We have a long way to go before the vision of natively probabilistic computation is realized. I devote one chapter of this dissertation to a brief discussion of some of the remaining challenges, which center on the problems of probabilistic computer architecture, probabilistic compilation (to bridge the gap between probabilistic programs and either reconfigurable arrays of probabilistic circuits or general-purpose probabilistic machines), and probabilistic computational complexity.

However, I hope this dissertation shows that it is possible to construct the main layers of the stack today and understand conceptually how they fit together. I also hope it provides a convincing argument that, given significant problem specific effort and some tool support, it is possible to carry simple probabilistic reasoning problems involving probabilistic graphical models all the way down from probabilistic programs to probabilistic circuits. Figure 1-5 contains a different, slightly more detailed view of some of these contributions.

Along the way, we will note lessons about knowledge representation and computation stemming from this natively probabilistic view, summarized here:

1. To write down and efficiently manipulate large probabilistic structures, we should focus on building *samplers*, rather than functions that compute probabilities.

2. Probabilistic programs can be good vehicles for describing general-purpose models, unlike deterministic programs, which are best for describing specific reasoning algorithms.

3. Algorithmic ideas like higher-order procedures, memoization and interpretation can clarify and generalize widely used probabilistic modeling ideas, like clustering and sequence/time-series modeling.

4. Using sampling, it is possible to build reasoning algorithms that perform well on both soft, statistical problems and highly constrained, logical reasoning problems. We should try to find algorithms that work well in both cases, automatically adapting their behavior as appropriate, rather than viewing them as separate domains subject to separate techniques.

5. We can build useful stochastic algorithms for exploring large state spaces that leverages, rather than sidesteps, the rich body of work in discrete algorithms and combinatorial optimization, including recursion and backtracking.

6. Probabilistic procedures and sampling algorithms — and not just Monte Carlo estimators built out of them — can often be massively parallelizable, due to the presence of conditional independencies and exchangeability. Unlike in the deterministic setting, this fine-grained parallelism is often easy to identify, as representations of distributions often make the conditional independencies explicit.

7. Many sampling algorithms naturally require very low bit precision, can be implemented without large, slow floating-point circuitry. Thus accurate probabilistic computation does not require precise probability calculation.

8. Reconfigurable computing, where instead of reprogramming a processor one rewires a circuit, becomes an appealing strategy in the probabilistic setting, due to massive, exploitable concurrency and low bit precision.

9. Probabilistic machines can be far more physically robust to faults than their deterministic counterparts, because they are not expected to perform identically every time. They could thus be implemented on noisy, unreliable substrates.

1.3.1 Moral: Don't calculate probabilities; sample good guesses.

There is a simple way to sum up these lessons. Our most powerful computers are well suited for logical deduction and high-precision arithmetic, capable of performing billions of elementary Boolean inferences and floating-point operations per second. However, they are stymied by problems of uncertain reasoning and inductive learning over just tens of variables, despite the evidence that vastly more difficult probabilistic inference problems are effortlessly and unconsciously solved by the mind and brain. This is especially problematic given our increasing dependence on probabilistic inference across computational science and large-scale data analysis, where we would like to bring rich models of the world in contact with data.

By focusing our efforts on sampling, we may be able to avoid the worst computational bottlenecks associated with probabilistic reasoning. We may also be naturally led to emphasize the qualitative predictions made by our models, in terms of their typical simulated behavior, rather than insist on a quantitative precision that has often proved elusive (and, at least sometimes, excessive) outside of accounting, astronomy and quantum physics. The greatest potential for this approach, however, may be in AI. We can try to build agents that sample good guesses rather than calculate precise probabilities and reliably choose satisfactory actions rather than struggle to find guaranteed

optimal ones. By giving up precision and guaranteed optimality, we will be trying to gain robustness and flexibility. In this way, I hope that natively probabilistic computation will allow us to take a small step closer towards the goal of constructing a thinking machine.

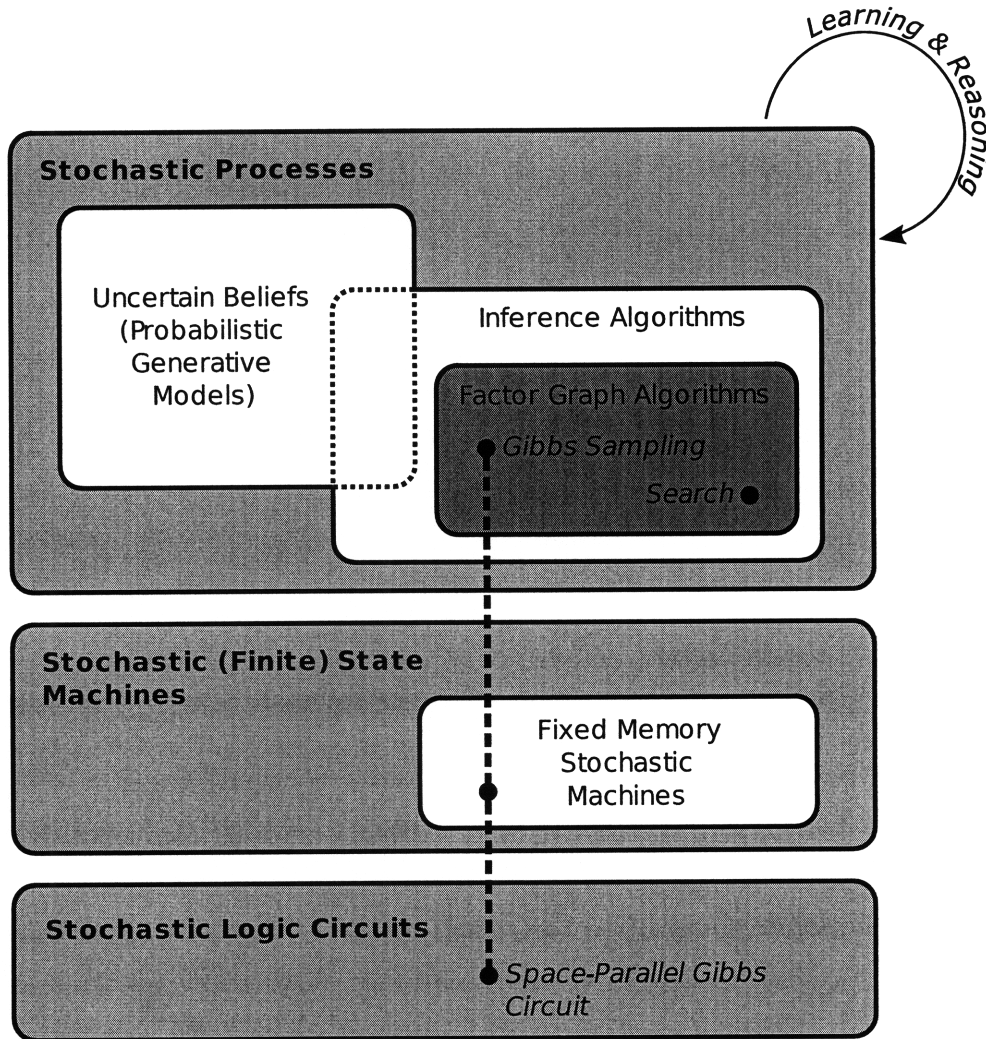


Figure 1-5: Probabilistic programming languages like Church, which describe stochastic processes or probabilistic generative processes, can be used to represent both uncertain beliefs and useful algorithms in a single notation. Learning and reasoning will involve executing probabilistic programs and learning probabilistic programs from data, using general machinery. For the subset of models described by factor graphs, I show how to solve them using a new, recursive sampling algorithm, as well as generate static-memory stochastic state machines for solving them, that can be implemented via massively parallel stochastic circuits.

Chapter 2

Church: a universal language for probabilistic generative models

> "The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is the emergence of what might best be called procedural epistemology – the study of the structure of knowledge from an imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects. Mathematics provides a framework for dealing precisely with notions of 'what is.' Computation provides a framework for dealing precisely with notions of 'how to.' "
>
> – Sussman & Abelson, *The Structure and Interpretation of Computer Programs*

> "93. When someone says 'I want a programming language in which I need only say what I wish done,' give him a lollipop."
>
> – Alan Perlis, *Epigrams on Programming*

2.1 Introduction

The introduction of probabilistic modeling has given rise to an extensive body of work in machine learning, statistics, robotics, vision, biology, neuroscience, AI and cognitive science. However, many of the most innovative and useful probabilistic models currently being developed far outstrip the representational capacity of our most popular formalism - probabilistic graphical models - and are outside of the scope of the associated inference techniques. They also bear very little resemblance to the linear regression models that have historically been most popular in statistics

and data mining. Consider the following recent examples from artificial intelligence and cognitive science:

- To model biochemical regulatory networks inside cells, Freidman, Koller and colleagues (23; 24; 21) have introduced models that learn simple probabilistic graphical models from data, with complex, biologically informed schemes for capturing higher-order structure in the network fragments (67).
- To model scientific inferences such as Mendeleev's discovery of the periodic table, Kemp and Tenenbaum (41) employed a joint probability model over graph grammars, structures generated by those grammars, diffusion processes over those structures, and data generated by those processes. They used probabilistic inference over this space to learn forms and structures from data, automatically discovering whether the data was best organized in terms of clusters, a line, a low-dimensional space, a ring, a chain or a tree.
- To model the extraction of structured visual percepts from images, Zhu et al have used probability models over parses of color images into regions, brush strokes, and objects (92; 83), building up to variants of Marr's primal sketch (47; 31). Probabilistic inference in these models can simultaneously segment an image, detect faces, and parse text, as well as hallucinate completions for obscured regions.

Due to the lack of formal support, these models are communicated using a mix of natural language, pseudo code, and mathematical formulae and solved using special purpose, one-off inference methods. Rather than precise specifications suitable for automatic inference, graphical models typically serve as coarse, high-level descriptions, eliding critical aspects such as fine-grained independence, abstraction and recursion.

The lack of good languages has also limited the complexity and quality of the models we have been willing to entertain. This is especially apparent in problems of metareasoning, or reasoning about reasoning. If we do not have a single representation language that describe both what an agent believes and how it reasons, it will be hard to build agents that have beliefs about the reasoning processes of other agents. We would like it if we had a language which also could represent desires and plans, without requiring us to reinvent basic concepts like procedures and evaluation each time they are needed in these different settings.

Church is a probabilistic programming language motivated by these problems. It provides a universal notation for probabilistic generative processes and the distributions that they induce.

	Structured Formalism	Model Classes	Languages & Systems
	Atomic world-states	Probability theory	
I	Finite graphs over variables	Probabilistic graphical models	BUGS (45), BNT (52), HBC (13)
II	Relational Databases	RPMs (56), PRMs (22)	
II	First-order logic, closed world	Markov (Weighted) Logic (63)	Alchemy
II	First-order logic, open world	BLOG models (51)	BLOG in Java, PyBLOG
III	Programming languages	Computable probability models	Church (30), IBAL (59), Infer.NET/CSoft (89), PTP (55) ...

Table 2.1: An overview of some of the mathematical and computational formalisms for uncertain knowledge representation. Modern systems have come in three generations, using graph theory, logic, and programming languages to manage structure and probability to manage uncertainty.

Processes can be simulated, as in a normal programming language. They can also be viewed as generative models and conditioned on. The main idea behind Church is to represent a probabilistic generative process in terms of a procedure that makes stochastic choices, where executing that procedure simulates the process and samples a value from the distribution on values it induces. Church generalizes the pure, or side-effect free, subset of Scheme, a dialect of Lisp used to describe deterministic processes that induce deterministic functions.

In this chapter, we will focus on the ways Church relates to previous functional programming languages and probabilistic modeling languages, paying particular attention to the advantages gained by descending from Lisp. We will see how to write modern nonparametric, hierarchical probabilistic models in Church by marrying memoization with the Dirichlet process and using higher-order procedures like `UNFOLD`. We will also take a look at new kinds of models enabled by Church, including universal planning and inductive probabilistic programming, as well as the general shape of a Church implementation. We will close with the perspective probabilistic programming provides on old issues like programming style and program synthesis.

2.1.1 Languages for Knowledge Representation

As the popularity of probabilistic modeling has grown, there have been a series of efforts to develop mathematical formalisms for describing complex models and software systems for automating aspects of modeling and inference. These can be grouped into three generations.

The first generation of probabilistic modeling tools focused on probabilistic graphical models, based on directed and undirected graphs of variables. The central idea was to build big distributions out of pieces by leveraging conditional independence, translating conditional independencies into

decompositions of the joint probability distribution (for directed models) and the unnormalized joint probability distribution (for undirected models). In the process, we learned how to formalize aspects of causality in probabilistic terms (57), as well as how to connect the probabilistic models and inference algorithms in use in AI and statistics with models and algorithms from computational physics (90; 42). Software tools for graphical modeling came from the Bayesian statistics (45) and machine learning communities. The simplest graphical models, known as Bayesian networks, have even seen commercial development as part of probabilistic expert systems for decision support.

The second generation was motivated by the desire to build probability models over sets of objects and structured relationships between them. These approaches use ideas from first-order logic (and closed-world relatives, such as relational databases) to represent distributions on worlds with objects and relations without decomposing them into elementary random variables.

One popular approach, typified by Markov Logic, involves using weights to directly induce a distribution over the models of a theory written in first-order logic. he probability distribution on worlds comes from renormalizing the weight function over the set of admissible models. This is guaranteed to be well defined when there is a finite set of models, for example when all domains have a finite number of entities, and parallels undirected graphical modeling. Another approach, typified by probabilistic relational models or PRMS, involves extending Bayesian networks to the setting of relational databases, by developing a template language for Bayesian networks based on the schema of a relational database, and leveraging this structured template for learning and reasoning. BLOG extends this idea to the setting of worlds with unknown numbers of objects and uncertain identity[1].

The third generation of modeling languages are based on combinations of probability and programming languages. The key idea is that a program which makes probabilistic choices induces a distribution on outputs. Thus programs can represent probabilistic models in terms of the probabilistic generative processes that generate samples from them. Put differently, in probabilistic programming, one writes a program which hallucinates possible worlds with probabilities that follow a desired probabilistic model. All the typical tools progamming languages provide to control the complexity of large knowledge structures become available in probabilistic modeling. For example, IBAL modelers can use the ML typechecker to help debug their constructions, and exploit algebraic data types to simplify the construction of combinatorially defined models. The underly-

[1] I personally prefer to interpret BLOG as a kind of nontraditional probabilistic programming language, since the immediate semantics of a BLOG model is in terms of a generative process which happens to induce a distribution on models of a logic, but it seems the inventors of BLOG prefer to emphasize its declarative, logical aspects.

ing programming language and the approach to integrating inference provide the main axes along which probabilistic programming languages vary. For example, IBAL and Infer.NET/CSoft both use inference machinery that grounds out in finite graphs, and therefore do not support random choices within recursive procedures.

2.1.2 Why Build on Lisp?

> "Programming languages should be designed not by piling feature on top of feature, but by moving the weaknesses and restrictions that make additional features appear necessary."

– Rees & Clinger (eds), *Revised Revised Revised Report on the Algorithmic Language Scheme*

> "Scheme is an especially good vehicle for exhibiting the power of procedural abstractions because [...] Scheme does not distinguish between patterns that abstract over procedures and patterns that abstract over other kinds of data."

– Abelson & Sussman, *Lisp: a language for stratified design*

The essence of probabilistic generative modeling is its focus on processes as mechanisms for representing uncertain knowledge. The essence of probabilistic programming is its use of formal languages for describing processes, in terms of procedures that generate them. Our ability to flexibility manipulate and abstract over procedures in the deterministic limit will bound our ability to flexibly manipulate and abstract over probabilistic generative models. Thus Scheme, with its support for anonymous, untyped, higher-order procedures and the dynamic, procedure-based programming style it supports, was a natural starting point.

The simplicity and elegance of the Scheme evaluator, and the explicitness of the substitution and environment models of Scheme evaluation (1), were additional factors in our choice. Scheme's `EVAL` has only a small number of cases, and a compact, recursive definition that respects the lexical structure of statements written in the language. Ultimately, this simplicity allowed us to develop both a clean metacircular implementation of inference - which, almost magically, interacted correctly with the introduction of controlled forms of mutation into the language - as well as come up with machinery for inverting Church programs that handle random procedures and arbitrary probabilistic recursions.

Lisps are rare among functional languages in that they tend to be strongly but dynamically typed - that is, values can be thought of as having types, but symbols are not restricted according

to type and a program need not pass a type-checking step to be viewed as valid. This means it is feasible to expose procedures for evaluation - and, as we will see, its probabilistic generalizations to simulation and conditioning - within the language, without creating thorny typing issues. This will allow us to remove unnecessary distinctions between "learning" and "inference", and thereby enable a variety of interesting kinds of models for metareasoning.

In addition to these technical considerations, some aesthetic considerations contributed to our choice to work with Lisp. Although these are somewhat intangible, I feel that in the long run they will prove to be the most important differentiators of Church, at least in artificial intelligence.

We would like to build the knowledge structures in our agents using the tools of stratified design, organizing our agents' knowledge in terms of layers of conceptual language that moves from the general needs of probabilistic knowledge representation to the special needs of knowledge representation in particular domains. For example, we might like to develop a language for worlds with objects that have properties, say, corresponding to the modeling style encouraged by BLOG. We might also like a more specialized language for describing systems of physical objects, including balls and blocks, that carry with them knowledge from intuitive physics, such as persistence and continuity of motion. Furthermore, we would like our agents to be able to *learn* these languages from experience, along with particular programs written in these languages.

We thus need to work with a language that encourages the definition of sublanguages within it, burdening the programmer - and, ultimately, the learner - with a minimum of overhead. This stratified style of programming is the essence of the Lisp approach, supported by both its expressive constructs for procedural abstraction and the "code is data" maxim of Lisp. As we want to be able to someday learn not just probabilistic programs but domain-specific probabilistic programming languages from data, we want to be sure that we can at least manually nest languages within languages without needing to re-invent machinery at each stage. For example, it should be possible to compactly write the primitives of the language, including those involved in interpretation of the language and reasoning over programs written in the language, in the language itself. Such primitives, once defined, should be indistinguishable from the built-in versions that have been provided, and freely interoperate with them. This reflectiveness lies at the heart of the Scheme design philosophy.

2.2 A Brief Tour of Church

Church is based on a call-by-value Scheme with first-class environments and no mutation. In this chapter, I assume familiarity with Scheme, and introduce Church by example.

The first way in which Church departs from Scheme is in its introduction of random primitive procedures or *ERP*s. `FLIP` is the only logically necessary one, which generates a sample from a Bernoulli distribution, with a default coin weight of 0.5. Technically, the contract that a random primitive procedure must satisfy is that each application generates a fresh sample from some *exchangeable sequence*; we will return to this later in the chapter. For the moment, imagining ERPs to introduce independent, identically distributed randomness won't be too misleading, and will in fact be a useful special case.

2.2.1 Stochasticity and Call-by-Value

In Scheme, `(EVAL <expression> <environment>)` returns the value of the expression in the given environment. In Church, every expression-environment pair induces a *distribution* on values, where the ERPs introduce the randomness. That is, each expression defines a conditional probability distribution on results, which must be conditioned on a set of symbol bindings accessed through an environment structure. While expressions which involve no ERPs can be evaluated as regular Scheme expressions, we must decide how to handle expressions containing ERPs.

We make the choice that *evaluting an expression in an environment draws a fresh sample from its induced distribution on values*. Evaluating a Church expression produces a definite value, stochastically drawn from its induced distribution on values in the current environment. This choice composes naturally with the Scheme call-by-value rule for procedure application, where one evaluates the operands and operator of a compound expression in arbitrary order, then applies the value of the operator to the values of the operands. The net effect is that random worlds can be built up in environments, by binding sampled values to symbols. If one wants to bind a distribution, one instead binds a procedure - representing a potentially probabilistic generative process - that generates a sample from it.

Other languages, including IBAL, have chosen differently, reflecting the idea that they are meant to be *embedded* within deterministic languages, rather than probabilistic generalizations of them. The other main approach is to view the "value" of an expression with randomness in it as a "distribution object" represented by a table of values with probabilities. For example, `(flip)` in IBAL might deterministically evaluate to `((#t 0.5) (#f 0.5))`, making probabilistic pro-

```
((lambda (x) (+ x x)) (flip))
> 0, 2, 0, 0, 2, ...

(= x x)
> #t, #t, #t, #t, #t, #t, ...

(= (+ x y z) (+ x y z))
> #t, #t, #t, #t, #t, #t, ...

(= (flip) (flip))
> #t, #t, #f, #f, #f, #t, ...
```

Figure 2-1: Inputs and repeated outputs from a Church machine session showing the interaction of call-by-value semantics with stochasticity. The first example shows interference stemming from binding at lambda application. The second and third examples show how random worlds can be built using binding, where the rules of logic apply to the values of expressions that only involve deterministic procedures and symbols. The fourth example shows what happens when trying to compare the invocation of two coin flips. We argue this is appropriate behavior for a knowledge representation system. `(x)` denotes the result of simulating the (potentially stochastic) process named `x`, where `x` denotes the fixed value of the symbol `x`.

gramming into a kind of weighted variant of nondeterministic programming. Of course, this approach leads to all the difficulties with recursion and complexity mentioned in the introduction, as the size of these probability tables grows exponentially (or infinitely, for certain stochastic recursions). Replacing these tables with functions that compute probabilities is no better, as these functions then can easily require exponential (or infinite) time to evaluate.

The choice to consistently generalize evaluation to stochastic simulation is one of the distinguishing features of Church. Interaction with a Church machine is done through a REPL, or Read-Eval-Print Loop, where Church expressions are entered, parsed, and evaluated in the top-level environment. The behavior of such a machine is indistinguishable from (and can be implemented by) the results of calling `(EVAL <parsed-expression> *global-environment*))`. Figure 2-1 shows example interactions which illustrate the interaction of stochasticity and call-by-value semantics.

Probabilistic purity admits exchangeable mutation

Some deterministic functional programs are *pure*: if I evaluate an expression in an environment multiple times, I always get the same value. More formally, the sequence of values obtained by

evaluating an expression in an environment is a constant sequence. A language is pure if it only permits pure programs to be written. For example, a minimal Scheme without `DEFINE` and `SET!` is pure. If I write some function `(foo x)` in such a pure Scheme and evaluate `(foo 3)` twice, I will get the same value back both times. The procedure `foo`, like all pairs of an expression and an environment, is guaranteed to represent a well-defined deterministic function as a consequence of purity.

Church programs that involve random primitives like `flip` are, clearly, not pure. If I evaluate `(flip)` twice in some environment, I will get the same answer only with probability 0.5. This is a direct consequence of evaluation as stochastic simulation. However, `flip` induces a well-defined distribution on return values: each draw is taken independently from some Bernoulli distribution.

We might think that this defines the natural probabilistic notion of purity to be "evaluating an expression in an environment multiple times results in a sequence of independent, identically distributed random variables". However, it turns out this is too restrictive. Consider the program in Figure 2-2, which describes a pure procedure `sample-coin` that generates pure procedures. If we have our hands on one such returned procedure, `my-coin`, and apply it repeatedly, we will be drawing from some Bernoulli distribution whose coin-weight is hidden (or closed over) in the environment associated with `my-coin`.

The sequence of values corresponding to calls to `my-coin` is not independent and identically distributed, conditioned only the contents of the calling environment. Intuitively, knowledge that the first 10 values are `#t`, for example, makes it seem more likely that the hidden coin weight is high so next value will probably be `#t`. Of course, conditioned on the contents of the closure inside `my-coin`, the sequences of values is actually IID. If we chose to define probabilistic purity by requiring the sequence resulting from repeated evaluation of expressions to be IID, we would have to view call-by-value languages as impure, since simple binding of randomness inside a procedure can result in dependencies.

In fact, we can implement `sample-coin` differently, in terms of mutation. This implementation is shown in Figure 2-3. A caller unable to peer inside procedures would be unable to distinguish between these two implementations. Both generate procedures that produce individual samples from a sequence of coin flips $p(x_1, ..., x_n)$. The version with mutation never generates an independent, identically distributed sequence. We can analyze the resulting distribution on values in terms of two different decompositions. In the first case, we sample a coin weight θ and bind it in the environment. To a caller of the returned procedure, this coin weight is integrated out, and so

we have

$$p(x_1, \cdots, x_n) = \int d\theta p(\theta) \prod_i p(x_i|\theta)$$

In the new version with mutation, we sample directly from a sequence of predictive distributions, where each new coin flip is conditioned on the particular history of flips that have happened previously:

$$p(x_1, \cdots, x_n) = p(x_1)p(x_2|x_1) \cdots p(x_n|x_1, \cdots, x_{n-1})$$

To advance along the sequence of predictive distributions each time the returned procedure is called, we must remember the statistics of past coin flips and update them on each flip using mutation.

We thus suggest that the right generalization of purity to the probabilistic setting is "the sequence of values obtained by evaluating an expression in an environment is exchangeable". A sequence of random variables is called *exchangeable* if the joint distribution on the variables is invariant to the ordering in the sequence. For finite sequences $X_1 \cdots X_N$ and permutations σ, we can write this condition as:

$$Pr[X_1 = x_1, \cdots, X_N = x_N] = Pr[X_{\sigma(1)} = x_{\sigma(1)}, \cdots, X_{\sigma(N)} = x_{\sigma(N)}]$$

Exchangeability intuitively corresponds to a commonly valued *consequence* of functional purity, namely that the order of evaluation of a sequence does not matter. As long as this exchangeability property holds, there are a family of representation theorems, commonly referred to as "De Finetti's Theorem" (14), that guarantee (subject to various technical conditions) a representation exists in terms of independent, identically distributed draws given some latent random variable. In fact, Freer and Roy (64) have shown that a large class of computable exchangeable sequences have a computable representation in the style of the first version of `sample-coin`. Thus requiring exchangeability of evaluation sequences corresponds to the idea that some distribution is being drawn from, without requiring that distribution to be made explicit, and thus allowing random values to be bound in environments. Mutation is permitted, as long as it preserves independence to order of evaluation (and therefore does not generate a system that counts).

Memoization

Church includes a primitive procedure `(mem <proc>)`, which takes a procedure and returns a memoized version — one that only applies itself once for each set of arguments. In the determin-

```
(define (sample-coin)
  (let ((coin-weight (random-beta 1 1)))
    (lambda () (flip coin-weight))))

(define my-coin (sample-coin))
(my-coin)
> #t
(my-coin)
> #f
(define your-coin (sample-coin))
(your-coin)
> #t
```

Figure 2-2: Expressing the Beta-Bernoulli model as a pure thunk valued procedure. To construct a coin, we return a procedure that closes over a single randomly chosen coin weight. Calls to such procedures are *exchangeable*, not independent and identically distributed: the order of the calls doesn't matter, but knowledge of one return value does change our expectations about other return values. However, if we can look inside the closed-over environment, we see the representation in terms of independent, identically distributed draws.

istic setting, this trades space (to store old values) for time (as each value is only computed once). In the probabilistic setting, memoizing a procedure in general changes its meaning. For example, Figure 2-4 shows a simple example where memoization is used to build infinite random streams. We will see other uses of memoization later on in this chapter.

