FROM EXECUTION TRACES TO SPECIALIZED INFERENCE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Lingfeng Yang
August 2015

This dissertation is online at: http://purl.stanford.edu/kq822ym0815

Includes supplemental files:
1. Thesis with high-resolution images *(et2si.pdf)*

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Pat Hanrahan, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Alex Aiken

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Noah Goodman

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

We often envision a future where computers answer our questions. This is inference: coming to conclusions about the world based on a limited amount of information. One formulates a hypothetical world (model), then simulates and analyzes its behavior with respect to evidence.

Probabilistic programming languages are a recent approach where we can express any hypothetical world as a program with random choice primitives. However, this descriptive power often sacrifices performance of inference. In my work, I explore execution traces as a solution that enables high performance while preserving descriptive power. This leads to three subsequent developments.

The first is Shred, a tracing compiler that generates efficient Metropolis-Hastings MCMC code from probabilistic programs. Performance was seen to be competitive with hand-coded MCMC in some cases. Like the tracing just-in-time (JIT) compilers that served as the inspiration, there is a sacrifice of efficiency in representing multiple control flow paths.

Unlike with traditional JIT compilers, the execution target of probabilistic programming language traces is not limited to straightforward execution on a low-level language, but also includes state-of-the-art inference engines. I developed Solitaire, a language for procedural content generation with constraints, combining trace graph compilation with SMT solving and probabilistic languages.

Finally, rather than being limited to an intermediate representation, execution traces can also be treated as abstract objects of inference. The third development is model accretion, a stochastic search-based MAP inference algorithm that improves performance of procedural content generation by re-using previous executions.

Acknowledgements

I would like to thank my advisor Pat Hanrahan, who has been very patient and supportive with me over the years, and has contributed much to my intellectual development. Noah D. Goodman has also been a steady source of learning and knowledge, especially in strenthening my connections between computer science, math, and physics. Programming languages and compilers have been a difficult field to enter for my Ph. D work, and Alex Aiken has been very supportive and generally a confidence builder when it comes to learning these subjects. I would also like to thank my wife Jasmine for her patience and support. Finally, I thank my parents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Execution traces are a feasible and effective way to achieve specialization of inference languages to particular problems.

Inference by computer is playing an increasingly large role in science and our daily lives. The most prominent is inference for analysis. Our email programs infer which incoming messages are likely to be spam and filter them away. The weather is often predicted by simulating the interactions among wind, temperature, and the clouds. Physicists simulate individual interactions between magnetic domains in a material in order to discover properties about the material at critical temperatures.

Inference can also be used to make things. In the procedural generation of trees and flowers, it can be useful to describe the set of all valid plants as a probability distribution. We can then use probability queries to control which ones are generated.

The general setup is to take some part of the world, describe its possibilites in terms of a probability distribution

$$p(x_1 \dots x_n),$$

and express inference as a conditional distribution

$$p(x_1 \dots x_i | x_j \dots x_n)$$

where $x_j \dots x_n$ correspond to observations and/or constraints of interest.

Traditionally, this has been accomplished by constructing a bespoke implementation of an inference algorithm and model representation for a given application. Although this is effective, it is difficult to rapidly prototype models and inference algorithms.

To solve this problem, there have been recent efforts to build probabilistic programming languages whose purpose is inference. They allow us to express any set of possible worlds as the steps of evaluation of a program. By having a formal, executable definition of models, we can separate the implementation of inference algorithm from the specification of the model. This makes machine learning and AI techniques much more accessible.

However, while probabilistic programming languages can describe almost any probabilistic model, this can come at a cost of performance, which tends to be inferior to that of hand-written algorithms tuned for particular problems. In fact, probabilistic languages are in a uniquely difficult place, as our probabilistic language runtime must essentially take as input a program whose time and space usage generally cannot be predicted.

In this thesis, we explore an approach to improving performance by generating execution traces of the probabilistic program. The idea is that no matter how sophisticated the input program, the execution trace exposes a finite sequence of primitive operations and random choices taken on a particular control flow path. With a proper choice of primitive operations, this allows specialization of inference by generating fast code to compute probabilities and to use existing powerful inference engines as backends. Finally, the execution trace can also inspire an inference algorithm that works by saving previous "good" execution traces.

This thesis is organized into 3 sections, each of which highlights a different use of execution traces.

1. Traces can be compiled to fast scoring code for the Metropolis-Hastings algorithm on a variety of machine learning problems (inference for analysis). We are able to acheive the speed of hand-written implementations through analyzing and compiling the trace to a lower-level langauge.

2. Traces allow interoperation with state of the art inference engines. In the domain of constrained procedural modeling (inference for synthesis), I show that by compiling traces to SMT formulae and feeding them to the Z3 SMT solver, there are several classes of models that cannot be feasibly synthesized using MCMC-like stochastic search methods alone.

3. Finally, traces are not just useful as an intermediate representation, but also as an abstract object of inference in a novel MAP inference algorithm: model accretion. Model accretion applied to procedural modeling shows a significant speedup even compared to the SMT solving backend.

Chapter 2

Background

This work focuses on the use of execution traces to increase performance of inference. We draw on concepts from both *probabilistic inference* and *probabilistic programming*. We prefer to characterize our methods in the context of an application of inference. The field of *procedural content generation* thus features as an application throughout this work. In this chapter, I introduce relevant topics from each of these fields.

2.1 Probabilistic inference

Probabilistic inference is the computation of event probabilities given possibly incomplete observation data. The degree of plausibility of an event is tied to a real number in the range $[0, 1]$. The world is modeled as a chain of interacting *events* that possibly affect the plausibility of other events down the chain.

For example, if we model event B as happening after event A, and we model B as having influence on whether A happens, then the number $P(B|A)$, representing how likely it is that B happens if A happens, is different from $P(B)$ which only represents how likely B is to happen given no information about whether A happened.

An interesting result is that the entirety of the probabilistic framework of thinking—Bayes' rule [23], dependent/independent probability calculation, expected value, variance, etc.—can be derived from just the two assumption above, that probabilities are real numbers in the range $[0, 1]$, and we have decided on a particular sequence of

events and interactions. This is covered in more detail in "Probability Theory: The Logic of Science" by E.T. Jaynes [18].

To provide intuition, we will now cover a simple example of probabilistic inference. Suppose we are playing a dice game and we are interested in the probability distribution over the resulting sum of two independent dice rolls. If betting and money is involved, it will be important to compute these probabilities correctly.

We can represent the outcome of the first dice roll as a *random variable* X, and the second, Y. Probability distributions are defined by combining these random variables. In this case, we are interested in the distribution over the sum of the outcomes, or $P(X + Y)$.

Situation 1: No observed variables. Suppose no dice have been thrown yet; we have no observation data. We must then rely completely on *prior knowledge* (i.e., *assumptions*) of how dice rolls and addition works. One assumption is to suppose both dice are fair dice; this can be specified by assigning a *uniform distribution* U to each of X and Y over the integers $\{1, 2, 3, 4, 5, 6\}$:

$$X \sim U\{1, 2, 3, 4, 5, 6\},$$
$$Y \sim U\{1, 2, 3, 4, 5, 6\}.$$

This means the probability of some number showing up for the first dice X is equal to that of any other number; the probability $P(X = i) = 1/6$ for all $i \in \{1 \dots 6\}$, and the same goes for $P(Y)$. We have thus cast our assumptions in terms of probability distributions. These are called *prior* probability distributions. Inference necessarily requires a set of such prior distributions; otherwise no probabilities can be calculated.

We can then consider computing (inference over) $P(X + Y)$, describing the outcome in terms of the sum of the numbers on the dice. We need to compute the probability $P(X + Y = k)$ for all possible k, given that $P(X), P(Y)$ have the uniform distributions shown above.

Table 2.1: Outcomes of two dice rolls

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Inference is in general difficult because there are many ways that random variables can combine to produce different outcomes. It is often infeasible to compute them all. In this case, however, we can feasibly enumerate all possible outcomes of the two individual dice rolls along with their sum. Table 2.1 shows the possibilities. The first row and column enumerate the possible outcomes (X, Y) of the two dice rolls, respectively. The remaining cells show the sum $X + Y$ given the values of (X, Y) at the row and column.

Each pair of dice rolls can be simulated by selecting a number from the first row and first column uniformly, and then pointing to the "outcome cell" at the selected row and column. Since the selection of the two numbers each has probability 1/6, the probability of selecting the outcome cell is $(1/6)(1/6) = 1/36$. We can then assign the probability 1/36 uniformly to each outcome cell.

However, we see that not every outcome of $X + Y$ occurs uniformly; 7 occurs the most, while 2 and 12 occur only once. Thus, in computing $P(X + Y = k)$, we need to properly count multiple ways of producing the same outcome from different settings of X and Y. In this case, since the occurrence of outcome cells is mutually independent, we can just add up the probabilities:

$$P(X + Y = k) = \frac{1}{36} N_k,$$

where N_k is the number of ways we can obtain k as the sum of two dice rolls, and

is the number of occurrences of k as an outcome cell in the table above. The resulting distribution $P(X+Y)$ is then:

$$
\begin{aligned}
P(X+Y=2) &= 1/36 \\
P(X+Y=3) &= 2/36 = 1/18 \\
P(X+Y=4) &= 3/36 = 1/12 \\
P(X+Y=5) &= 4/36 = 1/9 \\
P(X+Y=6) &= 5/36 \\
P(X+Y=7) &= 6/36 = 1/6 \\
P(X+Y=8) &= 5/36 \\
P(X+Y=9) &= 4/36 = 1/9 \\
P(X+Y=10) &= 3/36 = 1/12 \\
P(X+Y=11) &= 2/36 = 1/18 \\
P(X+Y=12) &= 1/36
\end{aligned}
$$

We see that 7 has the highest probability of occurring, while 12 and 2 have the least. Craps, a casino game, utilizes this sum of two independent dice rolls in its beginning stages. We can come up with strategies for these games using inference techniques similar to those shown here.

Situation 2: First dice rolled. Now suppose the first dice has been rolled. What is the distribution of $P(X+Y)$? This is unlike Situation 1, where there was no observable data to influence our inference. In terms of probability theory, we may recast our inference query $P(X+Y)$ to the conditional distribution $P(X+Y|X)$, which takes into account the observation. Inference is most commonly applied in this setting where incomplete observations are given and we would like to know something about the outcome.

There are many ways to represent conditional distributions, but in this work, the most relevant way to represent a conditional distribution $P(A|B)$ is as a *function*

that takes a sampled value b of B as input, and returns a distribution over A taking b into account.

For the conditional dice roll, we are interested in coming up with a function F that takes as input the value of the first dice roll X, and then returns a distribution over the sum $X + Y$.

Going back to our tabular representation, we can think of this function as first selecting a column for the outcome of X, and then noting that whatever value Y can be, the resulting $X + Y$ will be in the same column. The resulting distribution over $X + Y$ is then over the column. The conditional distribution then takes the following form, with one entry for each possible value of the second dice:

$$
\begin{aligned}
&P(X + Y | X = x) : \\
&P(X + Y = x + 1) = 1/6 \\
&P(X + Y = x + 2) = 1/6 \\
&P(X + Y = x + 3) = 1/6 \\
&P(X + Y = x + 4) = 1/6 \\
&P(X + Y = x + 5) = 1/6 \\
&P(X + Y = x + 6) = 1/6
\end{aligned}
$$

We see that the condition distribution $P(X + Y|X)$, given x the resulting value of the first dice roll, is uniform over $\{x + 1 \dots x + 6\}$.

Building probability distributions by composition The above example illustrates a very simple probability distribution: the sum of two independent random variables. The distributions in this work will not contain such simple structure and will not feature as many convenient independencies. Yet, they are very similar in their essence; one forms complex distributions by *composing* together a set of *primitive* distributions on which no further decomposition is possible (here, the uniform distributions over integers). Our composition operators can be any deterministic

function (here, it was the addition operator).

In this work, we deal in *probabilistic programming languages*, which take this idea of composition to the extreme and consider, as composition operator, the action of a *computer program* on the random variables.

Probability distributions as event trees To summarize the intuition, Figure 2.1 shows a general way to visualize inference and probability distributions through an *tree of events* (aka decision tree). Each event corresponds to a node, and the outcome of each event corresponds to following an edge from that node.

Each path from the root of the tree to a leaf corresponds to a possible sequence of event outcomes. Every such sequence is assigned a probability. The path probabilities then must add up to 1. Inference can be seen as counting the paths of this tree that correspond to some real-world observation or constraint

Note that each event can be of varying data type; e.g., we can have the first event have an integer outcome, the second event a boolean, the third a floating point number, and so on. We exploit this capability when using probabilistic programs to describe outcomes of events.

In addition, note that not every possible outcome of an event holds given outcomes of past events. For instance, if $E_1 = 0$, then E_2 can only be `true`. In a similar sense, we can incorporate observations by actively restricting the outcomes of certain events to their observed values.

Difficulty of inference First, note that this restriction of outcome values can complicate the computation of the number of paths in support ($P(x) > 0$) of the distribution, which is crucial in normalizing the path probabilities to add up to 1. The computation of thios number is not a simple matter of multiplying together all domain sizes of each event, and in general is as difficult as inference itself. The number of points in support weighted by the path probabilities is known as the *normalizing constant*, and is of complexity class *# P-complete*.

In addition, some of the path probabilities can be quite small, which will make naive algorithms fail. For example, if we simply follow the event tree according to

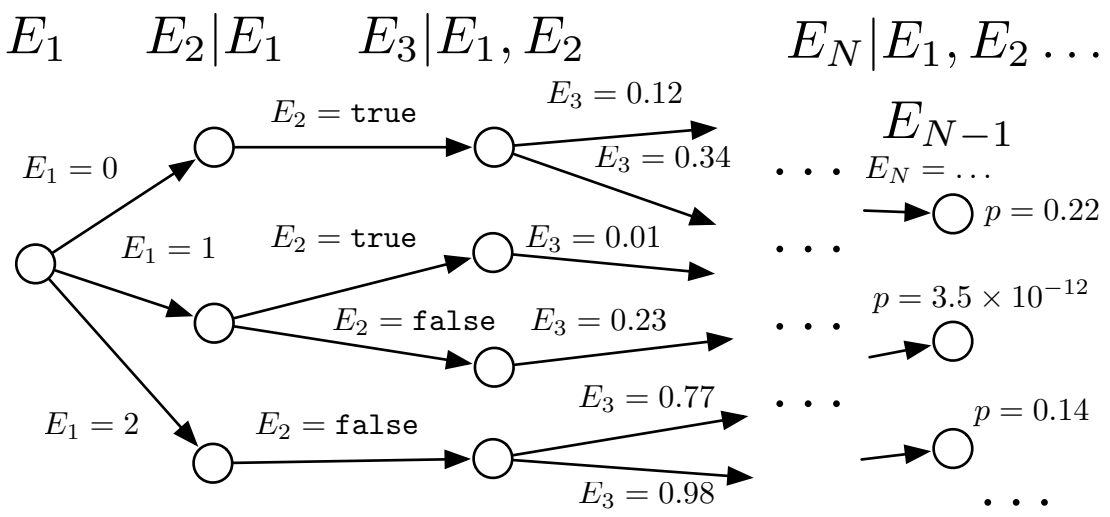


Figure 2.1: A probability distribution visualized as a tree of events. Every possible sequence of events is a path through this tree. Every path through the tree is assigned a probability.

the probabilities of individual event outcomes, we will have a hard time counting these small-probability events. Overall, the full event tree is usually quite large and is infeasible to compute over directly. This also makes inference difficult.

Inference algorithms and this work In the dice roll example, the inference algorithm used was direct enumeration of all possible outcomes. While this can be fast for simple distributions with mostly independent parts, more sophisticated techniques are necessary in order to obtain answers in feasible amounts of time; the full event tree is usually infeasibly large to operate over directly.

In this work, the focus is on increasing the efficiency of inference in two main ways: making an existing inference algorithm faster or more easily applicable, or designing an algorithm around a different perspective of the distribution.

In the following section, we will introduce a key class of inference algorithms, Markov Chain Monte Carlo (MCMC), that features throughout the work.

2.2 Markov Chain Monte Carlo

The inference techniques in this work are mostly Markov Chain Monte Carlo (MCMC) inference methods. In this section, we give background on MCMC.

2.2.1 Approximate inference

We begin by discussing the class of inference methods MCMC falls under: approximate inference.

In inference, we are prohibited from using direct enumeration in all but the simplest cases. In fact, such *exact* inference methods are usually infeasible, and we must turn to *approximate* ones.

Approximate inference covers such methods as:

1. Belief propagation [32]: locally updating interaction terms between random variables to approximate marginal probabilities.

2. Variational inference [9]: using a simple distribution to approximate a more complex one.

3. Monte Carlo sampling: producing random outcomes and estimating probabilities by binning the random outcomes.

Although the techniques described in this work can apply to any inference scheme, here, we are mainly concerned with the random sampling methods. Markov Chain Monte Carlo (MCMC) falls under this third category.

MCMC is a class of approximate, sampling-based inference methods that works by repeatedly perturbing a vector that represents the current setting of random variables: an outcome that is treated as the "current" one. This is also called the *state*. The next setting (or state) is obtained by applying the perturbation operator.

As we keep applying perturbations, we visit more states. The idea of MCMC is that we can design a perturbation operator so that if we record all the states that have been visited, we obtain an approximation of the distribution of interest; there will be more states clustered in areas of high probability, and states with lower probability will occur less often.

MAP inference. Note that in many situations where MCMC is used, we in fact cannot hope to obtain a reasonable approximation of the entire distribution. However, we are also often interested in inferring only the *modes* of the distribution (the states with the highest probabilities).

There are many situations in which MCMC is not usable for characterizing the entire distribution, but can easily find modes. This is called maximum a posteriori (MAP) inference, and many examples in this work are examples of MAP inference.

As a further note, if MAP inference is desired, the "MCMC" perturbation operator need not even approximate the distribution in the limit, but is still useful for finding modes and conducting MAP inference. The stochastic perturbation of MCMC serves as a simple and general scheme for global optimization.

2.2.2 Formulation of MCMC

MCMC methods largely consist of two elements:

1. A *perturbation operator* $K(x'|x)$.
2. The distribution of interest $P(x)$, and a way to evaluate (possibly unnormalized) densities $P(x)$.

$P(x)$ gives us the (possibly unnormalized) probability at any point in the state space, and $K(x'|x)$ tells us how to perturb one state x to find the next state x'.

Approximation criteria. Clearly, not every choice of $K(x'|x)$ will end up generating a trajectory of states that approximates $P(x)$. In fact, many things about $K(x'|x)$ can "go wrong" along the way.

Therefore, practitioners have come up with a set of criteria for the resulting trajectory of states to approximate $P(x)$. The first two, given below, are what it takes for the resulting trajectory of states to even approach a well-defined distribution in the first place:

1. *Recurrent*: that every state x with $P(x) > 0$ is reachable from any other such state using $K(x'|x)$. In other words, if $K^n(y|x)$ represents the application of the

perturbation n times, we can always pick n large enough so that $K^n(y|x) > 0$ for all (x, y).

2. *Aperiodic*: that the greatest common divisor of *revisit times* is equal to 1; that is, the g.c.d. of all the least k such that $K^k(x|x) > 0$ (a revisitation) is equal to 1.

With these two conditions, it is guaranteed that the resulting trajectory of states will converge to some distribution of states in the limit of infinite perturbations. The third condition, *stationarity* or *global balance*, is a check that the resulting distribution is the distribution of interest:

$$\sum_{x'} P(x)K(x'|x) = \sum_{x'} P(x')K(x|x') = P(x).$$

In other words, $P(x)$, the probability of interest, should be the fraction of time the chain spends in state x. This fraction is equal to both the total probability of being in some other state x' (possibly equal to x) and arriving at x, and the total probability of being at x and leaving to some other state x' (possibly equal to x).

In this work, we mainly focus on the Metropolis-Hastings MCMC method. For Metropolis-Hastings, a stronger condition, *reversibility*, is imposed on $K(x'|x)$ that allows more user-friendly constructions of perturbation operators:

$$P(x)K(x'|x) = P(x')K(x|x').$$

2.2.3 Metropolis-Hastings

We now explain Metropolis-Hastings, which features in many of the inference methods in this work. Metropolis-Hastings (M-H) is a very general, widely used, and simple MCMC inference algorithm. A more detailed treatment can be found in the original paper by Hastings et al [17]. M-H takes a probability density function $P(X)$ as input. $P(X)$ does not need to be normalized to 1. M-H repeatedly applies a "proposal

operator" $Q(X'|X)$ to a current state (assignment to X). As with other MCMC methods, the trail of states visited comprise a collection of states whose distribution approximates $P(X)$. The M-H algorithm is given below.

Algorithm 1: METROPOLIS-HASTINGS

Input: Density P, # iterations N, proposal operator Q
Output: Set S of samples approximating P

$S \leftarrow \{\}$
$X \leftarrow$ INITIALIZE()
for $i \in 1 \dots N$ **do**
  $X' \sim Q(X'|X)$
  $\alpha \leftarrow \min\{1, \frac{Q(X|X')P(X')}{Q(X'|X)P(X)}\}$
  $X \leftarrow X'$ w.p. α
  $S \leftarrow S \cup \{X\}$
return S

At each iteration, if the proposal state density $P(X') > P(X)$, X' is accepted as the next state. Otherwise, acceptance depends on the ratio of probabilities.

The combination of applying the proposal and accepting or rejecting the result induces a MCMC kernel $K(X'|X)$. We assume a choice of Q that makes $K(X'|X)$ *irreducible*; that is, any state can be visited. $K(X'|X)$ also satisfies *detailed balance*:

$$P(X)K(X'|X) = P(X')K(X|X').$$

In other words, the probability of going from any state A to any other state B is the same as the other way around. A consequence of detailed balance is the aforementioned *stationarity*; that the sequence of states approximates the distribution of interst.

In practice, however, M-H, like most MCMC methods, may take an unreasonable amount of time to converge. The utility of M-H is that it is a very general inference algorithm that is simple to apply; many choices of $Q(X'|X)$ are possible as long as the forward/backward transition probabilities can be computed, and the resulting

$K(X'|X)$ always satisfies detailed balance, and thus will always result in a Markov chain that converges to the distribution of interest.

In this work, we will assume that the current state consists of a vector of more primitive random choices: $X = (x1 \dots xn)$. For instance, in the Ising model, such a primitive choice could be a binary random variable (0 for one spin, 1 for the opposite spin).

Single-site and block perturbations. The proposal operator $Q(X'|X)$ is applied to the current state, producing X'. In many cases, only one component of the vector X is perturbed. This is known as a *single-site* proposal. Single-site proposals work well in many situations, as they minimize the amount of change the perturbation applies and thus maximize the probability of acceptance.

However, in situations where there are strong correlations between random choices, the single-site proposal is inefficient. To address this, many practitioners have adopted perturbations that change multiple components $x_{b_1} \dots x_{b_k}$ This is known as a *block* proposal. If block proposals are carefully designed in such a way that they respect correlations between variables, they can be more efficient.

2.3 Probabilistic programming languages

This work concerns the problem of how to increase efficiency of *probabilistic programming languages*, which are a general way to represent probability distributions.

The specification of probability distributions and inference algorithms can involve much labor; one is often programming the low-level representation of distributions, random variables, MCMC perturbations, and how these parts all work together.

It is desirable to have a more high-level solution: write down the distribution once, and have the inference automatically set up. This is the idea behind probabilistic programming languages.

A probabilistic programming language is an *executable specification* of a probability distribution. This goes beyond simply a way to write down the distribution; a probabilistic program runs on an execution engine for performing inference. Once

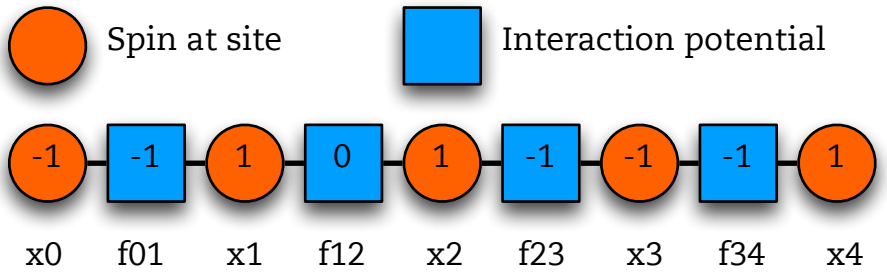


Figure 2.2: Ising model on 5 sites.

the distribution is specified by the program, the execution engine is free to employ a variety of inference techniques on the distribution.

In this work, I employ a variant of the Church [16] probabilistic programming language. This section gives background on how Church and probabilistic programming languages work.

Consider an example from statistical physics: the 1-D Ising model. Although this is a simple model that has been solved analytically, it well illustrates all of the essential mechanisms of a probabilistic programming language. The model (Figure 2.2) describes a line of sites, each of which has one of two spin states. There are interactions between every consecutive pair of spins, encouraging neighboring spins to be aligned (same state).

Figure 2.2 shows a 5-site 1-D Ising model. Orange nodes denote spins (of value -1 or +1), and square, blue nodes denote interaction potentials (-1 for pairs of opposite spins and 0 for pairs of like spins). The probability of any state $\mathbf{x}$ consists of the individual site probabilties times the weights due to interaction:

$$P(\mathbf{x}) = p(x_1) \prod_{i=2}^{N} f(x_{i-1}, x_i) p(x_i)$$

f are the interaction potentials, also called "factors" or "assertions" throughout this thesis. In Church, the Ising line model on N sites is expressed through the

following program:

```
(letrec ([site (lambda () (= 0 (randint 0 1)))]
         [ising-loop
            (lambda (prev n)
               (if (= n 0) #t
                   (letrec ([next (site)]
                            [interact (assert (= next prev))])
                           (ising-loop next (- n 1))))))])
   (ising-loop (site) (- N 1))
```

Everything is now described in terms of how it was generated. `site` is a function to sample a spin state. This function is used in a looping procedure `ising-loop`, which takes as argument the previous site sampled and the number of sites yet to generate.

