Incrementalizing MCMC through Tracing and Slicing

Yang, Yeh, Goodman, Hanrahan.
Presented by: Lakhan Kamireddy

Abstract

GOAL of the paper: Increase the efficiency of MCMC inference on probabilistic programs.

How: In the Church setting, MCMC is a random walk over program paths, very expensive as each iteration requires complete program execution with side computations. The paper exploits the fact that only some random choices influence control flow. The authors use techniques like tracing JIT for structural variables and Incremental computation for non-structural variables and thus optimize the MCMC execution.

Results: The paper presents a dynamic compilation technique. Their optimization is orders of magnitude faster than unoptimized church MCMC.

Introduction

This paper focuses on Church, a probabilistic extension of untyped call-by value lambda calculus developed by Goodman et al. Inference in Church is accomplished through MCMC Metropolis Hastings Algorithm.

Recap: In MH, we come up with a proposal by perturbing a set of random choices made during program execution. We then accept or reject the perturbation according to acceptance probability.

We accept or reject our perturbation depending on this, The q(x'|x) and q(x|x') factors serve to compensate for bias in the random walk. The MH accept ratio.

$$\alpha = \min \left\{ 1, \frac{p(x')\, q(x|x')}{p(x)\, q(x'|x)} \right\}$$

Introduction

- Anyone would advocate for the need of performance improvement in performing MH efficiently. The authors proposed some techniques to get orders-of-magnitude improvement in MH performance. Their work is a continuation to the MH implementation we saw in Wingtae et al.

-**MH** on Probabilistic programs **can be accelerated**. How ? We identify choices that may be reused (Note that we require a naming scheme we saw earlier to name each random choice).

- Many MH proposals will cause no change in the control flow of the (probabilistic) program. To exploit this we distinguish two types of random variables. **Structural** and **non-structural**

Structural and non-structural

Structural Variables: Random Variables that affect the control flow of the program.

Non-structural Variables: Random Variables that **don't** affect the control flow of the program.

- We accelerate MH by compiling away structural choices. This is done using **just-in-time (JIT) tracing** technique. Optimizations such as dead code elimination and allocation removal are applied to the trace. This is 1st major performance improvement technique .

- **Incrementalization -** or avoiding rerunning of parts of the trace that donot depend on the recently perturbed variable. 2nd performance impr technique.

Tracing JIT compilation

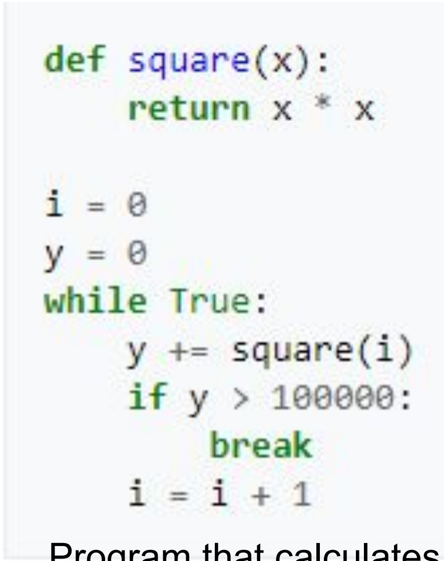
- **Tracing just-in-time compilation** is an optimization technique to optimize execution of program at runtime. We identify **hot** (frequently operated) bytecode sequences, record them, compile them to fast native code. We call such a sequence of instructions a trace. Method calls, loops are generally the targets.

- We generally use the existing interpreter, for a dynamic language and tweak it to achieve this type of compilation.

- Once we have the trace, we can apply many possible optimizations. It makes a significant difference to performance because ?