Memoization requires some form of mutation — or an unpleasant, whole-program transformation, prior to interpretation — to implement. However, it is clearly exchangeable: the joint distribution on simulated values of a memoized procedure does not depend on the order in which they are drawn. This allows memoization to be treated as an ordinary random primitive procedure. We think other impure higher-order functions and linguistic constructs that seem to preserve the identity of the procedures and programs they act on may be helpfully viewed as generating exchangeable sequences. Siskind and Pearlmutter's differential operator (73; 58) is one interesting candidate. Lisp's `gensym` is another: the meaning of a program with `gensym` does not depend on the values it returns, but only on the property that they be distinct, which is invariant to their ordering. Thus we see that exchangeability recovers an intermediate point between purity and mutation in the deterministic limit, while serving as the natural notion of purity in the probabilistic setting.

```
(define (sample-coin)
  (let ((counts (list 1 1)))
    (lambda ()
      (let ((result (flip (/ (car counts) (+ (car counts) (cdr counts))))))
         (if result (set-car! counts (+ (car counts) 1))
                    (set-cdr! counts (+ (cdr counts) 1)))
         result))))

(define my-coin (sample-coin)
(my-coin)
> #t
(my-coin)
> #f
(define your-coin (sample-coin))
(your-coin)
> #t
```

Figure 2-3: Mutation is allowable as long as it preserves exchangeability. Using this idiom, we give an alternate representation for the Beta-Bernoulli from Figure 2-2, which operates by sampling from a sequence of predictive distributions, updating sufficient statistics after each call.

2.2.2 Universal Bayesian Inference and Conditional Simulation

We have seen how it is natural to generalize the notion of executing processes that evaluate functions to the idea of simulating probabilistic generative processes that sample from distributions. This allows us to write programs that correspond to probabilistic generative processes, giving us a new, procedural notation for distributions in probabilistic modeling. To have an effective modeling language, we need to somehow support inference, which involves *conditioning* the model.

In Church, we support inference by generalizing evaluation further, to *conditional simulation* of any probabilistic generative process, where both the process and the condition can be arbitrary probabilistic procedures. Rather than organize computation around the procedure `(EVAL <expression> <environment>)`, we organize computation around the procedure `(QUERY <expression> <environment> <predicate>)`. The job of `QUERY` is to simulate from the induced distribution on values for the given expression in the given environment when conditioned on the predicate being true on its output.

To understand this choice, it is worth contrasting it with the prototypical view of Bayesian inference. In a typical Bayesian problem, one has a prior probability distribution $Pr(H = h)$ on

```
(define notastream (lambda (idx) (flip)))
(define bitstream  (mem (lambda (idx) (flip))))

(notastream 0)
> #t
(notastream 1)
> #t
(notastream 0)
> #f
(notastream 0)
> #t

(bitstream 0)
> #f
(bitstream 1)
> #t
(bitstream 0)
> #f
(bitstream 0)
> #f
```

Figure 2-4: Memoization, an exchangeable operation, can be used to construct infinite random objects whose pieces are indexed by the arguments to procedures, such as streams of random bits. This permits delayed evaluation in the probabilistic setting.

hypotheses h, capturing beliefs before a given set of observations will be taken into account. One also has a data model $Pr(D = d|H = h)$, that specifies the probability of any given data set d assuming that the hypothesis h holds. Often, both these probability functions are individually easy to evaluate, and are induced by complex, probabilistic generative processes. Bayes' Rule, or really some applications of the rule of conditional probability, is then used to describe the posterior distribution $Pr(H = h|D = d)$:

$$Pr(H = h|D = d) = \frac{Pr(H = h)Pr(D = d|H = h)}{Pr(D = d)} = \frac{Pr(H = h, D = d)}{\sum_H Pr(H = h, D = d)}$$

This new object is a conditional distribution on hypotheses h given data sets d. Its probability function is usually difficult to evaluate, because it requires a sum over exponentially many hypotheses h. It is also a "large" object, and difficult to manipulate even if it were easy to evaluate for any

given pair of d and h. That is, being able to evaluate the posterior probability would not necessarily make it easy to evaluate posterior expectations, examine typical world states responsible for the data, or make predictions about future data sets. This complexity issue is just a special case of the complexity issues involved in all probability function based representations for distributions.

In Church, one way to specify the canonical Bayesian reasoning setup is by providing a procedure that takes a prior and a likelihood, both represented as samplers, and produces a procedure that samples from the posterior whenever applied:

```
(define (make-posterior-sampler prior-sampler likelihood-sampler)
  (lambda (observed-data)
    (query '(let ((H (prior-sampler))
                  (D (likelihood-sampler H)))
              (cons H D))
           (get-current-environment)
           (lambda (HD)
             (equal? (cdr HD) observed-data)))))
```

The following properties of Church's `QUERY` construct stem from its Lispiness and distinguish it from most other probabilistic programming langauges:

1. It recovers `EVAL` when `pred` is always true (for example, `(lambda (x) #t)`). Thus it contains unconditional simulation as a special case, respecting environment structure, and can be applied recursively to conditionally simulate from a conditionally simulated distribution.

2. It can be implemented exactly, even when the space of possible causal histories associated with `exp` is exponentially large or infinite, as the process of simulating `exp` halts with probability 1. This exactness in the presence of an infinite support is hard to archieve with any description of a distribution that is not based on probabilistic procedures that produce samples.

3. It permits arbitrary, probabilistically computable predicates, which themselves might include randomness (arising, for example, when representing a collapsed rejection sampling algorithm).

```
(define (query exp env pred)
  (let ((val (eval exp env)))
    (if (pred val)
        val
        (query exp env pred))))
```

Figure 2-5: A metacircular description of QUERY, which generalizes EVAL to provide lexically-scoped *conditional* simulation of an admissible stochastic process. QUERY takes an expression, an environment and a predicate, and returns a sampled value from the *conditional* distribution on values induced by the expression, given the constraint that the predicate applied to that value is true.

4. As we will see in the next section, QUERY can be written within the language, and a native implementation is indistinguishable from the built-in one (due to first-class environments). Thus it demonstrates universality of Church for conditional simulation: in Church, you can write a reasoner that will halt with probability 1 for all queries where the expression and predicate both halt with probability 1.

Metacircular Implementation

We provide a metacircular implementation of QUERY in Figure 2-5, building on the well-known metacircular implementation of EVAL in pure Scheme (1). We use rejection sampling, a version of guess and check: simulate the underlying process, and accept the results if and only if the target condition is met, retrying as many times as necessary.

We note that when designing Church, our original notion of conditional simulation was not compatible with metacircular interpretation, nor did it correctly recover evaluation as a special case. We did not arrive at a version of QUERY which correctly functioned outside of the top-level — removing arbitrary restrictions on when and where probabilistic modeling and inference, as opposed to just forward simulation, could occur — until we were able to provide a definitional interpreter for Church's QUERY (62). This admittedly abstract constraint yielded a *definition* of query in terms of rejection sampling that has repeatedly held up to subtle extensions to the language, including the introduction of random primitive procedures that include exchangeable mutation. While we do not fully understand the reason that preserving metacircularity has led to such stability — avoiding errors in the conditioning constructs in some other probabilistic programming languages — we have found its results indispensable.

Efficient algorithms for conditional simulation have been the subject of a vast literature. This

rejection sampling based implementation has the property that it will only be efficient — in both time and number of attempted samples — when the conditioning operation does not change the distribution "too much", so most samples are accepted. In the special case of the standard Bayesian model, this means the prior and posterior are relatively similar. We will touch on approximation algorithms with different efficiency properties later on. We emphasize the rejection sampler, however, because it defines the desired behavior for query in a stable and procedural way, and forces us to develop algorithms for exact and approximate reasoning that are appropriately generable and recursively composable.

Stochasticity, not nondeterminism

The idea of using a program to represent the models of some declarative theory has a long history in logic, going back to McCarthy's AMB construct (1; 49; 72). The idea behind AMB is to represent an ambiguous value via a nondeterministic choice, subject to Boolean constraints, induced via the construct REQUIRE. A search algorithm operating behind the scenes is responsible for finding answers. For example, this nondeterministic program represents a SAT problem:

```
(let ((a (amb #t #f))
      (b (amb #t #f))
      (c (amb #t #f)))
  (require (or (and a (not b)) c))
  (list a b c))
```

Implementations of systems for so-called "nondeterministic" programming also frequently provide procedures like (ALL-SOLUTIONS <expr>), which list all possible values of an ambiguous expression. This same basic approach can be used to understand reasoning systems like Prolog, and procedural embeddings of a range of logics.

In Church, we might encode the related but different (!) problem of sampling a satisfying assignment for a Boolean formula as follows, using a simple lexicalized variant of QUERY that takes a let-style named list of variables, an expression to evaluate against these names to return, and a predicate expression:

```
(lex-query '((a (flip))
             (b (flip))
             (c (flip)))
```

```
'(list a b c)
'(or (and a (not b)) c))
```

Church and amb share the idea of capturing a space of possible outcomes using programs with some kind of choice operator and a predicate written as an arbitrary Boolean-valued expression. To convert Church into a language for nondeterministic programming we need to convert stochastic choices into nondeterministic ones and stochastic conditions into satisfiability or counting problems. For example, one could imagine first replacing (flip) with (amb #t #f), and then replacing (query <exp> <pred> <env>) with two procedures, (some-satisfying-value <exp> <pred> <env>) and (all-satisfying-values <exp> <pred> <env>).

However, Church is *stochastic*, capturing models of *probabilistic* knowledge, where amb is *nondeterministic*, capturing models of *logical* knowledge. This means, for example, that Church programs can potentially manage exceptions and weight alternate explanations, where a logical reasoner can only report the presence of perfectly consistent solutions or failure. Furthermore, because we sample, although the space of possible solutions grows exponentially in the number of stochastic choices (as with AMB), the cost of running the program forward to produce a typical solution grows only linearly. Of course, the complexity of adding conditions is more complicated and currently only poorly understood.

An Ergodic Walk On Execution Histories of a Lisp Machine

Another advantage of the QUERY construct is that it provides a uniform target for the design of universal approximation algorithms. Several such algorithms have been developed; here, I present the simplest one based on Markov chain Monte Carlo methods, to give a flavor for the general problem. A description of more advanced methods and associated experiments is left for future work.

The idea behind MCMC methods is to construct a sample approximately according to some distribution of interest by inventing a Markov chain whose long run distribution is the target and which is itself cheap to simulate. One then iterates this chain repeatedly to obtain the desired answer. One approach to inference in Church, then, is to define a state space and a random walk over this space such that the long run distribution is the conditional distribution on values implied by some query.

The state space we choose is the space of *computation traces* of a Church program. A computation trace is a directed acyclic graph composed of two directed trees, one tree for the envi-

ronment structure of the program and one tree for the reduction steps involved in simulating each subexpression. It makes the dependency structure of the computation — including its deterministic subparts — completely explicit, and in particular records the dependencies between stochastic choices and their particular outcomes. Figure 2-6 shows an example schematic trace for the expression `((lambda (x) (+ x 1)) (flip))`, while Figure 2-7 shows a complete computation trace captured from an early implementation of this algorithm on a simple satisfiability problem.

A detailed discussion of our Metropolis-Hastings algorithm for approximating `QUERY` is beyond the scope of the dissertation. However, the basic idea (due to Noah Goodman) is to:

1. Initialize the trace by executing the query expression, recording the `EVAL` recursion and environment structure as it progresses.

2. Select a place in the trace where a random choice was made, uniformly at random. Re-evaluate the choice. Propagate changes in the values of subexpressions and of bound symbols along the trace in all directions, conservatively keeping it consistent, stopping when no more changes are necessary. If the predicate of a conditional is reached, produce a trace fragment for the newly taken branch and consume the old branch, keeping track of the probabilities of each.

3. Accept or reject the new proposed trace according to the Metropolis-Hastings rule, rejecting if the outermost predicate is violated, and thus guaranteeing asymptotic convergence. Periodically mix it with a proposal that re-evaluates the trace by running the program forward to ensure ergodicity.

Random choices can either correspond to random primitive procedures or to procedures that have been tagged with a means of computing the *marginal* probability of an outcome given the input. Other procedural tags that are sometimes useful include the ability to "undo" sampling a value (rolling back any internal, exchangeable state associated with that value, such as sufficient statistics) and alternate MCMC kernels for resampling the value (such as Gaussian perturbation kernels for continuous values). Taken together, these additions constitute the information needed to use a procedure in not just forward simulation but MCMC-based approximate conditional simulation, and allow the general framework described here to recover popular MCMC algorithms for specific models. Of course, Markov chain iteration is an inherently stateful process, and thus the Metropolis-Hastings algorithm cannot be straightforwardly implemented within Church without costly build up and repeated, recursive recopying of the entire history of the Markov chain.

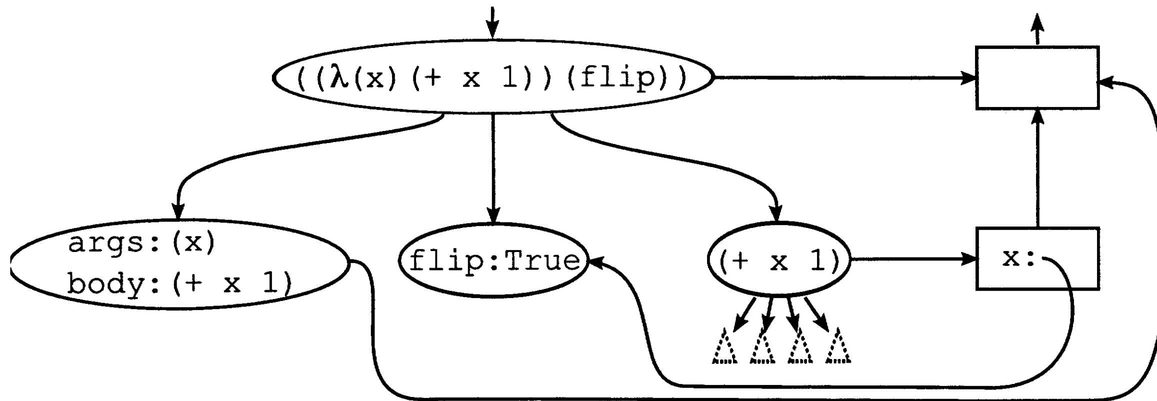


Figure 2-6: A schematic computation trace, showing the DAG obtained by connecting the tree of Church expressions and subexpressions with the tree of environments during the evaluation of the expression `((lambda (x) (+ x 1)) (flip))`.

This algorithm has the appealing property that the complexity of basic steps of reasoning — here, proposing new substructures in a trace, and updating a trace to reflect these proposals — follows directly from the complexity of simulating from subpieces of a structure of knowledge. It thus preserves the ability for a programmer — or an inference compiler, or uncertain reasoner — to rewrite a belief in a form that makes reasoning over it more efficient without changing its meaning and without leaving the basic Church language.

2.2.3 Expressiveness

The expressiveness of Church can be seen in the concise descriptions of recently introduced models that it enables as well as the ways in which it supports new kinds of models that were previously difficult to formally capture. In this chapter, I will focus on nonparametric Bayesian models, stochastic memoization, undirected models, inductive programming, and sequential decision making, since these exemplify the kinds of problems that Church can handle but other programs cannot.

Dirichlet Processes and Nonparametric Hierarchical Bayes

We start with the Dirichlet process, a popular object from the nonparametric Bayesian statistics literature, that is an infinite dimensional generalization of the Dirichlet distribution on the simplex. Figure 2-8 shows one implementation in Church.

The Dirichlet process maps a distribution onto a "clumpy" version of itself, where early draws are more likely to recur as the process is replayed. From this perspective, the Dirichlet Process can

Figure 2-7: A complete computation trace saved using the trace debugging facility of the Blaise implementation of Church.

```
(define (DP concentration base-measure)
  (let ((sticks (mem (lambda j (random-beta 1.0 concentration))))
        (atoms  (mem (lambda j (base-measure)))))
    (lambda ()
      (let loop ((j 1))
        (if (flip (sticks j))      ;; with probability (stick j)
            (atoms j)              ;;   return j'th sample from base measure
            (loop (+ j 1))))))))   ;; otherwise move to (j+1)'th stick
```

Figure 2-8: The Dirichlet Process can be naturally expressed as a higher-order Church procedure. Since its de Finetti or stick-breaking representation is infinite, we use MEM to delay the evaluation of the sticks until needed.

```
(define (DPmem alpha proc)
  (let ((restaurants (mem (lambda args (DP alpha
                                           (lambda () (apply proc args)))))))
    (lambda args ((apply restaurants args)) )))
```

Figure 2-9: Using the DP as a primitive, we can stochastically generalize memoization, yielding a version where a procedure is re-evaluated (possibly resulting in a new value, if the procedure is stochastic) with probability following the Chinese restaurant process each time it is called. With $\alpha = 0$ we recover deterministic memoization, where the procedure is only evaluated once, and with $\alpha = \infty$ we recover the absence of memoization (but wasting space to store the intermediate values). This idiom helps to explain the widespread popularity of the Dirichlet Process in probabilistic generative modeling, where it often serves the role of "stochastically caching" fragments of a generative process.

be usefully viewed as the key component of a kind of stochastic generalization of memoization, which yields procedures that sometimes return previously sampled values and sometimes sample new (and potentially different) ones. Figure 2-9 shows an example stochastic memoizer built using the Dirichlet process.

We can combine this procedural design idiom with ideas from functional programming to compactly express a range of widely used models based on stochastic transitions. First, we express the general process of unfolding, or building a structure recursively:

```
(define (unfold expander symbol)
  (if (terminal? symbol)
      symbol
      (map (lambda (x) (unfold expander x))
           (expander symbol) )))
```

Given this, we can build a Church model for PCFG transitions via a fixed multinomial over expansions for each symbol:

```
(define (PCFG-productions symbol)
  (cond ((eq? symbol 'S) (multinomial '((S a) (T a)) (0.2 0.8)))
        ((eq? symbol 'T) (multinomial '((T b) (a b)) (0.3 0.7))) ))

(define (sample-pcfg) (unfold PCFG-productions 'S))
```

The HDP-HMM (5; 81) uses memoized symbols for states and memoizes transitions. Fresh symbols are generated by the exchangeable (but stateful) primitive `gensym`, which returns distinct symbols on each call:

```
(define get-symbol (DPmem 1.0 gensym))
(define get-observation-model (mem (lambda (symbol) (make-100-sided-die)))
(define ihmm-transition (DPmem 1.0 (lambda (state)
                                     (if (flip) 'stop (get-symbol)) )))
(define (ihmm-expander symbol)
  (list ((get-observation-model symbol)) (ihmm-transition symbol)) )
(define (sample-ihmm) (unfold ihmm-expander 'S))
```

The HDP-PCFG (44), a nonparametric, recursive, probabilistic grammar, is also straightforward in Church, although it has no graphical model representation:

```
(define terms      '( a  b  c  d))
(define term-probs '(.1 .2 .2 .5))
(define rule-type (mem (lambda symbol)
                         (if (flip) 'terminal 'binary-production))
(define ipcfg-expander (DPmem 1.0 (lambda (symbol)
                                    (if (eq? (rule-type symbol) 'terminal)
                                        (multinomial terms term-probs)
                                        (list (get-symbol) (get-symbol)) )
(define (sample-ipcfg) (unfold ipcfg-expander 'S))
```

Making adapted versions of any of these models (38) only requires stochastically memoizing `unfold` (although there are some subtle issues here involving recursive, stochastically memoized functions that are beyond the scope of this dissertation):

```
(define adapted-unfold
  (DPmem 1.0 (lambda (expander symbol)
               (if (terminal? symbol)
                   symbol
                   (map (lambda (x) (adapted-unfold expander x))
                        (expander symbol)) ))))
```

In all these settings, we see how higher-order programming idioms from the functional world, like memoization and unfold, can be generalized probabilistically and used to create new models. A wide variety of nonparametric models for relational clustering are also naturally describable in Church using similar techniques.

Constraint Networks, Undirected Modeling and Integrable Functions

Probabilistic generative models can be used to capture systems of constraints as well as systems of directed (or causal) relationships. The dominant metaphor for such models comes from statistical mechanics: every configuration $\vec{x}$ of a system is assigned an "energy" $E(\vec{x})$, and the distribution on configurations is defined to be the Boltzmann distribution:

$$p(\vec{x}) = \frac{1}{Z} e^{-\frac{1}{T} E(\vec{x})}$$

Such "Gibbsian" systems favor configurations which have low energy. As the temperature T approaches 0, they place all their mass on configurations with the lowest energy, and as T approaches infinity, they place probability mass uniformly over their support.

A wide variety of "soft" or "probabilistic" constraint networks can be described using this formalism. Markov random fields or "undirected" graphical models (27; 42) form one important class, where nodes correspond to variables and the graph structure captures an additive decomposition of the energy function $E(\vec{x})$ in terms of factors on neighboring variables. Stereo vision problems provide one important example.

We can induce such systems using generative processes via the metaphor of thermodynamic equilibration: we build an ergodic Markov chain whose long run distribution follows $p(\vec{x})$ above. Thus one option for representing constraint systems in Church would be to write down probabilistic fixed-point iterations that converge to the desired distribution. However, without information on the mixing rate of the Markov chain (or termination techniques like coupling from the past), running the iteration for a finite number of steps could yield an approximate sample. Alternately, we can describe these systems using a `QUERY` where the predicate is probabilistic:

```
(query '(uniformly-sample (variables-with-domains system))
      (lambda (sampled-world)
        (flip (exp (* -1 (energy-of-setting sampled-world system))))))
```

By modeling such systems in terms of conditional simulation, we achieve two goals simultane-

ously. First, we capture their intractability — in these models, forward simulation is a challenge, unlike directed models (like Bayes nets) where simulation only becomes difficult after conditioning. Second, we capture the separation between the specification of an undirected model in terms of conditional simulation and the particular generative process we use to simulate from it. For example, approximate Markov chain based implementations of `QUERY` can be swapped in for the above definition without changing the model.

We can also use this technique to model optimization of integrable functions[2] without committing to a particular optimization algorithm: we let the energy equal the integrable function and anneal to some appropriately low temperature.

Inductive Probabilistic Programming

In true Lisp spirit, `EVAL` and `QUERY` are exposed as ordinary Church procedures, and can be written in a form indistinguishable from the built-in versions even if not. Exposing `EVAL` means that one can write a program that simulates a randomly chosen probabilistic program. Nesting `EVAL` within `QUERY` means that one can conditionally simulate from a probabilistic process that samples probabilistic programs.

From the standpoint of probabilistic modeling and Bayesian learning, this means one can do inference over a hypothesis space of probabilistic programs, and induce probabilistic programs from data, without needing any special machinery. We can thus avoid the common distinction between inference "within" a modeling framework and making inferences about (or learning) models written in that framework — for example, inferring the value of unobserved variables in a Bayes net versus learning the Bayes net parameters from data. Learning a Church program from data is no more difficult, conceptually, than implementing `QUERY`, since Church programs are just another kind of Church data.

Much work remains to be done developing this approach further, especially in defining hypothesis spaces (and prior distributions) over programs and in identifying algorithms for conditional simulation that work well in these general spaces. However, to concretize the basic concept, I include a very simple example in Figure 2-10. The `gen-exp` procedure returns a symbolic arithmetic expression drawn from a simple grammar, and the query returns samples from the posterior distribution on expressions given that the result is 24.

[2]Technically, if the domain is infinite, we may need to augment the predicate of the query with a bound, so that the rejection sampler is well-defined.

```
(define (gen-exp)
  (if (flip .4)
      (list (if (flip) '+ '*)
            (gen-exp)
            (gen-exp) )
      (+ 1 (sample-integer 10)) ))

(lex-query
  '((exp (gen-exp)))
  'exp
  '(begin
     (equal? (eval exp (get-current-environment)) 24) )

> ((* 4 6) . -859.5675368561048)
> ((* 3 8) . -1122.505663959104)
> ((+ (+ 5 (* 1 6)) (+ 6 7)) . -38.9947074465309)
> ((+ 6 (+ 10 8)) . -4268.056871640514)
> ((+ (* 4 5) 4) . -3828.4984617978816)
> ((* 4 6) . -2427.0969393313953)
```

Figure 2-10: Nesting EVAL inside QUERY allows one to express the problem of learning Church programs from data, without having to reinvent the representational machinery of a programming language. This program learns various ways of expressing the value 24, consistent with a particular context-free grammar on program text in symbolic expression form.

Planning as Sampling

We can also express softmax-optimal planning as inference in Church, following Toussaint et al. (82), though working in the policy free setting. Figure 2-11 shows an example skeleton of a planner, which chooses an action today under the assumption that future actions will be chosen approximately optimally. Here, we are using a lexicalized form of query, `(lex-query <object-association-list> <query-expression> <conditioning-expression>)`, which lets you build a random world out of named pieces and refer to those pieces in later expressions. This example illustrates three aspects of Church's expressiveness, resulting from combining sampling with structured processes:

1. Decision making is normally cast in terms of a maximization: choose an action a according to the program $max_{actions\ a} E_{P(future\ f|action\ a)}[value(f)]$. However, we can approximate both the maximization and the expectation using conditional sampling: we sample from a distribution proportional to a Monte Carlo estimate of the objective. We can make the approximation exact by combining annealing (for the max) with increasing the number of samples drawn for the Monte Carlo estimation. Thus the same algorithmic machinery used for implementing query can be used for other computationally difficult tasks in probabilistic AI, and improved algorithms for conditional sampling can immediately yield improvements to planners.

2. Planning represents a simple form of metareasoning, where the reasoning agent being reasoned over is a model of the future version of the agent doing the planning. This requires building probability models over reasoning processes, or in Church terminology, the nesting of queries. Church enables compact representation of versions of this problem using general machinery, where the reasoning algorithm, the optimization algorithm for decision making, and the world being reasoned over are all represented in Church.