Interaction between sites is accomplished through the assertion statement `(assert (= next prev))`, which specifies constraints on what is generated, "encouraging" the program to execute in a way that does not violate any assert statement.

In the formulations, each program execution is assigned a probability, and each violated assert statement weights the probability of the execution down by a predefined factor.

If more control over such weighting is desired, we use the `(factor ...)` primitive, which weights the probability of the current execution by log of whatever number is fed to it. We could have rewritten the `[interact ...]` binding using factors like so:

```
[interact (factor (? (= next prev) 0.0 -10.0))]
```

where `?` chooses between the second and third argument depending on if the first argument evaluted to true. Unlike normal if-statements, both alternatives are evaluated. Note that this is important for tracing, because `?` will be treated as another primitive operator and not incur overhead by depicting more program paths.

To sum up, with probabilistic programming languages, one expresses a probability distribution (and thus inference problems) by writing a program that shows how to generate each random variable. Assertions and factors provide a conditioning construct, allowing arbitrary constraints to be applied to the set of things generated.

2.3.1 Lightweight M-H

The promise of a probabilistic program is the ability to run inference out of the box using its execution engine. Much of this work can be seen as addressing inefficiencies in current execution engines for probabilistic programming languages.

To give intuition about how probabilistic program inference works, this section describes Lightweight M-H [46], a general and simple Metropolis-Hastings MCMC scheme for probabilistic programs that will also feature throughout the work in this thesis.

Need for naming random choices. One does not simply plug a probabilistic program into an existing inference algorithm. Probabilistic programs can express models with a varying set of random choices; randomness combined with the full expressivity of control flow operators allows many different sets of random choices to result from different exeuctions. This presents a problem for formulating inference algorithms such as M-H on probabilistic programs, as the vector of random choices will not be consistent.

Lightweight M-H [46](LWMH) is a recent, accepted method for running M-H on probabilistic programs. One of its key contributions is that it solves this naming problem; LWMH defines a scheme for naming random choices, allowing a well-defined scoring procedure. With each execution of the program, the set of random choices are computed, common choices are kept, and choices not common to both are discarded, with their density added to a transition factor that recapitulates the reversible-jump MCMC (RJMCMC).

Figure 2.3 is a flowchart captures the behavior of LWMH. We see that the scoring step involes some potential inefficiencies. First, it can be expensive to re-compute and look up the name of each random choice. Second, the whole program is being re-run from scratch, while many models do not require all random variables to be regenerated per scoring step. In the chapter on Shred, I will describe how to use execution traces to address these inefficiencies.

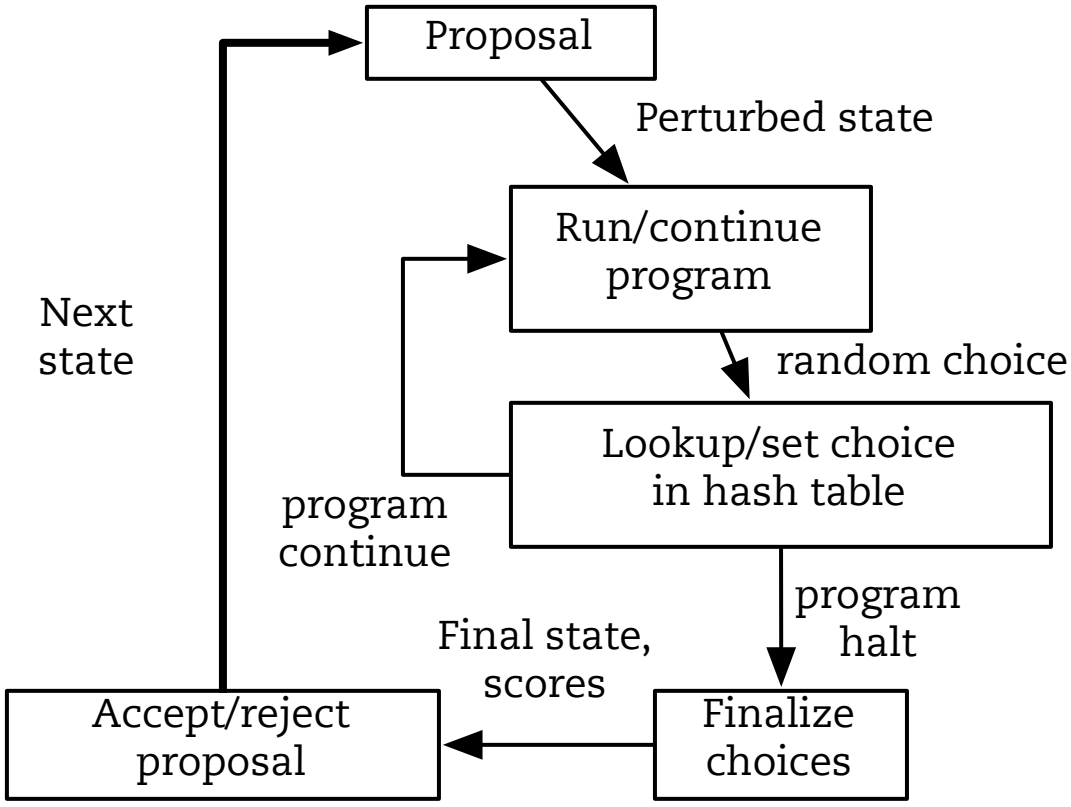


Figure 2.3: Lightweight M-H scheme.

2.4 Execution traces

Key to this work is the concept of the *execution trace*. As a probabilistic program (or any program) runs, one can record all of the primitive operations performed, such as `factor`, `=`, `+`, etc. This is called the *execution trace* and is the focus of this thesis. We will discuss how to use execution traces in order to specialize inference. The following is an execution trace of the 1-D Ising program:

```
x1 <- randint(0,1)
x2 <- randint(0,1)
b0 <- x2 == x1
assert(b0)
x3 <- randint(0,1)
b1 <- x3 == x2
assert(b1)
...
xN <- randint(0,1)
b_{N-1} <- xN == x_{N-1}
assert(b_{N-1})
```

`randint` is produced by all of the calls to 'site' in the original program. The repeated visits to `[interact (assert (= site prev))]` show up as the repeated `xi == x_{i - 1}` and `assert(bi)`. We see that all control flow and non-primitive function calls have been removed, leaving behind only the sequence of primitive operations performed during execution.

The advantage of an execution trace is that it is in an extremely simple, portable format; it can then be re-compiled to a low-level language, or translated to a format suitable to run on a state-of-the-art inference engine.

Note that this depicts only one control flow path through the program. Any compilation or analysis of this trace would apply only to this control flow path. There is a tradeoff inherent in using traces; tracing becomes more applicable the better the set of traces captures interesting program behavior. On the other hand, if it takes too many traces to capture any program behavior of interest, it is better to use a different method.

In this case of the 1-D Ising model, there was only one possible program path. Throughout this work, I will show that there are many other useful problem domains that can be depicted using just one or a few traces.

2.5 Procedural content generation

Throughout this work, *procedural content generation* features as an application of probabilistic inference and probabilistic programming languages. Procedural content generation is the production of artwork and graphics through specification as a program. It has a long history, such as L-systems being used to create models of plant life [35] in the 1960s.

Probabilistic programming languages, being a general way to specify probability distributions, are also a viable method for procedural content generation; one simply defines the distribution over the artwork of interest. Inference can then be used to zero in on "desirable" variations.

In this section, we summarize the context in which procedural content generation is used, and its relationship to probabilistic programming.

2.5.1 The need for content

The booming entertainment industry (and increasingly other fields such as fashion, industrial design and architecture) relies to a large extent on visual spectacle. Leaving aside the question of whether this should be the case, the top grossing movies in Hollywood all feature the latest and greatest in computer generated imagery, such as the Transformers series of movies. Additionally, the video game industry will eclipse both movie and music industries, and their primary mode of interaction is through a virtual 3-D world.

Some of the technological issues in what *can* be generated and displayed by computer have been solved. There has been progress in this area. Rather, I focus on the problems that remain in *what to* display and *how*. In order to generate, say, a cityscape in which the Transformers play around and destroy things, it is traditionally

ω : **plant**
p_1 : **plant** $\rightarrow$ **internode** + [**plant** + **flower**] − − //
[− − **leaf**] **internode** [+ + **leaf**] −
[**plant flower**] + + **plant flower**
p_2 : **internode** $\rightarrow$ F **seg** [// & & **leaf**] [// ∧ ∧ **leaf**] F **seg**
p_3' : **seg** $\xrightarrow{.33}$ **seg** [// & & **leaf**] [// ∧∧ **leaf**] *F* **seg**
p_3'' : **seg** $\xrightarrow{.33}$ **seg** *F* **seg**
p_3''' : **seg** $\xrightarrow{.34}$ **seg**
p_4 : **leaf** $\rightarrow$ [' { +f−ff−f+ | +f−ff−f }]
p_5 : **flower** $\rightarrow$ [& & & **pedicel** ‘ / **wedge** //// **wedge** ////
wedge //// **wedge** //// **wedge**]
p_6 : **pedicel** $\rightarrow$ FF
p_7 : **wedge** $\rightarrow$ [‘ ∧ F] [{ & & & & −f+f | −f+f }]

Figure 2.4: A stochastic L-system (left) and the generated flowers (right). This is taken from Figures 1.26 and 1.28 of *The Algorithmic Beauty of Plants* [35]

the job of many artists working together, with each artist creating a small part.

Modeling and placing objects in a virtual environment represents a tremendous amount of manual labor. This makes the automatic generation of such environments by computer a very attractive idea. It saves the time of artists, who then after using a procedural modeling system become more critics and judges—selectors—of computer generated art than the laborers involved in creating the pieces.

2.5.2 The easy case

In many cases, automatic content creation by computer works very well, such as with the generation of trees and cityscapes by grammars. This is because trees and cityscapes are extremely hierarchical and self-similar, which is natural to express by computational process. With a grammar, one can manually specify merely the *rule* to generate a single tree branch, and then rely on the grammar derivation process to produce trees of astounding complexity.

Figure 2.4 shows an example domain, flowers, that are amenable to this easy case of rule-based generation. The flowers were created by a *L-system* [35]. The L-system can be seen as a program that executes by repeatedly expanding the bold symbols according to what is on the RHS of the arrow ($\rightarrow$). The L-system may do

some stochastic sampling, but the simplicity is in the fact that every "run" of the "program" yields a plausible looking model.

2.5.3 Content generation is hard in general

In general, things are not so easy. In other cases, the purely forward-generative approach of grammars does not work so well. Consider the placement of furniture in a room according to practical and aesthetic constraints. It is difficult to specify a rule that will produce all plausible arrangements of a certain class of objects subject to constraints. If one wants the sofa to face the TV and to be flush against the wall, which do we generate first? The TV? The sofa? The wall?

What if we generated the TV and it was directly up against the wall, disallowing the placement of a sofa? With a purely forward generation approach, we are generally stuck; we need to make up further rules that tell us what to do in this situation, and more and more until we admit that is easier simply to place the furniture manually in Maya or 3ds max.

2.5.4 Solving for constraints by inference

This is where the relationship to probabilistic inference and probabilistic programming languages takes place. Instead of hard-coding forward generation rules that grow ever more cumbersome, one can simply encode *what* to generate, possibly as a probability distribution. The problem of solving constraints then becomes a problem of probabilistic inference.

Merrell et al [29] considered this problem. Their solution was to employ a general inference technique, parallel tempering Markov Chain Monte Carlo. Similar ideas were tried in the paper on Metropolis procedural modeling by Talton et al [38], where an existing forward-generating grammar is tagged with random variables and inference is performed over them, constraining the resulting model.

The general idea behind all of this work is that if one can specify some kind of *scoring function* or *soft recognizer*—a function that is simple to compute that tells us how far off we are from a satisfying model—one can then apply general optimization

techniques in order to produce satisfying models, rather than painstakingly make up forward generation rules for satisfying constraints.

In this work, we are interested in increasing the efficiency of inference by specialization through the construction of execution traces. Our techniques therefore apply to procedural content generation by probabilistic programs. In particular, in the chapter on model accretion, we show how execution traces can underpin the design of a more efficient algorithm to generate content by inference.

We will now go over the three contributions of this thesis in detail.

Chapter 3

Shred: Compiling Efficient MCMC Kernels

Shred is a tracing compiler for Metropolis-Hastings (M-H) on probabilistic programs. Shred is able to achieve MCMC kernel throughput (iterations per second) that is competitive with hand-coded solutions do not use probabilistic languages, using tracing and slicing to determine a minimum amount of computation to perform per M-H iteration.

3.1 Optimizing M-H: Ising example

The 1-D Ising model illustrates well how LWMH can be inefficient, and how Shred improves efficiency. Recall that the probability density of the N-site 1-D Ising model is

$$P(\mathbf{x}) = P(x_1 \dots x_N) = p(x_1) \prod_{i=2}^{N-1} p(x_i) f(x_i, x_{i+1}).$$

An iteration of M-H would proceed by constructing a perturbed vector $\mathbf{x}'$ that is different from $\mathbf{x}$ at a single site. Then the M-H acceptance ratio is

$$\min\{1, P(x')/P(x)\}.$$

Recall that our goal is to minimize the cost of this perturbation-acceptance/rejection procedure. It may be needlessly expensive to re-compute the entire ratio in each iteration of M-H. In fact, only a small, constant part needs to be recomputed if only one site is different. This can be seen by expanding the acceptance ratio out to the individual factors:

$$P(x')/P(x) = \frac{p(x'_1)f(x'_1,x'_2)p(x'_2)f(x'_2,x'_3)p(x'_3)f(x'_3,x'_4)\dots p(x'_{N-1})f(x'_{N-1},x'_N)}{p(x'_1)f(x_1,x_2)p(x'_2)f(x_2,x_3)p(x'_3)f(x_3,x_4)\dots f(x_{N-1},x_N)}.$$

If only say, x_3 is different from x'_3, then the above ratio reduces to

$$\frac{\cancel{f(x'_1,x'_2)}f(x'_2,x'_3)f(x'_3,x'_4)\cancel{f(x'_5,x'_6)\dots f(x'_{N-1},x'_N)}}{\cancel{f(x_1,x_2)}f(x_2,x_3)f(x_3,x_4)\cancel{f(x_5,x_6)\dots f(x_{N-1},x_N)}} = \frac{f(x'_2,x'_3)f(x'_3,x'_4)}{f(x_2,x_3)f(x_3,x_4)}$$

From how these terms line up, we see that if x' is only different from x at one site, only one or two $f(\cdot,\cdot)$ need be recomputed. In general, $P(x')$ is only different from $P(x)$ by at most two factors f. If the number of sites is n, we have achieved an optimization from $O(n)$ to $O(1)$. The situation is summarized below in a diagram for the case of 5 sites and 3 M-H iterations.

Shred automatically performs this optimization by tracing and slicing. In the next section, I explain how slicing works.

3.2 Slicing for the minimal change

Recall the trace:

```
x1 <- randint(0,1)
x2 <- randint(0,1)
b0 <- x2 == x1
assert(b0)
```

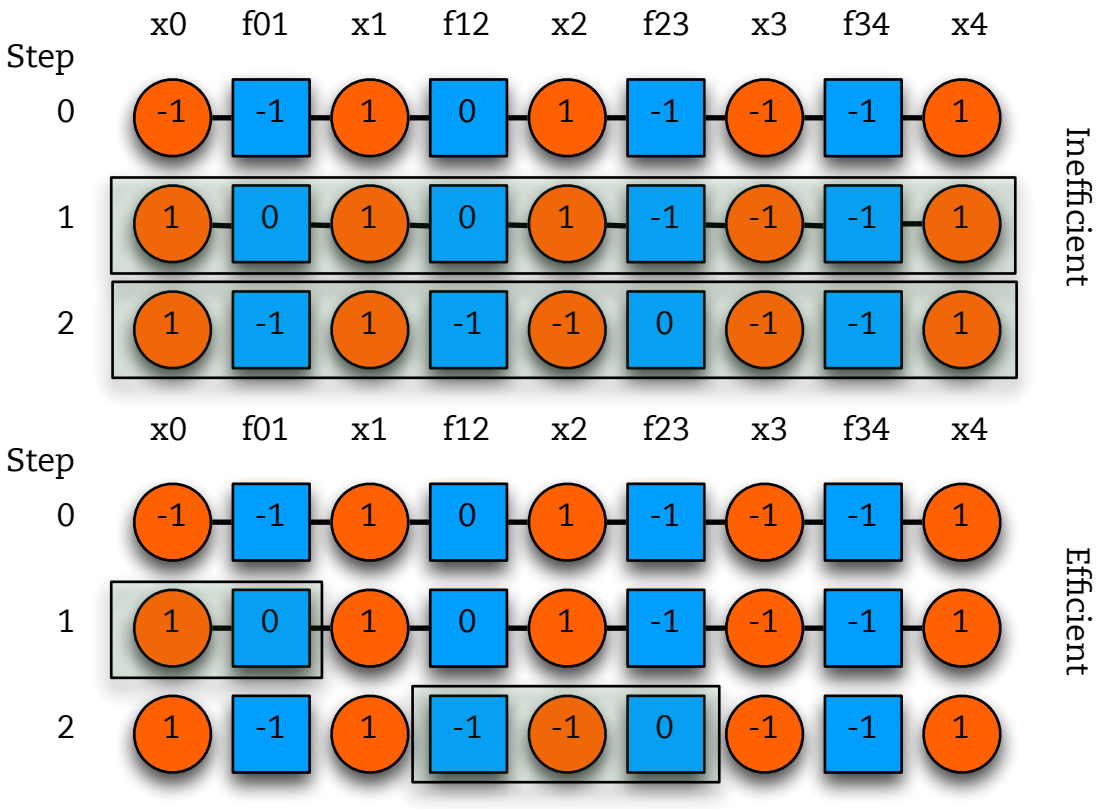


Figure 3.1: More efficient M-H by running the minimal necessary change.

```
x3 <- randint(0,1)
b1 <- x3 == x2
assert(b1)
...
xN <- randint(0,1)
b_{N-1} <- xN == x_{N-1}
assert(b_{N-1})
```

Slicing is formulated in terms of computing a minimal subset of the statements above for each possible proposal. Each 'randint' represents a choices that may be the target of a proposal. For example, suppose a proposal is made to `x1 <- randint(0,1)`. We then look for statements that use `x1`. We find one: `b0 <- x2 == x1`. So that's a statement that needs recomputation. We now also look for statements using `b0` in addition to those that use `x1`. We only find `assert(b0)`. The slice then consists of

```
x1 <- randint(0,1)
b0 <- x2 == x1
assert(b0)
```

Note that when we run a slice, these statements by themselves are not sufficent; `x2` is free and must be defined somewhere else already. We assume an enclosing scope where all LHS variables of statements are already defined. Running the slice then amounts to modifying some of these variables.

In general, for each random choice statement in the trace, we compute the entire set of statements *directly or indirectly* dependent on the value computed by the random choice statement. I will give a formal definition later.

3.3 Structural changes

The basic idea of tracing and slicing to recover the minimal M-H update is now clear, but what if the program has more than one possible trace? I call this phenomenon "structural change." Because the program execution is determined given random choices, we can isolate all choices that may result in a different trace or structure change.

Naturally, I will refer to these random choices as *structural choices*. All other random choices are assumed not to affect the set of random choices and are called *structure-preserving* choices.

3.4 Shred System

Shred then consists of the following components:

1. A trace generator and slicer for fast computation of M-H scores.
2. A mechanism to decide when a structural change has occurred, upon which a new trace is formed, or the existing trace remains in usage.

Since both are only concerned with what happens inside each iteration of M-H, the interface to Shred can then be essentially identical to that of Church Metropolis-Hastings (M-H): the program is run for a specified number of M-H iterations (expressed in terms of samples and lag) and a set of samples is returned.

The differences now primarily lie in the underlying behavior during each iteration of Metropolis-Hastings. Both systems will run the probabilistic program against a given set of random choices. However, when Shred runs the program, it will do so in one of two modes: 1) compiling traces and 2) running compiled traces. The first path is slow and the second is fast.

Figure 3.2 illustrates this design. For each MCMC proposal, Shred first executes the perturbation. If the perturbation happened to a structural variable, Shred first checks if there is an existing trace in the trace cache. This cache is keyed on the list of all values of structural variables. If the trace does not exist, Shred compiles and stores the trace as it execute the program to determine the score. The trace is stored and the proposal accepted or rejected according to the M-H acceptance ratio.

If the trace already exists, then Shred loads the trace, install the random choices in its memory, and runs the trace, which is fast. The trace produces a score and then the proposal state can be accepted or rejected. Note that if the perturbation is done on a structure-preserving proposal, we are assured that the current trace can be used, so Shred immediately runs the fast path.

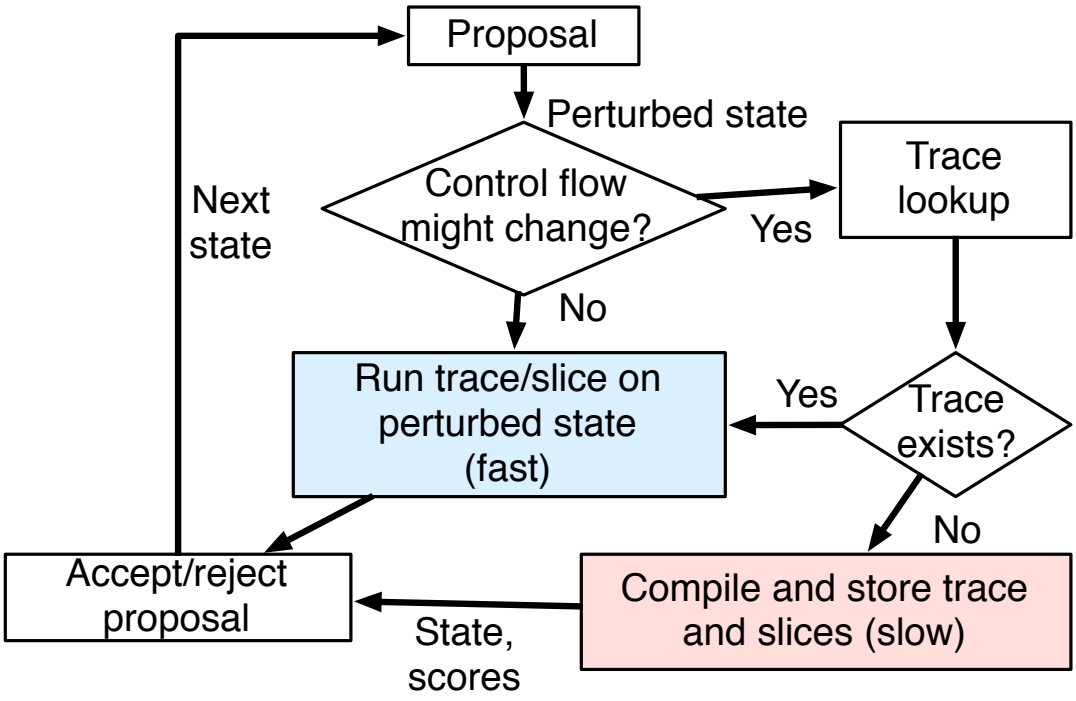


Figure 3.2: Shred system design.

Again, this performs better the less often traces are compiled; that is, the fewer paths occur in program execution.

3.5 Tracing and slicing algorithm

I now formally describe the tracing and slicing technique in this section. I build on the formulation used in the LWMH paper [46], which we briefly review here. Each run of the probabilistic program assigns values to a vector of random choices $X = (x_1...x_K)$, where K is assumed to upper bound the maximum number of steps in the computation. This leaves some elements in the vector to correspond to non-"existent" random choices. Let θ_i be the parameteres associated with each component x_i of X. The distribution associated with the program is then $P(X) = \prod_{i=1}^{K} p(x_i|\theta_i, x_1...x_{i-1})$. To save space, let f_i denote $p(x_i|\theta_i, x_1...x_{i-1})$.

We split the x_i into *structural* choices x_S that affect existence of other choices and *structure-preserving* choices x_N that do not. Without loss of generality, let $x_S = x_1 \dots x_S$ and $x_N = x_{S+1} \dots X_K$ be the structure-preserving choices. Note that this does not violate the conditional dependencies $p(x_i|\theta_i, x_1 \dots x_{i-1})$. No structure-preserving choice may influence any structural choice, otherwise it would be structural. Then the probability density of program executions is

$$P(\mathbf{X}) = \prod_{i=1}^{S} f_i \prod_{i=S+1}^{N} f_i = P(\mathbf{S})P(\mathbf{N}).$$

3.5.1 Tracing

Our tracing algorithm dynamically constructs subsets of $P(\mathbf{N})$ that correspond to extant sets of structure-preserving choices. We define our tracing interpreter as acting on a Church-like, call-by-value functional probabilistic language. This is for clarity and precision; in principle, our technique applies to any language admitting an implementation of Lightweight-MH.

Our language syntax is a variant of Church. Below is a grammar defining the

syntax. `v` are variables, `c` constants, `p` density functions, `op` primitive operations (such as $+, -, \times$, `cons`, `car`, and `cdr`), and `smp` are primitive sampling functions. Brackets denote a list of the included element.

```
e = lambda [v] e | app e [e]
    | if e e e | op [e]
    | v | c | letrec v = e in e
    | S p smp [e] | N p smp [e]
```

This is lambda calculus with structural (`S`) and structure-preserving (`N`) elementary random choice primitives (ERPs) [16]. `S,N` are user annotations that distinguish structural versus structure-preserving ERPs. Each ERP is parameterized by a scoring function, a sampling function, and a list of parameters.