Let us look at an example:

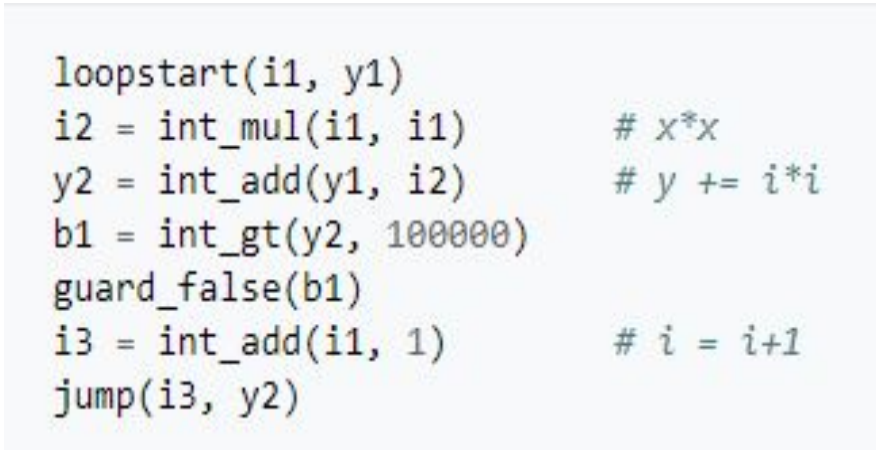
Tracing JIT Compilation



```python
def square(x):
    return x * x

i = 0
y = 0
while True:
    y += square(i)
    if y > 100000:
        break
    i = i + 1
```

Program that calculates sum of squares of successive whole numbers until sum exceeds 100000



```
loopstart(i1, y1)
i2 = int_mul(i1, i1)          # x*x
y2 = int_add(y1, i2)          # y += i*i
b1 = int_gt(y2, 100000)
guard_false(b1)
i3 = int_add(i1, 1)           # i = i+1
jump(i3, y2)
```

A trace could like this

Tracing

- Goal of tracing in this paper is to optimize the acceptance ratio calculation, specializing it to a particular set of structural choices. Traces are functions that take an assignment to the remaining non-structural choices.

-They are used to compute unnormalized probability score corresponding to these choices, forming numerator and denominator of acceptance ratio. Traces also compute the sampled value of the program.

- Furthermore we optimize the straight-line traces by applying standard techniques like dead code elimination and allocation removal

Incrementalization

- **Incremental computation**: Concerns with the problem: Given a function *f* , previous input *x*, previous output *$y = f(x)$*, and new input *x'*, how to calculate *$y' = f(x')$* as efficiently as possible ?

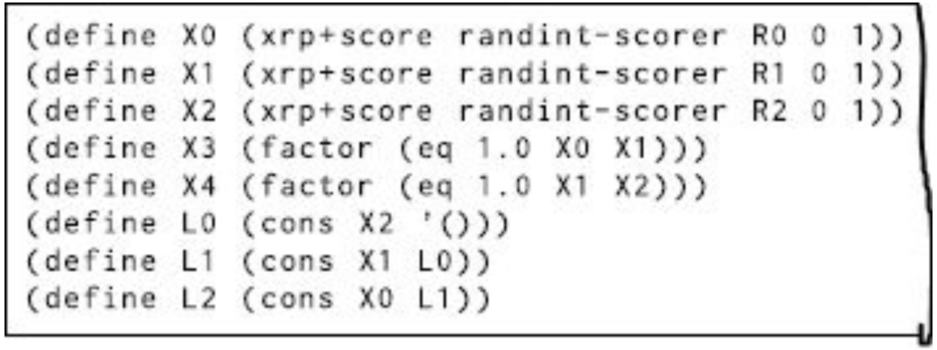
- We are interested in answering the question when *x* and *x'* are successive states of MH and *f* is the acceptance ratio as expressed through semantics of the probabilistic program.

Idea: Probability score associated with a run will be included in numerator and denominator of the acceptance ratio. Some factors will cancel out as same value is computed before and after the perturbation.

Incrementalization

Yang, et al.: **Reaching-definitions analysis** on the traces we record.

Each trace represents complete computation of numerator and denominator of acceptance ratio. We want to perform minimum computation possible between perturbations.