3. Actions are arbitrary Church objects, and rewards are computed by evaluating an arbitrary energy-valued Church expression. This means that the same planning kernel could be used to select program-valued actions from a large space of possible programs, and that all the machinery of Lisp is available to represent hierarchical, recursive structure in actions, plans, and rewards.

```
(define (choose-action state)
  (lex-query '((action (action-prior))
             'action
             '(flip (normalize-reward
                     (sample-reward action state)))))

(define (sample-reward action state)
  (let ((next-state (state-transition state action)))
    (+ (reward next-state)
       (if (terminal? next-state) 0
           (sample-reward (choose-action next-state)
                          next-state)))))
```

Figure 2-11: Planning as inference — or, more formally, softmax-optimal decision making in a sequential decision problem — is one instance of metareasoning, or reasoning about reasoning. The essential recursion captures the question "how should I act today, given that I will reason and act approximately optimally after taking that action?".

2.3 Discussion and Directions

We obtained Church by (1) adding stochasticity to a lexically scoped, call-by-value Lisp with first-class environments, (2) generalizing the idea of evaluation to stochastic simulation, with purity becoming exchangeability and (3) generalizing simulation to conditional simulation for universal Bayesian inference. We then saw how higher-order programming idioms from functional programming could be generalized probabilistically and used to simplify nonparametric Bayesian modeling, and how the Lispiness of Church opens the door for inductive programming and probabilistic metareasoning. Work on probabilistic programming, however, is just beginning.

2.3.1 Inference and Complexity

The main practical challenge facing the Church project is the development of better algorithms for `query`, along with massively parallel implementations.

One starting point is the "initialization" of an MCMC-based `QUERY`, where we currently use rejection, supported by a suite of constraint propagation heuristics. Finding a computation trace that is compatible with the predicate of a query from which to "start" MCMC is a full inference problem in its own right. An exact sample from the posterior would be the perfect initialization. A means of building up an initial trace out of pieces, biased towards high probability traces, would

be a significant contribution. The systematic sampling algorithm described in the next chapter may provide some useful leads. The main outstanding challenge is finding a means to "reverse" evaluation locally.

Partial evaluation provides another avenue of attack on the inference problem. In probabilistic programming, partial evaluation may be generalizable to marginalization by integration. If I know the distribution on values for some argument to a probabilistic procedure, I can potentially use that knowledge to simplify its contents by integrating out the variable. Query decomposition to syntactically leverage conditional independencies in a Church program seems like another promising technique. The procedural nature of Church and the close coupling of algorithm and belief give us hope that the inference problem may be tractable without an infinite regress into algorithm design. However, much of compiler research for deterministic languages can be interpreted as the art of accelerating `EVAL`, so our work is certainly cut out for us.

The theoretical study of the complexity of Church programs is another important area for future work. Only a small subset of Church programs naturally coincide with the decision or evaluation based complexity classes traditionally studied in theoretical computer science. However, there are some interesting analogies that might suggest useful generalizations of the theory. For example, one can view the set of Church `QUERY` expressions where both the expression and the predicate are easy to evaluate as a kind of stochastic analogue of the problem class NP: it is easy to check if a candidate solution satisfies the desired constraints, and easy to generate candidate solutions, although possibly hard in general to generate candidate solutions that satisfy the desired constraints. Finding an appropriate embedding of existing complexity classes into Church, and designing Church implementations which behave appropriately even on deterministic subproblems, seems like an important challenge.

2.3.2 Semantics and First-class Objects

The semantics of Church and connections with advanced constructs in functional programming are another potentially interesting area of research. For example, at present, because of the semantic restriction to programs that halt with probability 1, programming in Church may be viewed as a stochastic generalization of *total functional programming*. On the one hand, it seems difficult to avoid this restriction and retain the essential idea of Church, where distributions are induced by stochastic procedures. It is at present unclear how to define the "value" of an expression for which evaluation does not halt. On the other hand, this restriction may rule out certain potentially

sensible objects, such as conditional samplers for PCFGs which place some mass on infinite strings but which, given a finite yield, place all their mass on finite parses. Better understanding these technical limitations may help to clarify the formal foundations of Church.

Exploring fully lazy (or call-by-need) variants of Church also seems interesting. Currently, delayed evaluation is introduced by MEM, requiring a special exchangeable random primitive. The choice to make Church call-by-value was partly driven by expediency and partly driven by the observation that it is easy to simulate lazy behavior in a call-by-value language, but difficult to simulate eager behavior (with its associated potential savings of time and space, especially for recursive processes) when one only has lazy primitives. However, some programs that do not halt in eager languages do halt in ones that are lazy by default, and laziness may mesh well with the declarative nature of many Church programs.

The exclusion of SET! from the Church language has been the cause of considerable debate among the coinventors of Church. On the one hand, it has clearly proven helpful to be able to define primitives that mutate local state but satisfy an exchangeable contract. On the other hand, it is hard to decide how SET! should be treated if it occurs inside a QUERY (in some sense requiring the maintainence of a temporal history of the environment of the computation) or to weigh the cost of giving up exchangeability by construction. This issue connects more broadly with the question of the flow of time in Church programs. Currently, all Church programs define "timeless" distributions, meshing well with the declarative uses of Church; this is at the heart of exchangeability. However, this very timelessness makes it difficult to implement an efficient single-threaded Church engine within Church, and impossible to implement a massively parallel Church engine. Furthermore, it is currently unclear how to model continuous time stochastic processes in Church without leaning on complex strategies for nonuniform discretization. The development of concurrent stochastic programming primitives might well simultaneously settle both the practical issues of concurrent inference and allow for the natural modeling of continuous time stochastic processes.

Finally, we have avoided introducing first-class continuations into Church or thinking carefully about the stochastic significance of generalized tail recursion. We are not sure what a good notion of a stochastic delimited control construct is, or how such a construct would interact with the declarative semantics of a Church program. We suspect tail-recursive processes using arguments with bounded space may correspond to distributions with finite dimensional sufficient statistics (e.g. exponential families), since tail recursion seems analogous to closure under sequential conditioning. We also wonder if continuations could enable the expression of exchangeable primitives without mutation. We regretfully leave these expressiveness questions to future work.

2.3.3 Probabilistic programs, integer programs and logic programs

In computing, programming has come to mean two different things. Programmers often think of programming as defining or describing procedures and processes that machines will execute. Applied mathematicians often think of programming in the sense of linear programing and integer programming (and AI researchers in terms of logic programming): the definition of declarative problems involving maximization or satisfiability, solved by particular algorithms.

Conditional simulation in Church captures important aspects of both of these approaches. A query expression is a program in the programmer's sense, while the predicate defines a constraint in the applied mathematician's sense. Taken together, they define a declarative program over a space of values that was specified procedurally. In fact, by taking the view of Church queries as undirected models from computational physics and applying techniques like annealing, one can write down linear programs, integer programs, and weighted logic programs as probabilistic programs.

A Church engine solves these programs by using general-purpose reasoning machinery (e.g. in MCMC) that repeatedly simulates parts of the query expression to probabilistically satisfy the constraints of the query. Algorithmic knowledge is thus split between the choice of one particular query expression among some equivalent set and the implementation of query in the underlying Church engine. Finding representations of linear programs as query expressions which allowed the Church MCMC algorithm to automatically recover the techniques in, say, interior point methods, seems like one particularly interesting way to explore these procedural/declarative relationships.

2.3.4 Inductive Programming and Programming Style

> "54. Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy."
>
> — Alan Perlis, *Epigrams on Programming*

> "The real power of Lisp is that its unrestricted abstractions support the construction of new languages, greatly facilitating the strategy of stratified design."
>
> — Abelson & Sussman, *Lisp: a language for stratified design*

We have been trying to produce programs that are capable of programming since almost the origins of the field. While the deepest problems of program synthesis remain very much open, probabilistic programming may offer a new perspective for these problems and for problems of formalizing programming style.

It seems to us that programming languages are deemed "expressive" when it is possible to write a range of interesting programs and languages in them compactly, and when these new programs and languages will themselves be easy to extend. Probabilistic programming lets us begin to explore this issue formally, by viewing a "style" as a probabilistic program or grammar that generates programs in a given style. We can measure the ease of writing a program in a style by asking about the probability of the program under that style. For styles that adapt as programs are generated — for example, because they involve defining reusable symbols and programs — we can even study how the probability of a given program varies as a function of the number of symbols that have already been defined. These same issues will let us understand the inductive biases that result from different prior probability distributions on the space of programs.

The ability to study nested hypothesis spaces of programs — where a programming style or embedded language is learned, along with a program in it — is one new tool that probabilistic programming has to offer efforts in program synthesis. Another difference is that the goal is framed in terms of sampling a program from the induced posterior on programs, as opposed to finding the best program, so all the algorithmic techniques for approximately sampling over nonconvex energy landscapes can be brought to bear. The most important reason, however, may be that probabilistic programs can get partial credit for good answers. A probabilistic program does not need to deterministically produce a desired answer to be represented in the posterior on programs. Small changes to a program might increase the probability that it produces a desired output, allowing search algorithms to get partial credit.

In fact, this softening suggests other ways to generate new probabilistic programming constructs. We can take deterministic operators — like `AND` and `IF` — and view them as limits of appropriate families of probabilistic variants. For example, a noisy IF might take the wrong branch with some probability, and noisy logical operators could be built in terms of noisy IFs:

```
(IF (IF (flip *noise-level*) <pred> (not <pred>))
    <consequent>
    <alternate>)
```

This kind of softening was a key contributor to the success of neural networks over multilayer

perceptrons, by making gradient based methods possible. It can be applied to a program as a program transformation, and can itself be written as a simple Church macro. By noising up more general operators in programming languages, even structured, deterministic programs could be automatically relaxed into probabilistic programs that get partial credit.

Chapter 3

Systematic Stochastic Search

> "I think all our favorite [inference] methods – Gibbs sampling, overrelaxation, hybrid Monte Carlo, variational methods, EM, gradient descent – are all too creepy-crawly slow... The world isn't an adversary. It should be possible to solve many learning problems in a couple of iterations through a reasonable data set, rather than thousands. It may be too much to ask for a one-shot learning method, but maybe we should be looking for one-and-a-half-shot learning algorithms."
>
> – David J. C. MacKay, quoted in *Graphical Models for Machine Learning and Digital Communication*

> "Eliminate the impossible, and whatever remains, however improbable, must be the truth."
>
> – Sir Arthur Conan Doyle, *Sign of Four*

3.1 Introduction

Efficient inference in high-dimensional, discrete probability models is a central problem in computational statistics and probabilistic AI. It arises in a variety of situations, including medical diagnosis, when trying to reason from the results of tests to underlying symptoms, and in computer vision, when trying to reason from observed pixel values to depth maps.

In this chapter, we introduce a recursive algorithm for exact and approximate sampling aimed at solving this problem in the presence of multimodality. Our goal is to develop samplers that work well at diagnosing causes even when there are only a small number of high probability explanations for the data that are very different. In these settings, we want reasoning algorithms that recover this

qualitative structure, and capture the relative probability of these 'main' explanations as reliably as possible. We would like to do this efficiently even when there are exponentially many possible explanations and so the search problem seems difficult and prone to local minima.

We do this by generalizing ideas from classic AI search to the stochastic setting. Just as systematic search algorithms like A* recursively build complete solutions from partial solutions, sequential rejection sampling recursively builds exact samples over high-dimensional spaces from exact samples over lower-dimensional subspaces. Our method exploits and generalizes ideas from classical AI search for managing deterministic dependencies, including depth first search with backtracking, to tractably generate exact and approximate samples.

3.1.1 Sampling can manage multimodality where other inference methods fail

Many popular approaches to inference, such as mean-field variational methods (39), convex relaxations (88; 75), and generalized belief propagation (91), focus on approximating MAP assignments or (low-dimensional, e.g. 1 or 2 variable) marginal distributions. That is, they ignore the correlations between variables, instead looking for the "average" value of each variable individually, hoping that will lead to sufficiently accurate inferences. While effective in many settings, low-dimensional marginals (and MAP assignments, or "most probable" configurations) often do not capture the essential features of a distribution, especially in the presence of multimodality. Ferromagnetic Ising models provide one source of extreme examples:

$$p(x) = \exp\Big\{-J \sum_{(i,j)\in E} x_i x_j\Big\}, \quad x_i \in \{-1, 1\}. \tag{3.1}$$

As the coupling parameter J increases, the joint distribution on spins approaches a 2-component mixture on the "all up" and "all down" states, which has only 1 bit of entropy. Put differently, as J increases, the Ising approaches a SAT formula where all variables are constrained to be equal. In physical terms, we see a phase transition on the structure of solutions, from a disordered mess through a graded region of high complexity to a region where two qualitative states dominate.

A MAP approximation misses the fundamental bimodality of the distribution. Thus optimization-based approaches will ignore all but one of the valid explanations. A minimum-KL product-of-marginals approximation (as in naive mean field) confuses this distribution with the uniform dis-

tribution on spins. That is, ignoring correlations between variables in cases where there are only a small number of qualitative explanations destroys the critical structure these situations naturally exhibit.

When a high-dimensional distribution contains many widely separated, hard-to-find modes, it becomes difficult to parametrically approximate. Accordingly, simulation-based inference, where we represent the distribution in terms of a stochastic procedure that samples from it, seems ideal. If an efficient sampler[1] can be found, we can invoke it to produce accurate Monte Carlo estimates that take all modes into account or use it as a subroutine in the construction of ever-larger samplers, leveraging well-known composition laws (8). Fundamentally, a sampler-based (not sample-based!) representation of a distribution allows us to work with a program that can, when invoked, lets us explore the qualitatively important region of the distribution on demand. We thus avoid the need to come up with a complete, exponentially large, quantitative description. If we wish to ultimately build probabilistic agents that maintain and reason over uncertain beliefs for long periods of time, we will need to work with a representation for distributions that has this basic property.

Exact and approximate sampling are also problems of intrinsic interest in computational physics (60; 19), allowing us to directly inspect typical configurations of model systems defined in terms of thermodynamic equilibrium. Unfortunately, popular methods like Gibbs sampling often run into severe convergence problems in precisely those settings where the distribution of interest is highly multimodal and therefore sampling would be most desirable. These difficulties have historically motivated a number of specialized samplers that exploit sophisticated data augmentation techniques (78), as well as a variety of model-specific proposal designs. Our algorithm mitigates the problems of multimodality by generalizing ideas for managing deterministic dependencies from the constraint satisfaction literature. In particular, it operates by a stochastic generalization of *systematic* search.

3.1.2 Systematic vs Local Algorithms, Fixed Points, and Samplers

In deterministic systematic search, solutions to a problem are built up piece-by-piece, according to some scheme that ensures the first complete solution found will be good. This approach can be usefully contrasted with with *local* searches, generally based on fixed-point iteration, where a

[1] We note that practical efficiency here means no worse than *linear* in problem size with a manageable constant. This is a far stricter criterion than the theoretical notions of efficiency, which generally settle for performance that scales polynomially in problem size.

Domain	Systematic	Local
Sorting	MergeSort	BubbleSort
SAT solving	DFS, BFS, arc consistency, DPLL	Min constraint
ODEs/PDEs	Shooting (e.g. Euler, RK4)	Relaxation (e.g. Gauss-Seidel)
Linear systems	Algebraic (e.g. LU/QR)	Iterative (e.g. conjugate gradient)

Table 3.1: Systematic algorithms build up complete solutions out of sequences of partial pieces, while local algorithms attempt to iteratively improve structurally complete candidate solutions. The systematic/local distinction cross-cuts many fundamental algorithmic problems.

complete but possibly poor quality approximate solution is repeatedly iterated on until it stabilizes. Table 3.1 provides a domain-general list of example algorithms of both sorts.

More formally, most local searches can be cast as fixed-point iteration algorithms. One is looking for some specific $x^* \in \mathcal{X}$, for example a permutation that sorts the elements of some list. One then designs an f such that $f(x^*) = x^*$, finds some $x^0 \in cX$, and computes $x_{out} = f(f(\cdots f(x^0)))$. In bubble sort, for example, f can be viewed as a map from the initial permutation of the list to one after a full pass of "bubbling up" has taken place. Alternately, one can augment the space $\mathcal{X}$ to include a variable indicating what element of the permutation is currently being modified, and have f represent a single bubble-sort swap. In deterministic algorithm design, one typically attempts to argue that iterating f from the chosen x^0 will yield $x_{out} = x^*$ in some finite - and ideally both short and automatically computable - number of steps. One way to do this is by finding a Lyapunov function which is 0 for the desired x^* and decreases strictly monotonically under the action of f. In the case of bubble sort, one can use the number of out-of-order elements in the list, and it is easy to show that this reaches 0 when the algorithm has completed the nth full sweep of the initial list.

Markov chain based methods for simulation generalize this basic idea of fixed-point iteration to the broader setting of stochastic simulation[2]. Here, one wants to generate a sample from some distribution of interest $\pi(\cdot)$. Note that this stochastic goal of generating a sample from some fixed distribution strictly generalizes the goal of deterministically obtaining a particular value x^*. Obtaining such a value can be viewed as evaluating a no-argument function - that is, a function of

[2]While they have historically been used as part of Monte Carlo estimators for functions, yielding the label "Markov chain Monte Carlo" or MCMC methods, I argue it makes sense to view them as a tool for sampling in their own right. Monte Carlo estimation can then be viewed as the addition of a subsequent stage, resulting in a sample from a distribution whose mean is hopefully close to some quantity of interest.

no arguments that deterministically returns x^*.

For simplicity, we will assume that $\pi(\cdot)$ is a distribution on a discrete space $\mathcal{X}$, and represent $\pi(\cdot)$ as a vector $\vec{\pi}$ and our Markov chain transition kernel $T(x) : \mathcal{X} \to \mathcal{X}$ as a stochastic matrix $\mathbf{T}$. Markov chain based methods work by exploiting Ts with respect to which π is invariant, or in terms of probability functions, $\mathbf{T}\pi = \pi$. This is the stochastic generalization of the fixed-point condition above. Samples are generated by drawing x^0 from some initial distribution $\pi^0(\cdot)$ - again, analogous to the choice of x^0 above - and returning $T(T(\cdots T(x^0)))$ as a sample drawn approximately from π. Termination analysis is more complex in the stochastic setting, but recent advances like coupling from the past (60; 34; 12) provide one set of approaches, by enabling automatic determination of the number of iterations needed for exact convergence. Up to the technical constraints and regularity conditions imposed by discreteness, this embedding is fully general. Accordingly, we can view each of the deterministic local searches above as a special case of a Markov chain simulation method.

For example, bubble sort can be fruitfully viewed as an MCMC algorithm whose goal is to sample permutations to approximately sort the list. The energy function π counts the number of neighboring pairs under the permutation that are out of order. The transition kernel of the MCMC algorithm is a deterministic cycle of kernels that make Metropolis-Hastings proposals for pairwise swaps. When we anneal the energy so it is 0 if all elements are in order and ∞ otherwise and we use Gibbs kernels on swaps instead of Metropolis-Hastings, we will always swap elements if they are out of order, and swap with probability 0.5 if they are in order. Repeated iteration thus recovers the swaps made by bubble sort, with additional random choices to guarantee a uniform sample from all consistent partitionings.

The view of MCMC methods as stochastic local searches invites the question "what are useful stochastic generalizations of systematic searches?". In systematic searches, the first complete candidate solution is either exact (as in backtracking search or A*) or approximate (as in beamed searches), and strategies for search tree expansion are often used to manage deterministic dependencies among chains of choices. We introduce a method that automatically recovers a randomized variant of one such method, depth first search with backtracking, when applied to constraint satisfaction problems, generalizing these ideas to the setting of sampling. Furthermore, if the rejection step in our algorithm is replaced with importance sampling with resampling (and particular restricted choices of variable orderings or choice points for divide-and-conquer are used) we recover widely-used particle filtering algorithms for approximate sampling.

In this chapter, we present the mathematical and algorithmic underpinnings of our approach and

measure its behavior on ferromagnetic Isings and other probabilistic graphical models, obtaining exact and approximate samples on a range of realistic sampling problems with surprisingly low algorithmic cost.

3.2 The Adaptive Sequential Rejection Algorithm

Consider the problem of generating samples from an unnormalized, high-dimensional distribution $p(x)$, $x \in \mathcal{X}$, where $\mathcal{X} = \mathcal{Y} \times \mathcal{Z}$ and $p(x) = p(y, z) \propto \psi_1(y)\, \psi_2(y, z)$. This setting arises in factor graph inference, where the unknown constant in p is due to either the partition function of the underlying undirected model or the marginal likelihood of the evidence in the underlying directed model. For concreteness, in stereo vision, we might have x be a complete depth map, and z be the depth value at a single pixel, and y be the depth values for all other pixels.

Our algorithm generates exact samples from p by recursively generating an exact sample from $p'(y) \propto \psi_1(y)$ (which we assume has an analogous decomposition, i.e. $\mathcal{Y}$ and ψ_1 split into pieces), and then extending it to an exact sample from p by rejection. The idea is to use the standard computer science device of wishful thinking: find a way to solve the problem when the last pixel is removed, and then find a way to extend that partial solution into a solution to the full problem.

To apply our algorithm to an arbitrary factor graph, we need a way of recursively decomposing the model into distributions that are themselves decomposable. That is, we need general methods of divding factor graph inference problems into subproblems. To recover backtracking, so that we don't bother reconsidering states that we know to be inconsistent, we need to adapt proposals over time to reflect information we have learned during sampling. We will return to these issues later in the chapter, initially focusing on the main recursion.

First, we define[3] the Gibbs transition kernel

$$q_p(z \mid y) \triangleq \frac{p(y, z)}{\sum_{z'} p(y, z')} \tag{3.2}$$

$$= \frac{\psi_1(y)\, \psi_2(y, z)}{\sum_{z'} \psi_1(y)\, \psi_2(y, z')} \tag{3.3}$$

$$= \frac{\psi_2(y, z)}{\sum_{z'} \psi_2(y, z')} \tag{3.4}$$

and use $p'\, q_p$ as the proposal distribution for a rejection sampler for p: i.e., we first generate an

[3]We will use $\triangleq$ to denote definitions.

exact sample $\hat{y}$ from p' (by induction) and then generate an exact sample $\hat{z}$ from $q_p(\cdot \mid \hat{y})$. Let $\hat{x} = (\hat{y}, \hat{z})$ and define the weight of the sample $\hat{x}$ as

$$\begin{aligned} w_{p' \to p}(\hat{x}) &\triangleq \frac{p(\hat{y}, \hat{z})}{p'(\hat{y})\, q(\hat{z} \mid \hat{y})} && (3.5) \\ &= \frac{p(\hat{y}, \hat{z}) \sum_{z'} p(\hat{y}, z')}{p'(\hat{y})\, p(\hat{y}, \hat{z})} && (3.6) \\ &= \frac{\sum_{z'} p(\hat{y}, z')}{p'(\hat{y})} && (3.7) \\ &= \frac{\sum_{z'} \psi_1(\hat{y})\, \psi_2(\hat{y}, z')}{\psi_1(\hat{y})} && (3.8) \\ &= \sum_{z'} \psi_2(\hat{y}, z'). && (3.9) \end{aligned}$$

Note that the weight does not depend on $\hat{z}$ and so we will consider the weight $w_{p' \to p}(\hat{y})$ as a function of $\hat{y}$.

Recall that in rejection sampling with proposal p' and target p, we need to find some $c_{p' \to p} > w_{p' \to p}(y)$ for all $y \in \mathcal{Y}$. We can then accept $\hat{x}$ as an exact sample from p with probability

$$\frac{w_{p' \to p}(\hat{y})}{c_{p' \to p}}, \tag{3.10}$$

In general, loose upper bounds on each c are easy to find, but will introduce unnecessary rejections. Overconfident values of c are also easy to find, but will result in approximate samples. Both these variants may have practical value. Here, we focus on the setting where we actually use the optimal rejection constant:

$$c^*_{p' \to p} \triangleq \max_y w_{p' \to p}(y) = \max_y \sum_{z'} \psi_2(y, z'). \tag{3.11}$$

If $y = (y_1, \ldots, y_n)$ is high-dimensional, then the worst case complexity of calculating $c^*_{p' \to p}$ is exponential in n. However, when the sequence of distributions we are using has sparse dependencies (i.e., when $\psi_2(y, z)$ is a function of only $O(\log n)$ dimensions y_i), then we can calculate $c^*_{p' \to p}$ in polynomial time. For example, in grid Ising graphs, ψ_2 depends on at most three neighbors and therefore c^* can be calculated in constant time.

This inductive argument describes the non-adaptive sequential rejection sampler. Later on, we will describe a way to recursively build a good sequence of decompositions of the problem

by borrowing from the constraint propagation literature. However, before we do this, it is worth considering cases where we believe that rejection will be intractable, either because cs will be too hard to compute or because the rejection rate will be too high. In such settings, we might be content with *approximate* samples, but we would still like to leverage the systematic approach presented here.

There are many ways to relax our method to yield potentially useful approximate variations. The simplest is to replace the rejection step with k particle importance sampling plus resampling, avoiding the need to compute c or wait for acceptance at the cost of a biased approximation. Another option is to collapse all the rejection steps into one large sequence where sequential importance sampling with resampling is performed: rather than propagate 1 exact sample, we propagate k weighted particles which approximate the distribution of interest. This provides a kind of stochastically beamed, breadth-first alternative to the depth-first approach of the rejection sampler. We will later see how this version recovers widely used particle filtering algorithms as a special case.

The choice of the Gibbs transition kernel is important. Incorporating the $\psi_2(y, z)$ factor into the proposal prevents the algorithm from proposing samples $\hat{z}$ that are already known to be incompatible with the setting $\hat{y}$. Put differently, in the process of building up a sample, we shouldn't bother trying out things we can already know for certain will be inconsistent or unlikely.

3.2.1 Adaptation Stochastically Generalizes Backtracking

Systematic searches typically avoid reconsidering partial solutions that have been discovered inconsistent, eliminating the impossible case by case until a valid answer is found. This behavior is known as backtracking, and requires dynamically recording the information about inconsistent states obtained over the course of search. We accomplish this in the broader setting of sampling by introducing an adaptation rule into our sampler, which recovers this deterministic avoidance in the limit of deterministic inconsistency.