Our tracing interpreter is a procedure that produces a trace along with the program execution. Traces take this form:

```
t = s t | END
s = tv <- T-Score p tv [tv] | tv <- T-PrimOp op [tv]
```

`T-PrimOp` represents a primitive operation. `T-Score` takes a scoring function, the value of a random choice, and parameters as input, returning the value and incrementing a global score variable as a side effect. Executing a trace computes a corresponding probability density.

I now describe how traces are produced from Church programs. Our tracing interpreter, with a few exceptions, works just like a normal interpreter for a call-by-value functional language. See SICP 4.1 [3] for a canonical definition. We first run a pre-process that puts a unique label ‘lab‘ at each syntax element for computing ERP addresses [46]. Actual tracing starts at a structure-preserving ERP:

```
T(addr, env, N lab p smp [e]):
    v+ = next_trace_variable();
    i+ = next_trace_variable();
    x+ = [ T(addr, env, e_i), e_i <- [e] ];
    x- = [ trvals[v], v <- x+];
    this_addr = cons(lab, addr);
    if ERP_exists(this_addr) then
        trvals[v+] = ERP_val(this_addr);
```

```
        this_score = p(trvals[v+], x-);
        score = score + this_score;
        update(this_addr, p, smp, this_score, trvals[v+], x-);
    else
        v- = smp(x-);
        trvals[v+] = v-;
        this_score = p(v, x-);
        score = score + this_score;
        update(this_addr, p, smp, this_score, v-, x-);
    add_stmt(v+ = T-score p i+ x+)
    return v+;
```

This behaves identically to the inner loop body of Algorithm 3 in the paper [46] describing Lightweight-MH, except we also add a `T-score` statement to the trace and return a trace variable, not the sampled value. `update` changes the database of ERP values, parameters, and scores to reflect the latest program execution. The detailed workings of `update` can be found in Algorithm 3 in the LWMH paper [46]. `next_trace_variable` produces a new unique symbol for use as a trace variable. `i+` is an free *input variable* which will be set by the MCMC kernel at each proposal. The result is a direct correspondence between structure-preserving choices and input variables. `[ f(x) , x <- l]` depicts a mapping operation over elements `x` in the list `l`, producing a new list with the transformed elements. The function `add_stmt` appends its argument, a trace statement, to the global trace. `trvals` associates trace variables with the normally interpreted values (if a normal value is input, it acts as the identity function).

For structural ERPs, `T` works the same way except no tracing is done and all parameters are normally interpreted:

```
T(addr, env, N lab p smp [e]):
    x+ = [ T(addr, env, e_i), e_i <- [e] ];
    x- = [ trvals[v], v <- x+];
    this_addr = cons(lab, addr);
    if ERP_exists(this_addr) then
        this_score = p(ERP_val(this_addr), x-);
        score = score + this_score;
        update(this_addr, p, smp, this_score, ERP_val(this_addr), x-);
```

```
    else
        v- = smp(x-);
        this_score = p(v-, x-);
        score = score + this_score;
        update(this_addr, p, smp, this_score, v-, x-);
    return v+;
```

This is key to our tracing technique: it has the effect of *partially evaluating* (pre-computing) away the computation associated with $P(\mathbf{S})$ and all naming computation.

The interpreter than resumes, working (mostly) just like a normal interpreter for a call-by-value language, treating the trace variable as another kind of value. In fact, the interpretation of `letrec` bindings, non-primitive function defintiions `lambda` and non-primitive function calls `(f x1 x2 ...)` is the same as that of a normal interpreter for Church. For branching and primitive operations, it is necessary to deal with trace variables and their associated values explicitly. Branches are handled as follows.

```
T(addr, env, If e1 e2 e3):
    v1 = T(addr, env, e1);
    if trvals[v1] then
        return T(addr, env, e2);
    else
        return T(addr, env, e3);
```

At this point, we are already in the structure-preserving part $P(\mathbf{N})$ of the distribution. This allows us to assume that if the conditional part of the branch `e1` is a trace variable, the branch resulting from using its actual value `trvals[v1]` is the one always taken.

For primitive operations, we trace if the arguments contain trace variables and evaluate normally if not. This is because it is not necessary to trace things like `X3 = 2 + 2`; it is faster to use the computed result somewhere else, e.g. `X6 = 4 + X5`. This is known as *constant folding* in the compilers literature.

Let `any_trace_var?` be a function that checks whether or not a trace variable occurs in a list of values.

```
T(addr, env, op [e]):
    vs = [T(addr, env, e_i), e_i <- [e]];
```

```
if any_trace_var?(vs) then
    v+ = next_trace_variable();
    trvals[v+] = op([trvals[v_i], v_i <- vs]);
    add_stmt(v+ = T-PrimOp op vs);
    return v+;
else
    return op(vs);
```

Our final step is to translate the traces to a low-level language. The generated traces do not contain any control flow, looping, or recursion. Additionally, they only assign a value to each variable *once*. Such programs are said to be in *static single assignment* (SSA) [5] form. SSA programs are easily translated into a number of low-level languages.

We generate a C++ function that calculates the subset of $P(\mathbf{N})$. Its arguments are the input variables corresponding to each structure-preserving choice, and its body is the C++-translated version of each trace statement specialized to the types used at each point. This is then used in the MCMC loop as a way to compute the density. Note that it is also possible to compile to other low-level languages such as LLVM [24] and Terra [14], which can more directly map to machine instructions on common hardware.

Stochastic memoization

Although it is not essential for tracing, models with Dirichlet processes [40] are included in the benchmarks used in this weork. I elected to represent them using Church-inspired syntax constructs: `DPmem alpha f` and `mem f` [16].

`DPmem` takes a concentration parameter `alpha` and a function `f` representing the base distribution, producing a function whose distribution is a Dirichlet process with the given parameters. `mem` takes a function, transforming it to only run once and have one return value for each unique incoming argument. In this work, we assume that no structural choices take place inside the procedures called by `DPmem` or `mem`, and that all procedures input to `DPmem` take no arguments. This is sufficient to express complex machine learning models.

We deal with these primitives in a manner that avoids trace switching, but generates code with control flow. We track which primitive operations and random choices take place under a `DPmem` or `mem` call. These form sub-traces. These sub-traces are then compiled to loop-free C++ programs that conditionally execute based on the corresponding (stochastic) memoization semantics.

For example, the `f` in `DPmem` is traced out as a sub-trace, and for each call to the `DPmem`-transformed `f`, a call out to the sub-trace is surrounded by an if-statement that involves a draw from the uniform distribution u $U[0,1]$. If $u < \alpha$ (`alpha` parameter), the sub-trace is re-run. Otherwise, a value is re-computed by re-drawing from a collection of samples.

Slicing

We adapt the general concept of *slicing* [45] to generate computations that do the least amount of work in accept/rejecting proposals to structure-preserving choices. We define our *slice* $S(v_i)$ as all trace variables (i.e., statements) needing recomputation upon change to v_i. Each such trace variable corresponds to a trace statement, so it tells us which statements to re-run upon the change to v_i.

$S(v_i)$ computes all direct and indirect dependencies. For a given trace variable v_i, v_i may appear on the right hand side of the statements of other trace variables v_j: `v_j = T-PrimOp op ... v_i ..` or ‘`v_j = T-Score op v ... v_i ....` In this case, we define v_j as *directly dependent* on v_i. We use $D(v_i, v_j)$ to denote the set of all such pairs of trace variables.

We now define *indirect dependency*. For the `T-PrimOp` above, it may be necessary to further compute the direct dependencies $D(v_j, v_k)$ of the LHS variable, and so on. For `T-score` statements it is not, as they merely return the value of some other input variable. Let $D_p(x, y) = \{(x, y) | \texttt{y = T-PrimOp op vs}, x \in vs\}$ capture this fact. The *indirect dependence relation* $I(\cdot, \cdot)$ is inductively defined as

$$I(x,y) = D(x,y) \cup \{(x,z) | (x,z) \in D_p(x,z) \wedge (z,y) \in I(z,y)\}.$$

Then, the slice is the collection of trace statements corresponding to the following set of trace variables:

$$S(v_i) = \{v_i\} \cup \{v_j | v_j \in I(v_i, v_j)\}.$$

In code generation, we associate a function `S_vi` for each trace variables slice $S(v_i)$ that executes its corresponding statements in order. We then promote all trace variables to global scope so that they are visible from every slice. We also avoid redundantly generating code. For example, if we already computed $S(v_4)$, and $v_4 \in S(v_2)$, we replace the resulting set of statements with a call to `S_v4`. Computing the density relevant to each structure-preserving choice x_k then amounts to setting the input variable `ik` to x_k and running the corresponding slice `S_ik`.

Our algorithm for computing $S(v_i)$ is the memoized depth-first search suggested by the definition of $I(x,y)$ above and our avoidance of generating redundant code. It produces a directed graph where each node represents a trace variable and each edge a direct dependency. The set of descendants of a node is the slice.

3.6 Results

We evaluated:

1. Effect of tracing and slicing on kernel speed (in iterations per second).
2. Cost of tracing and slicing.

We used the following probabilistic programs to evaluate tracing and slicing. These are widely used in machine learning.

1. Linear regression. This performs linear regression on a set of 100 (x,y) data points using Gaussian priors on the slope and intercept of the unknown line. There is almost no independent structure; we expect slicing to not benefit.

2. Hierarchical regression. This is an adaptation of the Rats example model in Volume I of the OpenBUGS examples repository. 30 rats are modeled, each with five data points describing weight per week. There is much independent struture; the estimated slope and intercept of the growth model for a particular rat can be updated independently of the others. We expect slicing to help a lot.

3. Dirichlet-multinomial mixture. This classifies four objects into three categories. This is a small model.

4. Infinite mixture model. Same as the previous model, but with a Dirichlet process (DP) prior over the set of categories.

5. LDA topic model. This is a synthetic example with 21 documents, vocabulary size 4 and two topics.

6. HDP topic model. Same as the LDA topic model, but with a DP prior on topics.

7. Citation matching. There is a DP prior on papers. There are 5 citations along with paiwise citation-paper similarity factors and paper-paper dissimilarity factors.

These models are all *closed-universe* models; i.e., they only have one possible trace. No extra time is spent compiling further traces and switching between them, as would be the case in open-universe models. We include a separate analysis of open-universe models later that factors in compilation time.

We ran these benchmarks with four different MH implementations: Church's Lightweight-MH, Shred, and hand-coding. The OS/hardware used was a Mac OS X 10.9 laptop running on a 2.3 GHz Intel Core i7 with 8 GB RAM. Ikarus, a native-code Scheme compiler, was used to implement Lightweight-MH and our tracing/slicing algorithms. Hand-coded models were programmed in C++. For probabilistic language

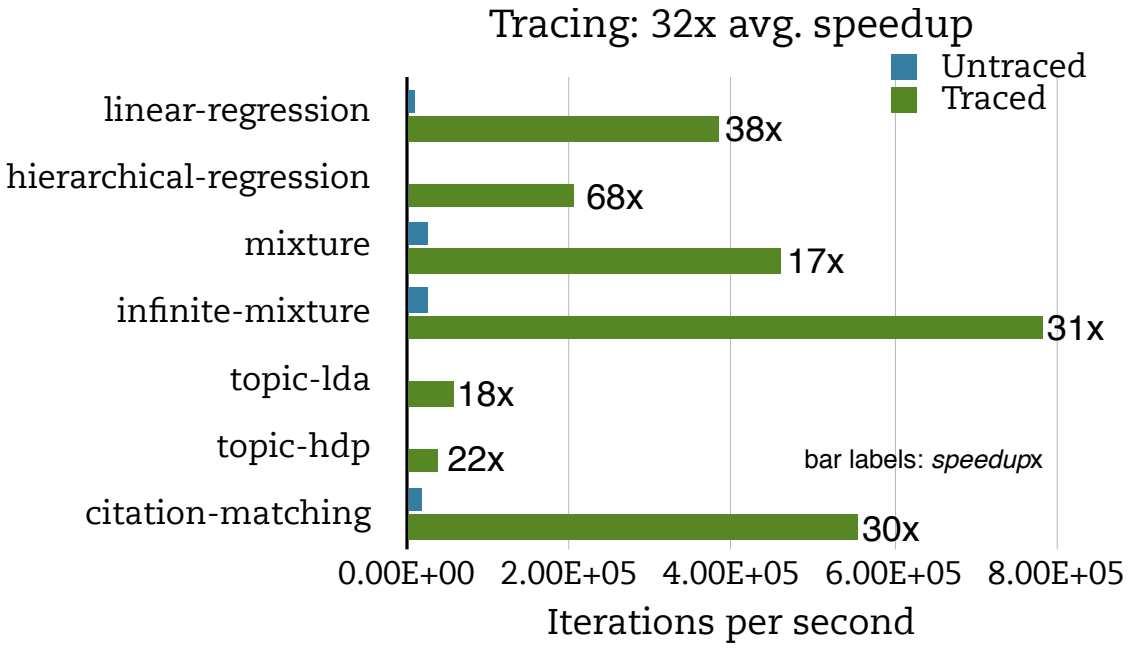


Figure 3.3: Speedup due to tracing.

implementations, we ran each model for 10^5 iterations. Hand-coded models were run for 10^6 iterations each.

3.6.1 Effect of tracing

First, we compare our tracing interpreter with Church and hand-coding. Figure 3.3 shows the change in iterations per second of the MH kernel using tracing versus without tracing. We see that tracing results in an average spedup of 32x, well over an order of magnitude.

The speedup also varies between models. Hierarchical regression model is sped up by 68x, while the mixture model is sped up by only 17x. We attribute this to the tracing interpreter removing overhead of the addressing scheme, and there being different amounts of addressing overhead for different models. Hierarchical regression contains many more random variables that go through a much longer loop than that of the mixture model, resulting in longer addresses and thus more addressing

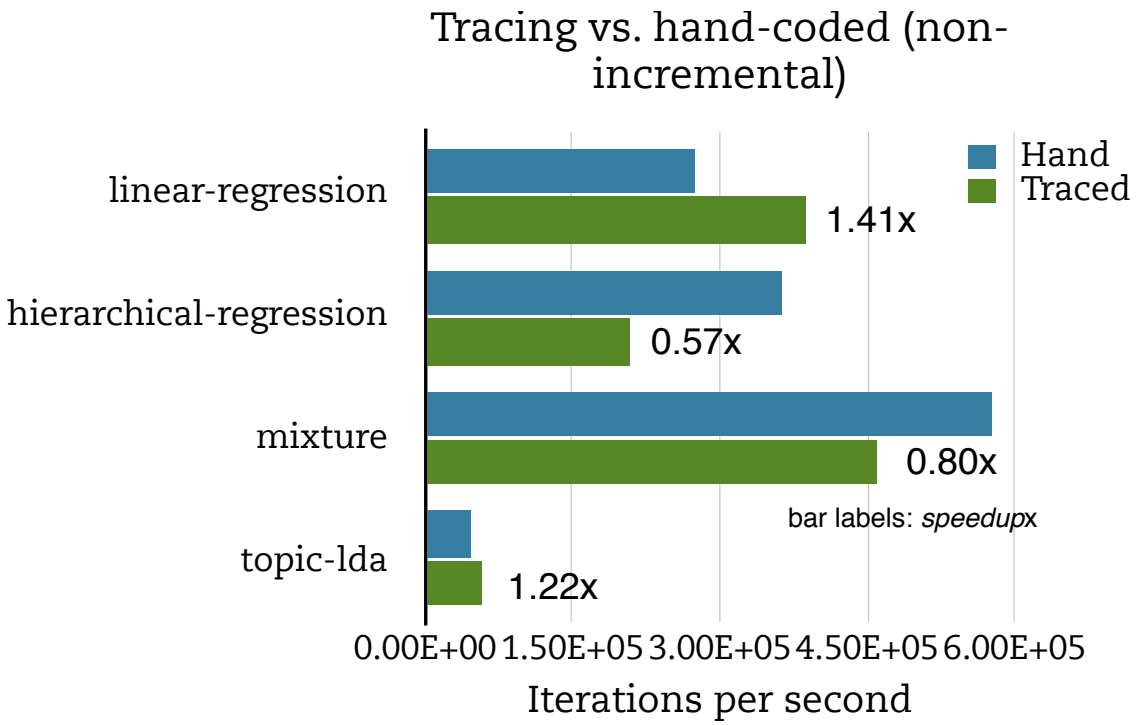


Figure 3.4: Performance of hand-coded C++ versus traced Church programs.

overhead. We also observe that larger models such as the LDA topic model achieve fewer iterations per second than the simpler models (such as the mixture model). This is a consequence of the amount of computation needed per iteration.

Figure 3.4 compares performance relative to that of a hand-coded implementation. We see that the code generated by tracing is about as fast as that of hand-coding, being close to the same speed on average.

3.6.2 Effect of slicing

Now we consider the effect of slicing, which attempts to recover the minimal computation per M-H iteration. Figure 3.5 shows speedup of slicing versus tracing. We see that slicing can sometimes be of a huge benefit, such as with hierarchical regression and the LDA topic model (8.8x and 19x speedups), while it can make the performance of the linear regression model worse.

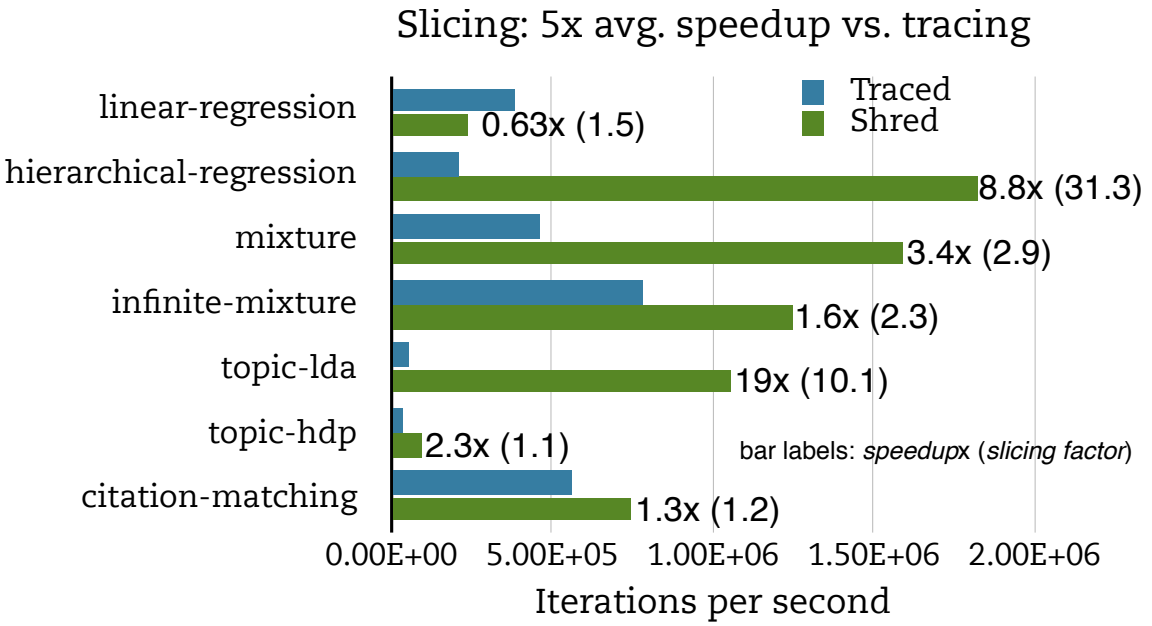


Figure 3.5: Speedup due to slicing.

We hypothesize that the longer (on average) the slices are in relation to the original trace, the smaller the speedup will be. We thus define a *slicing factor*: the trace length (number of statements in the trace) divided by the average length of a slice (averaged over all variables). A slicing factor of 1 means that slices are each as long as the full trace and we do not expect any speedup. A slicing factor of 10 means that slices have on average 1/10 the number of statements as compared to the full trace, and we expect the speedup to be around 10 times faster. Note that this is a coarse approximation; not every trace statement takes the same amount of time. Nevertheless, we see that the speedup is on the same order of magnitude as the slicing factor, and the two are correlated.

3.6.3 Costs of tracing and slicing

Figure 3.6 shows the contribution of slicing to the time taken to compile a trace.

We see that cost versus tracing alone is generally within a factor of two. This is despite that our algorithm for slicing is $O(N \log N)$ (N is the number of trace

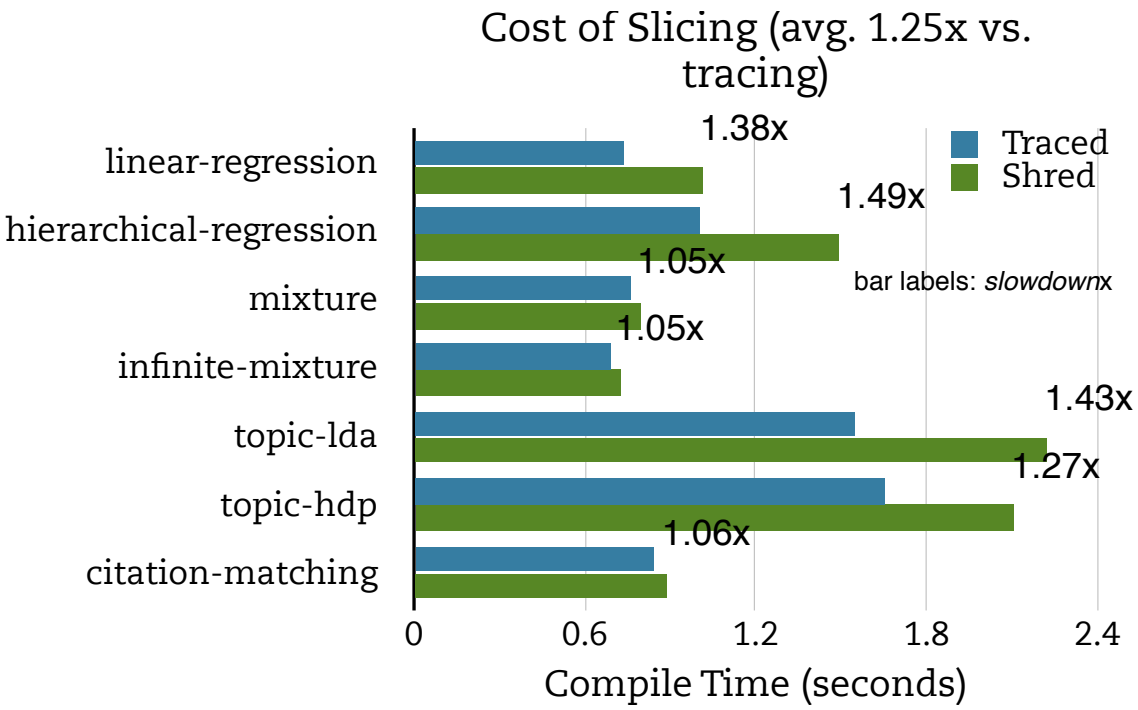


Figure 3.6: Cost of slicing.

statements) while tracing is $O(N)$. We attribute this to the fact that tracing has a much bigger constant factor involved in interpreting the original program step by step, while we employ data structures for slicing that do not have almost no interpretative overhead.

In absolute terms, we observe that the cost of slicing is small relative to the speedup. For instance, the LDA topic model costs 43 percent longer compile time to give a M-H kernel that is 1200 percent faster.

Overall our technique generates MCMC kernels that are dramatically faster than the unoptimized Lightweight-MH. This is especially true when models exhibit lots of independent structure, allowing slicing to isolate smaller computations. In the hierarchical regression and LDA topic model benchmarks, we achieve speedups of 598x and 342x relative to the untraced version!

3.6.4 Open-universe models

Open-universe models require the compilation of multiple traces and impose an overhead in both generating and compiling the extra traces and switching between them at runtime. We ran two open-universe models:

1. Model selection, choosing between the sum versus the product of two Gaussian variables.

2. Polynomial regression for degrees of polynomial 1 through 4 on 9 (x, y) data points.

We compared SHRED against hand-coding and Lightweight-MH, evaluating both iterations per second and total runtime.

We see that although our technique results in a clear speedup of the kernel by around an order of magnitude, hand-coded versions of the open-universe models are still much faster, by around an order of magnitude. We attribute this to the need to repeatedly save and load traces whenever a structural change occurs.

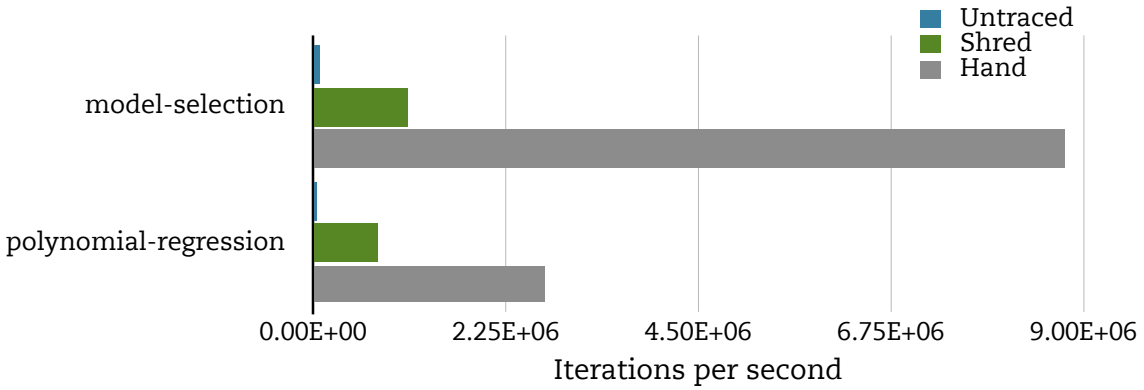


Figure 3.7: Performance on open-universe models.