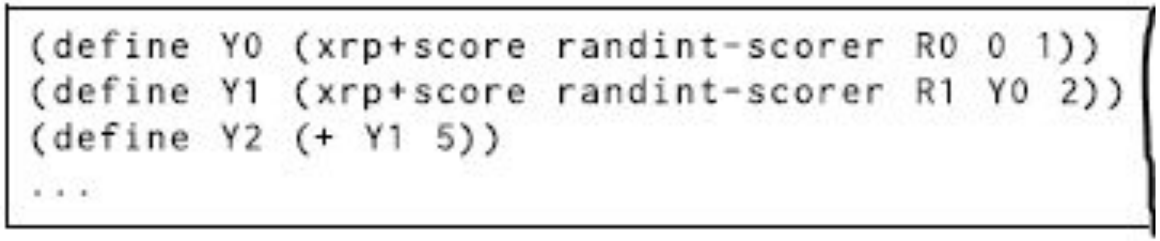


```
(define X0 (xrp+score randint-scorer R0 0 1))
(define X1 (xrp+score randint-scorer R1 0 1))
(define X2 (xrp+score randint-scorer R2 0 1))
(define X3 (factor (eq 1.0 X0 X1)))
(define X4 (factor (eq 1.0 X1 X2)))
(define L0 (cons X2 '()))
(define L1 (cons X1 L0))
(define L2 (cons X0 L1))
```

Incrementalization

Slicing by **Reaching Definitions-** Slice: The subset of variables that need to be recomputed. We compute each slice by performing a reaching definition analysis on the trace.

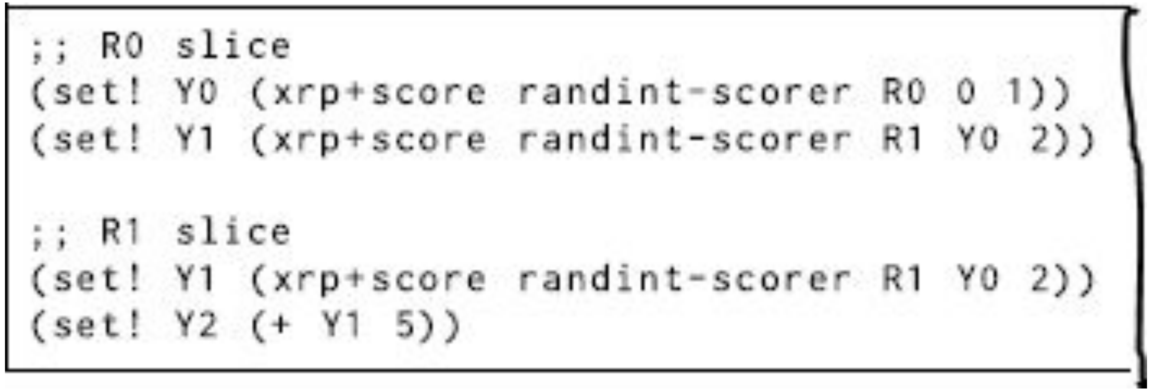
Given a input variable Ri, we identify all definitions that variable reaches by constructing a least fixed-point. Modification introduced: Input variables may reach the definition of other input variables, but no further



```
(define Y0 (xrp+score randint-scorer R0 0 1))
(define Y1 (xrp+score randint-scorer R1 Y0 2))
(define Y2 (+ Y1 5))
...
```

Incrementalization

Slices:



```
;; R0 slice
(set! Y0 (xrp+score randint-scorer R0 0 1))
(set! Y1 (xrp+score randint-scorer R1 Y0 2))

;; R1 slice
(set! Y1 (xrp+score randint-scorer R1 Y0 2))
(set! Y2 (+ Y1 5))
```

Caching with closures

Problem: We cannot run these slices in isolation.

Solution : To generate closures for each slice with an enclosing scope where the full set of variables is defined.

- Incrementalization involves elaborate caching schemes. We achieve caching by employing closures.

I am not talking in detail about models with structural changes:

- In models with structural changes, authors extended the caching that they do, to cache each trace where keys are values of structured variables involved in the change. In addition, they also compile and cache a structure change function for each structure proposal (Please read about this in section 6 of the paper).

Thank you!