Our non-adaptive sampler accepts samples with probability

$$\alpha_{p' \to p} = \frac{\mathbb{E}_{p'}\{w_{p' \to p}(\hat{y})\}}{c*_{p' \to p}}. \tag{3.12}$$

Let $p(y) = \sum_{z'} p(y, z')$ be the marginal (unnormalized) distribution of y under p; let $Z_p = \sum_y p(y) = \sum_y \sum_{z'} p(y, z')$, be the normalization constant of the distribution p; and let $Z_{p'}$ be

the normalization constant for p'. From Eq. 3.7, we have that

$$w_{p' \to p}(\hat{y}) = \frac{p(\hat{y})}{p'(\hat{y})}, \tag{3.13}$$

therefore,

$$\begin{aligned}
\mathbb{E}_{p'}\{w_{p' \to p}(\hat{y})\} &= \mathbb{E}_{p'}\left\{\frac{p(\hat{y})}{p'(\hat{y})}\right\} && (3.14)\\
&= \sum_{\hat{y}} \frac{p'(\hat{y})}{Z_{p'}} \frac{p(\hat{y})}{p'(\hat{y})} && (3.15)\\
&= \frac{Z_p}{Z_{p'}}, && (3.16)
\end{aligned}$$

and finally,

$$\begin{aligned}
\alpha_{p' \to p} &= \frac{Z_p}{Z_{p'}} \frac{1}{c*_{p' \to p}} && (3.17)\\
&= \frac{Z_p}{Z_{p'}} \frac{1}{\max_{y'} \frac{p(y')}{p'(y')}} && (3.18)\\
&= \min_{y'} \frac{\frac{1}{Z_{p'}} p'(y')}{\frac{1}{Z_p} p(y')}. && (3.19)
\end{aligned}$$

Note that the acceptance probability $\alpha_{p' \to p}$ depends only on the choice of p' and p and is precisely the largest ratio in absolute probability assigned to some $y \in \mathcal{Y}$.[4] An interesting special case is when the simpler distribution p' matches the marginal $p(y)$. In this case, $w_{p' \to p} = 1$ and we always accept.[5] Assuming each attempt to generate samples from p' by rejection succeeds with probability $\alpha_{p'}$, the entire rejection sampler will succeed with probability $\alpha_{p'} \alpha_{p' \to p}$. If this probability is $O(2^{-w})$, where w is the tree width of the factor graph, then, in expectation, we will be no better off than using variable clustering and dynamic programming to calculate marginals and sample exactly.

Our goal then is to drive $\alpha_{p' \to p} \to 1$ (and inductively, $\alpha_{p'} \to 1$). Consider the extreme case

[4]In particular, the acceptance is positive only if $p(y) > 0 \implies p'(y) > 0$ (i.e., p' is absolutely continuous with respect to p).

[5]While it may be tempting to think the problem is solved by choosing $p = p'$, if each stage of the algorithm performed this marginalization, the overall complexity would be exponential. The key to adaptation will be selective feedback.

where a sampled value $\hat{y}$ is revealed to be inconsistent. That is, $\psi_2(\hat{y}, z) = 0$ for all z and therefore $w_{p' \to p} = 0$. We should then adjust p' (and its predecessors, recursively) so as to never propose the value $\hat{y}$ again. Certainly if p' is the marginal distribution of p (recursively along the chain of rejections), this will take place.

Consider the distribution

$$p'_S(y) \propto p'(y) \prod_{y' \in S} \left(\frac{w_{p' \to p}(y)}{c^*_{p' \to p}} \right)^{\delta_{yy'}} \tag{3.20}$$

where $S \subset \mathcal{Y}$ and δ_{ij} is the Kronecker delta. Then

$$\begin{aligned} w_{p'_S \to p}(\hat{x}) &\triangleq \frac{p(\hat{y}, \hat{z})}{p'_S(\hat{y})\, q(\hat{z} \mid \hat{y})} && (3.21) \\ &= \frac{w_{p' \to p}(y)}{\prod_{y' \in S} \left(\frac{w_{p' \to p}(y)}{c^*_{p' \to p}} \right)^{\delta_{yy'}}} && (3.22) \\ &= \begin{cases} c^*_{p' \to p} & y \in S \\ w_{p' \to p}(y) & y \notin S. \end{cases} && (3.23) \end{aligned}$$

Therefore $c^*_{p'_S \to p} = c^*_{p' \to p}$. In particular, if $S = \mathcal{Y}$, then $w_{p'_S \to p} = c^*_{p'_S \to P} = c^*_{p' \to p}$ and every sample is accepted. In fact,

$$\begin{aligned} p'_{S=\mathcal{Y}}(y) &\propto p'(y) \prod_{y' \in \mathcal{Y}} \left(\frac{w_{p' \to p}(y)}{c^*_{p' \to p}} \right)^{\delta_{yy'}} && (3.24) \\ &\propto p'(y)\, w_{p' \to p}(y) && (3.25) \\ &= p'(y)\, \frac{p(y)}{p'(y)} && (3.26) \\ &= p(y) && (3.27) \end{aligned}$$

and therefore an exact sample from $p'_{\mathcal{Y}}$ is a sample from the marginal distribution of p. The Gibbs kernel exactly extends this to a sample from the joint.

Adaptation then involves the following modification to our algorithm: after proposing a sample $(\hat{y}, \hat{z})$, we augment S with $\hat{y}$. As $S \to \mathcal{Y}$, $p'_S(y) \to \frac{1}{c^*_{p' \to p}} p(y)$ pointwise.

This change can be implemented efficiently by storing a hashmap of visited states for every dis-

tribution in the sequence and modifying density evaluation (and, therefore, the Gibbs kernels) to reflect hashmap contents. Each stage of the sampler pushes states to the previous stage's hashmap as adaptation proceeds, moving each proposal distribution towards the ideal marginal. Because such adaptation leaves c^* unchanged (see Appendix), adaptation increases the algorithmic complexity by only a linear factor in the number of sampling attempts, with overall complexity per attempt still linear in the number of variables. Taken together, the hashmaps play the role of the stack in a traditional backtracking search, recording visited states and forbidding known bad states from being proposed.

3.2.2 Sequences of Distributions for Graphical Models

To apply this idea to graphical models, we need a way to generically turn a graphical model into a sequence of distributions amenable to adaptive sequential rejection. We accomplish this - and introduce further ideas from systematic search - by introducing the idea of a *sequence of restrictions* of a given factor graph, based on a *variable ordering*, i.e. permutation of the variables in the model). Each sequence of restrictions can be deterministically mapped to a nested sequence of factor graphs which, for many generic orderings, capture a good sequence of distributions for sequential rejection under certain analytically computable conditions.

We denote by X_i a random variable taking values $x_i \in \mathcal{X}_i$. If $V = (X_1, \ldots, X_k)$ is a vector random variables, then we will denote by $\mathcal{X}_V$ the cartesian product space $\mathcal{X}_1 \times \cdots \times \mathcal{X}_k$ in which the elements of V take values $v = (x_1, \ldots, x_k)$.

Definition 3.2.1 *A* factor graph $G = (X, \Psi, V)$ *is an undirected* X, Ψ*-bipartite graph where* $X = (X_1, \ldots, X_n)$ *is a set of random variable and* $\Psi = \{\psi_1, \ldots, \psi_m\}$ *is a set of factors. The factor* ψ_i *represents a function* $\mathcal{X}_{V_i} \mapsto [0, \infty]$ *over the variables* $V_i \subset X$ *adjacent to* ψ_i *in the graph. The graph represents a factorization*

$$p(v) = p(x_1, \ldots, x_n) = \frac{1}{Z} \prod_i \psi_i(v_i) \tag{3.28}$$

of the probability density function p, where Z *is the normalization constant.*

Definition 3.2.2 *The* restriction G_S of the factor graph $G = (X, \Psi, V)$ to a subset $S \subset X$ *is the subgraph* (S, Ψ_S, V_S) *of* G *consisting of the variables* S*, the collection of factors* $\Psi_S = \{\psi_i \in \Psi \mid$

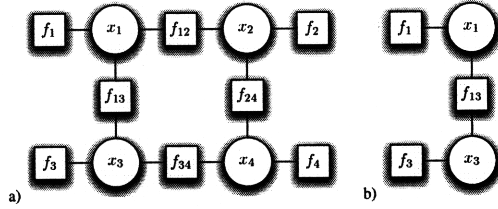


Figure 3-1: A four node Ising model, and its restriction to the variables x_1 and x_3.

$V_i \subset S\}$ *fully determined by the variables in* S*, and the edges* $V_S = \{V_i \mid \psi_i \in \Psi_S\}$ *connecting the edges* S *and factors* Ψ_S*. We denote by* Z_S *the normalization constant for the restriction.*

See Figure 3-1 for an example restriction of an Ising model. Consider a factor graph $G = (X, \Psi, V)$ and let $X_{1:k} = \{x_1, \ldots, x_k\} \subset X$, $(k = 1, \ldots, n)$ be the first k variables in the model under some order. The sequence of distributions we consider are the distributions given by the restrictions $G_{X_{1:k}}$, $k = 1, \ldots, n$.

We recover likelihood weighting (generalizing it to include resampling) on Bayesian networks when we use the importance variant of our algorithm and a topological ordering on the variables. Similarly, we recover particle filtering when we apply our method to time series models, with resampling instead of rejection and an ordering increasing in time.

In this chapter, we focus on generically applicable strategies for choosing an ordering. All our exact sampling results use a straightforward ordering which first includes any deterministically constrained variables, then grows the sequence along connected edges in the factor graph (with arbitrary tie breaking). This way, as in constraint propagation, we ensure we satisfy known constraints before attempting to address our uncertainty about remaining variables. If we do not do this, and instead sample topologically, we find that unlikely evidence will lead to many rejections (and approximate rejection, i.e. likelihood weighting, will exhibit high variance). In general, we expect finding an optimal ordering to be difficult, although heuristic ordering information (possibly involving considerable computation) could be exploited for more efficient samplers. An adaptive inference planner, which dynamically improves its variable ordering based on the results of previous runs, remains an intriguing possibility.

3.3 Experiments

First, we measure the behavior on ferromagnetic Ising models for a range of coupling strengths, including the critical temperature and highly-coupled regimes where Gibbs samplers (and inference

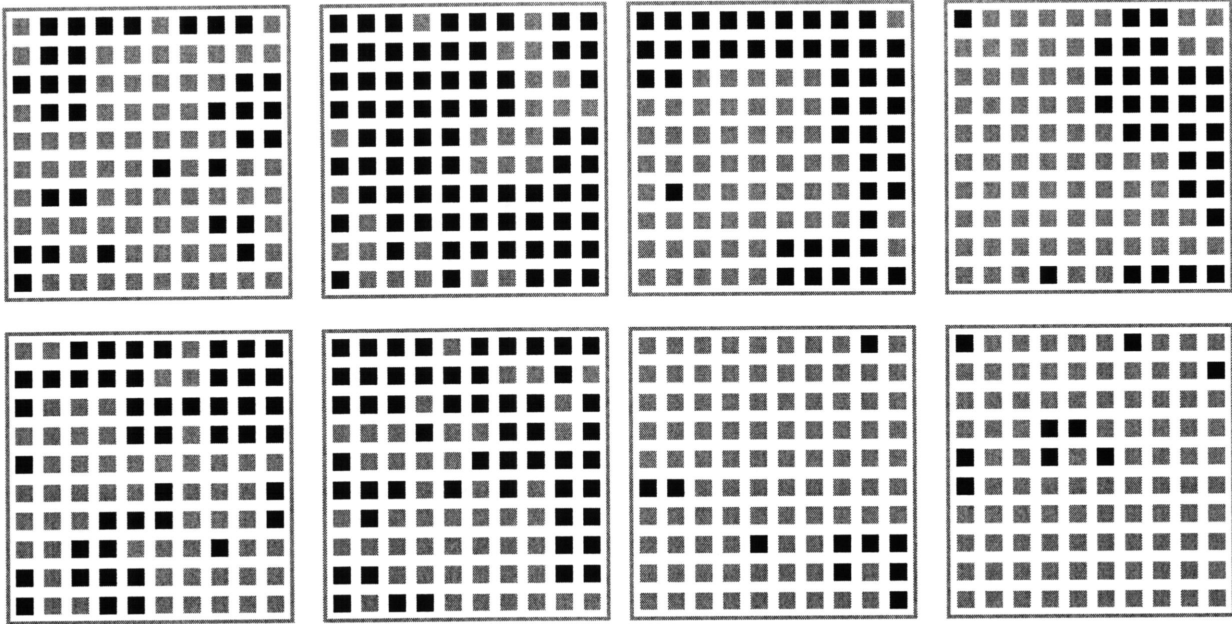


Figure 3-2: (left 4) Exact samples from a 100-dimensional grid ferromagnetic Ising just below the critical temperature. (right 4) Exact samples from a 100-dimensional grid ferromagnetic Ising just above the critical temperature.

methods like mean-field variational and loopy belief propagation) have well-known difficulties with convergence; see Figure 3-3 shows some of our results.

We have also used our algorithm to obtain exact samples from 10,000-dimensional antiferromagnetic (repulsive) grid Ising models at high coupling, with no rejection, as is expected by analytic computation of the αs, describing probability of acceptance. At this scale, exact methods such as junction tree are intractable due to treewidth, but the target distribution is very low entropy and generic variable orderings that respect connectedness lead to smooth sequences and therefore effective samplers. We have also generated from exact samples from 400-dimensional ferromagnetic grid Isings at more intermediate coupling levels, where adaptation was critical for effective performance.

We also measured our algorithm's behavior on randomly generated (and in general, frustrated) Ising models with coupling parameters sampled from $U[-2, 2]$. We report results for a typical run of the adaptive and non-adaptive variants of sequential rejection sampling on a typical problem size; see Figure 3-4 for details. We also note that we have successfully obtained exact samples from 64-dimensional Isings with randomly generated parameters, using adaptation. On the models we tested, we obtained our first sample in roughly 5000 attempts, reducing to roughly one sample per 1000 attempts after a total of 100,000 had been made.

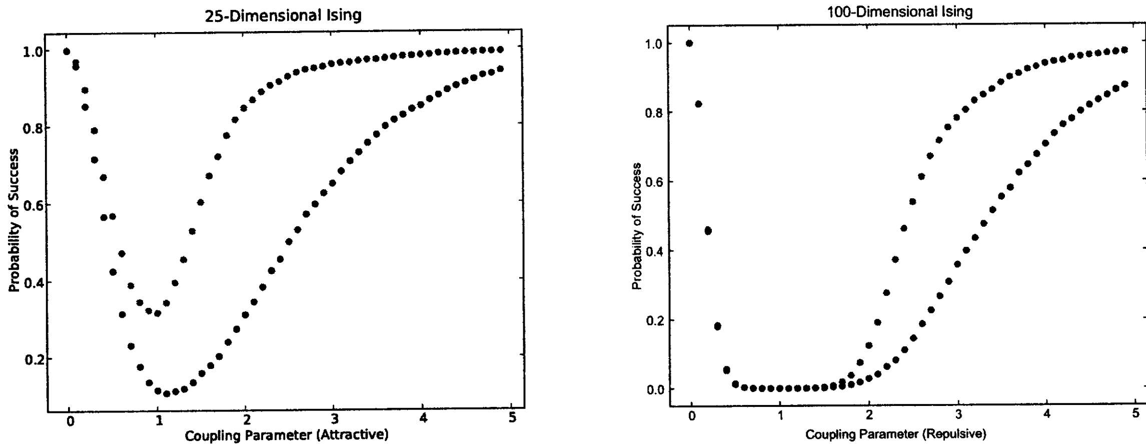


Figure 3-3: (left) Ferromagnetic (right) Antiferromagnetic. (both) Frequency of acceptance in nonadaptive (blue, lower) and adaptive (red, higher) sequential rejection as a function of coupling strength J. Naive rejection approaches suffer from exponential decay in acceptance probability with dimension across all coupling strengths, while generic MCMC approaches like Gibbs sampling fail to converge when the coupling reaches or exceeds the critical value. Note that adaptive rejection improves the bound on the region of criticality.

Given the symmetries in the pairwise potentials in (even) a frustrated Ising model without external field - the score is invariant to a full flip of all states - our algorithm will always accept with probability 1 on tree-structured (sub)problems. This is because the combination of Gibbs proposals with the generic sequence choice (connected ordering) can always satisfy the constraints induced by the agreement or disagreement on spins in these settings. Accordingly, our algorithm is more efficient than other exact methods for trees (such as forward filtering with backward sampling) in these cases. If, on the other hand, the target distribution does not contain this symmetry (so some of the initial choices matter), there will be some probability of rejection, unlike with forward filtering and backward sampling. This helps to explain the bands of rejection sometimes seen in the nonadaptive algorithm and the opportunity for adaptation on Ising models, as it is impossible for the algorithm to reject until a variable is added when its already added neighbors disagree.

We also applied our method to the problem of diagnostic reasoning in bipartite noisy-OR networks. These problems motivated several variational inference algorithms and in which topological simulation and belief propagation are known to be inaccurate (66). Furthermore, as in the ferromagnetic Ising setting, it seems important to capture the multimodality of the posterior. A doctor who always reported the most probable disease or who always asserted you were slightly more likely to be sick having visited him would not have a long or successful professional life.

The difficulty of these problems is due to the rarity of diseases and symptoms and the phenomenon of "explaining away", yielding a highly multimodal posterior placing mass on states with very low prior probability. We explored several such networks, generating sets of symptoms from the network and measuring both the difficulty of obtaining exact samples from the full posterior distribution on diseases and the diagnostic accuracy. Figure 3-5 shows exact sampling results, with and without adaptation, for a typical run on a typical network, generated in this regime. This network had 20 diseases and 30 symptoms. Each possible edge was present with probability 0.1, with a disease base rate of 0.2, a symptom base rate of 0.3, and transmission probabilities of 0.4.

The noisy-OR CPTs result in large factors (with all diseases connected through any symptoms they share). Accordingly, the sequential rejection method gets no partial credit by default for correctly diagnosing a symptom until all values for all possible diseases have been guessed. This results in a large number of rejections. Adaptation, however, causes the algorithm to learn how to make informed partial diagnoses better and better over exposure to a given set of symptoms.

Finally, we applied our method to a larger-scale application: approximate joint sampling from a Markov Random Field model for stereo vision, using parameters from (79). This MRF had 61,344 nodes, each with 30 states; Figure 3-6 shows our results. We suspect exact sampling is intractable at this scale, so we used the sequential importance relaxation of our algorithm, leveraging our general recursive machinery to introduce variables one at a time. We compared versions using 1 and 5 particles to widely-used Gibbs sampling approaches. Because of the strong - but not deterministic - influence of the external field, we needed a more informed ordering. In particular, we ordered variables by their *restricted entropy* (i.e. the entropy of their distribution under only their external potential), then started with the most constrained variable and expanded via connectedness using entropy to break ties. This is one reasonable extension of the "most constrained first" approach to variable choice in deterministic constraint satisfaction. The quality of approximate samples with very few particles is encouraging, suggesting that appropriate sequentialization can leverage strong correlations to effectively move through the sample space.

3.4 Discussion

Our experiments suggest that tools from systematic search, appropriately generalized, can mitigate problems of multimodality and strong correlation in sampler design. When variables (and their attendant soft constraints) are incorporated one at a time, a sampler may be able to effectively find high probability regions by managing correlations one variable at a time. This brings some

of the benefits of the divide-and-conquer approach to algorithm design to the broader setting of stochastic simulation. Additionally, any sample produced by sequential rejection is, by definition, exact. Aside from providing comfort to pracitioners nervous about the equilibration rate of their Markov chains, this exactness allows for our method to be embedded as part of larger stochastic processes in places where the theory does not readily admit approximations. For example, one might use adaptive sequential rejection to perform blocked Gibbs sampling on a subset of the variables in a very large graphical model, where use of an approximate variant would introduce multipath problems.

It would be interesting to compare and combine our algorithm with Markov chain methods, yielding hybrid systematic/local algorithms for stochastic simulation. Such work would likely require the theory of sequential Monte Carlo samplers (and in particular, backward kernels) from (15) and (32). In the case of approximate sampling, other work has used deterministic search as the subroutine of a sampler (29), but to the best of our knowledge there has been no other work on samplers that recover search behavior when applied to deterministic problems.

The phase transition plots for Ising models and our (unreported) preliminary experiments applying adaptive sequential rejection to Boolean satisfiability problems (expressed as deterministic MRFs) suggest it would be interesting to compare success probability to known results on phase transitions of hardness of SAT. It would also be interesting to see how the rejection rate and memory consumption of adaptation in our algorithm relate to the cost of dynamic programming (ala junction tree), and to explore the behavior of straightforward blocked variants of our method where multiple variables are added simultaneously.

This chapter only begins to explore the possibilities afforded by properly recursive sampling. For example, many deterministic divide and conquer algorithms operate by more aggressive recursion. The cost of merge sort, for example, can be characterized by a recurrence of the form $T(n) = 2T(n/2) + O(n)$, involving a recursion depth that is logarithmic in the length of the input. Our method recurses with linear depth. Sampling algorithms that leverage conditional independence for more aggressive recursions could potentially be tremendously more effficient than the sampling methods that are typically constructed today.

More generally, we believe the design space for sampling algorithms is far, far larger than people typically consider. Every deterministic algorithm, be it a local search, a systematic search, or some hybrid, is the deterministic limit of a potentially useful sampling algorithm. The control structures routinely leveraged by deterministic algorithm designers are far richer than those that have been exploited in the stochastic setting, even given languages for composable inference (8).

For example, first-order systematic searches like FDPLL (4) dynamically construct and solve abstract subproblems on their path to satisfying solutions. We feel that bringing the tools from this space into the stochastic setting represents a fruitful and pressing area for future research. This control expressiveness may also be of benefit to cognitive modelers, as the execution histories of a recursive sampler sometimes seem to better match the intuitive feel of real-world probabilistic reasoning than the wandering paths taken by stochastic fixed-point iteration.

Deductive reasoning problems - like deterministic constraint satisfaction - are special cases of probabilistic reasoning problems, recovered as soft constraints harden. Sampling naturally bleeds into satisfiability in this limit. Despite this natural embedding, the algorithmic literature on sampling-based inference and deductive reasoning has historically been mainly disjoint, with many practitioners of the opinion that logical reasoning and probabilistic inference are fundamentally different problems that should be tackled with fundamentally different techniques. Instead, we argue that good sampling algorithms can and should automatically become effective algorithms for deduction when appropriate, and that adopting this constraint will lead to more effective algorithms for both probabilistic and logical reasoning.

3.5 Appendix

If the distribution p is not the target distribution but instead the distribution at some intermediate stage in a sequential rejection sampler, the downstream stage will adapt p to match its marginal. We show that these additional factors not only satisfy the existing pre-computed bound c^*, but also that sequential rejection on the adapted distribution p eventually accepts every sample. Consider now that p has its own set of additional factors $x \in R$ for which there are weights $\phi_x \in [0, 1]$. If $x \notin R$, let $\phi_x = 1$. Then

$$w_{p' \to p_R}(y, z) \triangleq \frac{p_R(y, z)}{p'(y)\, q_R(z \mid y)} \tag{3.29}$$

$$= \sum_{z'} \psi_2(y, z') \prod_{x \in R} \phi_x^{\delta_{x(y,z')}} \tag{3.30}$$

and therefore

$$w_{p' \to p_R}(y, z) \leq \sum_{z'} \psi_2(y, z') = w_{p' \to p}(y). \tag{3.31}$$

We claim that $w_{p'_S \to p_R}(y,z) \leq c^*_{p' \to p}$. Let R' and ϕ' be the set of $x = (y,z)$ and weights that have been fed back to p in previous iterations of the algorithm. Consider

$$w_{p'_S \to p_R}(y,z) \triangleq \frac{p_R(y,z)}{p'_S(y)\, q_R(z \mid y)} \tag{3.32}$$

$$= \frac{w_{p' \to p_R}(y)}{\prod_{y' \in S} \frac{w_{p' \to p_{R'}}(y')}{c^*_{p' \to p}}} \tag{3.33}$$

$$= \begin{cases} w_{p' \to p_R}(y) & y \notin S \\ \frac{w_{p' \to p_R}(y)}{w_{p' \to p_{R'}}(y)} c^*_{p' \to p} & y \in S. \end{cases} \tag{3.34}$$

Eq. (3.31) implies that when $y \notin S$, we have $w_{p'_S \to p_R}(y,z) \leq w_{p' \to p}(y) \leq c^*_{p' \to p}$. Therefore, the claim is established for $y \in S$ if $\frac{w_{p' \to p_R}(y)}{w_{p' \to p_{R'}}(y)} \leq 1$. We have that

$$\frac{w_{p' \to p_R}}{w_{p' \to p_{R'}}(y)} = \frac{\sum_{z'} \psi_2(y,z') \prod_{x \in R} \phi_x{}^{\delta_{x(y,z')}}}{\sum_{z'} \psi_2(y,z') \prod_{x \in R'} \phi'_x{}^{\delta_{x(y,z')}}} \tag{3.35}$$

First note that, $x \in R' \implies x \in R$. Therefore, the inequality is satisfied if $\phi'_x \geq \phi_x$ for all x. We prove this inductively. When a value x is first added to R, $x \notin R'$, hence $\phi'_x = 1 \geq \phi_x$. By induction, we assume the hypothesis for ϕ_x and show that $\phi'_y \geq \phi_y$. Consider Eq. (3.34). If $y \notin S$, then $\phi_y = \frac{w_{p' \to p_R}(y)}{c^*_{p' \to p}} \leq \frac{w_{p' \to p}}{c^*_{p' \to p}} \leq 1 = \phi'_y$ by the optimality of c^* and Eq. 3.31. If $y \in S$, we have $\phi'_x \geq \phi_x$ for all x by induction, proving the claim.

Evidently, the weights decrease monotonically over the course of the algorithm. Of particular note is that when $R = \mathcal{X}$, and $S = \mathcal{Y}$. In this case the acceptance ratio is again 1 and we generate exact samples from p_R. Of course, $|R|$ and $|S|$ are bounded by the number of iterations of the algorithm and therefore we expect saturation $|R| = |\mathcal{X}|$ only after exponential work.