3.7 Related Work

Probabilistic programming languages. Our system provides a much more efficient implementation of Metropolis-Hastings-based queries [46] for the Church probabilistic programming language [16]. Other inference algorithms for Church can be better for some classes of programs [37]. Our techniques could likely be used to accelerate these other Church algorithms, and also algorithms for a variety of other universal probabilistic programming languages (such as Hansei [22] and Figaro [34]).

Other probabilistic programming languages take a different approach, using a simpler, non-Turing-complete language specialized for certain kinds of probabilistic computations. BUGS [41] and JAGS [1] focus on simpler probabilistic models with fixed control flow. Our techniques could apply to these languages as well.

Just-in-time compilation. Our technique has some aspects in common with Just-in-time (JIT) compilation, where we opportunistically replace segments of a running program with optimized versions of it to improve performance. JIT has a long history [6], with trace-based JIT compilation receiving renewed interest. Much of the work focuses on applying such techniques to JavaScript, a widely used language for specifying client-side scripts on web pages. In particular, by specializing the code in loops to the types actually used in them at runtime, speedups of an order of magnitude are possible [15].

Incremental computation. The discipline of *incremental computation* aims at minimizing computation in the situation of a function receiving a series of changing inputs. We do not address incremental computation in the same generality as those who have worked on self-adjusting computation [4]. Rather, we focus incrementalizing only our traces, which do not contain control flow and consist of a static sequence of function calls. We use a program slicing-like [45] method to compute the set of statements affected by a proposal to a random choice.

3.8 Discussion and Future Work

We have shown that compilation techniques—tracing and slicing—can generate MCMC kernels whose performance is on par with hand-written code. While the initial compilation overhead, which is sometimes in seconds, is not justified for simple models that are only run once, it is very useful in even somewhat complex models. Nevertheless, improving compilation efficiency is a viable avenue of future work. Adapting further techniques from the compilers literature such as hot path detection and trace trees [40] would allow online detection of such complexity, adaptively selecting paths to compile based on execution frequency. For MCMC, this would mean only tracing paths where MCMC will spend the most iterations. In addition, for some programs it is not necessary to generate the entire trace to determine the form of the slices. For example, for an Ising line model on a thousand sites, there are only three different forms the slice can take, not a thousand.

Our technique also motivates further work that simply use the optimized code. Two such ways include compiling ahead of time for an unknown data set and running several copies of the same compiled code in parallel, which both further amortize compilation time. We are exploring the generation of probabilistic hardware, which has extreme "compilation overhead" but lends itself well to these use cases. Moreover, there are inference algorithms such as locally-annealed reversible jump [49] that rely on running on a fixed set of variables for many iterations.

The original LIGHTWEIGHT-MH paper [46] highlighted the possibility of using general code transformations to improve performance of probabilistic inference. We

have largely succeeded, and we attribute this to the fact that the model representation is programmatic, allowing application and adaptation of general compilation techniques. Our technique could be the first of many ways in which such general techniques can be adapted to perform efficient inference.

Chapter 4

Solitaire: Traces for Procedural Modeling

Now, we consider how traces are useful for performing an altogether different kind of inference: systematic search. Systematic search, at its essence, is going through all possibilities one at a time until one finds some answer that satisfies some desired constraints.

1. In probabilistic inference, it can be useful to use systematic search to find points in the support of a distribution, or to repeatedly use it to find the MAP assignment.

2. For inference languages in general, systematic search can be quite useful when the constraints are difficult to satisfy and there are not that many satisfying assignments compared to the product of all domains of choice variables.

In this chapter, we investigate the use of systematic search for procedural content generation in graphics. We also address a limitation of the Shred system described in the previous chapter: having to generate and switch between multiple traces.

I therefore introduce Solitaire: a trace graph compiler for systematic search problems. Solitaire, like Shred, traces, but does not save individual paths; it saves a trace graph that can represent multiple control flow paths in one loop-free program. This

is a very flexible representation that can then be compiled to a formula for any state of the art constraint solver, such as Satisfiability Modulo Theories (SMT).

4.0.1 How versus What

The game of procedural content generation with constraints is finding the right spot in the classic tradeoff in difficulty between recognizing and generating. If it's easy to recognize it might be hard to generate. If it's hard to generate it's probably easy to recognize. With procedural modeling with constraints, it's replacing user-specified generation rules with user-specified recognizing functions. Although the difficulty is then left to the computer to generate satisfying models, we obtain an interface with greater expressive power.

And in this game, languages for inference such as probabilistic programming languages are the endpoint. This is because they allow an arbitrary mix of constraint-based modeling with forward generation. One can forward generate the parts that are obvious and leave the difficult parts in the form of constraints. The inference language's inference algorithm then (hopefully) produces satisfying models.

This is not a new idea; the notion of a language with both generation and constraints goes as far back as the 'amb' operator for Lisp (aka *nondeterministic programming*). In this work, we apply probabilistic/nondeterministic programs to procedural modeling. We produce a trace graph, allowing us to leverage the state of the art in existing algorithms for finding satisfying assignments (SMT solving).

4.1 Solitaire Overview

We cast procedural modeling with constraints as non-deterministic programs with `assert` statements used to denote constraints. Then, we repeatedly trace the original program, building up a trace graph. We then compile the trace graph to a SMT formula. Satisfying models correspond to satisfying assignments to the formula. This figure summarizes the situation:

To guide intuition, we will first explain how Solitaire is used to describe and

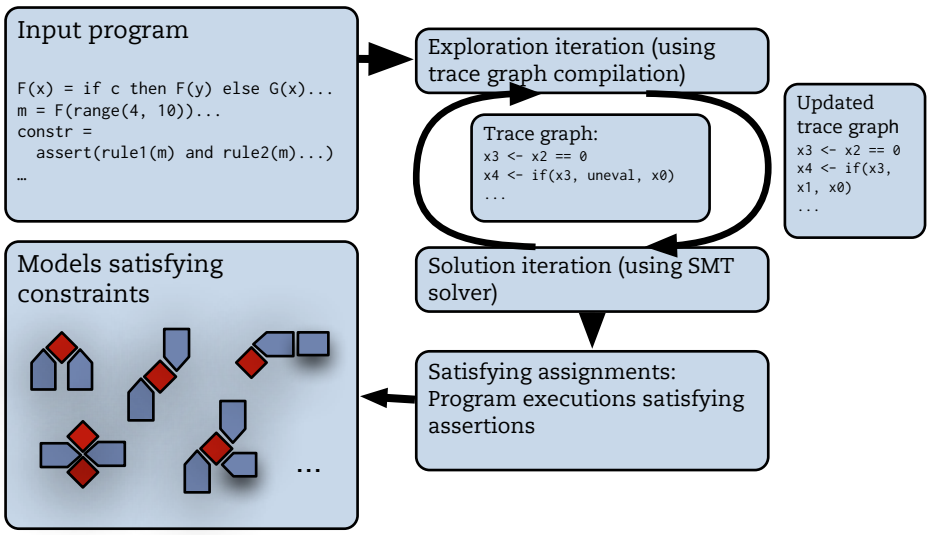


Figure 4.1: Solitaire system design.

generate a group of tables and chairs, with the tables adjacent to each other to form a larger table, and the chairs all facing the tables. Such layouts featuring furniture have featured in several instances of previous work [29, 50, 49] in which random walk techniques based on Meteropolis-Hastings were used to generate them. The constraints used in this example are as follows:

1. The tables touch to form a bigger table, and there are two tables.
2. All furniture elements are inside the room, and the room's shape fits the group.
3. The first three seats face the first table, and the second three seats face the second table.

Figure 4.2 shows an example layout.

4.1.1 Non-determinism and constraints

In Solitaire, the user specifies constraints using a combination of *non-deterministic values* along with the `assert` primitive. For instance, this is the program describing a

Figure 4.2: Workspace layouts: a domain where forward generation is difficult.

pair of numbers `x, y`, each between 0 and 10, such that their sum was 10.

```
x = range(0,10);
y = range(0,10);
assert(x + y == 10);
return x,y;
```

`range` specifies a non-deterministic value, much like a random choice in the probabilistic programming languages. `assert` essentially blocks any execution where its argument is false. In this case, satisfying assignments to (x, y) would then include pairs such as $(4, 6)$ or $(1, 9)$.

For a more complex usage of non-determinism and constraints, below is the definition of the facing constraint for furniture layouts:

```
facing = fun(from, to) {
    let rot = from.rot;
    assert(((from.x2 >= to.z1) => (rot == 0)) and
           ((to.x2 >= from.x1) => (rot == 2)) and
           ((from.y2 >= to.y1) => (rot == 1)) and
           ((to.y2 >= from.y1) => (rot == 3)));
}
```

`x1` and `x2` are accessors for the left and right edges of an object, respectively. `rot` is the orientation of the object. There are four allowed orientations corresponding to the cardinal directions; the object is rotated by `rot * pi / 2`. To perform non-deterministic object modeling with a facing constraint, the model writer creates objects where the `x1,y1,x2,y2` fields are assigned to non-deterministic values. Functions such as `facing` then restrict possible arrangements.

4.1.2 Concise specifications through recursion and iteration

The above mechanics of non-deterministic values, assetions, and systematic search are reproduced in existing constraint languages. However, Solitaire is built *on top of* a general purpose language, allowing such constraint programming to take advantage of the full expressives. This includes not just grammar-like recursion schemes, but arbitrary iteration patterns from programming languages.

In this instance, the layout includes two tables, four chairs, and one sofa. The code relevant to generation of tables follows:

```
make_table = fun () {
    width, height = 6, 3;
    rot = range(0, 3);
    x, y = range(0, 32), range(0, 16);
    return Obj("table", widht, height, rot, x, y);
}

repeat = fun(n, f) {
    if (n == 0)
        then return [];
        else return [f()] + fun(n - 1, f);
}

...
tables = repeat(2, make_table);
...
```

Note the definition of `repeat`, which is a recursive function for specifying multiple objects. This is similar in spirit to attribute grammars with recursive non-terminals carrying parameters, which are also suited for specifying multiple objects. Note that a recursive function was not necessary; it would have served just as well to employ an iteration construct such as `for` or `while`. Sofas and chairs are generated in a similar manner:

```
chairs = repeat(4, make_chair);
sofas = repeat(1, make_sofa);
```

Finally, the facing constraint is applied to the generated tables and chairs from before through *iterating* over each chair, selecting one of two tables non-deterministically, and applying the facing constraint:

```
chair_table_facing = fun(chairs) {
    t = tables[range(0,1)];
    for c in chairs:
        facing(c, t);
}
```

The utility of languages such as Solitaire for procedural modeling should now be clear: the user has *separated the concerns* of generating the model versus specifying constraints on it. This allows the user to deal with each one without having to account for idiosyncracies in the definition of the other.

4.1.3 Structural variation

Note that the layout discussed above constrains a fixed number of layout elements. With a non-deterministic/probabilistic language, one can easily express uncertainty in the number of elements. Solitaire can express arbitrary recursion and iteration with branch constructs, which makes it easy to express uncertainty in the number of elements in the layout. Design parameters over such structural properties can therefore be changed in a way that does not involve a protracted editing process. For instance, if the user wished that the number of tables and chairs were varying, and that there were more chairs than tables, only the lines specifying their generation need be changed, and the proper assertion added:

```
tables = repeat(range(1,5), make_table);
chair = repeat(range(2,6), make_chair);
sofas = repeat(range(1,3), make_sofa);
assert(length(tables) < length(chairs));
```

By changing just four lines, the program now specifies an *open-world* model; there are one to three tables, four to six chairs, and additionally the number of chairs is greater than the number of tables. We show the resulting layouts in the Results section.

4.1.4 Satisfying constraints by trace exploration and SMT solving

How does the algorithm behind Solitaire search for executions of the program that satisfy assertions? One can partition the entire space of possible executions (corresponding to models) into individual control flow paths, each of which are represented by a single trace.

In our case, we form traces and then pass each trace to a systematic search solver (here, the Z3 SMT solver). Our bet is that it is easy to find a path with solutions on it, and that satisfying models on these paths can be easily found using SMT solving.

Because each trace only represents a previously-seen control flow path, the possible program executions that a set of traces represent are in general a subset of all possible program executions. We construct a *trace graph*; an abstraction that compactly covers the program executions that are the union of all possible program paths represented by individual traces in aggregate.

Our algorithm can be separated into two separate phases, exploration and solving. The exploration phase simply constructs a trace graph. The solving phase employs the SMT solver to find satisfying assignments to the formula compiled from the trace graph. Below is pseudocode for our algorithm:

Algorithm 2: SOLITAIRE

Input: Program P, # exploration iterations E_N, # solution iterations S_N.
Output: Set S of models satisfying constraints

```
1  S ← {}
2  T ← EmptyTraceGraph()
3  for i ∈ 1...E_N do
4      T ← TraceExplore(P, T)
5  T_smt ← CompileToSMT(T)
6  for i ∈ 1...S_N do
7      s ← SMTSolve(T_smt, S)
8      if Unsat(s) then
9          return S
10     S ← S ∪ {s}
11 return S
```

We are given a program P describing a layout along with the number of exploration and solution iterations (respectively, E_N, S_N). The output is a set of models S satisfying constraints. Each exploration iteration updates a trace graph T using TRACEEXPLORE.

TRACEEXPLORE is the symbolic compiler heart of SOLITAIRE. TRACEEXPLORE runs the program freshly sampling all random choices, then adds the branches taken

by the resulting program path to T. Because we explicitly track control flow merges and common segments between paths, there may be many paths depicted for the cost of one extra random program run.

Next, the solution phase comes up, where we compile T to a SMT formula. Each solving iteration SMTSolve attempts to find one satisfying model that is not equal to any model found so far. Note that it is possible to run out of solutions because all of them have been found, or that the original model is infeasible to construct in the first place. In this case we exit early and return the current set of solutions.

4.1.5 Trace graphs

Trace graphs are a close cousin to the directed graph of basic blocks and trace trees in the compilers literature. Trace graphs avoid duplication of code that is common to multiple control flow paths. All control flow join/meet events are explicitly represented in the trace graph. This has two advantages, the first of which is to cut down on the amount of memory needed to represent the space of executions compared with just recording a flat list of traces. The second advantage is a little more subtle and has to do with how much of the space of execution is known after each exploration iteration. Our exploration step is based on running the program with a refreshed, random assignment of values to non-deterministic choices. While only one more concrete program path was actually visited by our exploration step, it may result in a disproportionate number of other paths that have been discovered by tracking join/meet events.

We illustrate the advantages of a trace graph over a flat set of traces by an example. Suppose the original program included many control flow events, such as the computation of a random subset of $\{1 \ldots n\}$:

```
flip():
    x <- range(0,1):
    return (x == 0);
subset(n):
    if n == 0:
        then return [];
```

```
else if flip():
        then return [n] + subset(n - 1)
        else return subset(n - 1)
```

`subset` forms a list of random numbers that is some subset of the integers $\{1 \ldots n\}$. On each iteration, `subset` first checks if `n == 0` in which case we are done with the subset. On other cases, it first samples a non-deterministic Boolean using the `flip()` function. If the `flip()` returns true, `subset` appends the current value of `n` to the result and continues. Otherwise it simply continues.

In terms of program paths, the possible set of paths is $O(2^n)$, exponential, because every different control flow path can represent only one possible subset. However, if we explicitly track control flow joins during trace exploration, and properly merge program states when control flow joins happen, we can cut this down to $O(n^k)$. Similar results have been reported in recent work on solver-aided DSLs and symbolic execution [42].

4.2 Formulation

At a high level, the SOLITAIRE algorithm combines symbolic execution/bounded model checking with a probabilistic/non-deterministic language [3], producing propositional logic formulae whose satisfying assignments correspond to program executions that do not violate any `assert` statement. In this section, we give the formulation of our symbolic execution in detail.

Interface. The input is how many times the program is symbolically explored: the number of exploration iterations. Each exploration iteration symbolically executes the program along a random control flow path until program halt, adding as much information as possible to an existing trace graph, which is compiled to a SMT formula. In each exploration iteration, every random choice is freshly sampled to encourage exploration of different paths.

4.2.1 Mitigating path explosion

The goal of symbolic execution coincides with ours: to find executions of a nondeterministic program that do not violate any `assert` statements. A major design decision is whether or not the space of program executions is explored on a path-by-path basis. In our setting, we would like to deal with 3D models that have a varying number of objects and consequently, a potentially very high number of valid program paths.

It may then be exceedingly costly to encode every path separately, which easily becomes a formula whose size is exponential in the average number of program steps. We would like a compact encoding that depicts possibilities of multiple paths in a way that maximizes the amount of shared structure, leading to a smaller formula.

Our solution is to allow path by path exploration, but at the same time, encode control flow merges explicitly, as is often the case in many symbolic execution/bounded model checking systems [12, 47]. Each exploration iteration on the program remembers the current trace graph, updating it in the portions where unexplored segments of paths exist.

This approach some similarities with a recently developed technique, the *lightweight symbolic virtual machine* [42], which addresses similar needs. SOLITAIRE is different in that it allows multiple executions of the input program. In particular, we do not eagerly evaluate both branches when encounting an `if`-statement; we only expand branches that have been concretely visited. Yet, due to explicit tracking of control flow merges, we are still able to obtain information about many unexplored paths.

Bitvector domain. Another set of design decisions concern the domains over which we are finding satisfying assignments. We chose to use integer arithmetic with booleans, with all values represented as 16-bit integers. It captures an interesting subset of all procedural models while at the same time, much work has been devoted to making this case run efficiently. In particular, Z3 [13], the solver we use, has optimizations specifically tailored to the bitvector case. While we can in principle extend our technique to handle arbitrary nonlinear real arithmetic with arbitrary data structures, such formulae can even be undecidable for satisfiability.

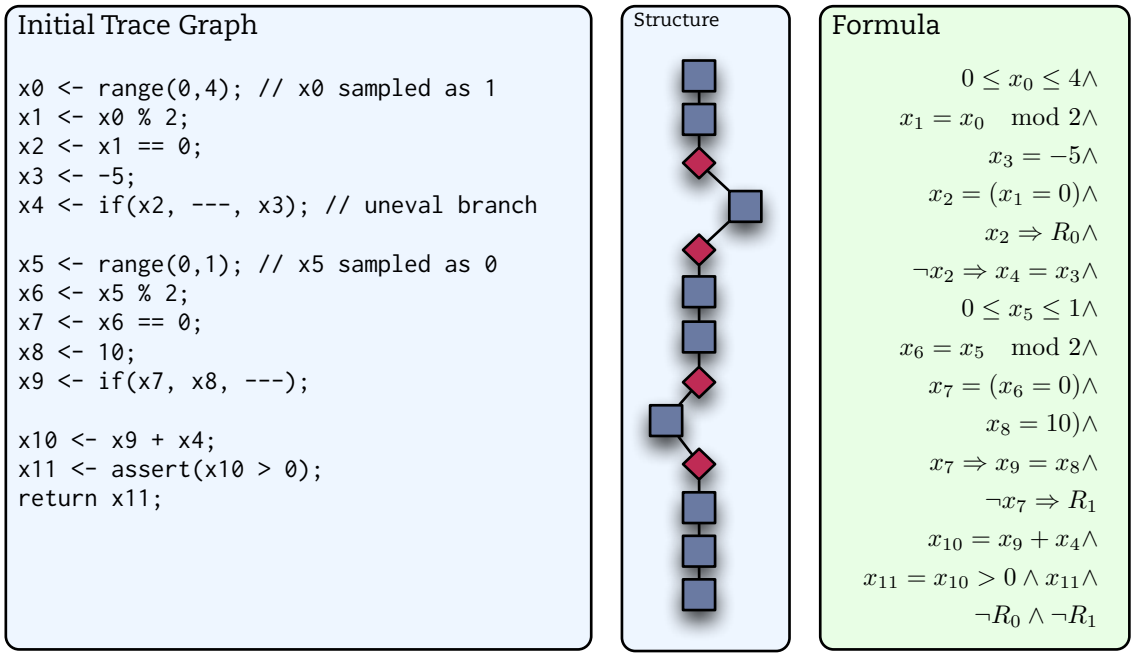


Figure 4.3: Initial trace graph and SMT formula.

4.2.2 Example

We show how our technique works through a simple example program:

```
F = fun(x) {
    if (mod(x,2) == 0) then return 10 else return -5;
}
n = F(range(0,4)) + F(range(0,1));
assert(n > 0);
return n;
```

The procedure `F` first checks whether or not its input is even with `mod(x,2) == 0`. If the input is even, 10 is returned. Otherwise, `F` returns -5. We calculate `n` by calling `F` twice, once with an integer in the range $[0, 4]$ (`range(0,4)`) and another time with a binary variable (`range(0,1)`). We then constrain `n > 0` using `assert`. Upon executing the program for the first time, we obtain an initial trace graph.

Figure 4.3 shows an initial trace graph along with the compiled SMT formula. The trace graph consists of all primitive operations taken during the program run. Note

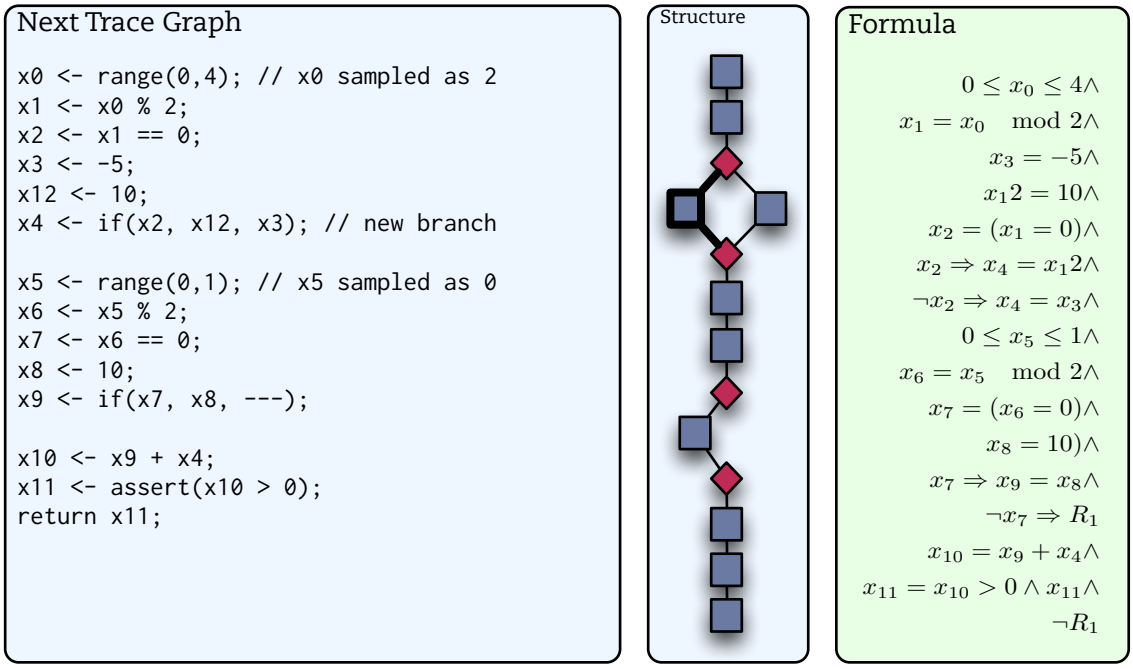


Figure 4.4: Next trace graph and SMT formula.

how we only took one branch of each `if` of the two calls to `F`. The first call, the input to `F`, `x0`, was sampled as `1`, which is odd, so the else branch was taken. The second call, the input to `F` was sampled as `0`, so the then branch was taken. Unevaluated branches are marked with `---`.

At any point, a trace graph can be compiled to a SMT formula, unevaluated branches and all. The right side of Figure 4.4 shows the resulting formula. Alternatives of branches have been compiled to logical implications. x_2 is the branch variable associated with the first branch. Note how we avoid unexplored branches, which is essential for the SMT solver to return a result consistent with program semantics. The first unexplored branch is represented as $x_2 \Rightarrow R_0 \wedge \neg R_0$; i.e., if the unexplored branch is taken, the formula cannot be satisfiable.

Now suppose we run the program again, keeping the previous trace graph in mind. Suppose all random choices were the same except for the first, where `x0` has been sampled as 2 (even). In this case, we have visited a new branch; the else-alternative of the first `if`. Figure 4.4 shows the resulting trace graph and SMT formula.

Note how the the unexplored branch is the only change to the formula. The rest of the trace graph was common to both paths and is therefore unchanged. In the SMT formula, R_0 has been taken out because both alternatives of the first branch have been explored. There is only one remaining unexplored branch, signified by the $\neg R_1$ constraint. Then, in order to find satisfying assignments, it is a matter of taking the generated SMT formulae and running a SMT solver.

4.2.3 Finding solutions

If a satisfying assignment exists, the SMT solver will return a *model*; a set of assignments $\{(x_i, v_i)\}$ that tell us what settings of variables it takes to make the entire formula evaluate to true. Some of these variables will correspond to particular random choices in the original program. We obtain the satisfying program execution (and model) by re-running the program against those satisfying random choices. To obtain further solutions, we ask for satisfying assignments that are different from all those produced so far: the *all-different* constraint.

Path-sensitive all-different constraint. In this all-different constraint, there is one more small though important implementation detail. Suppose the set of solutions found so far is the set $\{s_{ij}\}_{ij}$, where $i \in 1 \ldots n$ indexes the random choices and $j \in 1 \ldots m$ indexes over different solutions. The traditional all-different constraint would have us apply

$$\bigwedge_{j=1}^{m} \bigvee_{i=1}^{n} \neg(x_i = s_{ij}).$$

However, this is not desirable, because if a random choice does not exist in a particular solution, this can cause many additional solutions to be returned that appear no different from the existing one, since it is frivolously assigning different values to non-existent random choices. We also need to record the path conditions $p_i \ldots p_n$, one for each random variable. Then, we restrict solutions to those that are

different on existing random choices:

$$\bigwedge_{j=1}^{m} \bigvee_{i=1}^{n} (p_i \Rightarrow \neg(x_i = s_{ij}) \wedge \neg p_i \Rightarrow x_i = Q),$$

where Q is a reserved constant (any consistent choice will do). This causes the solver not to pick different assignments to non-existent random choices, and allows us to see a visibly different 3D model with each different solution.

4.3 Algorithm

We now give the full workings of the SOLITAIRE symbolic compiler.

4.3.1 Trace exploration

The main part is the TRACEEXPLORE procedure (see Algorithm 2), which takes a program P and current trace graph T, and returns an updated trace graph T' using the trace graph evaluator TGE. TRACEEXPLORE is run iteratively for the desired number of exploration iterations. Below is pseudocode for TRACEEXPLORE.

Algorithm 3: TRACEEXPLORE

Input: Program P, trace graph T
Output: Updated trace graph T (destructive update)

```
1 RewindState(T)
2 PushBranchFrame(T, top, true)
3 Tge(P, T)
4 PopBranchFrame(T)
5 return T
```

All the real action is in the call to TGE, but there are some salient details to introduce already.

Trace graph data structure. We represent the trace graph as a stateful object. The trace graph consists mainly of a hash table that maps from *branch frames* to

trace segments and *abstract stores*. Branch frames form *branch stacks*. We will further define these objects in the following paragraphs. The state also includes a *cursor*, which always stores the current branch stack and offset into the corresponding trace segment. RewindState restores the cursor to the empty branch stack and 0 offset.

Branch frames and branch stacks. A branch stack consists of a list of branch *frames*. Each branch frame consists of a dynamically unique (but same if re-visited on different runs) string label corresponding to the if-statement that caused it, along with `true/false` which describes which branch alternative is in question (`then/else` respectively). The current branch frame is the top of the branch stack, and the operations PushBranchFrame and PopBranchFrame push or pop one branch frame.

Unique but predictable variable names. We employ the *addressing scheme* [46] to obtain dynamically unique labels that are the same if re-visited on different runs. For the branch frame labels discussed above, a unique address is associated with the condition part of the if-statement. We also associate an address with every primitive operation. This is essential to being able to re-visit and update a trace graph on multiple program runs.

Trace segments and abstract stores. Trace segments are sequences of primitive operations that do not include branching. Trace segments are what is compiled to SMT formulae. Abstract stores depict concretely/partially evaluated values and list data structures that allow us to emit code to trace segments that do not include any list operations, leaving it in pure integer arithmetic and boolean logic. We can do this because our constraints all directly operate only on booleans and integers, save for `null?`, which we show can be reduced to decision trees over booleans.

4.3.2 Trace graph evaluator

Now we describe Tge, the trace graph evaluator. Tge is based on existing simple interpreters for call-by-value functional languages [3]. The difference is that when

executing primitive operations, statements are added to the trace graph, and when executing and exiting branches, the branch frame that is being pushed or popped is recorded, along with all integer/boolean values in the abstract store that may need to be used later on.

TGE differs from a traditional interpreter primarily in handling of primitive values, primitive operations, and branches. If a non-primitive function call or function definition occurs, we employ the same definitions as in a traditional call-by-value interpreter. These cases are exactly the `lambda`, `letrec` and `apply` (non-primitive) cases in SICP chapter 4 [3].

Primitive values. Our language operates on integers, booleans, and cons cells (lists). TGE, however, needs to be able to emit code for trace segments and to represent lists in the abstract store. We therefore pair every primitive value that is computed with a unique variable name. We call this pair of (variable, value) a *cell*. TGE expects all procedures to take and return cells.

Primitive functions. Primitive functions of the language are as follows:

```
+ | - | * | / | and | or | not | cond _ _ _
cons | car | cdr | null?
```

This is integer arithmetic, boolean logic with a ternary operator `cond`, and the classic list operations. For primitive functions that are not list operations, TGE works using the following pseudocode:

Algorithm 4: TGE-PRIM-NONLIST

Input: Address i, Primitive operator p, input cells $a_1 \dots a_k$, trace graph T
Output: Output cell r, updated trace graph T

1 $v_{1-} \dots v_{k-} \leftarrow \textsc{Vals}(a_1 \dots a_k)$
2 $r_- \leftarrow p(v_1 \dots v_k)$
3 $v_{1+} \dots v_{k+} \leftarrow \textsc{Vars}(a_1 \dots a_k)$
4 $r_+ \leftarrow \textsc{MakeVar}(i)$
5 $\textsc{Emit}(r_+ \leftarrow p(v_{1+} \dots v_{k+}))$
6 **return** $\textsc{Cell}(r_+, r_-)$

This works much like a normal tracing semantics; values and variables are extracted from the cells, the proper trace statement is emitted, the actual value is computed, and finally we return a cell as a result.

MakeVar is a function that deterministically computes a variable name given the current address. MakeVar in conjunction with the addressing scheme is used instead of a gensym, because we require consistent variable names on subsequent runs of the program.

Emit adds to the trace segment of the current branch frame. Note that Emit needs to only add code *when the current branch frame has never been visited before.* This is accomplished by querying the current trace graph's hash table for the current frame. If the current frame exists, Emit will do nothing. Otherwise, it will output its input code at the current cursor position, and advance the cursor by one step in its offset.

For the list operations

```
cons | car | cdr | null?
```

we need to do something more sophisticated, because we aim to emit only code that operates on integers and booleans. Instead of directly emitting a statement, these list operations act on the abstract store. A similar technique was used to achieve allocation removal optimizations in the PyPy tracing JIT compiler [10]. The pseudocode for each case is as follows.

Algorithm 5: Tge-cons

Input: Trace graph T, address i, input cells $(x_+, x_-), (y_+, y_-)$
Output: Output cell r, updated trace graph T

1 $r_+ \leftarrow \text{MakeVar}(i)$
2 $\text{StoreWrite}(T, r_+ : \text{Cons}(x_+, y_+))$
3 **return** $\text{Cell}(r_+, r_-)$

Tge-cons emits no code; rather, it updates the abstract store. StoreWrite takes the current branch frame's abstract store and updates it with the input association of variable and abstract frame value.

Abstract frame values. Abstract frame values can be the following:

$$V ::= \textsc{Cons}(s, s)|\textsc{Const}(v)|\textsc{Ref}(s)|\textsc{Phi}(s, s, s)$$

where s denotes some variable name used to lookup in the store, and v a primitive constant value (nil, integer or boolean). We can depict arbitrary pointer structures in this scheme. Cons represents a cons cell. Phi does the heavy lifting of representing control-flow-merged objects; the first argument is the condition variable of the if-statement of interest, and the second and third the identity of the object for true versus false evaluations of the condition variable, respectively.

Algorithm 6: Tge-car

Input: Trace graph T, address i, input cell (x_+, x_-)
Output: Output cell r, updated trace graph T

```
1 r+ ← MakeVar(i)
2 r− ← car(x−)
3 a ← StoreChase(T, x+, car, false)
4 if ¬List?(a) then
5     Emit(r+ ← FrameVal2Expr(a))
6 return Cell(r+, r−)
```

For car, we run Tge-car. In this case, StoreChase looks up the abstract frame value associated with the first member of the list being processed. If the result is not a list, we emit the resulting constant or reference. Otherwise, we simply continue interpretation. StoreChase can be considered the heart of the symbolic compiler. We will describe it in detail later.

Tge-cdr is similar, but uses cdr as the underlying primitive operation:

We now see the basic idea: perform an abstract store operation whenever a list operation is involved, and if the result is not a list, make it visible to the current trace segment. For Tge-null?, we know it will not return a list, so its implementation is shorter than the others:

Algorithm 7: TGE-CDR

Input: Trace graph T, address i, input cell (x_+, x_-)
Output: Output cell r, updated trace graph T

$r_+ \leftarrow \text{MAKEVAR}(i)$
$r_- \leftarrow \text{CDR}(x_-)$
$a \leftarrow \text{STORECHASE}(T, x_+, \texttt{cdr}, \texttt{false})$
if $\neg\text{LIST?}(a)$ **then**
  $\text{EMIT}(r_+ \leftarrow \text{FRAMEVAL2EXPR}(a))$
return $\text{CELL}(r_+, r_-)$

Algorithm 8: TGE-NULL?

Input: Trace graph T, address i, input cell (x_+, x_-)
Output: Output cell r, updated trace graph T

$r_+ \leftarrow \text{MAKEVAR}(i)$
$r_- \leftarrow \text{NULL?}(x_-)$
$a \leftarrow \text{STORECHASE}(T, x_+, \texttt{null?}, \texttt{false})$
$\text{EMIT}(r_+ \leftarrow \text{FRAMEVAL2EXPR}(a))$
return $\text{CELL}(r_+, r_-)$

Branching. Algorithm 9 describes the branching case of our trace graph evaluator. Depending on the result of the condition part, we evaluate the then- or else-expression using TGE. Branch frames are properly pushed and popped during this process.

We conceptualize branches as returning tuples of multiple values, each of which represents control flow merged symbolic integers or booleans. UPDATEPHI is used to add new return values associated with the branch frame. In this case, if the branch returns a non-list value, we need to update the associated tuple.

Abstract store update. Now we describe STORECHASE, the key mechanism to retrieve a value from the abstract store given some primitive list operation (`car/cdr/null?`). The pseudocode is given below:

v_+, is the variable representing the abstract store value on which we would like to perform the abstract operator f of interest. f returns `car/cdr/null?` of abstract store values, whose definition is shown in Algorithm 12.

First, we use LOADVAL to look up the abstract value associated with v_+ from

Algorithm 9: TGE-IF

Input: Trace graph T, address i, condition cell (c_+, c_-), then-expression t, else-expression e

Output: Output cell r, updated trace graph T

```
1  PushBranchFrame(T, c+, c−)
2  r+ ← MakeVar(i)
3  if c− then
4  |  (t+, t−) ← Tge(e, T)
5  |  (e+, e−) ← (uneval, uneval)
6  |  r− ← t−
7  if ¬c− then
8  |  (t+, t−) ← (uneval, uneval)
9  |  (e+, e−) ← Tge(e, T)
10 |  r− ← e−
11 PopBranchFrame()
12 if ¬List?(r−) then
13 |  UpdatePhi(c+, r+, t+, e+)
14 return Cell(r+, r−)
```

the store. The key to this algorithm is to avoid exponential time spent re-updating data structures when there are an exponential number of possible paths. We use memoization to accomplish this. e_{memo} is a string that represents the variable being operated on, v_+, along with the current abstract value to which v_+ is bound. If this changes, we know that we need to re-update. If re-updating is necessary (e_{memo} not found in memory), we attempt to apply the abstract operator f to the abstract value. If the abstract value merely points at other values, in the case of $\text{Ref}(v)$ and $\text{Phi}(c, v, v1, v2)$, we need to recursively call StoreChase to find what actual value they correspond to.

For control flow merged symbolic values $\text{Phi}(c, v, v1, v2)$, StoreChaseBr (Algorithm 11) recursively applies StoreChase and the operator f to each alternative of a branch, but memoizing the branch involved, not visiting any branch more than once for the current program run. This is key to achieving sub-exponential performance. In case e_{memo} is found in memory, the restriction R tells us whether or not to restrict residual updating. Abstract store entries from previous program runs may

Algorithm 10: TGE-STORECHASE

Input: Trace graph T, variable v_+, abstract operator f, is-restricted R
Output: Frame value a, updated trace graph T

$e \leftarrow \text{LOADVAL}(v_+)$
$e_{memo} \leftarrow \text{STRINGAPPEND}(v_+, \text{TOSTRING}(e))$
if $\neg\text{EXISTSINMEM}(e_{memo})$ **then**
  switch e **do**
    case $\text{CONST}(v)$
      $r \leftarrow f(e)$
    case $\text{CONS}(x, y)$
      $r \leftarrow f(e)$
    case $\text{REF}(u)$
      $r \leftarrow \text{STORECHASE}(T, u, f, R)$
    case $\text{PHI}(c, t, e)$
      $\text{REF}(t_+) \leftarrow \text{IF}(t =$
      $\texttt{uneval}, \text{REF}(\texttt{uneval}), \text{STORECHASEBR}(T, (c, \texttt{true}), t, f, R))$
      $\text{REF}(e_+) \leftarrow \text{IF}(e =$
      $\texttt{uneval}, \text{REF}(\texttt{uneval}), \text{STORECHASEBR}(T, (c, \texttt{false}), e, f, R))$
      $e_{merge} \leftarrow \text{STRINGAPPEND}(\texttt{ir}, v_+)$
      $e_{final} \leftarrow \text{PHI}(c, t_+, e_+)$
      $\text{WRITETOPARENTFRAME}(c, e_{merge}, e_{final})$
      if $\neg\text{LIST?}(t_+)$ **then**
        $\text{UPDATEPHI}(c, e_{merge}, t_+, e_+)$
      $r \leftarrow \text{REF}(e_{merge})$
  $\text{WRITEMEM}(e_{memo}, r)$
else
  if $\neg R$ **then**
    $\text{STORECHASE}(T, v_+, f, \texttt{true})$
  else
    return $LookupInMem(e_{memo})$

Algorithm 11: TGE-STORECHASEBR

Input: Trace graph T, branch frame (c_+, b), variable v_+, abstract operator f, is-restricted R

Output: Frame value a, updated trace graph T

```
1 e_memo ← StringAppend(v_+, c_+, ToString(b))
2 if ¬ExistsInMem(e_memo) then
3 |   PushBranchFrame(T, c_+, b)
4 |   a ← StoreChase(T, v_+, f, false, R)
5 |   WriteMem(e_memo, a)
6 |   PopBranchFrame(T)
7 |   return a
8 else
9 |   return LookupInMem(e_memo)
```

have been affected by the current run of StoreChase. If R is false, we continue to run StoreChase but with R set to `true`. This induces some slowdown, but is polynomial, not exponential.

Implementation. Solitaire and the above algorithms were implemented in Haskell [27]. As for the implementation of Tge, we used the *tagless-final* style [11]. In this style, metalanguage (Haskell) constructs dictate mechanisms of variable binding and function application for the non-primitive constructs `lambda,let,apply`. The implementation details then concern only the semantics of primitive functions and evaluation order. This is a good fit for our setting because we do not change how the non-primitive language constructs work; we only need to deal with definitions of primitive functions and branching.

As Haskell is a non-strict language, branching is easy to re-define in terms of lazy procedures that replace `if`. The one remaining issue is to make sure our language has call-by-value evaluation order. This is accomplished by wrapping all primitive functions in a continuation monad, which forces an eager evaluation order.

Algorithm 12: Abstract store operations

Input: Abstract value v, abstract operator f
Output: Abstract output value y

```
1  switch f do
2      case CAR
3          switch v do
4              case CONST(v)
5                  return CAR(v)
6              case CONS(x, y)
7                  return REF(x)
8              case OTHERWISE
9                  ERROR
10     case CDR
11         switch v do
12             case CONST(v)
13                 return CDR(v)
14             case CONS(x, y)
15                 return REF(y)
16             case OTHERWISE
17                 ERROR
18     case NULL?
19         switch v do
20             case CONST(v)
21                 return NULL?(v)
22             case CONS(x, y)
23                 return false
24             case OTHERWISE
25                 return false
```

4.4 Results

The purpose of SOLITAIRE, like grammars before it, is to replace a manual creation process with a concise description of the model in a formal language. It is about saving work in generating a *wide variety* of models, efficiently. Therefore, this section evaluates the SOLITAIRE algorithm in terms of its *efficiency* and the *visual diversity* of generated results. Efficiency will be evaluated against a random walk technique. Simulated annealing was chosen because of its effectiveness as a general optimization technique and its similar behavior to MCMC-based algorithms such as Metropolis-Hastings in this setting of finding satisfying solutions.

The following domains will be considered:

1. Workspaces. Refer to the overview section for a description of the constraints. The basic concept of a table and chair group is preserved. Here, I consider room layouts that include a varying number of groups along with secondary objects such as bookcases and whiteboards. The benchmarks evaluate fixed-structure layouts of varying size: `room-1` where there is 1 group of tables and chairs, `room-2`, where there are 2, up to `room-4`. `workspace-open` is a benchmark that combines the possible layouts of the other workspaces through making the number of table/chair groups uncertain over `1,2,3`. The resulting possible set of models is then the union of those of `workspace-1,2,3`. Exploration iterations produce larger formulas that represent a bigger subset of this model space.

2. Office building layouts. In these benchmarks, layouts of the interiors of office buildings are synthesized, including the arrangement of hallways and offices. Each office building consists of a set of connected hallways, each of which has one or more sets of incident offices. In each such set, the offices are constrained to be adjacent to the hallway and next to each other, so that they line up. Their dimensions are also constrained to be `4x8` or `8x8`, with positions and dimensions of all other elements constrained to be multiples of 4. I evaluated office buildings where the connectivity structure of the hallways formed a diamond (`office-d4-4-4`); that is, there are 4 hallways and each hallway is connected

to the next, including the last and first hallways. The first two hallways have two groups of 4 adjacent rooms adjacent and the last two hallways have just one group of 4 adjacent offices. There are also two other, smaller versions, `office-l3-3-4` and `office-l2-2-4`, where there are three (two) hallways connected in a chain with three (two) corresponding groups of offices.

4.4.1 Efficiency

This section compares SOLITAIRE against simulated annealing (SA). Simulated annealing is a global *optimization* technique, suited to *searching for a single solution*. In contrast, SOLITAIRE intends to produce a *different and satisfying* model every iteration. All experiments were run on an Intel Core i7 2.3 GHz on an early 2012 Macbook Pro.

Comparison with simulated annealing. SOLITAIRE was run for 8 exploration/-solution iterations (yielding 8 different solutions). Simulated annealing was run for 8000K iterations or 5400 seconds (90 minutes), whichever came first. The formulation of the SA kernel was a single-site proposal for each non-deterministic choice in the corresponding SMT formula. Every assertion statement violated was given a cost of 9000 to allow for partial credit. The annealing schedule increased the inverse temperature lineary from 0 to 100 in a number of steps equal to the number of iterations. Results are summarized in the table below. "sol" and "sa" refer to the average and standard deviation of time the solver spent to produce a satisfying assignment, taken over the single run of 8 iterations. Exploration iterations and the constraint to avoid previous solutions introduce sequential dependencies, resulting in increased variance.

Benchmark, Time (s)	sol	sa
`workspace-1`	4.45 avg, 0.10 stdev	-
`workspace-2`	16.89 avg, 0.59 stdev	-
`workspace-3`	53.43 avg, 2.46 stdev	-
`workspace-open`	59.70 avg, 41.15 stdev	-
`workspace-4`	121.91 avg, 7.07 stdev	-
`office-l2-2-4`	9.35 avg, 1.96 stdev	-
`office-l3-3-4`	45.67 avg, 13.17 stdev	-
`office-d4-4-4`	583.58 avg, 242.44 stdev	-

SA was *never able to find a solution* for any of the examples in the time allotted. It is clear, therefore, that the scope of problems SOLITAIRE capably handles is different from that of simulated annealing, and in turn, any MCMC method that shares its behavior. In addition, structural variation results in high variance in runtime. This could be due to the fact that at the beginning, the formula is small. Then as the exploration phase expands the formula, more time is needed to produce a solution.

4.4.2 Diversity

In this section, I show and discuss selected models taken from the benchmarks above. All models in the figures were taken from runs of 8 iterations (8 models) on the fixed-structure models (not `workspace-open`).

Workspaces. Figure 4.5 shows representative workspace furniture arrangements. The room's dimensions adapt to the number of groups. Furniture groups can either be laid out in a line or packed into a square. In addition, the pairs of tables can attach a variety of ways, subsequently allowing further variation in chair/sofa arrangements, even as they are restricted to face the tables. It is apparent that many visually and semantically different combinations of furniture are possible given the relatively simple set of constraints.

Office buildings. Even though the structural relationships between hallways was fixed, there is a large amount of variability of secondary structure and form, that is readily explored by the algorithm. Figure 4.6 shows the interiors synthesized from the corresponding benchmarks. Many of the buildings exhibit realistic features such as central courtyards, symmetric wings, and load-bearing spaces. These were emergent; i.e., not explicitly planned or specified as part of the constraints, and only apparent as the whole system including the solver is taken together.

As the difficulty of procedural modeling schemes increase, an important research area will be in combining probability and logic: what kinds of bias do randomly restarted systematic search techniques like the ones in Z3 exhibit for visually complex problems?

4.5 Discussion and Future Work

A non-deterministic language is capable of expressing and solving constrained procedural modeling problems. Program analysis provides tools and techniques to efficiently explore the execution space of a non-deterministic program and find solutions: directed testing and SMT solvers. Solitaire proved capable of generating constrained models in a wide range of domains: workspace furniture arrangements, office building interior layout, and platform game levels. All of these domains can be of sufficient complexity to be out of the reach of stochastic local search techniques like simulated annealing. The systematic search employed in SMT solvers is clearly more suitable for these domains.

This could be due to the higher level of coupling between constraints. In a furniture arrangement, it is sometimes possible to move just one furniture piece and improve the constraint satisfaction, but in a this setting, there is not much space for furniture pieces to move freely to an optiaml solution. In office building layouts, the situation becomes harder: no changes can be made to any model parameter without breaking some constraint. For the connected platforms, there were constraints dictating the position and length of walkways in relation to platforms, which themselves

needed to be away from other platforms by certain distances. Finally, there were constraints that the platforms touched particular points in space. It would be impossible to move or resize a walkway or platform without violating a constraint.

Broadly speaking, the potential for techniques like SOLITAIRE is to uncover new domains that were previously closed to procedural modeling techniques due to expressivity or efficiency problems. Game content synthesis is such a domain, with many complex pieces being required to fit together in space and satisfy semantically important constraints. Beyond that, computer-aided design (CAD) has traditionally dealt with the specification and direct manipulation of single models with many design constraints. It would be fruitful to explore how interaction techniques in CAD can integrate with the database-amplification advantages of procedural modeling, and conversely, to see how constraint satisfaction techniques employed in CAD apply to procedural modeling.

Another potential line of inquiry will be into refining the distinctions between the set of constrained procedural modeling problems efficiently solvable by systematic search such as SMT solving versus random walk techniques. SOLITAIRE is simply one of the early steps in a line of development of algorithms that bridge random sampling techniques with systematic search, where random sampling is only used to explore the set of possible paths. What essentially makes a constraint hard to solve by random walk? In what settings does systematic search of SMT solvers fail compared to random walk? And finally, is there a viable way to combine the techniques? A good answer to the last question demands further exploration of both fields as they apply to procedural content generation.

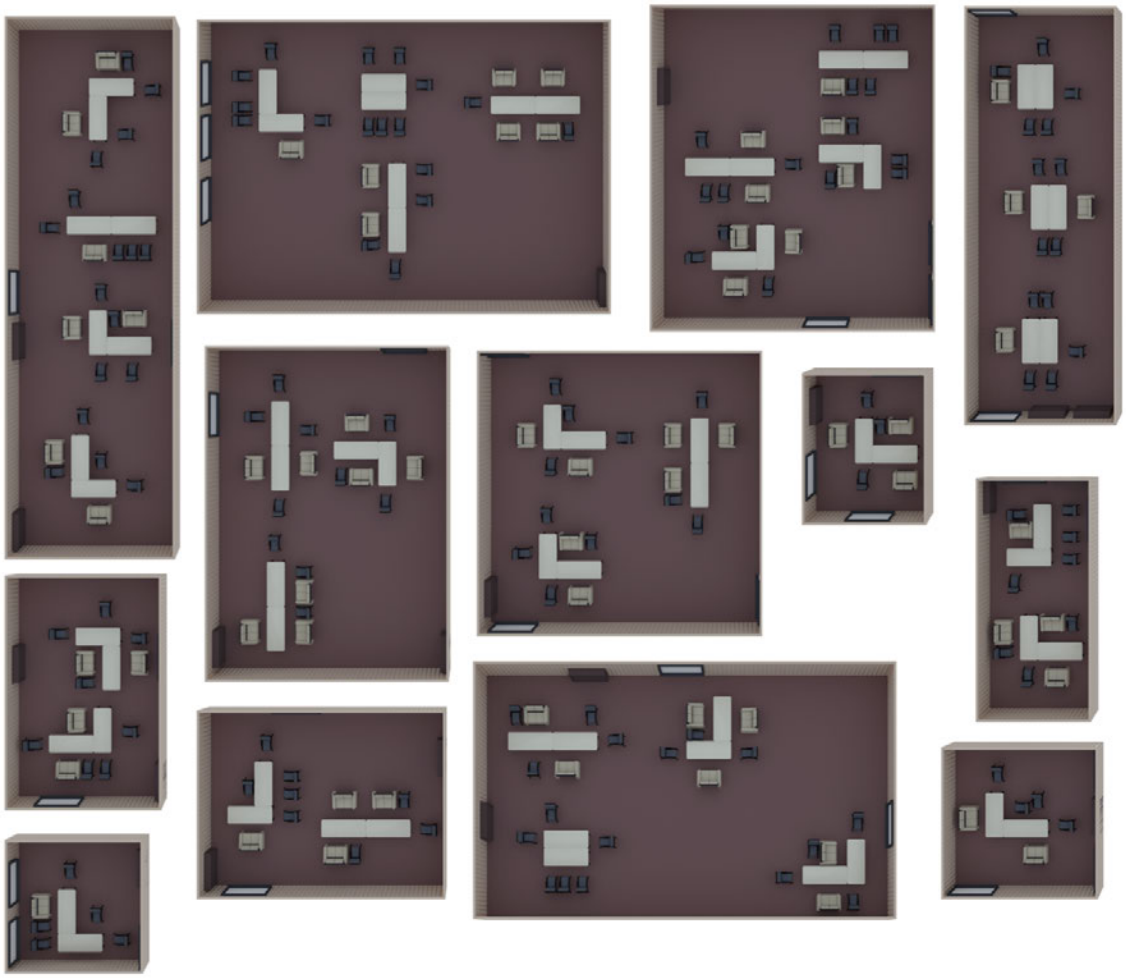


Figure 4.5: LEGO spaceships generated by SOLITAIRE.