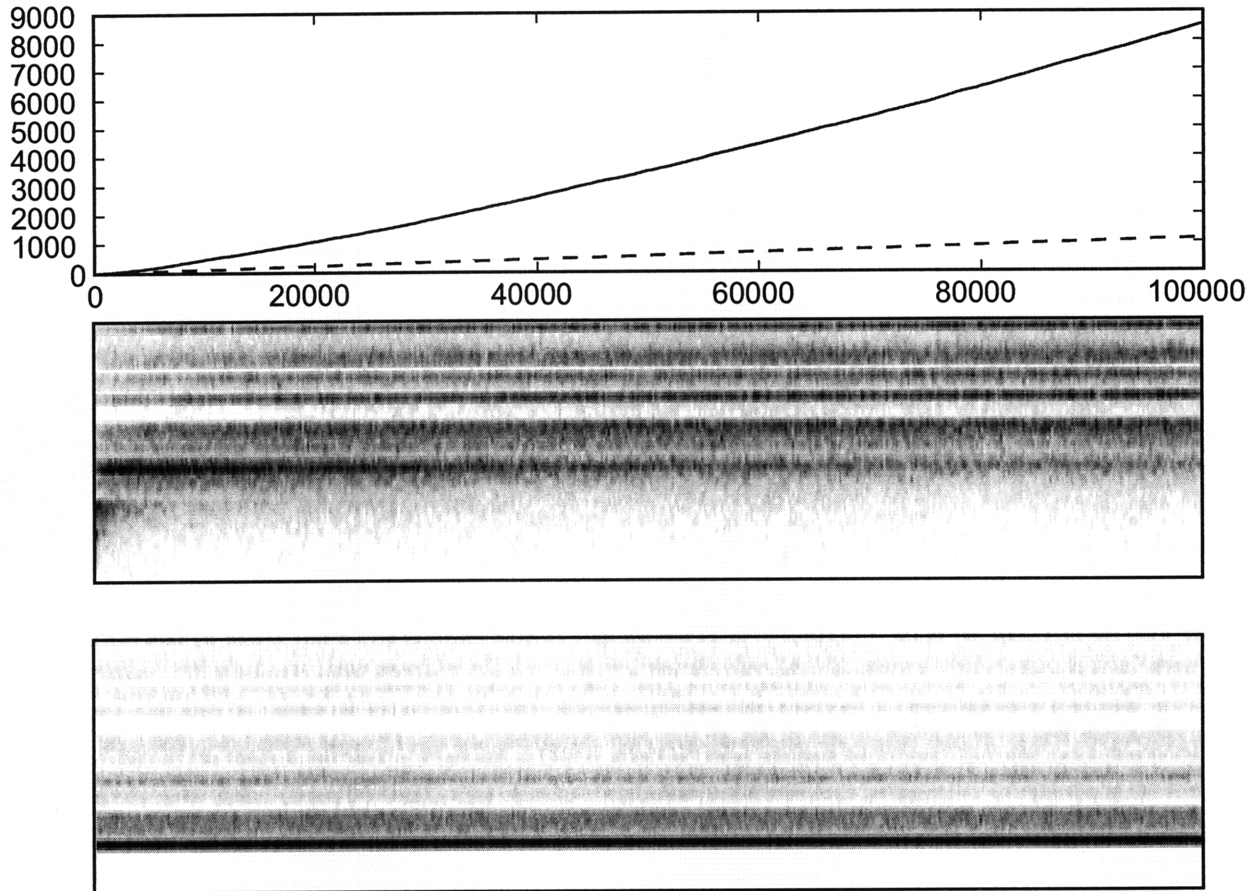


Figure 3-4: Comparison of adaptive (top-red, center) and nonadaptive (top-blue/dashed, bottom) rejection sampling on a frustrated 36-dimensional Ising model with uniform $[-2, 2]$ distributed coupling parameters. (top) Cumulative complete samples over 100,000 iterations. (lower plots) A black dot at row i of column j indicates that on the jth iteration, the algorithm succeed in sampling values for the first i variables. Only a mark in the top row indicates a successful complete sample. While the nonadaptive rejection sampler (bottom) often fails after a few steps, the adaptive sampler (center), quickly adapts past this point and starts rapidly generating samples.

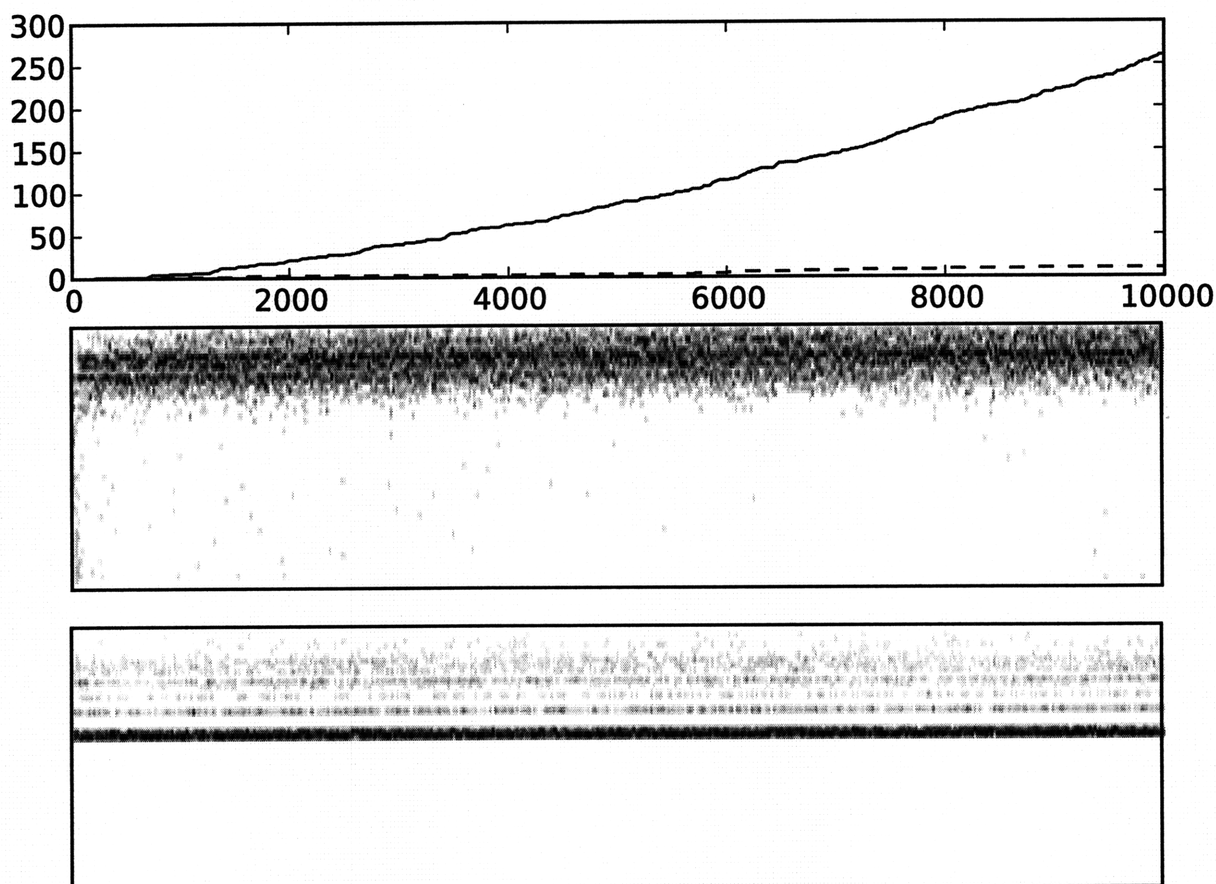
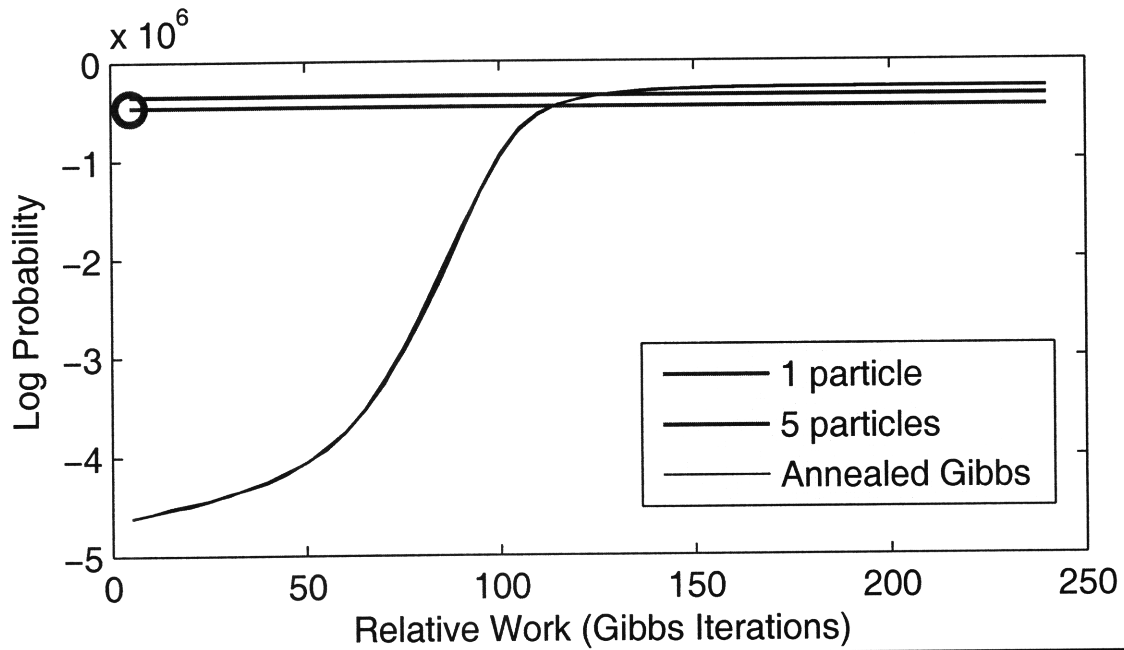


Figure 3-5: Comparison of adaptive (top-red, center) and nonadaptive (top-blue/dashed, bottom) rejection sampling for posterior inference on a randomly generated medical diagnosis network with 20 diseases and 30 symptoms. The parameters are described in the main text. (top) Cumulative complete samples over 100,000 iterations. (lower plots) show the trajectories of a typical adaptive and non-adaptive run in the same format as Figure 3-4. Here, adaptation is critical, as otherwise the monolithic noisy-OR factors result in very low acceptance probabilities in the presence of explaining away.



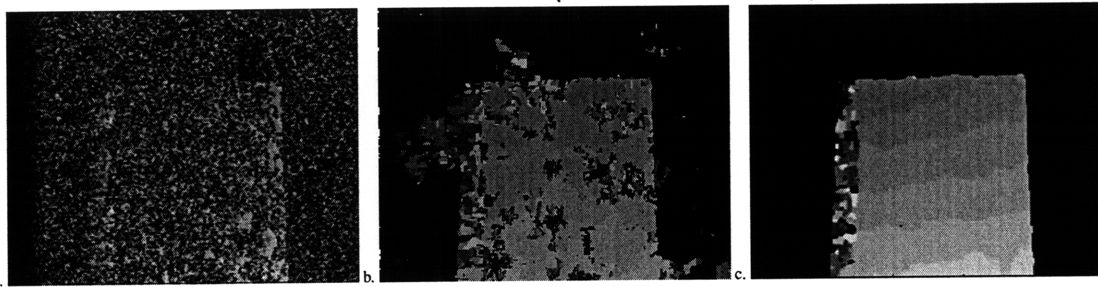


Figure 3-6: Comparison of an aggressively annealed Gibbs sampler (linear temperature schedule from 20 to 1 over 200 steps) to the non-adaptive, importance relaxation of our algorithm. The red circle denotes the mean of three 1-particle runs. The horizontal bars highlight the quality of our result. (a) Gibbs stereo image after sampling work comparable to an entire 1-particle pass of (b) our algorithm. (c) Gibbs stereo image after 140 iterations.

Chapter 4

Stochastic Digital Circuits for Probabilistic Inference

> "I basically know of two principles for treating complicated systems in simple ways; the first is the principle of modularity and the second is the principle of abstraction. I am an apologist for computational probability in machine learning, and especially for graphical models and variational methods, because I believe that probability theory implements these two principles in deep and intriguing ways – namely through factorization and through averaging. Exploiting these two mechanisms as fully as possible seems to me to be the way forward in machine learning."
>
> – Michael I. Jordan, quoted in *Graphical Models for Machine Learning and Digital Communication*

4.1 Introduction

Structured stochastic processes play a central role in the design of approximation algorithms for probabilistic inference and nonlinear optimization. Markov chain (50; 27) and sequential (17) Monte Carlo methods are classic examples. However, these widely used algorithms - and probabilistic reasoning and Bayesian statistics in general - can seem unacceptably inefficient when simulated on current general-purpose computers.

This high apparent cost should not be surprising. Computers are based on deterministic Boolean circuits that simulate propositional deduction according to the Boolean algebra (9; 69), while problems of inference under uncertainty - and many stochastic algorithms for solving these problems -

are best described in terms of the probability algebra (36). To perform probabilistic inference on computers based on all-or-none, deterministic logic circuits, one typically rewrites algorithms in terms of generic real-number arithmetic, which is then approximated by general-purpose Boolean circuits for floating-point arithmetic (40). This indirection has many disadvantages: it obscures fine-grained parallelism, complicates algorithm analysis, and is needlessly costly in both time and space.

In this chapter, I present an alternative approach, based on a novel circuit abstraction called *combinational stochastic logic*. Combinational stochastic logic circuits stand in relation to the probability algebra as Boolean gates do to the Boolean algebra. Every single-output combinational Boolean circuit evaluates the truth value of some propositional sentence, given the truth values of its inputs. Analogously, every single-output combinational stochastic logic circuit *samples* the truth value of some propositional sentence from its associated probability, given sampled truth values for its inputs. As in Church, we are choosing to represent distributions using samplers, rather than probability evaluators, recovering function evaluators as a deterministic special case. In this case, however, our representation language is graph-based, making time, space and bit-width requirements explicit - we are playing the same game at the level of circuits, rather than the Lisp.

We make three contributions. First, we show how combinational stochastic logic circuits generalize Boolean logic, allowing construction of arbitrary propositional probabilistic models. Second, we combine our stochastic logic gates with ideas from contemporary digital design, showing how to build stochastic finite state machines that implement useful sampling algorithms. In particular, we show how to directly implement MCMC algorithms for arbitrary Markov random fields in hardware in a massively parallel fashion. Finally, we estimate the performance of our approach when implemented on commodity reconfigurable logic, finding substantial improvements in time efficiency, space efficiency and price. We also show that stochastic logic circuits can perform robustly in the presence of a range of transient and persistent faults, suggesting interesting possibilities for distributed computing on unreliable substrates.

4.2 Stochastic Logic Circuits

Our central abstraction, *combinational stochastic logic*, generalizes combinational – or stateless – Boolean circuits to the stochastic setting, recovering Boolean gates and composition laws in the deterministic limit. A Boolean gate has input bit lines and output bit lines, and puts out a Boolean

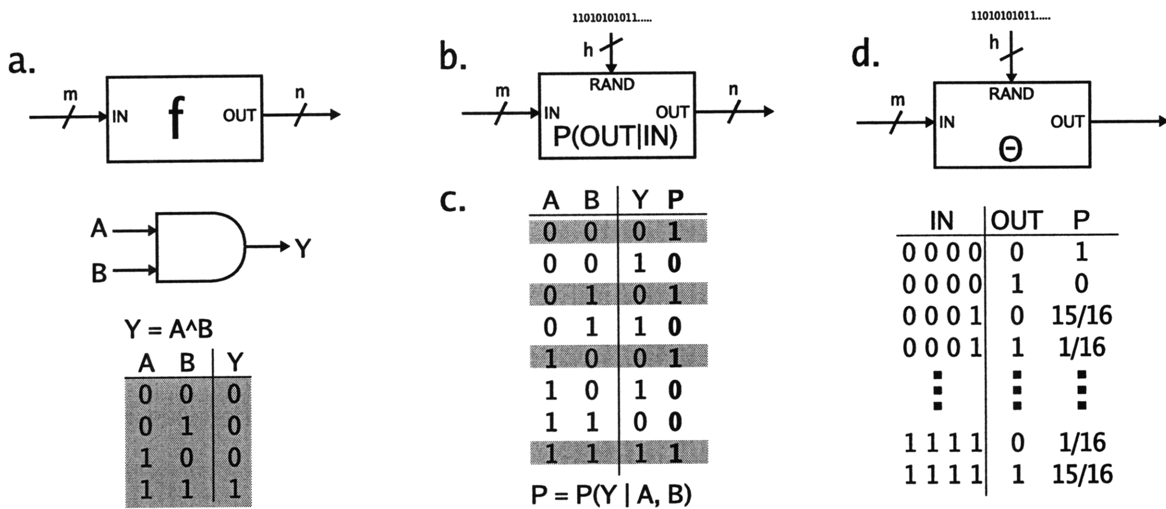


Figure 4-1: **Combinational stochastic logic.** (a) The combinational Boolean logic abstraction, and one example: the AND gate and its associated truth table. (b) The *combinational stochastic logic* abstraction. On each work cycle, samples are drawn on OUT from $P(\mathtt{OUT}|\mathtt{IN})$, consuming h random bits on RAND to generate nondeterminism. (c) An AND gate can be viewed as a combinational stochastic logic gate that happens to be deterministic. (d) The conditional probability table and schematic for a Θ gate, which flips a coin whose weight was specified on IN as a binary number (e.g. for IN = 0111, $P(\mathtt{OUT} = 1|\mathtt{IN})$ = 7/16). Θ gates can be implemented by a comparator that outputs 1 if RAND $\leq$ IN.

function of its inputs on each work cycle. Each gate is representable by a set of truth tables, one for each output bit; the abstraction and an AND gate example are show in in Figure 4-1a. Figure 4-1b shows a combinational stochastic logic gate, which adds random bit lines. On each cycle, the gate puts a sample from $P(\mathtt{OUT}|\mathtt{IN})$ on its output lines, using the random bits – which must each be flips of a fair coin – to provide the nondeterminism. Just as Boolean gates can be represented by families of truth tables, individual stochastic gates can be represented by conditional probability tables (CPTs), where all the probabilities are rational with finite expansions in base 2.

By explicitly representing the bitwidths of values and the entropy requirements per sample from each CPT, we can directly map stochastic gates onto discrete, physical machines for performing computation. Figure 4-1c shows how to recover deterministic Boolean logic gates by zero-entropy CPTs, using the AND gate as an example. Figure 4-1d shows the conditional probability table and schematic for a unit called Θ, which generates flips of a weighted coin whose weight is specifed on its IN lines. The Θ gate is one important, recurring stochastic primitive, and the designs for properly stochastic units that we have developed so far depend heavily on its use. We note that for any combinational stochastic logic element, it is possible to abstract away the entropy lines; this is

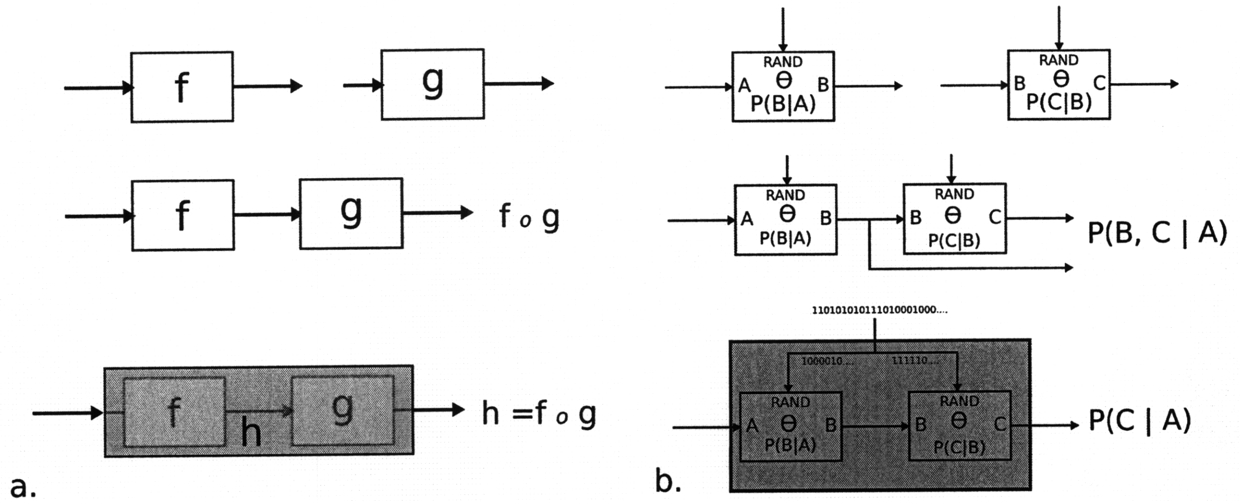


Figure 4-2: **Composition and abstraction laws.** (a) Boolean gates support expressive composition and abstraction laws. The law of composition states that any two Boolean gates f and g with compatible bitwidths can be composed to produce a new Boolean gate h. The law of abstraction states that h can now be used in designing further Boolean circuits without reference to the components f and g that it is composed of. (b) The analogous laws for combinational stochastic logic: one can sample from joint distributions built out of pieces, or view a complex circuit abstractly as a primitive that samples from a marginal distribution. Note that in this case the input entropy is divided among the two internal elements; the abstraction preserves the notion of a single incoming source of randomness, even though internally the elements receive effectively distinct streams.

implementable in a variety of ways, using pseudorandom or physical entropy sources bundled with the stochastic elements. However, we prefer to emphasize both the simulability of these elements on a deterministic logic substrate and to explicitly track flow of entropy throughout the computation, both to facilitate debugging of stochastic circuits in simulation and because we suspect the entropy flow through a circuit may relate to its fundamental physical resource requirements.

Boolean gates support expressive composition and abstraction laws, shown in Figure 4-2, physically reflecting the composition and abstraction laws supported by Boolean functions. By repeatedly composing and abstracting, digital designers build up adders, multipliers, and complex switching networks out of basic Boolean logic operations. Adding stateful elements then leads to flip-flops, accumulators and vector memories, ultimately leading to universal processors. The ubiquity of digital logic depends critically on these laws: they allow designers to build on each

others' work, without having to concern themselves with the details of each others' implementations. For example, recent pentiums have had over 800,000 Boolean gates, and VLSI circuits are routinely designed by teams of tens to hundreds of engineers using programs that automate aspects of the low-level design. These artifacts, collaborative processes and software programs rely critically on the fact that compositions of Boolean elements can be analyzed simply in terms of their pieces.

We have developed stochastic generalizations of these basic laws. Feeding the output of one gate into the input of another results in samples from the joint distribution of the two elements, allowing construction of samplers for complex distributions from simpler pieces. Furthermore, one can abstract away the details of a complex stochastic circuit, viewing it as a single combinational stochastic gate that simply generates samples from the *marginal* distribution of the original output gate's value given the original input gate's input value. Taken together, these laws support the construction of arbitrarily complex probabilistic (and Boolean) systems out of reusable components. In this chapter, we will start with stateless Θ gates that flip weighted coins, and build up to circuits for general purpose, MCMC based approximate inference in factor graphs. The detailed design of circuits for systematic algorithms (like sequential Monte Carlo and sequential rejection sampling) and for conditional sampling over recursive stochastic processes or more structured stochastic processes will be largely left to future work.

4.2.1 Sampling is linear, while probability evaluation is exponential

It is important to note that our choice to generalize function evaluation to sampling - rather than, for example, probability computation - is a critical enabler of the composition and abstraction properties stated here. This is because sampling inherits the complexity structure of evaluation, as follows. The cost of evaluating $g \circ f(x) = g(f(x))$ is at most equal to the sum of the cost of evaluating f and the cost of evaluating g, in time and space on a serial computer and in circuit depth and width. On a serial machine, one stores $f(x)$ in some temporary location y, then computes $g(y)$. In a circuit, one feeds the output of a circuit that computes f as the input into a circuit that computes g. More generally, then, the cost of a k-step composition (or a k-stage circuit, if you like) is only linear in k.

In the stochastic setting, evaluating the probability of the composition of two conditional distributions requires computing a marginal probability, $P(C|A) = \sum_B P(C, B|A) = \sum_B P(C|A, B)P(B|A)$. If our basic units can only answer probability queries on input/output pairs, then constructing a

composite unit out of pieces will require a sum equal to the number of possible states of the intermediate quantity in the composition. This means that the cost of evaluating $P(C|A)$ would be the *product* of the costs of $P(C|A,B)$ and $P(B|A)$, and that more generally, the cost of k-step compositions grows exponentially in the depth k. This cost, which would be paid in the space and complexity of a circuit that iterated over possible outcomes, say, by time-division multiplexing, and in the time complexity associated using such a circuit, is prohibitive. Sampling, on the other hand, brings the cost back to the linear one we expect, as we can generate samples from the marginal distribution $P(C|A)$ by sampling from $P(B|A)$ and feeding that sample as input to a sampler for $P(C|A,B)$.

This complexity difference is one fundamental argument in favor of viewing probabilistic systems as a generalization of deterministic ones, via stochastic simulation, instead of trying to embed them inside deterministic ones by probability calculation. The ability that samplers provide to tractably represent exponentially large objects - and to estimate global properties of these objects in constant time, via the Monte Carlo method, escaping the "curse of dimensionality" - may be the key to computationally tractable probabilistic reasoning.

4.2.2 Probabilistic Completeness and Finite Precision

An important aspect of Boolean logic circuits is their logical completeness. That is, we can build a Boolean circuit using AND, OR and NOT gates that evaluates the truth value of any Boolean formula that uses a finite set of propositions (69). This corresponds to the class of Boolean functions. To see this, first enumerate all Boolean functions on k input bits. There are 2^k distinct inputs, and the function could output a 0 or a 1 for each of these. Thus there are 2^{2^k} distinct Boolean functions, listable in dictionary order. For any function in this list, we first construct a sentence in the propositional algebra with k atomic propositions that is true exactly when the function is 1 and false otherwise. One way to do this is via turning it into a sum of products in the Boolean algebra, also known as writing it in disjunctive normal form. This yields a c-clause formula, where each clause uses AND and NOT to pick out exactly one unique setting of the k inputs where the function is 1, and the clauses are connected by ORs. By construction, this formula is true for a given setting of the input propositions if and only if the function is 1 for the analogous inputs. We can then construct a Boolean circuit by replacing each variable in the formula with a wire (or "node") in the circuit, and each logical connective with its corresponding Boolean gate. While we can often find smaller circuits by, for example, applying the laws of the Boolean algebra to the

normal form representation above, this "lookup table" representation is logically sufficient. Given this ability to compute Boolean functions, we can build machines to compute any *discrete* function simply by fixing a coding scheme that maps the discrete outputs to sequences of 0s and 1s and then computing a bank of Boolean functions using the above method, one for each bit.

Assuming one can implement Θ gates with arbitrarily precise probabilities[1], we can show the analogous probabilistic completeness properties of our stochastic logic circuits. We build a Θ gate for each of the 2^k input values, fixed to their base probabilities, and use a standard digital multiplexer to select the output. Generalizations to l-bit outputs are straightforward though costly in circuit width; circuits which compress the table in terms of latent variables will be more efficient in general.

There are straightforward ways to implement arbitrarily precise Θ gates in either bounded time but unbounded space and entropy, by adding more wires, or bounded space but unbounded time, by arithmetic decoding or some other bitwise, iterative scheme. However, in practice, as with all high-precision or nearly smoothly varying quantities on current digital computers, we will typically approximate our probabilities to some specific, sufficient precision.