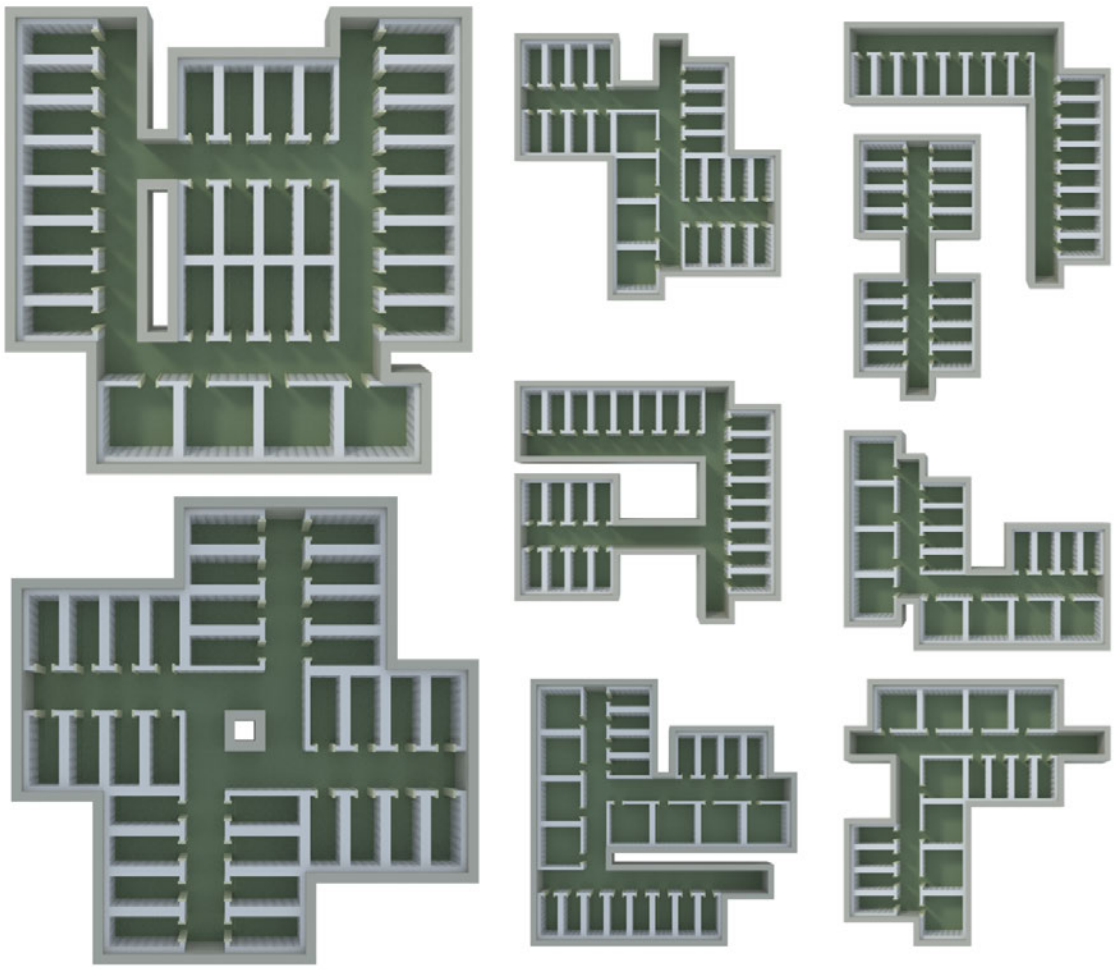


Figure 4.6: Office building interior layouts generated by SOLITAIRE.

Chapter 5

Model accretion

In this chapter, we present an algorithm that is inspired not with traces as intermediate representations for some other inference backend, but traces as abstract objects of inference themselves. Can we design an inference algorithm around recording previous program executions? What are its applications?

Procedural modeling using stochastic, generative processes can generate complex models starting from just a few symbols and rules. One common set of methods for creating generative models are L-systems [25] and shape grammars [36, 30]. Programming languages that make random choices can also be used to generate models. An office building with furniture in each office can be built using a program with nested loops. The outer loop generates high-level structure such as hallways, while the inner loops generate individual rooms, and then a furniture layout in each room. More recently there has been interest in using probabilistic programming languages to create generative models.

However, a long-standing problem with generative models is that it is hard to control the output. Without controllability it is difficult for artists to generate the content they want. One solution is to specify constraints on the output of the generative process. Hard constraints are logical conditions that must be absolutely satisfied and often deal with physical validity (e.g., two pieces of furniture do not intersect), and alignment (the left edges of the tables are along the wall). Soft constraints are continuous probability/cost functions describing relative desirability of models.

The most common method for finding models satisfying hard constraints is systematic search, such as using a SAT solver [28]. SAT solving techniques cannot guarantee a solution in less than exponential time $O(2^N)$, although in practice they employ heuristics that may allow them to find solutions more quickly. Stochastic search can also be used [29, 48]. Stochastic search-based solvers take $O(N^2)$ in the best cases, but unfortunately are not guaranteed to find a solution.

Our observation is that procedural models used in computer graphics often contain repetition and hierarchical constraints. Subsequently, one can rearrange or recombine portions of an existing model to generate new models. Since the existing models satisfy the constraints, the new combinations are also likely to satisfy the constraints. For instance, the furniture layout of one office can sometimes substitute for a layout in another office, since the constraints on furniture do not span multiple offices.

We thus propose a stochastic search-based synthesis algorithm, *model accretion* (MA). MA works by copying blocks of random choices from a set of initial solutions, recombining and rearranging known good configurations. The key idea behind MA is that the structure of the models reflects the hierarchical structure of the program. For example, the layout of furniture in an office is generated by a function call. If we group random choices within a function, we can copy them as a block into another run of the program calling that function.

We show that synthesis using MA is much more efficient than using existing systematic or stochastic search methods. We evaluated MA on office layouts, video game levels, and LEGO buildings. MA produced solutions 5-20x faster than systematic (by SMT solving) and stochastic (Metropolis-Hastings) searches. For video game levels, after MA generated the initial solution, MA produced further solutions 75x faster than the SMT solver.

5.1 Related Work

Example-centric synthesis. The synthesis of textures [44] and layouts, particularly from examples or a few template parts, is the most closely related set of techniques. There are image synthesis techniques that *reuse* or otherwise recombine its

portions, such as PatchMatch [8]. Synthesis with hard constraints can also benefit from examples [48]. There is also interest in synthesis off of the 2-D grid, such as with discrete element textures [26] and deformable templates [33]. We can also learn a complete generative process from the examples [19][39]. Model accretion adapts such example-centric techniques to general procedural modeling with constraints. Instead of a grid or a specialized set of geometric primitives, it employs the structure of a grammar or program (loops and recursion) to dictate the situations in which reuse happens.

Solver-based and procedural synthesis. Other approaches focus on using a solver and/or generative process to synthesize [30]. Quadratic programming [7], stochastic search [50] [29][49] and systematic search [28] are popular approaches. When one deals with arbitrary constrained generative processes, general stochastic searches [38] and systematic searches [28] are used. We thus compared model accretion (MA) against two leading approaches for synthesis of general procedural models with constraints: Metropolis-Hastings (M-H) and SMT solving. However, model accretion is more than another isolated solving technique; in the long term, its value is in amplifying the power of any given solver through recombination of the solver's outputs.

Tempering methods. Markov Chain Monte Carlo (MCMC) is a set of techniques for approximating a given probability distribution. The simplest and most general is Metropolis-Hastings (M-H) [17]. Optimization techniques such as simulated annealing [21] can also be formulated based on M-H. Based on this core concept, there are also *tempering* [31] schemes that employ a sequence of temperatures that serve as varying *levels of difficulty* for the search, so we can arrive at global optima sooner. Model accretion can be placed in the design space of such tempering methods. Rather than difficulty being controlled by an annealing sequence, it is controlled by copying random choices from less complex distributions. In our setting, easier models are not merely at a higher temperature, they are simpler models.

5.2 Overview

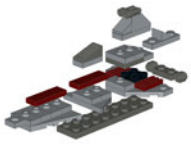
In this section, we give the intuition behind model accretion (MA) through an example: LEGO spaceship synthesis. Our spaceship program (Figure 5.1 left) generates a list of N spaceship structures.

Program generating the model:

```
main(N):
    res = struct_loop(N, [cruiser_base(0,0,0)]);
    pair_loop(res, lambda a, b:
                      assert(non_overlap(a, b)));
    return res;

struct_loop(n, acc):
    if (n == 0): return acc.
    else: let next = structure();
          assert(at_least_1_connected(acc, next));
          return struct_loop(n - 1, cons(next, acc))

structure():
    x <- randint(-20, 20); y <- randint(-20, 20);
    z <- randint(-20, 20);
    return with_type(x,y,z);

with_type(x,y,z):
    i <- randint(0, 9);
    if (i == 0): return cruiser_0(x,y,z);
    else if (i == 1): return cruiser_1(x,y,z);
    ...
```

callsite
random choice

Structures:



Initial solutions (N = 2):

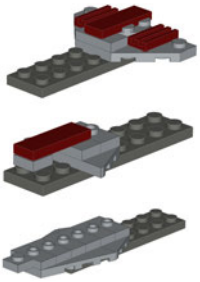


Figure 5.1: Model accretion (MA) input: program generating LEGO spaceships and initial solutions.

Each iteration of `struct_loop` calls `structure`, to randomly generate one structure. Each structure has a type and a position (x, y, z), which are random values. However, a random list of structures does not correspond to a valid spaceship. `assert` statements are used to apply hard constraints. `assert(at_least_1_connected(. . . ))` asserts that each new structure must connect to some structure already generated, and `pair_loop(res,. . . assert(non_overlap(. . . ))` ensures that structures do not intersect. A satisfying model is a program execution that does not violate *any* `assert` statement.

We define a probability distribution $p(\mathbf{x})$ over the possible program executions, where $\log p(\mathbf{x})$ is proportional to the number of satisfied constraints and $\log p(\mathbf{x}) = 0$ when all constraints are satisfied. We would like to produce samples $\mathbf{x}$ from the class $\log p(\mathbf{x}) = 0$. As this is also the maximum probability, synthesis can be seen as an optimization problem, using $p(\mathbf{x})$ as a landscape to navigate to $\log p(\mathbf{x}) = 0$. This is as opposed to other formulations that produce samples from the entire distribution.

MA begins with a set of satisfying *initial solutions*. In this thesis, we produced initial solutions by tracing the program and turning the trace into a logic formula [42], querying satisfying assignments using systematic search with the Z3 SMT solver [13]. In general, any method to obtain satisfying program executions (including by hand) is suitable.

Given program and initial solutions, MA repeatedly copies from the initial solutions using the function call hierarchy. For example, a call to `structure` causes sampling of the (x, y, z) and part type, defining a block of four random choices associated with that procedure. Similarly, `with_type` makes a single random choice over part type, and defines another block of choices. Each block is defined by the procedure call that produced it and the random choices made in that procedure call. Then, the block from the initial solutions overwrites a block in the current assignment. We call this a *copy proposal*. The mechanics of a copy proposal are shown in Figure 5.2.

This is in contrast to flat stochastic and systematic searches, such as single-site Metropolis-Hastings (M-H) and SMT solving, which perturb only one random choice at a time.

After the copy proposal is made, we still need to check that the program execution satisfies the constraints. In our spaceship example, copying a `structure` block copies in a spaceship structure of a given type at some (x, y, z) position, but the structure may not satisfy the constraint. To encourage movement through the space, we follow up every copy proposal with a single-site Metropolis-Hastings step.

The resulting synthesized models are shown in Figure 5.3, up to $\mathtt{N} = 6$. We can employ model accretion in a feedback loop; the models synthesized by MA in turn can be used as the initial solution set for yet more complex models. If $i = 1 \ldots N$, models from p_{θ_i} can be synthesized from all solutions over $p_{\theta_1} \ldots p_{\theta_{i-1}}$. By building

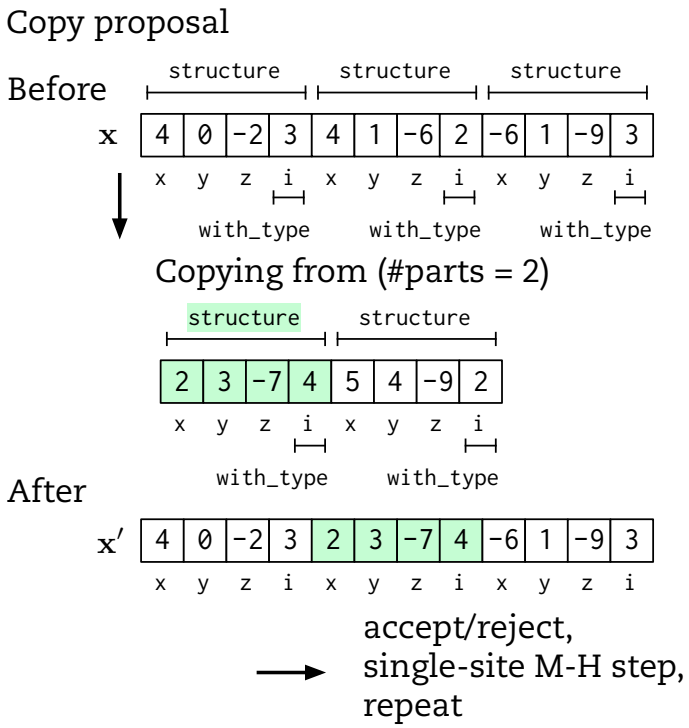


Figure 5.2: Mechanics of a copy proposal.

up complex models from previous simpler ones, there are situations where models can be synthesized faster than with techniques that proceed directly from scratch. We refer to such a sequence of parameters as an *accretion sequence*.

5.3 Formulation

MA is a stochastic search-based optimization algorithm using the copy proposal. The copy proposal randomly selects a solution $\mathbf{x}_s$ from the initial solutions along with a random block of choices $\mathbf{c}$ in $\mathbf{x}_s$. It then selects another random block of choices $\mathbf{r}$ in the current assignment $\mathbf{x}$, and replaces $\mathbf{r}$ with $\mathbf{c}$, resulting in $\mathbf{x}'$, which is accepted or rejected as the next assignment according to the ratio:

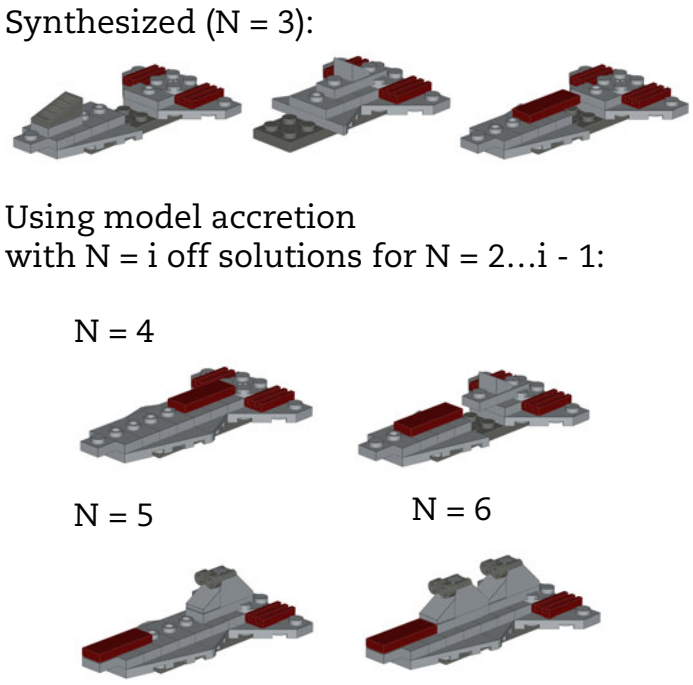


Figure 5.3: Synthesized LEGO spaceship models.

$$\min\{1, p_{\theta_i}(\mathbf{x}')/p_{\theta_i}(\mathbf{x})\}.$$

Note that MA does not satisfy detailed balance and is therefore not a valid producer of samples of $p_{theta_i}(\mathbf{x})$, despite the resemblance of the acceptance ratio to Metropolis-Hastings'. This is fundamentally because the copy proposal only copies *from* the initial solutions and never writes back *to* them.

We evaluate $p_{\theta_i}(\mathbf{x}')$ by running the program against a table of choices set to those in $\mathbf{x}'$ and keeping a running total of $\log p_{\theta_i}(\mathbf{x}')$. See the paper on Lightweight M-H [46] for details. We would like p_{θ_i} to be lower when constraints are violated, so every violated assertion multiplies the current value by some $w < 1$. We chose $\log w = -10$ in our examples, though the choice is not critical. Let a_i denote all assertion statements. Then the total unnormalized (log) probability is

$$\log p_{\theta_i}(\mathbf{x}) = \sum_{i=1}^{K} (\log w) \cdot I\{a_i = \texttt{False}\}.$$

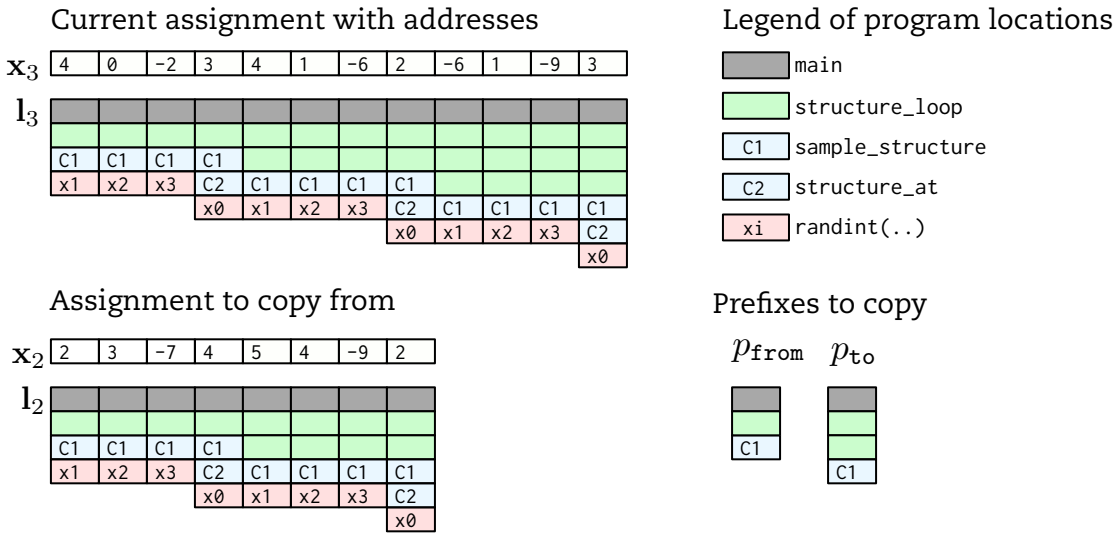


Figure 5.4: The start of a copy proposal for LEGO spaceships. (Left) The current assignment $\mathbf{x}_3$ and assignment copying from, $\mathbf{x}_2$.

We now describe how random choice blocks are defined. Figure 5.4 shows the start of the copy proposal depicted in Figure 5.2 in more detail. The left side shows the assignment $\mathbf{x}_3$ ($\theta = 3$) before the copy proposal along with the assignment being copied from: $\mathbf{x}_2$ ($\theta = 2$). Let $\mathbf{x}'_3$ be the assignment after the copy proposal.

Addressing scheme: a basis for copy proposals. The *addressing scheme*[46] tells us how to collect choices to form blocks and track them during program execution. In Figure 5.4, the vertical sequences of color labels underneath each random choice is an *address*. The address a_i of each random choice x_i records the stack of program locations at the time when the choice was made. For example, the first component of $\mathbf{x}_3$ has an address that representing the first call to `randint` in the first iteration of `struct_loop`. There is then a corresponding vector $\mathbf{l}_3$ of addresses parallel to random choices $\mathbf{x}_3$.

Consequently, a block of random choices corresponding to a procedure call is computed as a set of choices that share the same prefix in their addresses. Repeated calls to the same procedure are then any two prefixes that end with the same program location. We will call them *compatible*. This can be visualized (and implemented) in terms of prefix trees, depicted in Figure 5.5). Note that not every callsite will be useful. We have found it useful to annotate callsites to be used in copying.

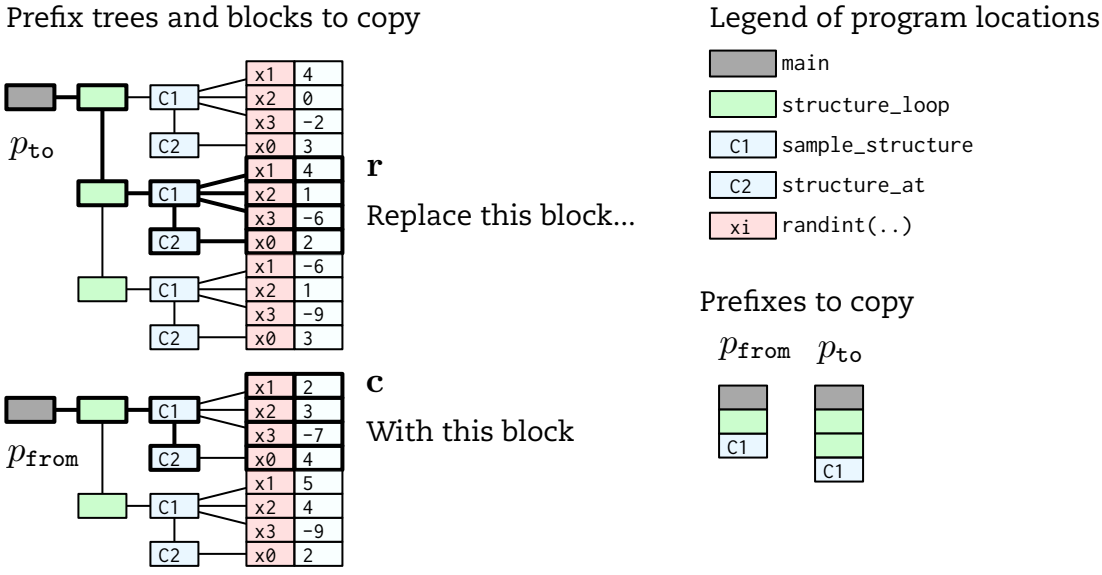


Figure 5.5: Choices to copy are grouped in prefix trees.

A copy proposal then proceeds by first selecting a pair of compatible prefixes $(p_{\texttt{to}}, p_{\texttt{from}})$, one for the current assignment and one for the assignment to copy from, respectively. These correspond to the two blocks $(\mathbf{r}, \mathbf{c})$: the choices to replace and the choices to copy in. This is shown in Figure 5.5.

Finally, replacing $\mathbf{r}$ with $\mathbf{c}$ comprises two steps: First remove all choices with prefix $p_{\texttt{to}}$ in $\mathbf{x}_3$. Then insert choices that have prefix $p_{\texttt{from}}$, after fixing up the addresses of the result so that $p_{\texttt{from}}$ is replaced with the original $p_{\texttt{to}}$. We show this replacement process in Figure 5.6.

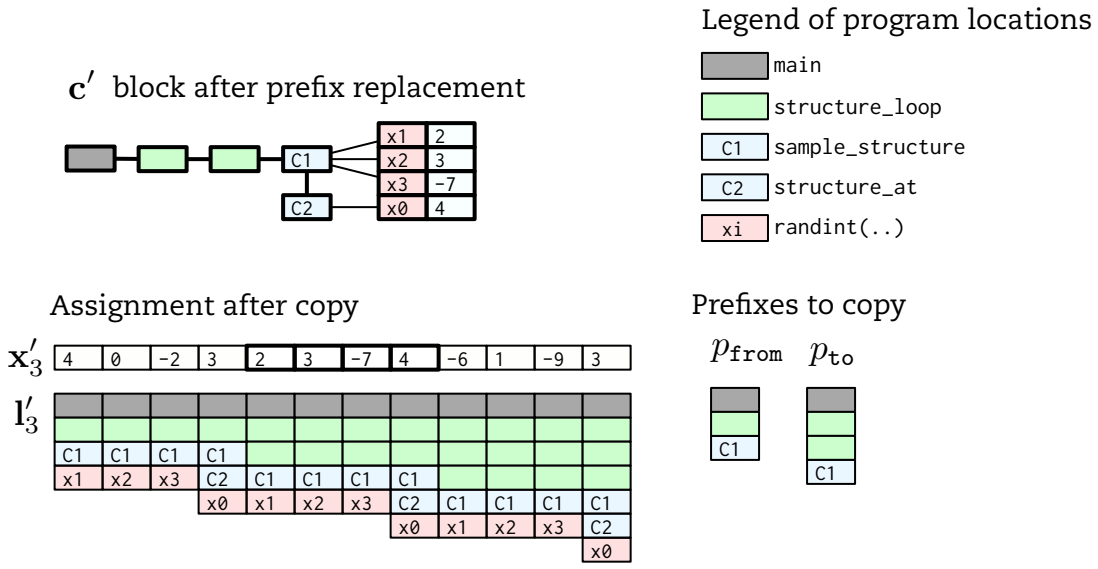


Figure 5.6: (Top left) $\mathbf{c}'$ is formed by replacing $p_{\texttt{from}}$ with $p_{\texttt{to}}$ in its addresses. (Bottom left) $\mathbf{x}'_3$, the assignment after the copy proposal.