We expect the actual probability precision needed for exact and approximate sampling algorithms in typical machine learning applications will be low for four reasons. First, fixed parameters in probability models are known only to finite precision, and often only to within an order of magnitude. Second, sampling variance masks small differences in sampling probabilities; approximation error will often be literally in the noise for approximate inference algorithms. Third, most approximate sampling algorithms (e.g. sequential and Markov chain Monte Carlo) depend on ratios of probabilities and weights to obtain the probabilities sampled from during simulation, further pushing dependence up to high order bits. Fourth, expected utility decisions only depend on the low order bits of estimated expectations as the outcomes become increasingly indistinguishable. We recognize that some applications (such as satellite tracking or ab initio physical calcuations) may require substantially higher precision. Obtaining tight analytical bounds on the precision requirements of stochastic circuits for exact and especially approximate sampling is an important open challenge, recovering the classic Boolean circuit minimization problem in the deterministic limit.

[1]That is, Θ gates for arbitrarily finely specified coin weights

4.2.3 Stochastic Circuit Complexity and Design Patterns

We now consider some recurring patterns in probability and stochastic processes, directly translating them to useful designs for stochastic circuits. Our first example is a binomial distribution; we use the conditional independencies in the process to show time and space tradeoffs. For example, Figure 4-3a shows a circuit for sampling from a binomial distribution on n coin flips using $O(log(n))$ time and $O(n)$ space by both sampling and adding in parallel (via a logarithmic adder tree implementation of +). Figure 4-3b shows another circuit for the same problem using $O(log(n))$ space and $O(n)$ time, operating by serial accumulation. Both require $O(n)$ bits of entropy.

The question of the intrinsic time, space and entropy - really, circuit depth, "width" and entropy - requirements for sampling seems like a particularly interesting direction for future work, as it gives us a new view of the hardness of sampling with all the benefits associated with a fully discrete circuit model of computation. For example, we can quantitatively study the hardness of sampling in terms of depth, width and entropy. For some distributions, we may be able to do no better than the standard upper bound resulting from a strategy of "build an exponentially large look-up table beforehand", but for others, stochastic circuit minimization may yield far more efficient alternatives. Furthermore, the non-uniformity of circuit models of computation seems especially appealing in the current age of massive parallelism. For example, Markov chain Monte Carlo algorithms build up approximate samples from distributions over exponentially large spaces by building on the ability to rapidly generate samples from smaller spaces. Whether or not they will be efficient depends in a large part on exactly how big the smaller spaces are, which will determine the amount of physical space we need to make simulation from those smaller spaces nearly instantaneous. More generally, as we start exploiting finer and finer time-space tradeoffs, we find that constant factors matter more and more; a circuit model for sampling, rather than for approximately evaluating binary decisions (as in BPP), may help us gain traction on these issues.

This view, and the binomial example, also exposes a fundamental property of sampling with potentially significant implications for computational tractability of probabilistic inference: *every conditional independence in a given stochastic process yields an opportunity for parallel evaluation*, or the shortening of circuit depth at the cost of circuit width. We will exploit this property in the main application later on in this chapter. A simple binomial distribution is embarassingly parallel, at the level of individual coin flips, or random bits. Many simple distributions can be simulated in a variety of ways, exploiting decompositions that reflect different independencies. Each such decomposition yields different time and space tradeoffs, which may be appropriate for

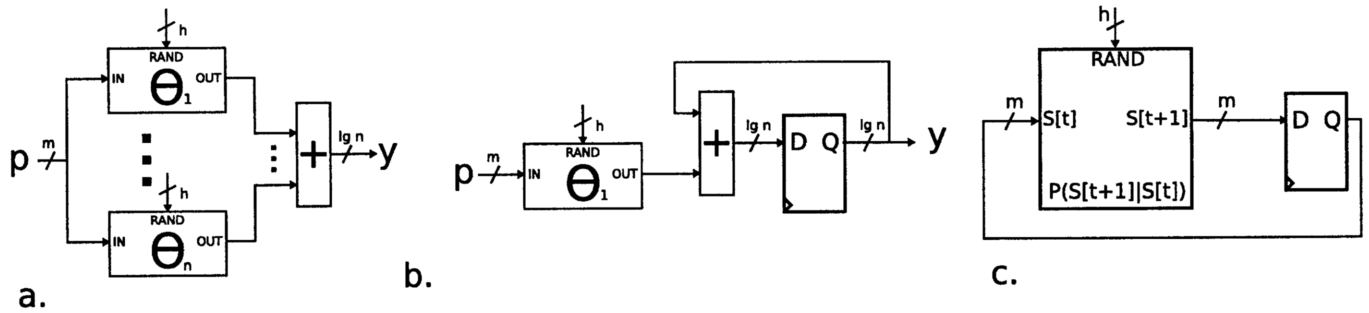


Figure 4-3: **Example stochastic circuit designs.** (a) and (b) show two circuits for sampling from a binomial distribution on n flips of a coin of weight p, consuming nh total bits of entropy. (a) shows a circuit where the coins are flipped in parallel and then summed, costing $O(n)$ space and $O(log(n))$ time per sample. (b) shows a serial ciruit for the same problem, using $O(log(n))$ space (for the accumulator) and $O(n)$ time. Clocked registers are shown as units with inputs and outputs labeled D and Q. (c) shows a stochastic finite state machine (or finite-state Markov chain), with a state register connected to a combinational state transition block.

different situations. Large probabilistic systems built up out of large numbers of coin flips will in general exhibit many opportunities for fine-grained space/time tradeoffs, and thus guide the circuit microarchitecture of machines for stochastic simulation.

Many more interesting circuits become possible when we combine stateless circuits with standard tools from contemporary digital design for maintaing state. For example, we can implement rejection sampling by combining a purely combinational proposal sampler circuit with a Boolean predicate as part of a state machine that loops until the predicate is satisfied. This circuit uses bounded space but possibly unbounded (and in general exponential) time. To implement MCMC, an approximate sampling method, we can combine a combinational circuit that samples from the MCMC transition kernel with a register that stores the current state of the chain. Figure 4-3c shows the circuit structure of a generic stochastic finite state machine (FSM) applicable to these sampling algorithms. The FSM could be implemented by storing the current state in a clocked register, with a stateless block consuming random bits to sample one next state via a stochastic transition model on each cycle. We will apply this design idiom to perform approximate joint sampling from large Markov random fields in the next section.

4.2.4 Approximate Inference using MCMC and Gibbs processors

We focus on tempered Gibbs sampling algorithms for Markov random fields (MRFs) because they are simple and general but usually considered inefficient. Many other Monte Carlo recipes, including Metropolis-Hastings and sequential importance sampling, can also be built using the same techniques.

To implement a Gibbs MCMC kernel for a given variable, we must score each possible setting given its neighbors under the joint density of the MRF, temper those scores, compute the (log) normalizing constant, normalize the energies, convert them to probabilities, and generate a sample. This pipeline is shown in Figure 4-4a, and can be implemented in linear time in the size of the variable by standard techniques combined with a simple stochastic accumulator for sampling (which can be thought of as the k-outcome generalization of a Θ gate, for generating draws from arbitrary discrete distributions). However, when the CPT for the variable has sufficiently small size, we can do better: by precomputing CPTs and using the design in 4-4b, we can obtain samples in constant time. This will be tractable whenever the energy precision requirements are not too high and the degree of the MRF is not too large[2].

We can then combine Gibbs kernels into massively parallel Gibbs samplers by exploiting conditional independencies in the MRF. Specifically, given a coloring of the MRF (an assignment of colors to nodes so no two adjacent nodes have the same color), all nodes of each color are conditionally independent of each other given all other colors, and thus can be sampled in parallel. This was first observed in (27) for square-lattice MRFs. Figures 4-4c and 4-4d show two example colorings. The degree of parallelism depends inversely on the number of colors, and the communication cost between Gibbs units is determined by the total bits crossing coloring boundaries in the MRF. Figure 4-4e shows a massively parallel circuit built out of Gibbs units exploiting a graph coloring, clocked in a distributed fashion to implement the two-phase structure of a parallel cycle of single site Gibbs kernels. For very large models this pattern can be tiled arbitrarily, preserving constant time Gibbs scans independent of lattice size at linear space cost in lattice area.

Gibbs sampling circuits can be viewed as breaking down the problem of generating a sample from the full joint distribution on variables into generating samples from individual variables in sequence and waiting for convergence. Of course, groups of nodes in a Markov Random Field

[2]We note that these nonlinear dependencies on size parameters provide further support for the appropriateness of a circuit model for probabilistic computation. Sampling methods become computationally favorable as representations for distributions as soon as the dimensionality of the distribution gets high enough that the exponential space or time costs associated with other representations become infeasibly large.

can be collapsed into supernodes, resulting in exponential growth in the domain of the variables, but potentially exponential savings in convergence times. Running a regular Gibbs sampler on the collapsed problem is the same as running a "blocked" Gibbs sampler on the original problem. This can sometimes result in a net savings improvement, since the bases of the exponents might be different. For example, one might only need to increase the cost of individual Gibbs kernels by a factor of 1000 - in time or in space - to improve convergence by a factor of 100,000. This is especially likely when blocking variables changes the dependency structure of the graph in ways that significantly improve its coloring number. Also, since the bandwidth needed between Gibbs kernels increases linearly with the number of nodes being blocked, such an approach might actually substantially increase exploitable parallelism.

More generally, we can see that Gibbs samplers can be viewed as a kind of parameterizable processor for building sampling algorithms, where we pay in space to view certain exponentially costly operations that have low exponents as cheap in time. The size and number of these processors will determine what problems are tractable on a given probabilistic computer, potentially changing tractability by an exponential factor if space is truly plentiful.

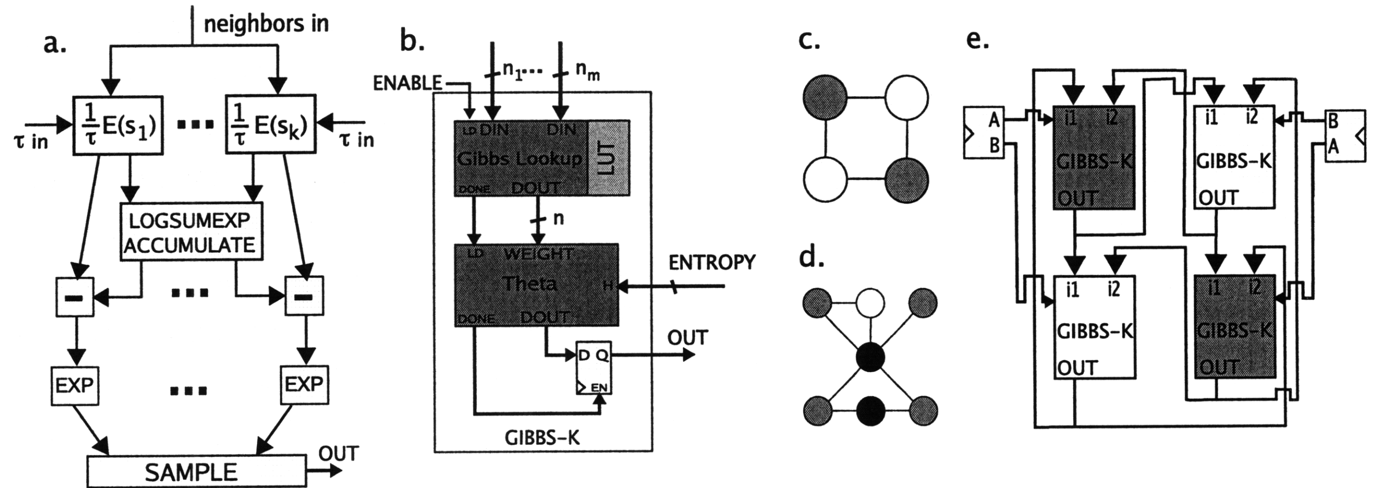


Figure 4-4: **Designs for Gibbs samplers.** (a) shows a schematic Gibbs pipline, outlining the operations needed to numerically sample a single arbitrary-size variable. (b) shows a condensed implementation of a Gibbsable binary variable unit, which internally stores both a current state setting and a precomputed CPT lookup table (LUT), and runs in 3 clock cycles. (c) and (d) show colored MRFs, where all variables of each color can be sampled in parallel. (e) shows a distributed circuit that implements Gibbs sampling on the MRF in (c) using the Gibbs units from (b).

4.2.5 Implementation via commodity FPGAs

To estimate the performance of stochastic circuit designs on current physical substrates for computation and compare to widely available general purpose processors, we implemented our stochastic circuits on Xilinx Spartan 3 family Field Programmable Gate Arrays (FPGAs). FPGAs provide digitial logic elements that can be reprogrammed arbitrarily to directly model any digital circuit. They are widely used in consumer electronics and signal processing because they offer lower development cost compared to traditional application specific integrated circuits (ASICs).

Stochastic circuits nominally require large quantities of truly random bits. However, in practice, almost all Monte Carlo simulations use high-quality pseudorandom numbers, which can be produced by well-known methods. For our FPGA implementation, we use the 128-bit XOR-SHIFT pRNG from Marsaglia (48), which has a period of $2^{128} - 1$, directly implementing one pRNG in digital logic per stochastic element in our circuit (each initially seeded differently). This implementation was chosen because of economic expediency, for debuggability (because pseudorandom bitstreams can be replayed) and because it highlights those wins that come from structuring a circuit for stochastic simulation according to our abstractions independent of the economics of the semiconductor industry. We expect physically stochastic implementation will result in substantial additional design wins.

Since logic utilization influences circuit cost, energy efficiency, and speed, we briefly mention some techniques we use to compress the logic in our circuits. We can use these ideas whenever the analytical relationships and conditional independencies that directly lead to exact, compact sampling circuits are unavailable, as is often the case in MCMC and SMC proposal generation. The key is to represent *state values*, *energies* (i.e. unnormalized log probabilities), and *probabilities* in a fixed-point form, with m bits for the integer parts of energies, n bits for the decimal part, and $1 + m + n$ total bits for probability values. We then compactly approximate the `logsumexp(`e_1`,`e_2`)` function (to add and normalize energies) and the `exp(`e_1`)` function (to convert energies to probabilities), and sample by exact accumulation. We note that numerically tempering a distribution - exponentiating it to some $\frac{1}{\tau}$ - can be parsimoniously implemented as energy bit shifting, for dyadic τ. Heating a distribution by 2^k causes k low-order bits of energy to be lost, while cooling a distribution causes low-order bits to become arbitrarily significant.

We have also built a compiler that produces finite-precision parallel automata for sampling-based inference in factor graphs that are bit-accurate representations of our circuits, substantially reducing development time. I defer discussion of this compiler to the next chapter.

4.2.6 Related Work

The pioneering work of von Neumann (85) and Gaines (25) addressed the reliable implementation of Boolean algebra and arbitrary real arithmetic using stochastic components. Our work is different in motivation and in application: we have introduced methods for engineering large-scale, probabilistic systems, without indirection through generic real arithmetic, which can be used with both deterministic and noisy substrates.

Adaptive or noise-resistant circuit implementations made from stochastic elements have arisen in analog VLSI (28) (53), ultra low power digital logic (via "probabilistic CMOS" (11)), and self-assembled nanoscale circuit fabrication (61). Our work is at a different level of abstraction, providing complete, compositional specifications of stochastic yet digital circuits for probabilistic arguments and circuit patterns for sampling algorithms. However, our stochastic circuits could be implemented on these substrates, potentially yielding cheaper, more efficient circuits than is possible with standard digital semiconductor techniques. Finally, in mainstream digital design, various application specific accelerators for particle filtering have been explored; see (7) for one detailed example. These efforts have focused on efficient parallel architectures for particle advancement and resampling, using classical methods – not direct rendering of the structure of stochastic processes into digital hardware – to simulate from forward models and compute weights.

4.3 Performance Estimates

We synthesized parallel Gibbs circuits on Spartan 3 family FPGAs, measuring the clock rate achieved, clock cycles per sample, and circuit space costs. Figures 4-5b and 4-5c show our results for a circuit for Gibbs sampling on a binary, square-lattice Markov Random Field, using the Gibbs lookup table design. We show estimated price/performance curves for 16 bit samplers. In many applications, quantization error due to 16-bit truncation of probabilities for binary variables will be washed away by noise due to Markov chain convergence, Monte Carlo variance, and model uncertainty. Each horizontal step corresponds to a range of problem sizes that fit on the same number of FPGAs; this tiling, and therefore our price/performance results, should actually be achievable in large-scale practice (with only a 3 clock-cycle overhead for FPGA to FPGA communication).

We include performance estimates for parallel sampling on microprocessors, to highlight the gains coming from *combining* parallel sampling with direct simulation in hardware. These estimates generously assume zero cost in time and dollars for interprocessor communication, and 20

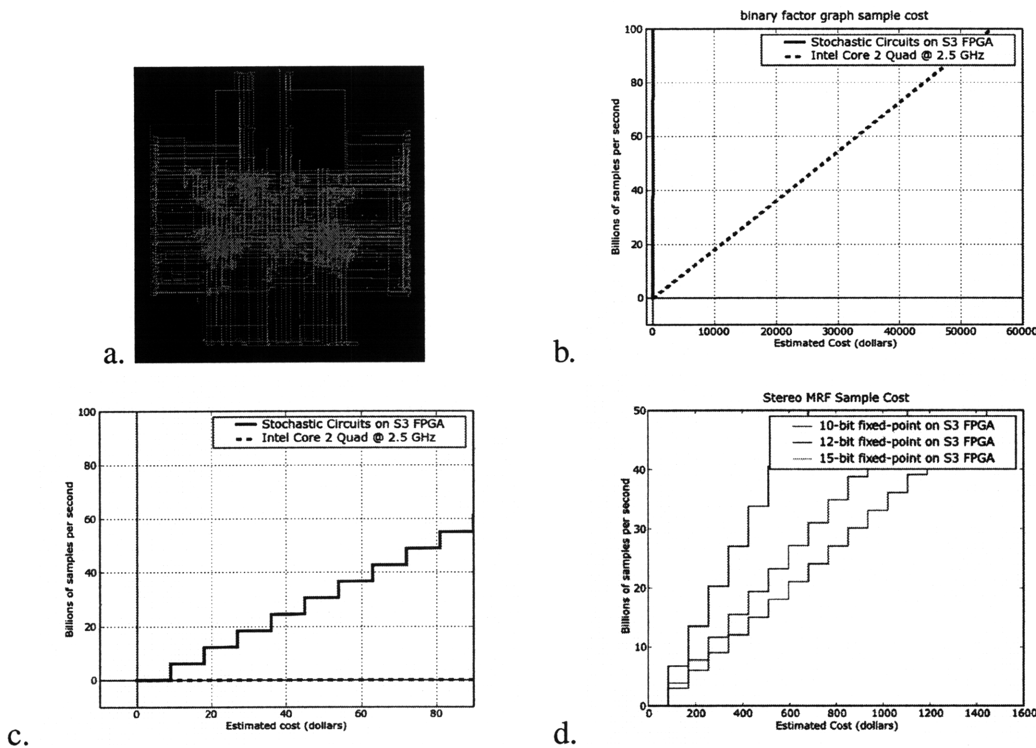


Figure 4-5: **FPGA price/performance estimates.** (a) shows an example synthesized FPGA layout for Gibbs sampling on a 9x9 lattice. (b) compares the price/performance ratio of a stochastic circuit to an optimistic estimate for conventional CPUs, in billions of single-site samples per second per dollar, for a binary square-lattice MRF. (c) shows a zoomed-in comparison. (d) shows price/performance results for stereovision. Approximately 10 billion samples per second are needed for real-time (24 FPS) performance at moderate (320x240 pixels) resolution.

clock cycles per sample, ignoring the serial costs due to memory accesses, branch prediction, cycle cost of floating-point operations, and random bit generation. In our experience the actual performance in practice of even highly efficiently programmed parallel Gibbs samplers on conventional CPUs is lacking, with these costs playing a substantial role. Since a conservative MCMC run typically entails hundreds of thousands of complete Gibbs scans, our circuits should make it possible to affordably obtain reasonable solutions to models with hundreds of thousands of variables in real time.

Another way to view these design wins is that massive parallelization - simultaneous execution of hundreds of thousands to millions of densely communicating "processes" - is only economical when the operations to be done in parallel are very simple. In that limit, parallel computation needs to look more like circuit design to be economically (and physically) sensible. Highly pipelined,

high precision numerical computation is just too costly in silicon area. However, by building stochasticity into the foundations of digital logic, we find that the operations needed for stochastic simulation and probabilistic inference are actually exceedingly simple, and map well onto small, fast, distributed circuits.

Figure 4-5d shows price/performance estimates for a realistic application: annealed Gibbs sampling on an MRF for stereovision (80). The model has 32 distinguishable disparities per pixel. Figure 4-6 gives a rough sense of the quality of the stereo results and their dependence on numerical precision. Our gains should allow generic, annealed Gibbs sampling for discrete variable MRFs to support affordable, dense, real-time stereovision (using standard digital techniques to stream calibrated images into the registers storing MRF potentials on an FPGA). For example, a 320x240 image requires $\sim$380 million single-site samples per frame, assuming 5000 full Gibbs sweeps of the image for MCMC convergence. With $\sim$\$300 of hardware, we should be able to solve this problem at 24 frames per second in 15-bit precision. For different models, the time per sample increases roughly linearly with MRF clique size and with graph coloring number, as individual sites take longer to sample and fewer sites can be sampled in parallel. Thus the complexity of the knowledge structure directly constraints the complexity of inference, reflecting its conditional independencies.

We have endeavored to make our price-performance comparison as meaningful as possible, comparing the off-the-shelf price for both commodity x86 hardware and commodity FPGAs. This does not take into account the (nontrival) cost of support hardware, such as PCBs, power, and cooling. It is not clear that this is an advantage on either side – while commodity x86 motherboards are cheaper than low-quantity custom-designed FPGA PCBs, it is also much easier to add dozens of FPGAs to a single PCB, which no commodity x86 motherboards support.

4.3.1 Robustness

> "The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong, it usually turns out to be impossible to get at or repair."
>
> – Douglas Adams

Classical digital logic yields state machines that are unstable to even transient faults: a one-bit error in a state representation can yield a next state that is arbitrarily far from the desired next state

in Hamming distance. This instability is inherited from the brittleness of combinational Boolean circuits, where the underlying design contract does not allow for the possibility of occasional error due to its fundamental determinism, and further amplified by the serial design idioms encouraged by deterministic digital design.

We expect stochastic circuits to be fundamentally more robust in many cases. First, even for serial designs, although a one-bit error - transient or persistent - might result in a particular sample being sampled from an incorrect distribution, the computation is specified in terms of a *distribution* on outputs. This specification space has a continuous topology, unlike deterministic circuits, whose specification space has a discrete topology. It is therefore at least possible for specifications of our digital circuits to be meaningfully violated by degrees, leaving wiggle room for errors to only distort the resulting answer. Second, if we attempt to analyze the behavior of a faulty stochastic circuit, we can bring the tools of probability to bear. The uncertainty we have as designers about the presence or absence of persistent faults from faulty fabrication, along with our best probabilistic model of the transient faults our devices will be subject to in a given environment, can be modeled by simply modifying the abstract model for the stochastic elements that we use. For example, a semiconductor implementation where a node shorts to ground with some probability on each clock cycle, say due to high temperature or cosmic rays, could be modeled as a weighted mixture between the desired FSM transition and one that includes a randomly chosen node fault.

Beyond the theoretical possibility that our stochastic circuits might exhibit robust behavior, we expect that the design idioms we have advocated - distributed finite state machines implementing methods like Markov chain Monte Carlo - will further encourage robustness behavior. First, our circuits are naturally distributed, translating the conditional independencies in the underlying stochastic process into high width. Accordingly, deviations from the specification in the state update stochastic logic will only directly influence the probabilities for some of the elements of the state vector, not all of them. Second, Markovian fixed-point iteration (as in ergodic convergence of MCMC) yields a "restoring force" that reduces the impact of transient faults over time and encourages local consistency despite persistent faults. That is, at each iteration, the local stochastic computation for each variable is (probabilistically) moving in a direction more consistent with its neighboring variables, under the local constraints from the global probability model. This encourages recovery from extreme transient faults.

We quantitatively explore these properties on a small but illustrative example shown in Figure 4-7, using numerical techniques to exactly compute the equilibrium distribution on states for a parallel Gibbs circuit with stochastic transient faults, where the state value for a given site is flipped

with some probability. The circuit performs inference in a butterfly-structured attractive binary Markov random field, and thus has a potential central point of failure. Deterministic digital design normally requires Boolean primitives to be extremely reliable, e.g. fault probabilities around 10^{-8}. Here we see that with fault probabilities of 10^{-2}, the equilibrium distribution of our Gibbs circuit is very close to the target, and is still reasonable even with 50% fault probability per site per cycle. If faults are restricted only to the critical central site, performance only degrades slightly, due to the Gibbs sampler's pressure for local consistency.

The core intution is that if we design our systems with stochastic behavior in mind, we no longer need to fear the stochasticity of the physical world. The kinds of error rates we are likely to incur due to physical faults may well be very small with respect to the intrinsic variablity (or entropy, or uncertainty) in the computation our machine is performing, and thus they can be disregarded. As we continue to explore increasingly unreliable physical substrates for computation, bring our computing machines into increasingly harsh physical environments, or attempt to understand how to compute with coffee cups and brain cells, this kind of robustness may become increasingly important. Detailed empirical and theoretical characterization of the robustness properties of large circuits - ideally by comparison to exact calculation or analytical bounds from the theory of Markov chains - remains an open challenge.

4.4 Discussion and Future Work

In this chapter, we introduced *combinational stochastic logic*, a complete set of stochastic gates for the probability algebra, naturally generalizing the deterministic, Boolean case. We have shown how to construct massively parallel, fault-resistant stochastic state machines for Monte Carlo algorithms, using designs quite unlike the real-valued, vector-arithmetic structures underlying current computers. Instead, we directly model the probability algebra in digital hardware, finding that the conditional independences in the processes of interest combined with the inherently low bit precision needed for stochastic simulation naturally lead to small, fast, parallel circuits whose physical structure matches the structure of the stochastic processes they are designed to simulate. When implemented on tiled arrays of commodity FPGAs, our circuits should support low-cost, real-time approximate inference on models with hundreds of thousands of variables.

Much work remains to be done. First, we should explore different implementation substrates. For example, we could use Gaines-style circuits built via analog VLSI to cheaply implement our Gibbs pipeline elements, combining the speed and energy efficiency of analog computation with

the arbitrary composability of digital machines. We could also build reliable stochastic circuits out of nanoscale substrates, exploiting the robustness of our approach. Second, we should explore hypotheses in computational neuroscience based on stochastic circuits implementing approximate inference. One starting point is the observation that time-averaging a wire in a stochastic circuit yields a "rate code" that approximately reports the wire's "instantaneous" marginal probability. Third, we should develop mathematical connections between the finite-size time, space and entropy requirements of stochastic circuits and asymptotic complexity results from randomized algorithms.

We should also construct more sophisticated circuits. We can start by building circuits for approximate inference in nonparametric and hierarchical Bayesian models by combining stochastic samplers with stack-structured memories (for growing statespaces) and content-addressible memories (for e.g. sufficient statistics). Just as with our MRF application, we expect the knowledge structures we are interested in performing inference over to be reflected in the physical structures for needed for conditional simulation. We can also directly use the pieces from our Gibbs pipeline to implement advanced techniques like sequential Monte Carlo and Swendsen-Wang.

More speculatively, we can consider what stochastic digital circuits suggest about the design of physical memories. The registers we currently use take the same amount of time to store and to recall all bit strings, and are thus (implicitly) optimal with respect to the uniform probability model on symbols. However, we know that all symbols in some set to be remembered are rarely equally likely, and that finding a coding scheme efficiently mapping symbols to be remembered to a string of bits that have uniform probability is the central problem in source coding. A stochastic logic circuit that samples symbols according to the code - by arithmetic decoding, say - can then provide the interface to the physical memory.

Finally, we are actively exploring novel reprogrammable computer architectures better suited to probabilistic inference than traditional stored-program machines. For example, we have begun development of the *IID*, an FPGA-hosted, stochastic version of the Connection Machine architecture (33), which will be programmed by specifying a probability model and relying on a compiler to automatically produce an appropriate inference circuit. Some of the current challenges involved in that effort, and in connecting these ideas up with Church, will be detailed in the next chapter.

The apparent intractability of inference has hindered the use of Bayesian methods in the design of intelligent systems and in the explanation of computation in the mind and brain. We hope stochastic logic circuits help to address this concern, by providing new tools for mapping probabilistic inference onto existing digital computing machinery, and suggesting a new class of natively stochastic digital computing machines.

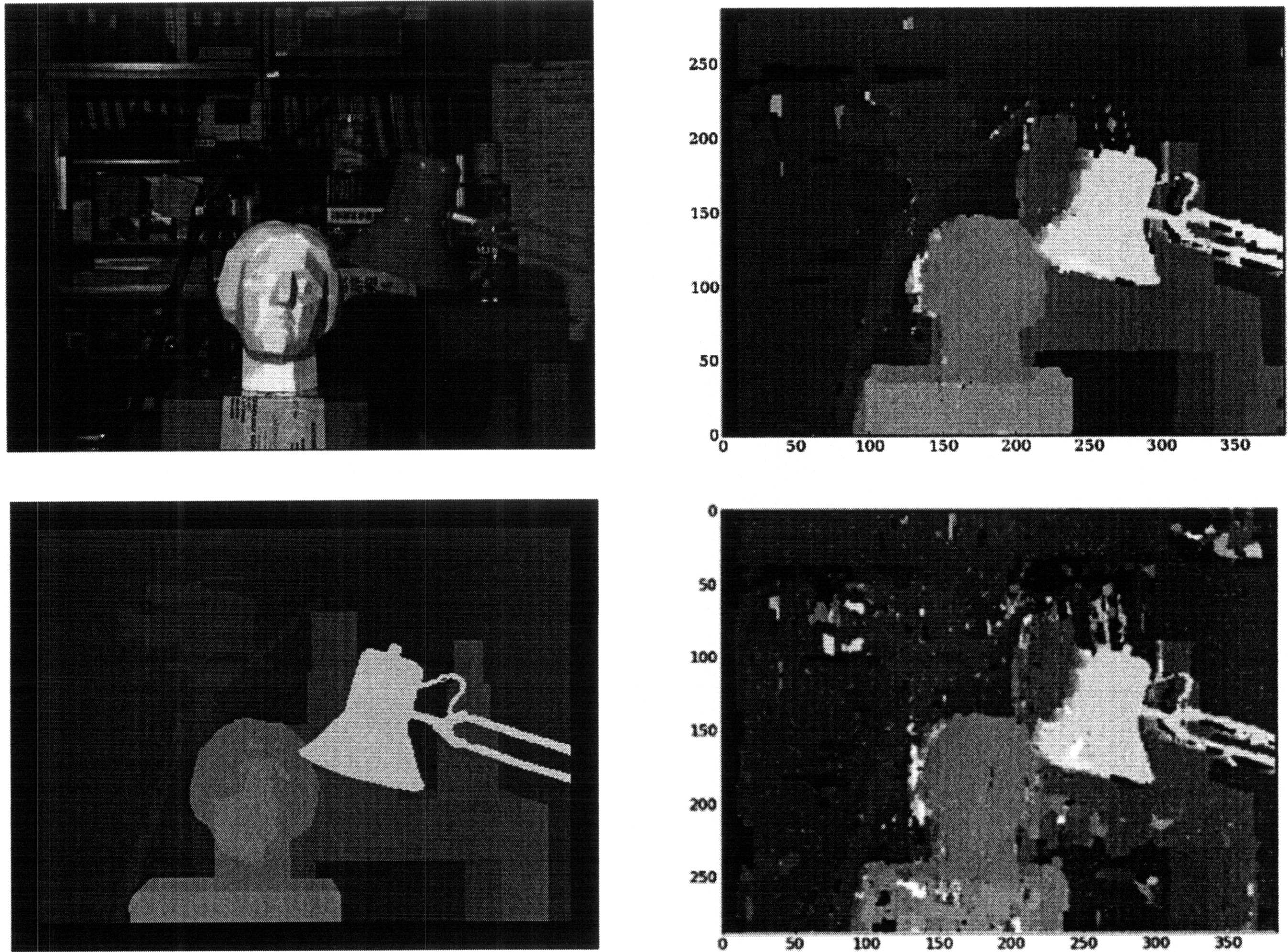


Figure 4-6: (Top left) The left camera image in the Tsukuba sequence from the Middlebury Stereo Vision database. (Bottom left) The ground truth disparity map. (Top right) The solution found by an annealed Gibbs sampler run at floating point precision after 400 iterations. (Bottom right) The solution found by simulation where probabilities were truncated after calculation but before sampling to 8 bits (which we expect is roughly comparable to full computation on our arithmetic pipeline with 10 to 15-bit fixed point). Note that quality is only slightly degraded.

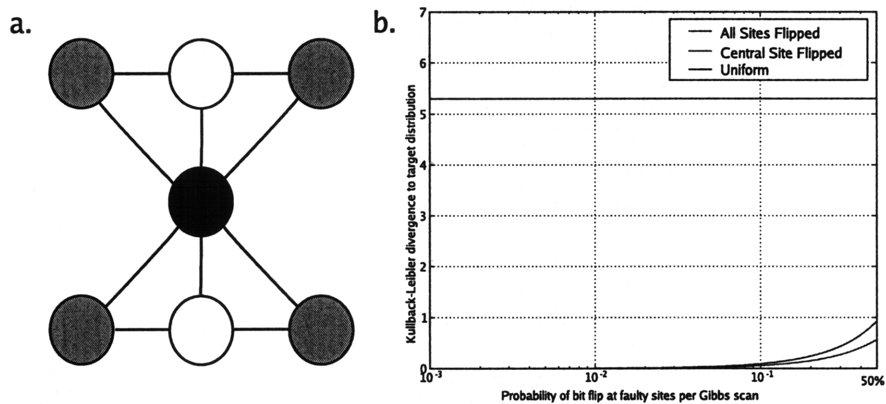


Figure 4-7: **Robustness to faults.** (a) shows a butterfly-structured binary Markov random field with attractive potentials. (b) shows robustness results to transient single site faults (assessed by deviation of the equilibrium distribution of a Gibbs sampler circuit from the target).

Chapter 5

Challenges in Connecting the Layers

> "There is, however, one feature that I would like to suggest should be incorporated in the machines, and that is a 'random element'."
>
> – Alan Turing, *Intelligent Machinery, A Heretical Theory*

> "83. What is the difference between a Turing machine and the modern computer? It's the same as that between Hillary's ascent of Everest and the establishment of a Hilton hotel on its peak."
>
> – Alan Perlis, *Epigrams on Programming*

When we program in a high-level language, we are usually oblivious to the fine-grained architecture of the machine we are programming, let alone the details of the digital circuitry that implements it. Instead, we pick algorithmic problems and evaluate programming choices based on approximate mathematical models of generic computing machines. We hope our computer architects design our physical computers such that our models are not too wrong to be useful and that a wide range of problems can be solved efficiently. We rely on compiler writers to build software that automatically translates our high-level programs into code that will run natively and efficiently on whatever machine we are actually programming.

If we want probabilistic learning and reasoning over rich knowledge structures to be easy and efficient in practice, we need to develop generalizations of these techniques. In particular, we need:

1. **Efficient, parallelizable inference algorithms for conditionally simulating from arbitrary probabilistic programs**. These algorithms must themselves be writeable as short probabilistic programs. Church's rejection and MCMC algorithms and systematic stochastic search represent just one step in this direction.

2. **Compilers that can turn recursive probabilistic programs into efficient circuits for massively parallel execution**. This will depend on a good, high-level language for circuits, and an effective scheme for making time/space tradeoffs for executing recursive processes. I briefly outline a prototype compiler based on State-Density-Kernel graphs (8) that takes a step in this direction, simplifying parts of this process for discrete factor graph models, but leaving the hardest problems of handling recursive processes left unsolved.

3. **Microarchitectures for probabilistic computers.** Circuit fragments that do elementary propagation steps for MCMC inference over Church traces or can handle conjugate Bayesian models may play analogous roles to register files and arithmetic logic units in classical architectures. I sketch some of the ideas we have been developing to manage these issues.

4. **Models of complexity for probabilistic computation.** Assessing the efficiency of a probabilistic program involving a query will require a better understanding of the complexity of sampling than seems to be available via current complexity theory.

In this chapter, I try to clarify some steps towards these goals. I first briefly outline a compiler that can produce circuit designs for problems with fixed-structure state spaces. I then describe two key challenges, namely parallelizing recursive processes and representing structured data, that both need to be solved if we want efficient Church MCMC in digital hardware. I also touch on some intuitions about computational complexity that may help us better understand what programs we expect to be efficient.

5.1 Generating Circuits That Solve Factor Graphs

Figure 5-1 shows the basic dataflow for a compiler we have written that automatically produces block diagram level circuit designs for inference in discrete factor graph models. This compiler sheds light on the basic challenges that more general efforts to connect the layers that future toolchains will face, automatically identifying and exploiting fine-grained conditional independencies implied by a simple but widely-used class of stochastic programs. Our goal was to automate

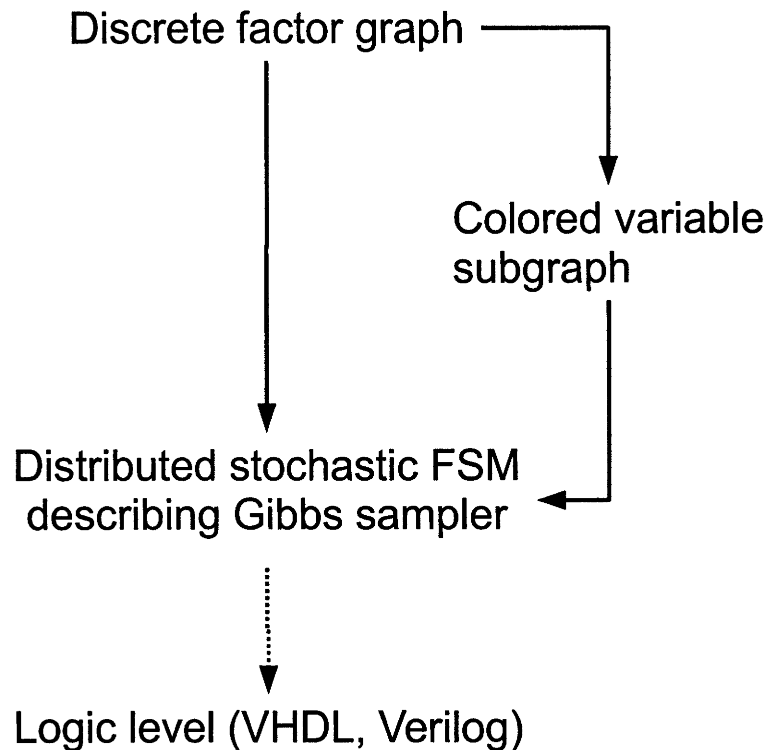


Figure 5-1: An overview of the compiler we have implemented to simplify the generation of stochastic digital circuits to simulate from factor graphs. Graph coloring is used to automatically identify fine-grained conditional independencies in the model. These independencies are used to automatically generate a graphical description of a parallel Gibbs sampler in the State-Density-Kernel language, serving as a block diagram level design for the digital circuit. A combination of software support and hand digital design (e.g. for implementing potentials) is needed for the final translation to executable digital circuit.

the process of producing circuits for solving propositional probability models, and to link this capability up with standard reconfigurable logic to obtain a general-purpose facility for stochastic computation.

Our compiler accepts as input a discrete variable factor graph, specified using a Python API. Figure 5-2 shows an example use. It operates by extracting the underlying undirected graph linking variables and coloring this graph to identify opportunities for conditional independence in Gibbs sampling: according to the semantics for Markov Random Fields, all nodes of the same color are conditionally independent of each other given all the nodes of all other colors. This means they can be sampled simultaneously, in principle requiring only local state and access to the state of their neighbors. Thus the topology of the graph, the number of bits needed to describe each variable and

the coloring number of the graph dictate the communication costs associated with fully exploiting its parallelism.

```
def testcompile():

    sdk = lattice.IsingFactorGraph(2,2)
    fgg.simpleGibbsKernelFromFG(sdk)

    sdk.drawGraph("/tmp/"+ getpass.getuser() +".test.png")

    module = helper.Module("sdk")

    compile.compile(module, sdk)

    module.writeBitcodeToFile("outputs/testCompile_simpleising.bc")
```

Figure 5-2: A screen capture of a Python fragment invoking our factor graph inference compiler. The displayed fragment generates a factor graph, identifies opportunities for exploitable parallelism, and generates an SDK description of a stochastic FSM for sampling from the distribution induced by the factor graph. The code fragment includes boilerplate to generate native code using LLVM for efficient, bit-accurate circuit simulation.

Our compiler uses this coloring to construct a State-Density-Kernel graph description of a distributed stochastic automaton that generates samples from the joint distribution implied by the factor graph. The State-Density-Kernel language (slightly modified here from its original form from (8)) allows one to specify a stochastic automaton out of pieces, making the dependencies between pure functions (such as probability densities or energy terms), pieces of state (such as variables) and stochastic transition functions (such as MCMC transition kernels) graphically explicit. Figure 5-3 shows the SDK generated for the inference problem from Figure 5-2.

With the SDK in hand, we can automatically generate native x86 code to perform bit-accurate simulation of the stochastic circuit the SDK represents, using LLVM (43) to manage native code generation. Critical low-level optimizations can also be managed at the LLVM level. The SDK description also serves as the block diagram level design of a digital circuit, which we then implement at the logic level using parametric instantiations of the states and kernels and hand engineering for the densities.

Currently, the supported language for potential functions is limited to a small, pre-specified list, with logic-level digital design work required to extend it. This is primarily because the contribution of the compiler lies in its ability to construct distributed stochastic circuits from factor graphs, and

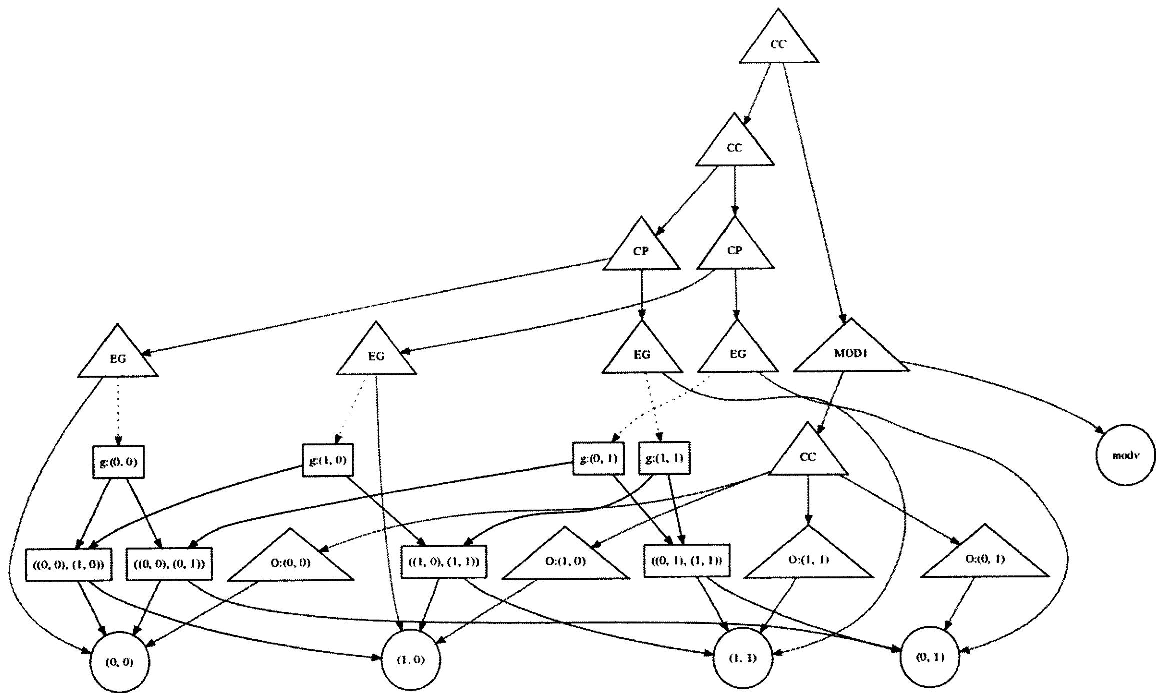


Figure 5-3: A State-Density-Kernel graph automatically produced by our compiler for performing parallel Gibbs sampling in a 2x2 Ising lattice. The density and kernel structure make explicit the full conditional independencies in the underlying factor graph for maximally parallel execution.

from the standpoint of this process the generation of circuits for potential evaluation can be viewed as calling out to a deterministic black box. If the arity of the discrete values in the variables is low and each factor is only connected to a small number of variables, then representing potentials as tables is feasible. However, as the complexity of the tables in terms of number of bits grows, it becomes important to compress the tables by writing them in terms of functions that compute their values, opening up the general problem of function-to-hardware compilation. We have not solved that general problem, although it is a special case of the more general problem of probabilistic compilation which we will describe in the next section.

5.2 Probabilistic Compilation and Architecture

"12. Recursion is the root of computation since it trades description for time."

– Alan Perlis, *Epigrams on Programming*

The discrete factor graph setting allowed us to focus on inference over a fixed, finite state space comprised of discrete values and work with Gibbs sampling, a probabilistic fixed-point iteration algorithm. We could thus construct a finite graphical description of a stochastic automaton for performing inference. We could also identify conditional independencies by graph coloring and use these to execute parts of the automaton in parallel. The exploitable parallelism is the source of the design wins in our circuitry, while its explicitness is what enables our compiler, and therefore our reconfigurable architecture.

If we want efficient probabilistic computation in the more general Church setting, we need to leverage conditional independence to yield exploitable parallelism, and simulate independent parts of our probabilistic processes in parallel using small, space-efficient processes. As this parallelism becomes increasingly finer grained, we will again approach reconfigurable computing, where programming becomes rewiring and compiler support becomes increasingly important. Moving to the broader Church setting adds two main layers of complexity over discrete factor graphs: executing recursive processes in parallel and representing structured data. A reconfigurable computer for Church inference that uses MCMC will need to have combined answers to these two problems, informing both its compiler and its architecture. Figure 5-5 captures a fantasy diagram for that situation.

5.2.1 How should we parallelize recursive processes?

Consider a machine that peforms Church inference using rejection sampling. We have seen how to implement this algorithm in Church, and thus in Scheme, assuming we are provided with random primitive procedures. The key insight needed to translate this implementation into a *serial* circuit for implementing it can be seen in Steele & Sussman's classic paper on the Scheme79 architecture (77):

> LISP, like traditional stored-program machine languages and unlike most high-level languages, conceptually stores programs and data in the same way and explicitly allows programs to be manipulated as data. LISP is therefore a suitable language around which to design a stored-program computer architecture. LISP differs from traditional machine languages in that the program/data storage is conceptually an unordered set of linked record structures of various sizes, rather than an ordered, indexable vector of integers or bit fields of fixed size. The record structures can be organized into trees or graphs. An instruction set can be designed for programs expressed as such

trees. A processor can interpret these trees in a recursive fashion, and provide automatic storage management for the record structures.

That is, we can leverage the "code as data" aspect of Lisp to build a stored-program architecture centered around symbolic, recursive processes rather than iterative, numerical ones. The same thing is possible for Church, and conceptually straightforward - although practically extremely involved - for the rejection sampler implementation of QUERY. However, such an architecture would make only peripheral use of stochastic digital elements, in precisely those places where primitive operations were performed. It would lack robustness to faults because it would not be performing inference by stochastic fixed-point iteration. It would not be tremendously faster than traditional architectures because it would ignore conditional independencies between subparts of the computation.

In any side-effect free deterministic functional language, all subevaluations are independent; the only sharing occurs through bound variables in the environment, though in principle they could be recomputed. In principle, every recursive call to EVAL in a meta-circular evaluator could be implemented by copying the environment and sending it off to a new thread (or copying it into the memory of a nearby Scheme79 circuit). However, a machine that did this would spend all its time shuttling argument and environment data to and from different evaluators.

One main problem, then, is to devise a means of deciding when to recurse in place and when to recurse by transferring data to another point in space and causing computation to proceed in parallel at that location. These choices are hidden when a process - including the process of evaluation - is described in Scheme notation.

In the probabilistic setting, this basic challenge - deciding when to exploit the opportunities for parallelism that conditional independence makes explicit, in the setting of recursion - remains. The one additional wrinkle is that exchangeable random procedures that are not IID introduce locking constraints on environments. This is because I cannot run two instances of an exchangeable random procedure that performs internal mutation at the same time, since both will be reading to and writing from the same internal state. An allocator must trade off the advantages of running a subprocess in parallel with the cost of transmitting environment data back and forth and the waiting due to locks.

5.2.2 How should we represent structured state for MCMC?

If we can obtain satisfactory solutions to the time/space allocation problem, we will still need two architectural innovations over the Scheme79 architecture to bridge the gap between structured Church programs and basic stochastic digital circuitry. First, we will need to build a stateful circuit fragment that can represent a piece of a computation trace in a local memory and execute the rules associated with trace updating by cycling the memory through trace recursions. To support concurrent trace updating, we will want this trace unit to have hooks for other trace units, including means of linking their subexpression and environment structures with the current one.

Second, we will want appropriate circuit fragments to compactly represent and compute over common exchangeable random primitives. These play the analogue of the units for applying primitive procedures in the Scheme chip, or of basic arithmetic operations in a standard FPU. Their job is to capture the state of an exchangeable random sequence, along with the basic operations needed for MCMC. Consider a discrete distribution over a finite list of symbols of known length. We would like to be able to represent this kind of random draw, both in the case where the distribution is explicitly represented (and so the probabilities of each outcome are provided as a parameter to the circuit fragment) and where the distribution is integrated out, yielding an exchangeable sequence of symbols. In the exchangeable case, we need to support the following operations:

1. Generating a sampled symbol from the distribution, conditioned on previous draws via its internal state.
2. Evaluting the probability of a symbol.
3. Incorporating a sampled symbol into the state.
4. Removing a sampled symbol from the state, i.e. "unsampling" the symbol, for use when moves are rejected in MCMC.
5. Copying the state to and from the state of another discrete distribution unit.

The state is just the sufficient statistics of the distribution. For a short list of possibilities, i.e. a low-dimensional multinomial distribution, a count array is likely to be the most efficient representation. However, as the list grows very long - for example, when we have a distribution over millions of word tokens - the symbols are likely to beecome sparsely represented as the distribution approaches a Dirichlet process. Accordingly, a sparse, list-based representation of the counts may be a more efficient use of time and space.

Similar fragments for IID and exchangeable binomials, Gaussians, and so on will form an important primitives library, and make the locus of many of the most important state management and time/space tradeoffs explicit. We imagine that much like a modern FPGA has banks of floating-point MULITPLY-ACCUMULATE units, reflecting its focus on convolution, a reconfigurable stochastic accelerator might have banks of units for sampling from various primitive distributions. While simulating these operations using general purpose memories and processors is of course possible, it is precisely this simulation that obscures the low intrinsic time/space efficiencies in sampling algorithms, and can make them seem unacceptably slow.

5.3 Probabilistic Computational Complexity

Computability and complexity classes are specified in terms of sets with definite boundaries, defined by the domains and ranges of definite functions, such as "the set of all pure Lisp programs that halt" and "the set of all Boolean formulae on n variables in 3-term conjunctive normal form which are satisfiable". Computability and complexity theorists study the logical possibility of evaluating these functions — "is it possible to decide if a given 3-CNF formula is satisfiable" — as well as the time and space requirements needed for such evaluations (71). The theory of descriptive complexity explicitly identifies the classes under study in complexity theory with the models of theories written in various logics (20; 35), identifying the difficulty of logical reasoning with traditional complexity classes. The amount of effort spent on "derandomizing" methods (74) for approximate function evaluation is another indication of the ways in which the theory of computation emphasizes determinism.

We need a new theory of computational complexity that addresses the main questions of interest in probabilistic computing, including:

1. A uniform theory of the complexity of sampling that can explain the costs in time, space and entropy for simulating Church expressions involving `QUERY` as well as those involving only `EVAL`.

2. A non-uniform theory of the hardness of sampling in terms of depth, width and entropy stochastic digital circuits, keeping in mind that the tractability of sampling depends crucially on the amount of exponentially hard work one can do cheaply (e.g. to set up a look-up table in a machine to build a fast Gibbs unit before a long sampling run).

These theories must explain the empirical fact that problems of Bayesian inference are often easy to approximately solve - at least, approximate well enough - in practice, especially by sampling methods. While I do not know how to bridge the gap between the practice of approximate sampling and probabilistic computing and complexity theory, I can suggest four independent but combineable ways in which the standard setting for complexity theory could be changed to bring it closer:

1. Consider the problem of sampling witnesses to problems in NP (or co-NP), uniformly at random, rather than finding satisfying assignments. For NP, this involves implementation as a Church query where the expression draws a candidate solution uniformly at random and the predicate checks the desired condition.

2. Sample from variants of the problem where the energy landscape is replaced by a smoothed version. For problems in NP, obtain this landscape by working with noisy versions of the predicate. For example, in SAT, replace AND with a noisy AND that sometimes returns true when one input is false. Sampling on the energy landscape in such a smoothed version allows for generic means of obtaining partial credit.