5.4 Algorithm

Algorithm 13 depicts the interface and main loop of model accretion. We are given the program, synthesis parameter, set of initial solutions, and number of iterations. Running Eval or Eval-Regen on an empty assignment initializes the assignment and probability density. While Eval will re-sample each choice individually on the first program run, Eval-Regen initializes using copy proposals, by copying blocks of choices from the solutions whenever a procedure call is encountered during execution that has not been seen. The effect is a slight boost in the initial density score; Eval-Regen is not critical to performance.

Then, for each of the K iterations, we first run Copy (the copy proposal operation), and then accept or reject the result according to our acceptance ratio. A following step of Lightweight M-H encourages movement in the space. If the resulting assignment has log density zero, it satisfies all constraints and we add it to our solution set. In this section, we describe Eval,Copy, and Replace (a subroutine of Copy).

Algorithm 13: MA

Input: Program P, parameter of synthesis θ, initial solutions $S = \{s_m\}$, iterations K
Output: Final solutions Q (initially empty)

$(\mathbf{x}, p_\theta(\mathbf{x})) \leftarrow$ EVAL-REGEN$(P, \theta, S, ())$
for $i \leftarrow 1 \ldots K$ **do**
  $(\mathbf{x}', p_\theta(\mathbf{x}')) \leftarrow$ COPY$(P, \theta, S, \mathbf{x})$
  $(\mathbf{x}, p_\theta(\mathbf{x})) \leftarrow (\mathbf{x}', p_\theta(\mathbf{x}'))$ w.p. $\min\{1, p_\theta(\mathbf{x}')/p_\theta(\mathbf{x})\}$
  $(\mathbf{x}, p_\theta(\mathbf{x})) \leftarrow$ LWMH$(P, \theta, \mathbf{x})$
  if $p_\theta(\mathbf{x}) = 1$ **then**
    $Q \leftarrow Q \cup \{\mathbf{x}\}$
return s

Eval. EVAL$(P, \theta, \mathbf{x})$ runs the program with parameter θ and choices $\mathbf{x}$, much like the TRACE-UPDATE procedure in Lightweight M-H. EVAL returns $p_\theta(\mathbf{x})$ given program P, parameter θ, and assignment $\mathbf{x}$. EVAL also returns $\mathbf{x}_c$ which is a "fixed up" version of $\mathbf{x}$ that corresponds to an actual program execution.

This "fixing up" needs to be done whenever $\mathbf{x}$ does not correspond to an actual program execution. We employ the same approachas in Lightweight M-H: through a trial program execution against the addresses and choices in $\mathbf{x}$ [46]. Recall that each component x_i of any state $\mathbf{x}$ there is a corresponding address a_i. If $\mathbf{x}$ does not correspond to a valid execution (which can happen often), then either 1) there are some addresses visited by the trial execution not in the assignment or 2) there are some a_i in the assignment not visited by the trial execution.

Copy. COPY updates its input state $\mathbf{x}$, copying a random block of choices from a random solution $\mathbf{x}_s$ from the initial solutions S. Algorithm 14 lists pseudocode for the copy operation. COMMONPROCS returns all relevant callsite locations common to both the current state and selected solution. One such location l is sampled using SELECT, which samples a member of its input uniformly at random. We then sample prefixes to copy from and to (p_1, p_2 respectively) that end in the selected location l using PREFIXESWITH to generate them and SELECT again to sample. Knowing p_1 allows us to query the assignment using TRIE-QUERY to return all components of $\mathbf{x}_s$

Algorithm 14: COPY

Input: Program P, parameter θ, solution sets S, current state $\mathbf{x}$
Output: Next state and density $(\mathbf{x}', p_\theta(\mathbf{x}'))$

$\mathbf{x}_s \leftarrow \textsc{Select}(S)$
$l \leftarrow \textsc{Select}(\textsc{CommonProcs}(\mathbf{x}_s, \mathbf{x}))$
$p_1 \leftarrow \textsc{Select}(\textsc{PrefixesWith}(l, \mathbf{x}_s))$
$p_2 \leftarrow \textsc{Select}(\textsc{PrefixesWith}(l, \mathbf{x}))$
$c \leftarrow \textsc{Trie-Query}(p_1, \mathbf{x}_s)$
$\mathbf{x}'' \leftarrow \textsc{Replace}(\mathbf{x}, p_2, p_1, c, \textsc{Flip}(0.5))$
$(\mathbf{x}', p_\theta(\mathbf{x}')) \leftarrow \textsc{Eval}(P, \theta, \mathbf{x}'')$
return $(\mathbf{x}', p_\theta(\mathbf{x}'))$

having p_1 as prefix, thus computing c, the block to copy in. Finally, we modify the actual state using REPLACE (explained next) and re-run EVAL to fix the state up and obtain the density.

Replace. Algorithm 15 lists pseudocode for REPLACE, which overwrites a block of choices in the destination state with $\mathbf{c}$. There are two basic steps, deleting and writing. Deleting is done by the first loop, which produces the input state *without* the choices matching p_2. The second loop writes. It creates $\mathbf{c}'$, which is $\mathbf{c}$ but with p_1 replaced with p_2 to fit, and then merges $\mathbf{c}'$ with $\mathbf{x}$. SET,LOOKUP set or retrieve random choices, keying on addresses. $\mathbf{x} \sqcup \mathbf{y}$ is the merge operation: combine all the addresses and choices, favoring $\mathbf{y}$ on duplicates. $\textsc{Drop}(n, x)$ returns x without the first n elements.

It can be beneficial to replace only the choices immediately below a given prefix. REPLACE comes with a flag choosing whether to do this *context-only replacement*. We use SHALLOW-MATCH to determine choices immediately below a prefix. During each copy proposal, $\textsc{Flip}(0.5)$ randomly chooses between a full versus context-only replacement with equal probability.

Implementation. The language and compiler used for the overall system was Haskell 2010 on GHC 7.8.2. Initial solutions were stored as text files where each solution was represented as a list of addresses and values of random choices. We

Algorithm 15: REPLACE

Input: Source state $\mathbf{c}$, source prefix p_1, destination state $\mathbf{x}$, destination prefix p_2, context-only flag f
Output: Next state $\mathbf{x}''$

$\mathbf{x}_d \leftarrow ()$
for *addreses* $a_i \in A(\mathbf{x})$ **do**
  if $\neg$PREFIX-MATCH$(p_2, a_i) \vee (f \wedge \neg$SHALLOW-MATCH$(p_2, a_i))$ **then**
    $\mathbf{x}_d \leftarrow$ SET$(\mathbf{x}_d, a_i,$ LOOKUP$(a_i, \mathbf{x}))$
$\mathbf{c}' \leftarrow ()$
for $a_i \in A(\mathbf{c})$ **do**
  if $f \wedge \neg$SHALLOW-MATCH(a_i, p_1) **then**
    CONTINUE
  $a'_i \leftarrow p_2 +$ DROP$(|p_1|, a_i)$
  $\mathbf{c}' \leftarrow$ SET$(\mathbf{c}', a'_i,$ LOOKUP$(a_i, \mathbf{c})$
$\mathbf{x}'' \leftarrow \mathbf{x}_d \sqcup \mathbf{c}'$
return $\mathbf{x}''$

used standard techniques [5] to parse and compile the input programs. There were two main compiler targets. One was a SMT solving component for generating initial solutions and comparing performance. The other component was for EVAL and EVAL-REGEN; to compute the probabilities $p_\theta(\mathbf{x})$ and update $\mathbf{x}$ to correspond to program executions.

The SMT solving component was built as a tracing compiler with the Z3 [13] solver as backend. The trace was converted to a formula representing the constraints at runtime. This is known as *symbolic execution* [20] and our implementation is similar to recent work in *solver-aided domain-specific languages* [42]. The Haskell SBV library greatly facilitated construction of this component. The EVAL component was built as a source-to-source transformation following the implementation of Lightweight M-H [46]. In order to perform copy operations efficiently, a trie data structure with address keys was used to store and prefix-query the sets of address. It was helpful to pre-compute tries for each assignment in the initial solutions.

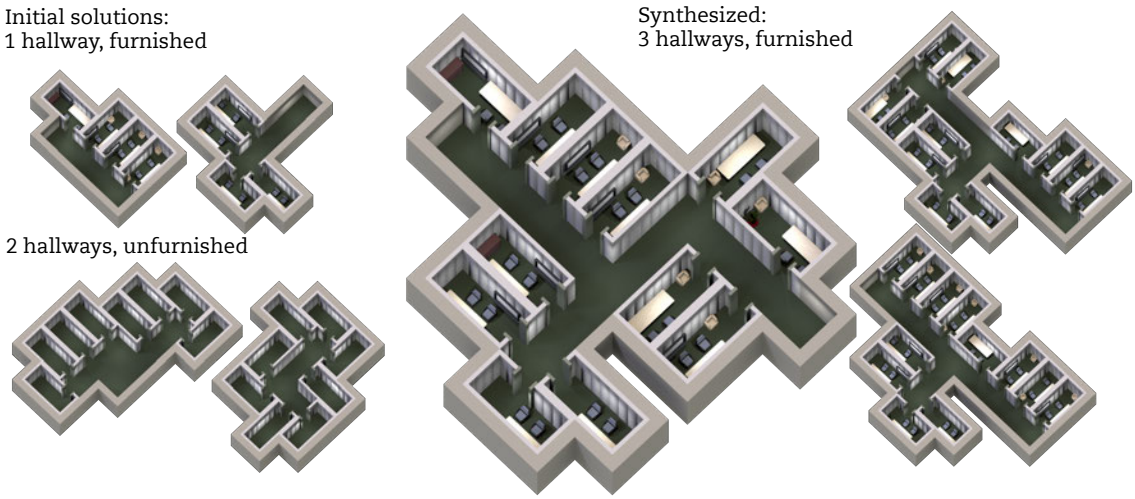


Figure 5.7: Office buildings with furniture synthesized using MA. (Left) The initial solutions include 1-hallway floor plans with furniture (top left) or 2-hallway floor plans without furniture (bottom left). (Right) MA-synthesized 3-hallway layout with furniture.

5.5 Results

In this section, we evaluate the performance of model accretion on three popular applications of procedural modeling:

1. Office buildings with furniture layouts.
2. Video game levels.
3. Architectural design prototypes.

The program for each domain is included in the supplemental materials. We compare the wall-clock time taken to synthesize using MA given initial solutions versus two other techniques: 1) single-site Metropolis-Hastings (M-H) and 2) SMT solving. M-H is a stochastic search technique applicable to general constrained procedural modeling [38]. SMT solving is an equally applicable systematic search.

All evaluations were performed in OS X v10.10 on a 2012 Macbook Pro (Intel i7-3615QM 2.30GHz, 8 GB RAM). For the SMT solver, the solver-aided DSL techniques

used to generate solutions involve an initial symbolic execution phase [42]. This time conservatively left out of timing numbers. Only the actual CPU time spent in systematic search is counted.

5.5.1 Building and Furniture Layout

In this section, we evaluate the use of MA to synthesize building and furniture layouts. These domains feature heavily in procedural modeling. It can be challenging to synthesize both in one model.

We used a program whose parameters were the number of hallways and whether or not to include furniture. Figure 5.8 shows the call graph. The main loop calls `wing`, which samples a building wing. Each wing consists of a `hall` and two groups of rooms, sampled through `roomgroups`, which has two inner loops sampling each group. `room` samples a room and furniture layout, each comprising 2 accessory elements and 1-2 tables with 2 chairs. Our copy proposals thus perturb (from coarse to fine): wings, room groups, rooms, furniture layouts, and then furniture elements. `offset` expresses the position of each building element as a (x, y) offset from the element at a coarser level.

The brackets in Figure 5.8 show selected constraints and approximate set of influenced random choices. Constraints mainly concern alignment and non-overlap. We see that the influence of some constraints (e.g., `furniture non-overlap`) is completely contained within some block or hierarchy level, while the influence of others (e.g., `table-room-align`, `room-hall non-overlap`) cuts across such boundaries. It is not guaranteed for copy proposals to be accepted at a high rate.

MA was used to synthesize furnished building layouts with 3 hallways. We employed MA in an accretion sequence, where the initial solutions were 10 1-hallway, furnished layouts and 10 2-hallway, unfurnished layouts (depicted on the left side of Figure 5.7). These were first used to synthesize 10 2-hallway, furnished layouts and then finally, these 30 solutions in aggregate were used to synthesize 10 3-hallway, furnished layouts. The right side of Figure 5.7 shows such 3-hallway furnished layouts.

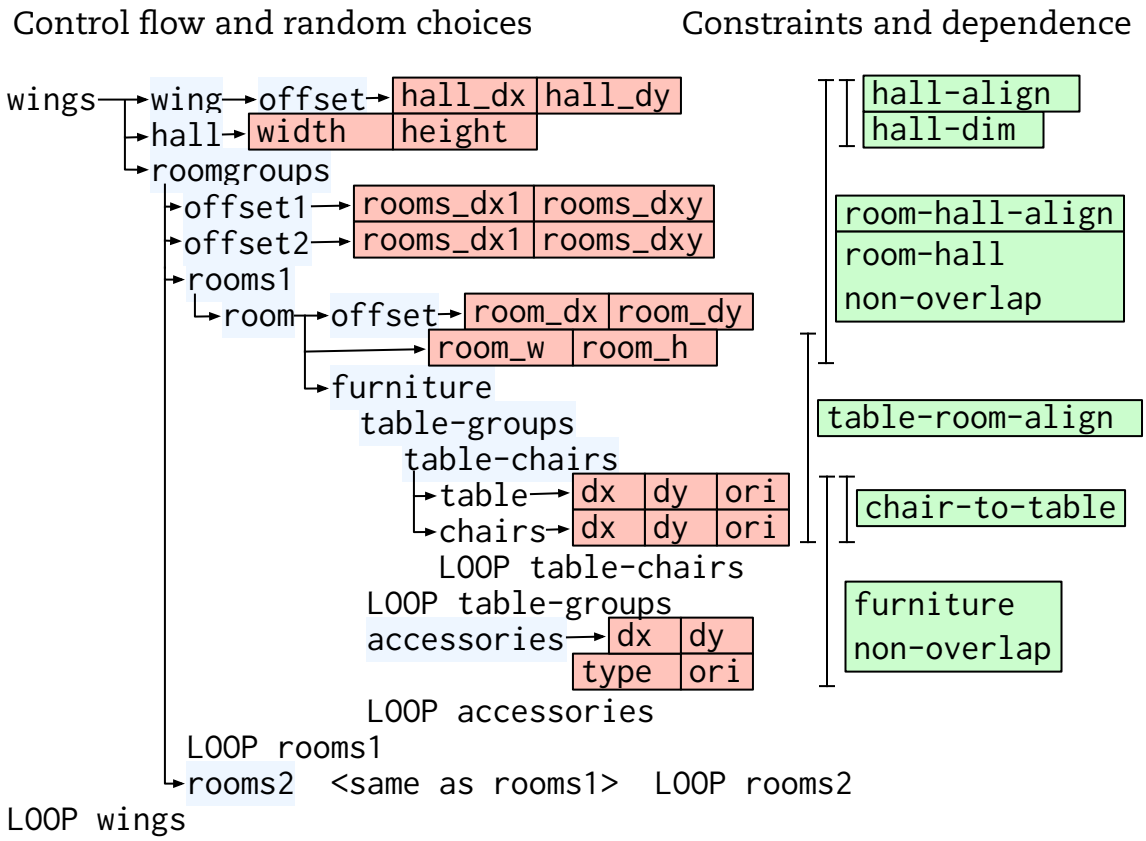


Figure 5.8: The control flow (blue) of the office building/furniture layout program along with random choices (red) and constraints (green). Brackets show dependencies of constraints. Some constraints are omitted due to space constraints.

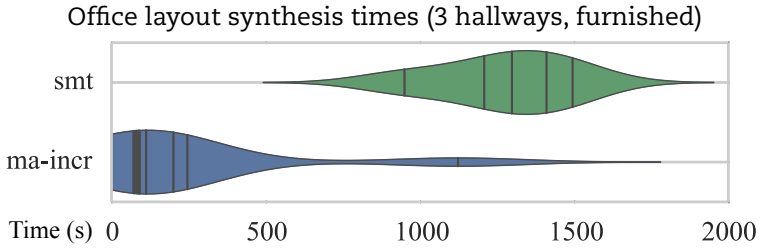


Figure 5.9: Distribution of time elapsed from start of run to the first satisfying office layout with 3 hallways and furniture, comparing model accretion (ma-incr) and the Z3 SMT solver (smt). M-H never finished in less than 1 hour (3600s). Vertical lines show individual samples.

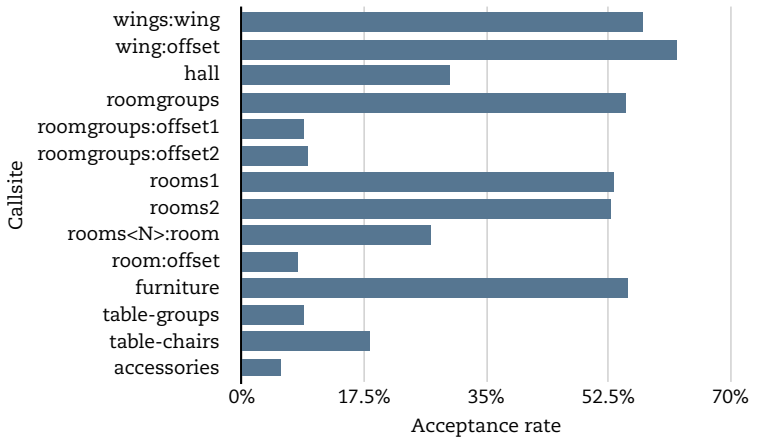


Figure 5.10: Acceptance rates of copy proposals arranged by callsite.

Comparison versus M-H and SMT solving. Figure 5.9 shows distributions of the time taken for model accretion and SMT. We do not show timing for M-H; M-H never synthesized a satisfying model in any reasonable amount of time (1 hour). We attribute this to the many interlocking constraints and high number of variables in the model. SMT solving produced solutions in 1272.4s on average (210.5s standard deviation); a systematic search can be more efficient than M-H in models with hard constraints. Model accretion took 232.1s on average (339.8s standard deviation), a 5.5x average speedup over SMT solving. It can be faster to generate from a given set of simple models than to solve completely from scratch.

Acceptance rate of copy proposals. Next, we evaluate how well individual copy proposals work. Figure 5.10 shows the acceptance rate of copy proposals by callsite for the above synthesis. `furniture` copies are accepted over 50% of the time. Figure 5.8 shows that `table-room-align` is the only constraint not contained completely within calls to `furniture`. Therefore, `furniture` copy proposals already satisfy everything except `table-room-align`. We attribute this high acceptance rate to this fact that most constraints on furniture are self-contained. On the other hand, `table-groups` copy proposals are accepted at a rate around 10%, as there is significant interaction

with other furniture objects through the `furniture-non-overlap` constraint. Analogously, at the level of building structure, it is simple to copy over an entire group of rooms (`roomgroups`, `rooms1`, `rooms2`) ($\sim$ 50% acceptance rate), but more difficult to copy over the offset and dimensions of an individual room (`room:offset`, with $\sim$ 10% acceptance rate). We conclude that copy proposals tend to work better when blocks being copied have more constraints internal to the block and impact fewer external constraints.

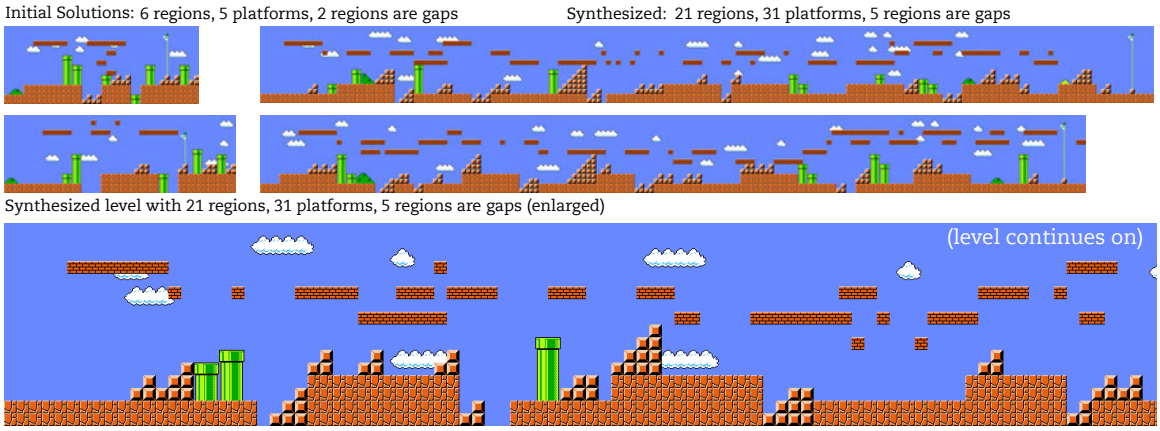


Figure 5.11: Mario levels synthesized using MA. (Top left) The short levels are the set of initial solutions. (Top right) The long levels are synthesized using MA. (Bottom) Enlarged view.

5.5.2 Video Game Levels

Video game levels exhibit hierarchical, repeated structure with a few constraints on individual pieces, such as non-overlap. There can also be a few global constraints as well, such as the number of particular objects or level features. Super Mario Bros. [2] is a prototypical example of this; there are several midair platforms within jumpable distance of each other, and each ground region can have an arrangement of pipes and blocks attached.

Figure 5.12 sketches out our program for synthesizing Mario levels. The level consists primarily of `region`'s generated in a loop. Each region is either solid ground

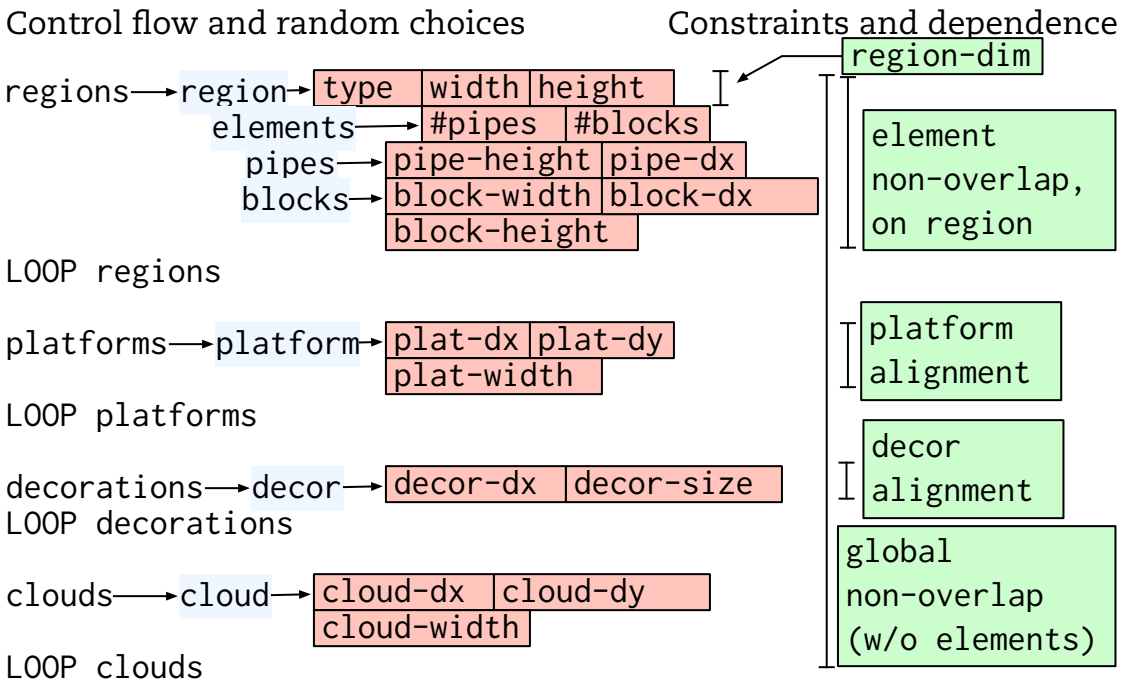


Figure 5.12: The control flow (blue) of the Mario level layout program along with random choices (red) and constraints (green). Brackets show dependencies of constraints.

or a gap that Mario has to jump over. There is a hard, global constraint on the number of gap regions. A varying number of pipes and blocks are arranged on each region using the calls to `elements`, `pipes`, and `blocks`. Copy proposals can then copy an entire `region` and its pipes and blocks, or just the pipes or blocks. Non-overlap and inside-region constraints control the location and dimension of pipes and blocks in each region. Our levels cannot be generated by a simple forward process alone. The procedure `platforms` generates the hovering platforms. Copy proposals would then transfer the offset and dimensions of an individual platform. Finally, decorations are generated in the loops `decorations` and `clouds`.

The program had 3 parameters: number of ground/gap regions, number of platforms, and the constraint on the number of gaps. The synthesis task was to produce Mario levels with 21 regions, 31 platforms and 5 regions constrained to be gaps.

MA had initial solutions of 10 levels with 6 regions 5 platforms, and 2 regions as gaps. Let these parameters be denoted as the triple (r, p, g). We synthesized according to the accretion sequence $(11, 11, 3), (16, 21, 4)$, and $(21, 31, 5)$. After 5 solutions were synthesized, MA moved on to the next parameter seting. From experiments, it took 200 iterations between solutions for them to appear significantly different. To evaluate the effect of the accretion sequence, this was compared against MA with no nontrival accretion sequence, directly using solutions from $(5, 4, 2)$ to synthesize those of $(21, 31, 5)$. Figure 5.11 shows the resulting levels.