3. Sample approximately, e.g. from a Markov chain that has not completely converged, rather than insisting on exact sampling. Exploit the fact that it may be easy to generate exact samples without being able to know for certain that they are exact.

4. Consider the probabilistic process by which the world generates problem instances, allowing the world to make errors if it attempts to generate (or transmit) hard instances (as in semi-random sources (84; 6) and smoothed analysis (76)).

There may be a way to view our current theories of complexity as focusing on a limiting case of probabilistic computation, with the setting most relevant for conditional simulation in probabilistic AI lying outside of this limiting plane. The theory of randomized algorithms typically focuses on (and has yielded positive results about) the use of samplers in function approximation (3; 18). However, this theory does not focus on the difficulty of sampling; it instead measures the difficulty of solving satisfiability problems (as in NP) or counting (as in #P).

There is work that directly bears on the difficulty of sampling, for example from the study of the convergence rates of certain Markov chains (37; 70). However, to the best of my knowledge, it has only been linked to "complexity theory" through the lens of approximating functions via

Monte Carlo estimation. By combining the four relaxations of deterministic complexity mentioned above, we might be able to develop stronger connections that shed light on the real computational tractability of both deterministic and probabilistic computation. The hope would be to recover the structures of complexity theory, focused on function evaluation, as the deterministic limits of a larger class of sampling problems. Enlarging the space might help us gain perspective on properties of complexity theory that are currently hard to explain. For example, many NP complete problems exhibit phase transitions of hardness and are often very easy in practice; see (68) for one example of work in this large literature. However, problems like graph isomorphism and factoring seem hard in practice, even though they are not NP-complete. One possibility is that NP complete problems relax to approximate sampling problems that are easy to solve, whereas the sampling problems that recover graph isomorphism or factorization in the limit are remain difficult. Regardless, understanding the structure of complexity for probabilistic computation and its relationships to existing structures remains an intriguing challenge.

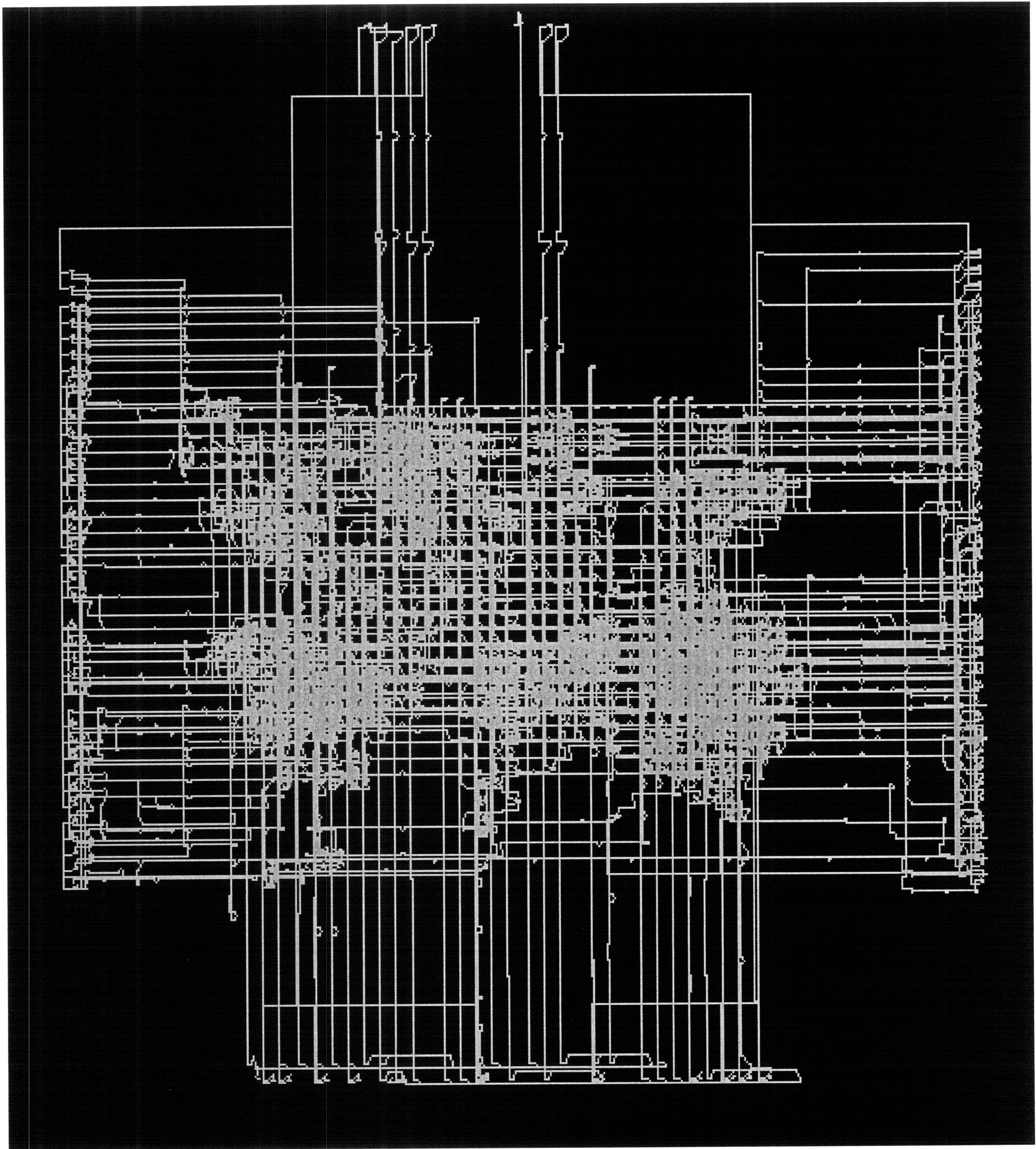


Figure 5-4: A synthesized FPGA layout for a circuit for a 9x9 Ising lattice, consistent with a 9x9 design (as SDK) produced by our compiler. This circuit assumes random bitstreams are provided from off the FPGA.

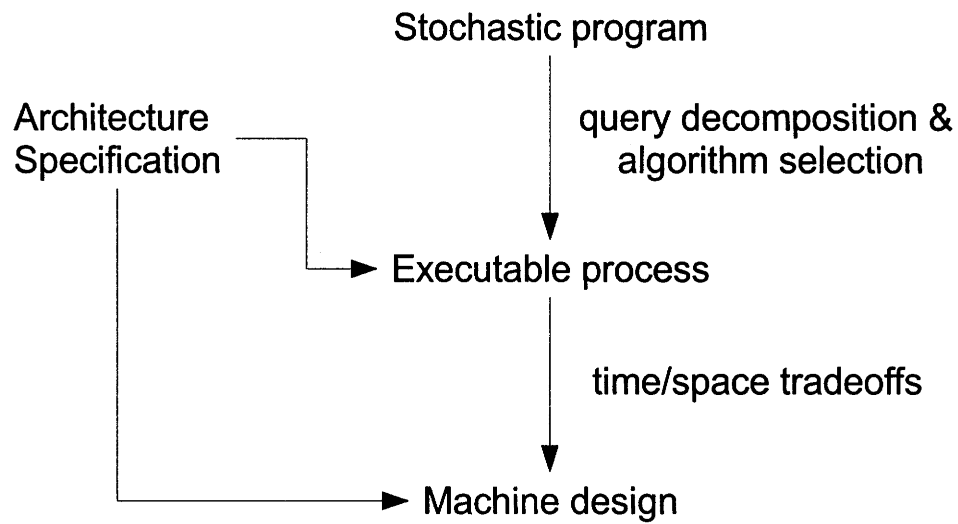


Figure 5-5: An ideal compiler for stochastic computation would involve automatic, universal conditional simulation (to remove all queries) and automatic identification and negotiation of time/space tradeoffs informed by the architectural constraints of the available machine.

Chapter 6

Conclusion

> "The possibility that the logic of Gibbsian systems (set up for physical chemistry) might be equally applicable to biological and social systems, was considered more and more seriously.."
>
> – Hans Lukas Teuber, personal notes

> "With our artificial automata we are moving much more in the dark than nature appears to be with its organisms. We are, and apparently, at least at present, have to be much more 'scared' by the occurrence of an isolated error and by the malfunction which must be behind it. Our behavior is clearly that of overcaution, generated by ignorance."
>
> – John von Neumann, *The General and Logical Theory of Automata*

In this dissertation, I introduced a new set of natively probabilistic computing abstractions, including probabilistic generalizations of Boolean circuits, backtracking search and pure Lisp. I showed how these tools let one compactly specify probabilistic generative models, generalize and parallelize widely used sampling algorithms like rejection sampling and Markov chain Monte Carlo, and solve difficult Bayesian inference problems. I will now close with some admittedly optimistic reflections.

Computation has changed the way we build models of the world, allowing us to specify and solve systems of equations and optimization problems that are vastly more complex than was previously possible. My hope is that the integration of probability into our programming languages and computing machines will help to continue this transformation in three main ways:

1. **Our programs themselves will become our general-purpose, generative models.** In classical AI and discriminative machine learning, programs typically describe algorithms for solving problems. They are written to go directly from percepts or assertions to causes. For example, one might write a vision program that takes in an image and produces a list of the objects in it. The reverse direction, which goes from causes to effects, correponds to modeling. However, because of determinism, programs written in the reverse direction — for example, computer graphics engines — are hard to directly employ as models. Without probabilities, it is unclear how to choose between the infinitely many inputs that are logically consistent. With probabilistic programming, we can write and extend programs that hallucinate possible worlds and explanations. In this example, such a program might choose typical objects and list them along with a rendered view. We could then explore the typical consequences of the model by running it forward and also bring it into contact to data by using universal probabilistic reasoning algorithms to run it backwards, going from the program that describes our model to an algorithm that solves the model given data via general purpose machinery.

   This approach also sidesteps the problems of feature engineering that have been at the heart of discriminative learning methods. Rather than search for features that are invariant to a transformation one would like to match across, one writes procedures that apply randomly chosen transformations. For example, to recognize images across rotations, one could write a probabilistic program that samples images and then rotates them, and invert this program to perform recognition. Of course, the algorithmic problem of inverting the program may still be quite hard in practice; building inference engines that are up to the task of inverting general programs remains an important challenge.

2. **We will begin to expect a probabilistically consistent treatment of uncertainty when we build and fit models.** We will expect to be able to simulate a model to get a sense of the variability in outputs or the stability of data-driven conclusions. We may also grow suspect of models that come with high-precision parameters, and be naturally led to focus on the qualitative (if uncertain) consequences, rather than trusting aspects that depend on low-order bits.

3. **Our machines will begin to sanity check implausible inputs and sample plausible alternatives rather than blindly follow our instructions.** Our interactions will someday be taken as noisy evidence, interpreted with respect to probabilistic programs that model our

intent, rather than taken as definite inputs to some deterministic function. This epistemological flexibility, arising from the wiggle room afforded by probability, could potentially allow us to one day build a probabilistic computer that is not well described by the phrase "garbage in, garbage out".

The flavor of probabilistic computing is interestingly different from both deterministic computation and quantum computation. Probabilistic algorithms and state machines work by massively parallel stochastic walks, rather than carefully coordinated sequences of deterministic steps. We expect them to eventually produce desired outputs in reasonable proportions, rather than perform any given step precisely. This may help us model biological, neural, psychological and social systems, which robustly exhibit reasonable behavior under a wide range of conditions but rarely - if ever - can be made to repeat themselves perfectly. Metaphors from statistical physics - and, ultimately, metallurgy and chemical engineering - may become increasingly appropriate for describing our algorithmic processes as our computers themselves are naturally viewed as Gibbsian rather than deterministic systems. Perhaps annealing was a particularly fortuitous beginning, showing how optimization, one of the key problems in deterministic computation, can be fruitfully viewed as the limit of a broader, probabilistic view. We also hope that in the process some of the more unpleasant mismatches between computation and classical physics, such as its lack of scale invariance and its instability to local failures (33), may begin to be mitigated. As we build computers out of larger and larger collections of smaller and smaller components, this kind of convergence between theories of computation and physics will probably grow more important.

Probabilistic computation may also provide clues for understanding neural computation and cognitive architecture. We can let go of our focus on calculating probabilities and optimal actions, instead favoring systems that sample good guesses. For example, neural systems may appear noisy because they are trying to solve problems of inference and decision making under uncertainty by sampling. The variability might not be Gaussian error around some linearized set-point, but rather the natural dynamics of a distributed circuit that is robustly hallucinating world states in accordance with a generative probabilistic model and the evidence of the senses. At the cognitive level, we may be able to build agents that use probabilistic programs as the inputs, outputs and executable descriptions of their learning, reasoning and planning systems, with their beliefs and desires represented using layers of increasingly domain-specific language in a stratified design.

When we build probabilistic circuits, algorithms and programs, we give up many of the conceptual and practical advantages of determinism. Our systems become harder to characterize, to test, to analyze and to debug, requiring new concepts, mathematics and practical tools. Furthermore,

some applications seem to require determinism; it is hard to imagine the IRS using probabilistic methods in the core of their accounting and bookkeeping systems (although they might be helpful in suggesting audits). However, it seems like there are some potentially powerful synergies between probability and the basic abstractions we use in computing:

- Programming languages give us tools for managing large structures of procedural knowledge. Marrying them with probability helps us use programs declaratively (as generative models), and also tolerate exceptions, manage uncertainty, and learn inductively.

- Search algorithms let us explore large spaces efficiently. Marrying them with probability helps us avoid local minima, account for multiple solutions to reasoning problems without combinatorial explosion, and exploit fine-grained parallelism that was exposed via conditional independence.

- Circuits let us reflect the structure of our computations in our machines. Marrying them with probability lets us save space by exploiting low bit precision, save time by exploiting exposed parallelism, and deal with very high fault rates.

We are a long way from fully developing the technology needed to make probabilistic modeling easy and efficient in everyday practice, and, in all probability, even farther from building a general purpose thinking machine. However, I hope I have convinced you that by viewing stochasticity and uncertainty as an ally, not an enemy, and marrying it with our basic computational abstractions, we have some new and interesting avenues to explore.

Bibliography

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, USA, 1996.

[2] C. Andrieu, N. de Freitas, A. Doucet, and M.I. Jordan. An Introduction to MCMC for Machine Learning. *Machine Learning*, 50(1):5–43, 2003.

[3] I. Bárány and Z. Füredi. Computing the volume is difficult. *Discrete and Computational Geometry*, 2(1):319–326, 1987.

[4] P. Baumgartner. FDPLLA First Order Davis-Putnam-Logeman-Loveland Procedure. In *Automated Deduction-Cade-17: 17th International Conference on Automated Deduction, Pittsburgh, Pa, Usa, June 2000: Proceedings*, page 200. Springer, 2000.

[5] M.J. Beal, Z. Ghahramani, and C.E. Rasmussen. The infinite hidden Markov model. *NIPS 14*, 2002.

[6] A. Blum and J. Spencer. Coloring random and semi-random k-colorable graphs. *Journal of Algorithms*, 19(2):204–234, 1995.

[7] M. Bolic. Architectures for Efficient Implementation of Particle Filters. *USA: State University of New York at Stony Brook*, 2004.

[8] Keith Bonawitz. *Composable Probabilistic Inference with BLAISE.* PhD thesis, 2008.

[9] George Boole. *An Investigation of the Laws of Thought.* 1854.

[10] N. Chater, J.B. Tenenbaum, and A. Yuille. Probabilistic models of cognition: Conceptual foundations. *Trends in Cognitive Sciences*, 10(7):287–291, 2006.

[11] S. Cheemalavagu, P. Korkmaz, and K.V. Palem. Ultra low-energy computing via probabilistic algorithms and devices: CMOS device primitives and the energy-probability relationship. *Proc. of The 2004 International Conference on Solid State Devices and Materials*, pages 402–403.

[12] Andrew M. Childs, Ryan B. Patterson, and David J. C. MacKay. Exact sampling from non-attractive distributions using summary states. 2000.

[13] H. Daume III. Hbc: Hierarchical bayes compiler. 2007.

[14] B. De Finetti. *Funzione caratteristica di un fenomeno aleatorio*. Soc. anon. tip." Leonardo da Vinci", 1930.

[15] P. Del Moral, A. Doucet, and A. Jasra. Sequential monte carlo samplers. *Journal of the Royal Statistical Society*, 68(3):411–436, 2006.

[16] P. Diaconis. The markov chain monte carlo revolution. 2008.

[17] A. Doucet, N. de Freitas, and N. Gordon. Sequential Monte Carlo in Practice, 2001.

[18] M. Dyer, A. Frieze, and R. Kannan. A random polynomial-time algorithm for approximating the volume of convex bodies. *Journal of the ACM (JACM)*, 38(1):1–17, 1991.

[19] Robert G. Edwards and A. D. Sokal. Generalizations of the Fortuin-Kasteleyn-Swendsen-Wang representation and Monte Carlo algorithm. *Physical Review*, 38:2009–2012, 1988.

[20] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In *Complexity of Comput., Proc. Symp. appl. Math., New York City*, 1973.

[21] N. Friedman. Inferring cellular networks using probabilistic graphical models, 2004.

[22] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *International Joint Conference on Artificial Intelligence*, volume 16, pages 1300–1309. LAWRENCE ERLBAUM ASSOCIATES LTD, 1999.

[23] N. Friedman and D. Koller. Being Bayesian about network structure. A Bayesian approach to structure discovery in Bayesian networks. *Machine Learning*, 50(1):95–125, 2003.

[24] N. Friedman, M. Linial, I. Nachman, and D. Pe'er. Using Bayesian networks to analyze expression data. *Journal of computational biology*, 7(3-4):601–620, 2000.

[25] B. R. Gaines. Stochastic Computing Systems. *Advances in Information Systems Science*, 2, 1969.

[26] A. Gelman, J. Carlin, H. Stern, and D. Rubin. *Bayesian data analysis*. Chapman & Hall, London, 1995.

[27] S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions and the Bayesian restoration of images. *Readings in Computer Vision: Issues, Problems, Principles, and Paradigms*, pages 564–584, 1987.

[28] R. Genov and G. Cauwenberghs. Stochastic Mixed-Signal VLSI Architecture for High-Dimensional Kernel Machines. *Advances in Neural Information Processing Systems*, 14.

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

[30] Noah Goodman, Vikash Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua Tenenbaum. Church: a language for generative models with non-parametric memoization and approximate inference. In *Uncertainty in Artificial Intelligence*, 2008.

[31] CE Guo, S.C. Zhu, and Y.N. Wu. Primal sketch: integrating texture and structure. *Computer Vision and Image Understanding*, 106(1):5–19, 2007.

[32] Firas Hamze and Nando de Freitas. Hot coupling: A particle approach to inference and normalization on pairwise undirected graphs. In *NIPS*, 2005.

[33] W.D. Hillis. *The Connection Machine*. MIT Press, 1989.

[34] Mark Huber. A bounding chain for Swendsen–Wang. *Random Structures & Algorithms*, 22(1):53–59, 2002.

[35] N. Immerman. *Descriptive Complexity*. Springer, 1999.

[36] E. T. Jaynes. *Probability Theory : The Logic of Science*. Cambridge University Press, April 2003.

[37] M. Jerrum and A. Sinclair. The Markov chain Monte Carlo method: an approach to approximate counting and integration. *Approximation algorithms for NP-hard problems*, pages 482–520, 1997.

[38] M. Johnson, T. Griffiths, and S. Goldwater. Adaptor grammars: A framework for specifying compositional nonparametric Bayesian models. *NIPS 19*, 2007.

[39] Michael I. Jordan, Zoubin Ghahramani, Tommi S. Jaakkola, and Lawrence K. Saul. An introduction to variational methods for graphical models. *Mach. Learn.*, 37(2):183–233, 1999.

[40] W. Kahan and J. Palmer. On a proposed floating-point standard. *ACM SIGNUM Newsletter*, 14:13–21, 1979.

[41] C. Kemp and J.B. Tenenbaum. The discovery of structural form. *Proceedings of the National Academy of Sciences*, 105(31):10687.

[42] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, 2001.

[43] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society Washington, DC, USA, 2004.

[44] P. Liang, S. Petrov, M.I. Jordan, and D. Klein. The Infinite PCFG using Hierarchical Dirichlet Processes. *Proc. EMNLP-CoNLL*, 2007.

[45] D.J. Lunn, A. Thomas, N. Best, and D. Spiegelhalter. WinBUGS-a Bayesian modelling framework: concepts, structure, and extensibility. *Statistics and Computing*, 10(4):325–337, 2000.

[46] D.J.C. MacKay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003.

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

[48] G. Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 2003.

[49] John L. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

[50] N. Metropolis, AW Rosenbluth, MN Rosenbluth, AH Teller, and E. Teller. Equations of State Calculations by Fast Computing Machines. *Journal of Chemical Physics*, 1953.

[51] B. Milch, B. Marthi, S. Russell, D. Sontag, D.L. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. *Proc. IJCAI*, 2005.

[52] K.P. Murphy. The bayes net toolbox for matlab. *Computing science and statistics*, 2001.

[53] AF Murray, D. Del Corso, and L. Tarassenko. Pulse-stream VLSI neural networks mixing analog and digitaltechniques. *Neural Networks, IEEE Transactions on*, 2(2):193–204, 1991.

[54] R.M. Neal. Probabilistic inference using Markov chain Monte Carlo methods. Technical Report CRG-TR-93-1, Department of Computer Science, University of Toronto, 1993.

[55] S. Park, F. Pfenning, and S. Thrun. A probabilistic language based upon sampling functions. *SIGPLAN Not.*, 40(1):171–182, 2005.

[56] H. Pasula and S. Russell. Approximate inference for first-order probabilistic languages. In *International Joint Conference on Artificial Intelligence*, volume 17, pages 741–748. LAWRENCE ERLBAUM ASSOCIATES LTD, 2001.

[57] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, San Mateo CA, 1988.

[58] B.A. Pearlmutter and J.M. Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. 2008.

[59] A. Pfeffer. IBAL: A probabilistic rational programming language. *Proc. IJCAI*, 2001.

[60] James Gary Propp and David Bruce Wilson. Exact sampling with coupled Markov chains and applications to statistical mechanics. *Random Structures and Algorithms*, 9(1&2):223–252, 1996.

[61] W. Qian, J. Backes, and M. Riedel. The Synthesis of Stochastic Circuits for Nanoscale Computation, 2007.

[62] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740, New York, NY, USA, 1972. ACM.

[63] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1):107–136, 2006.

[64] D. Roy and C. Freer. Computable exchangeable sequences have computable de finetti measures.

[65] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[66] L. K. Saul, T. Jaakkola, and M. I. Jordan. Mean field theory for sigmoid belief networks. *Journal of Artificial Intelligence Research*, 4(4):61–76, 1996.

[67] E. Segal, M. Shapira, A. Regev, D. Pe'er, D. Botstein, D. Koller, and N. Friedman. Module networks: identifying regulatory modules and their condition-specific regulators from gene expression data. *Nature genetics*, 34:166–176, 2003.

[68] B. Selman and S. Kirkpatrick. Critical behavior in the computational cost of satisfiability testing. *Artificial Intelligence*, 81(1-2):273–295, 1996.

[69] Claude Shannon. *A Symbolic Analysis of Relay and Switching Circuits*. PhD thesis, 1940.

[70] A. Sinclair and M. Jerrum. Approximate counting, uniform generation and rapidly mixing Markov chains. *Information and Computation*, 82(1):93–133, 1989.

[71] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, December 1996.

[72] J.M. Siskind and D.A. McAllester. Nondeterministic lisp as a substrate for constraint logic programming. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 133–133. JOHN WILEY & SONS LTD, 1993.

[73] J.M. Siskind and B.A. Pearlmutter. Nesting forward-mode AD in a functional framework. *Higher-Order and Symbolic Computation*, 21(4):361–376, 2008.

[74] D. Sivakumar. Algorithmic derandomization via complexity theory. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 619–626. ACM Press New York, NY, USA, 2002.

[75] David Sontag and Tommi Jaakkola. New outer bounds on the marginal polytope. In J.C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 1393–1400. MIT Press, Cambridge, MA, 2008.

[76] D.A. Spielman and S.H. Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM (JACM)*, 51(3):385–463, 2004.

[77] G.L. Steele and G.J. Sussman. Design of LISP-based Processors, or SCHEME: A Dielectric LISP, or Finite Memories Considered Harmful, or LAMBDA: The Ultimate Opcode. 1979.

[78] Robert H. Swendsen and Jian-Sheng Wang. Nonuniversal critical dynamics in monte carlo simulations. *Phys. Rev. Lett.*, 58(2):86–88, Jan 1987.

[79] M. F. Tappen and W. T. Freeman. Comparison of graph cuts with belief propagation for stereo, using identical mrf parameters. In *ICCV*, 2003.

[80] MF Tappen and WT Freeman. Comparison of graph cuts with belief propagation for stereo, using identical MRF parameters. *Proc. ICCV*, pages 900–906, 2003.

[81] Y. W. Teh, M. I. Jordan, M. J. Beal, and D. M. Blei. Hierarchical Dirichlet processes. *Journal of the American Statistical Association*, 101(476):1566–1581, 2006.

[82] Marc Toussaint, Stefan Harmeling, and Amos Storkey. Probabilistic inference for solving (PO)MDPs. Technical Report EDI-INF-RR-0934, University of Edinburgh, 2006.

[83] Z. Tu and S.C. Zhu. Image segmentation by data-driven Markov chain Monte Carlo. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5):657–673, 2002.

[84] U.V. Vazirani and V.V. Vazirani. Sampling a Population with a Semi-random Source. In *Proceedings of FSTTCS Conference*, pages 443–452. Springer, 1986.

[85] J. von Neumann. Probabilistic logics and synthesis of reliable organisms from unreliable components. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 43–98. Princeton University Press, 1956.

[86] M. Wainright and M. I. Jordan. Graphical Models, Exponential Families, and Variational Inference. *Foundations and Trends in Machine Learning*, pages 1 – 305, 2008.

[87] M. J. Wainwright and M. I. Jordan. Graphical models, exponential families, and variational inference. Technical Report 649, Department of Statistics, UC Berkeley, 2003.

[88] Martin J. Wainwright, Tommi S. Jaakkola, and Alan S. Willsky. A new class of upper bounds on the log partition function. In *In Uncertainty in Artificial Intelligence*, pages 536–543, 2002.

[89] J. Winn and T. Minka. Infer.net/csoft website.

[90] J. S. Yedidia. Generalized belief propagation and free energy minimization. Talk given at the Information Theory Workshop at the Mathematical Sciences Research Institute, 2002.

[91] Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. Generalized belief propagation. In *In NIPS 13*, pages 689–695. MIT Press, 2001.

[92] S.C. Zhu and A. Yuille. Region competition: unifying snakes, region growing, and bayes/mdlfor multiband image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(9):884–900, 1996.