Comparison versus M-H and SMT. Figure 5.13 shows synthesis times for Mario levels. We see that model accretion synthesizes the first solution from its simple initial ones much sooner than both M-H and SMT solving, producing the first solution in 36.7s average (9.5s standard deviation). Unlike with office building layouts, this domain features fewer constraints and is more amenable to M-H search: M-H can finish in a reasonable amount of time (704.2s average, 237s standard deviation). The sophisticated techniques of systematic search as captured by the SMT solver still leave it around 6x slower than model accretion, with 218.6s average solve time (47.2s standard deviation).

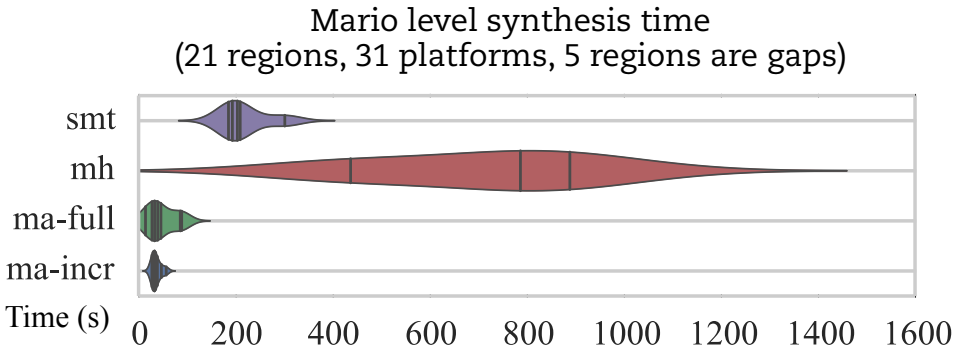


Figure 5.13: Distribution of synthesis times for Mario level synthesis, comparing model accretion (ma-incr), model accretion without accretion sequence (ma-noseq), M-H (mh), and Z3 SMT solver (smt). Vertical lines show individual samples.

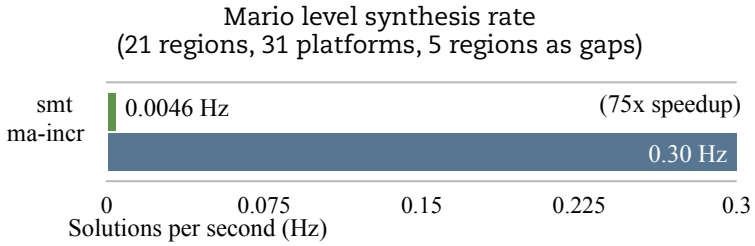


Figure 5.14: Number of visually different solutions generated per second (in Hz) after the first satisfying solution, comparing SMT solving (smt) and model accretion (ma-incr).

Effect of accretion sequence. Figure 5.13 also includes timing numbers for "ma-noseq", which is model accretion drawing directly on the set of initial solutions with 6 regions, 5 platforms and 2 regions as gaps, to the final set of parameters (21 regions, 31 platforms, and 5 regions as gaps), without creating and saving solutions at intermediate parameter settings. Without an accretion sequence, MA was a little slower, taking 47.8s on average (with higher variance: 28.8s stdev). The use of an accretion sequence can help performance.

Rate of solution production versus SMT. Because MA is a stochastic search, as soon as the first solution is produced, more tend to follow. We tracked any "different-enough" solutions that showed up *after* the first one. What defines "different-enough"? The satisfying solutions in the top right of Figure 5.11 are 200 iterations apart and are "different-enough." By this metric, Figure 5.14 shows the solutions per second (in Hz) comparing SMT solving and MA. We find that MA synthesizes models at a rate 75x that of the SMT solver, nearing 2 orders of magnitude. This suggets that MA is a very viable synthesis technique when used offline in the background to produce levels in a setting without user attendance, especially after the first solution has been found.

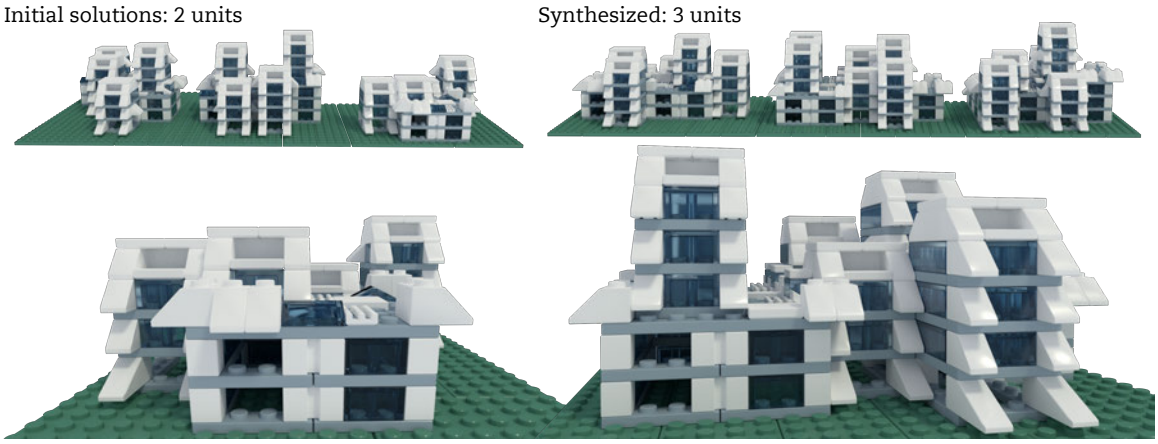


Figure 5.15: Synthesizing LEGO building designs. (Left) Initial solutions. (Right) Synthesized models using MA. Note how tower and building structures interact with roof decorations; there are many constraints that cut across hierarchies.

5.5.3 LEGO Buildings

Architecture is a common problem domain for procedural modeling, and LEGO can be a suitable medium for prototyping architectural designs [43].

We use a program that generates arrangements of 3 types of structures: buildings, towers, and roof decorations, shown in Figure 5.16. Building structures (middle) are relatively simple, consisting of a single floor unit that is repeated, ending in a roof

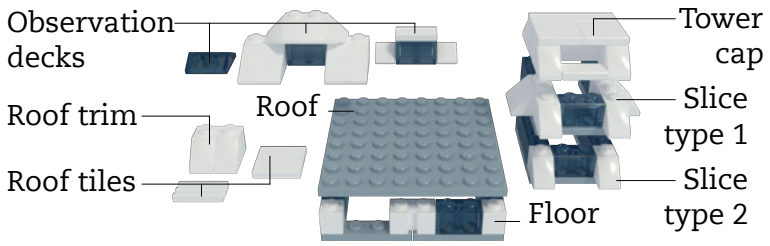


Figure 5.16: Breakdown of LEGO buildings into structures placed by program.

element. Towers (right) are made up of stacks of slices that end in a cap element. Some slices stick out more than others. Roof decorations (left) consist of observation decks, trim pieces, and tiles.

Figure 5.17 shows the control flow structure, random choices, and constraints of the program. The high-level organization is over *units* consisting of a tower next to a building, which has another tower on top. The top level function `bldg-tower-loop` generates these units in a loop. Each unit is generated by the procedure `b-t`. For each building, after the program samples the number of random choices, the procedure `roof-decor` samples roof decorations (trim, observation towers, and roof tiles). All decorations are sampled at some offset relative to the roof. Trim and observation towers are constrained to be on the edge of the roof, while roof tiles to be within the roof's area.

The tower next to the building is sampled starting from an offset computed by `tower-offset`. It is constrained to be next to the building through `bldg-tower-align`. Between consecutive iterations of `bldg-tower-loop`, `inter-bldg-tower align` constrains the building or tower in separate units to be aligned. Finally, there is a global non-overlap constraint.

The hierarchy of random choices then consists of units, the building and two towers within each unit, and roof element layout. Our program was parameterized on the number of units. We first synthesized 10 buildings with 2 units each using the SMT solver. The synthesis goal was to extend to 3 units given these 10 solutions. We ran model accretion until the first satisfying building appeared. Figure 5.15 shows the synthesized models, with initial solutions on the left and models synthesized with

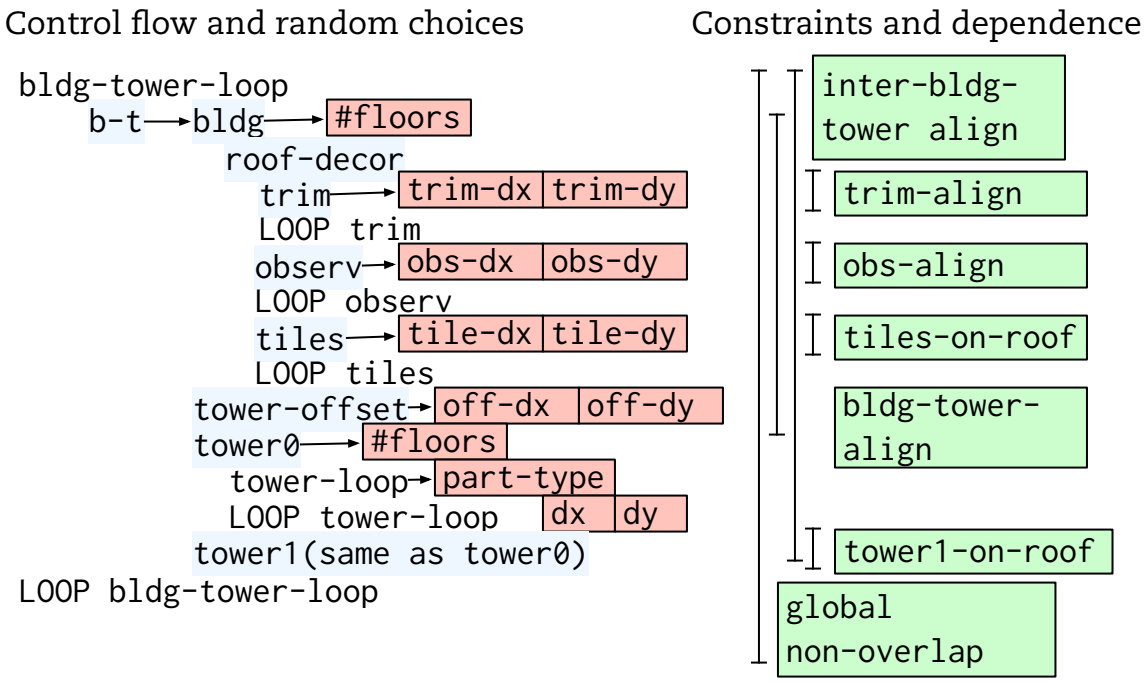


Figure 5.17: Call graph of program generating LEGO buildings.

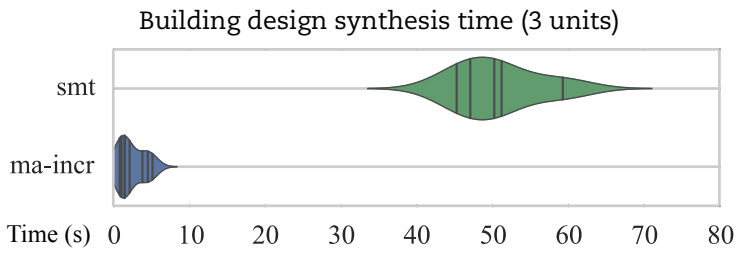


Figure 5.18: Distribution of first solution times comparing MA (3 units from 2) and SMT solving for LEGO buildings. Vertical lines show individual samples.

MA on the right.

Comparison with M-H and SMT solving. As before, we compared the time taken to synthesize the first solution against M-H and SMT solving. Figure 5.18 shows distributions of such times. M-H never produced a satisfying assignment in less than 3000 seconds. SMT solving was much faster, producing 3-unit buildings in 50.7 seconds on average (5.4s standard deviation). MA was the fastest, producing 3-unit buildings from 2-unit buildings in 2.4 seconds on average (1.7s standard deviation), a 20x average speedup.

5.6 Discussion and Future Work

We have demonstrated that MA can work for a wide range of constrained procedural models, from office layouts to LEGO buildings. MA amplifies the power of existing synthesis techniques, copying in and recombining parts of initial solutions. We see that there are two kinds of efficiency gains that result: 1) faster synthesis times for the first satisfying model and, as it is a stochastic search, 2) faster synthesis times for subsequent models.

The key to why MA works is in exploiting repetition and hierarchy in procedural models. Each sub-layout generated by the program can be considered as having constraints completely internal to the layout and constraints that interact externally with

the rest of the model. Each copy proposal clearly maintains internal constraints; however, external constraints still need to be satisfied. We have shown some procedural modeling domains where the external constraints are not too difficult.

The main avenue of future work is in further increasing the efficiency of MA. If we can analyze the program so as to completely bound the space of random choices, we can synthesize more efficient data structures over which to perform copy proposals. In addition, we can also compile a compact formula for computing the acceptance ratio for any copy proposal. Finally, we can apply techniques from machine learning in order to learn the best copy proposal to perform given features of the current assignment and solutions.

In the long term, we envision a learning system that performs synthesis over many different models offline, then selects appropriate copy proposals and previous solutions for any new synthesis problem that occurs. With sufficent generality in what is learned, this would lead to much more efficient procedural modeling.

Chapter 6

Conclusion

Tracing is a simple technique, yet it is a powerful souce of information about the execution space of an inference program. I have shown that execution traces are useful for inference in three ways:

Shred As traces map out the execution of a program in fine-grained detail, they allow us to improve the computational efficiency of a known inference algorithm, Metropolis-Hastings. It can be costly to compile such traces, but the efficiency of the resulting MCMC iterations more than makes up for it.

Simply by adopting tracing, a general method from compilers that is also quite simple, we are able to generate extremely fast code that runs as fast or faster than hand-coded versions.

The simple form of the trace allows us to perform fine-grained dependency analysis, recovering sophisticated incremental update schemes that are just about as fast as possible.

Solitaire Traces are also a flexible representation where we can take an arbitrarily complex inference program and run any state of the art inference engine, such as a systematic search, over it. The tradeoff is that we need to explore on a path by path basis.

By compiling traces to SMT formulae, which is also a simple concept, we are

able to solve procedural content generation problems that would be infeasible for techniques such as M-H, wspecially for office and furniture layouts, where constraints prohibit the use of forward generation.

We have shown that in contrast to traditional views on probabilistic programming, with traces we can take advantage of a state of the art inference engine and get good results without having to develop more algorithms from the ground up.

Model accretion We have shown that by recording and replaying segments of previous good program executions, we can greatly increase the efficiency of procedural content generation problems that would bring even the SMT solver to its knees.

These results show the benefits of departing from inference methods such as LWMH. In contrast to the previous approaches where a flat vector of random choices removes all structure and repetition, we instead treat program structure and repetition as a core variable that changes how we approach the construction of inference algorithms.

6.1 Broader impact

First, consider the impact of probabilistic programming languages. Probabilistic inference (including machine learning) is having an increasing impact in daily life, despite its workings being relatively inaccessible to the end user. As a most basic example, we would not be able to read through our email without Bayesian spam filters. Expanding the scope a bit, we would not be able to enjoy stable prices on many things without the power of inference in algorithms applied to trading in financial markets. We cannot count on our mail arriving in a timely manner unless automatic inference has been applied in optimizing the routing of mail.

Down the line, the role of the average person with a computer will become more important. There is not going to be an "inference app" for everything. We will need to put the tools of inference directly in the hands of the end user; to democratize the use of probabilistic inference. This has been a core motivation for probabilistic programming languages, and even today, there are practitioners in industry and research

who need specialized inference techniques.

For example, an aircraft engine manufacturer may use inference to evaluate the safety risks and expected costs and longevity of using different materials and techniques in the design of jet engine rotors. An analyst working in the campaign for the next U.S. President may want to write probabilistic programs that take into account the effects of both high-level campaign strategies and even the impact of individual words spoken at a debate, to produce materials targeted at individual voters. Finally, an inspector for the Occupational Health and Safety Administration (OSHA) may want to use a probabilistic program to simulate actual working environments, so as to preemptively infer the existence of hazardous situations; not every future safety regulation will have to be written in blood.

The costs of inference The role of this work is in increasing the efficiency of probabilistic programming languages. Many problems we would like to tackle with probabilistic programming languages have a prohibitive cost in computation cycles. All of the above examples, though tempting, would require many supercomputers worth of computing power in order to be feasible at scale. The average practitioner does not have access to supercomputers nor hired experts in compilers and inference for tuning the workings.

In this work, I have shown that by incorporating relatively simple ideas from programming languages and compilers, probabilistic programming languages can be made both general and high-performance. This reduces the cost of using inference techniques and brings the desired future above closer to reality.

Creative activities Finally, I expect that these techniques will be increasingly important also for bringing probabilistic programming languages and inference to areas of life not as tied to industry and research, but can be no less significant. The average person will have creative recreational interests, such as in art and music. Inference has a potentially big role to play in enhancing these activities. The procedural modeling applications in this thesis show how we can put more power in the hands of the individual artist.

6.2 Future directions

I see execution traces for specialized inference as the first step of several possible future lines of inquiry.

6.2.1 More inference algorithms run from traces

First would be cataloguing all possible inference algorithms that can run given a trace. We have only explored two existing algorithms: Metropolis-Hastings and DPLL(T). Belief propagation and gradient-based MCMC methods also seem amenable to being run from traces.

It would also be interesting to see which optimizations would apply best for each method. For instance, Hamiltonian Monte Carlo (HMC) operates on the entire vector of random choices, which would mean slicing does not apply to making HMC faster. Yet, it is nontrivial to find the simplest form of a gradient, as many algebraic simplification rules can apply, up to polynomial factoring which is intractable in genral.

6.2.2 Increasing performance of tracing

The primary way in which a tracing-based approach to specialized inference runs up against a wall in terms of models that are easily representable, is when these models contain a lot of control flow. In general, and especially for grammar-like models, the set of unique control flow paths is exponential in the number of program steps taken (or alternately, the length of any single path).

In order to address these models with tracing, we clearly need better tracing performance. Currently, tracing and trace graph formation are very expensive operations. In particular, optimizations and trace graph construction run what is essentially symbolic virtual machines over the trace.

I see two ways to increase the performance of tracing. The first is to perform the same constructions and optimizations, but more efficiently. A different implementation of the tracer may help here. I am looking toward implementations of fast JIT

compilers, such as those of V8 and the Firefox JIT compilers, which reduce overhead of tracing drastically compared to printing out C++ programs as traces and compiling the with GCC.

The second is to consider more compact representations of multiple paths. Trace graphs are such a method. However, like traces, they often repeat the same blocks of code over and over due to loops. While this can be a source of why traces are faster, there is no reason to perform the same symbolic virtual machine computations over and over; we should be able to compute exactly the segments of code that repeat, and patch together several trace segments to form any needed trace on the fly. These segments and their joining parts can be pre-compile to binary code, making the formation of any trace a matter of concatenating bits together.

6.3 Better representations of the space of executions

Next, traces rely on the overall concept that abstractions or approximations of the execution space of a program are useful. Given such representations, one can create a specialized version of an algorithm. This brings to mind some of the other ways to obtain such representations of execution space, namely partial evaluation and more broadly, supercompilation.

From this perspective, traces and trace graphs describe a very precise, underapproximate, finite approximation of the execution space. Partially evaluated/supercompiled representations describe programs in their full generality, but selectively evaluate some parts of the program ahead of time.

Making an inference algorithm run fast requires the proper representation, and exploring generally partial evaluated and supercompiled representations seems like a promising next step. In the future, I expect there to be a more thorough understanding of the best unfolded program representation for each inference algorithm.

Bibliography

[1] *Just Another Gibbs Sampler. Available at http://mcmc-jags.sourceforge.net.*

[2] Nintendo Research & Development 4. *Super Mario Bros. (video game)*. Nintendo, 1986.

[3] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[4] Umut Acar. *Self-Adjusting Computation*. PhD thesis, 2005.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[6] John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35:97–113, 2003.

[7] Fan Bao, Dong-Ming Yan, Niloy J. Mitra, and Peter Wonka. Generating and exploring good building layouts. *ACM Trans. Graph.*, 32(4):122, 2013.

[8] Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B. Goldman. Patchmatch: a randomized correspondence algorithm for structural image editing. *ACM Trans. Graph.*, 28(3), 2009.

[9] Matthew J. Beal. *Variational algorithms for approximate Bayesian inference*. PhD thesis, 2003.

[10] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Allocation removal by partial evaluation

in a tracing jit. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '11, pages 43–52, New York, NY, USA, 2011. ACM.

[11] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, September 2009.

[12] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, pages 168–176, 2004.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: a multi-stage language for high-performance computing. In Hans-Juergen Boehm and Cormac Flanagan, editors, *PLDI*, pages 105–116. ACM, 2013.

[15] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 465–478, 2009.

[16] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In *In UAI*, pages 220–229, 2008.

[17] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[18] E.T. Jaynes and G.L. Bretthorst. *Probability Theory: The Logic of Science*. Cambridge University Press, 2003.

[19] Evangelos Kalogerakis, Siddhartha Chaudhuri, Daphne Koller, and Vladlen Koltun. A probabilistic model for component-based shape synthesis. *ACM Trans. Graph.*, 31(4):55, 2012.

[20] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[21] Scott Kirkpatrick, C. D. Gelatt, and Mario P. Vecch. Optimization using simulated annealing. *Biometrika*, 57(1):97–109, 1970.

[22] Oleg Kiselyov and Chung-chieh Shan. Embedded probabilistic programming. In Walid Mohamed Taha, editor, *DSL*, volume 5658 of *Lecture Notes in Computer Science*, pages 360–384. Springer, 2009.

[23] P.S. Laplace and A.I. Dale. *Pierre-Simon Laplace Philosophical Essay on Probabilities*. Sources in the History of Mathematics and Physical Sciences. Springer, 1995.

[24] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[25] Aristid Lindenmayer. Mathematical models for cellular interaction in development: Parts i and ii. *Journal of Theoretical Biology*, 18, 1968.

[26] Chongyang Ma, Li-Yi Wei, and Xin Tong. Discrete element textures. *ACM Trans. Graph.*, 30(4):62, 2011.

[27] Simon Marlow. Haskell 2010 language report.

[28] Paul Merrell and Dinesh Manocha. Model synthesis: A general procedural modeling algorithm. *Visualization and Computer Graphics, IEEE Transactions on*, 17(6):715–728, 2011.

[29] Paul Merrell, Eric Schkufza, Zeyang Li, Maneesh Agrawala, and Vladlen Koltun. Interactive furniture layout using interior design guidelines. *SIGGRAPH 2011*, August 2011.

[30] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 614–623, New York, NY, USA, 2006. ACM.

[31] Radford Neal. Sampling from multimodal distributions using tempered transitions. *Statistics and Computing*, 6:353–366, 1994.

[32] Judea Pearl. Reverend bayes on inference engines: a distributed hierarchical approach. In *in Proceedings of the National Conference on Artificial Intelligence*, pages 133–136, 1982.

[33] Chi-Han Peng, Yong-Liang Yang, and Peter Wonka. Computing layouts with deformable templates. *ACM Trans. Graph.*, 33(4):99, 2014.

[34] Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. Technical report, Charles River Analytics, 2009.

[35] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer, 1991.

[36] George Stiny, James Gips, George Stiny, and James Gips. Shape grammars and the generative specification of painting and sculpture. In *Segmentation of Buildings for 3DGeneralisation. In: Proceedings of the Workshop on generalisation and multiple representation , Leicester*, 1971.

[37] Andreas Stuhlmüller and Noah D. Goodman. A dynamic programming algorithm for inference in recursive probabilistic programs. 2012.

[38] Jerry O. Talton, Yu Lou, Steve Lesser, Jared Duke, Radomír Měch, and Vladlen Koltun. Metropolis procedural modeling. *ACM Trans. Graph.*, 30(2):11:1–11:14, April 2011.

[39] Jerry O. Talton, Lingfeng Yang, Ranjitha Kumar, Maxine Lim, Noah D. Goodman, and Radomír Mech. Learning design patterns with bayesian grammar induction. In *The 25th Annual ACM Symposium on User Interface Software and Technology, UIST '12, Cambridge, MA, USA, October 7-10, 2012*, pages 63–74, 2012.

[40] 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.

[41] A. Thomas. Bugs: a statistical modeling package. 2:36–38, 1994.

[42] Emina Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 54, 2014.

[43] Christopher Turner, editor. *LEGO Architecture studio*. The LEGO Group, 2013.

[44] Li-Yi Wei and Marc Levoy. Fast texture synthesis using tree-structured vector quantization. In *SIGGRAPH*, pages 479–488, 2000.

[45] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[46] David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of AISTATS 2011, pp. 1–9.*, 2011.

[47] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.

[48] Yi-Ting Yeh, Katherine Breeden, Lingfeng Yang, Matthew Fisher, and Pat Hanrahan. Synthesis of tiled patterns using factor graphs. *ACM Trans. Graph.*, 32(1):3, 2013.

[49] Yi-Ting Yeh, Lingfeng Yang, Matthew Watson, Noah D. Goodman, and Pat Hanrahan. Synthesizing open worlds with constraints using locally annealed reversible jump mcmc. *ACM Trans. Graph.*, 31(4):56:1–56:11, July 2012.

[50] Lap-Fai Yu, Sai-Kit Yeung, Chi-Keung Tang, Demetri Terzopoulos, Tony F. Chan, and Stanley J. Osher. Make it home: Automatic optimization of furniture arrangement. *ACM Trans. Graph.*, 30(4):86:1–86:12, July 2011.