ORACLE®

Compilation of Probabilistic Programs

Jean-Baptiste Tristan
Principal Member of Technical Staff
Machine Learning Research Group - Oracle Labs
October 6, 2018

ORACLE

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

This Talk

For machine learning people:

Compilers are marvels of engineering,
we can borrow ideas and principles from 60 years of research

For compiler people:

Probabilistic programming compilation brings new
interesting and challenging problems

Compilers and Interpreters

An interpreter *simulates* a program

A compiler *translates* the program into machine code

Why Care about Compilers?

We (hopefully) agree that probabilistic programming is useful

But what about *compiling* probabilistic programs?

Isn't an interpreter good enough?

In 2013, we designed and implemented the Augur language:

MAP inference for Bayesian networks *compiled* to GPU code

Running LDA:

- Augur (compiled): 1 GB, less than 20 minutes on GPU
- FACTORIE (interpreted): More than 6 hours on CPU
- Jags (interpreted): More than 128 GB, failed

Why Care about Compilers?

We (hopefully) agree that probabilistic programming is useful

But what about *compiling* probabilistic programs?

Isn't an interpreter good enough?

In 2013, we designed and implemented the Augur language:

MAP inference for Bayesian networks *compiled* to GPU code

Running LDA:

- Augur (compiled): 1 GB, less than 20 minutes on GPU
- FACTORIE (interpreted): More than 6 hours on CPU
- Jags (interpreted): More than 128 GB, failed

ORACLE

Compiler Engineering: a Principled Trade

Some principles that can inform our work:

- Compiler architecture
  - Pipeline of well-defined transformations and analysis
    - Closure conversion to implement first-class functions
    - Register allocation to use CPU registers
    - Abstract interpretation to cast static analysis
  - Between intermediate representations
    - Abstract syntax trees
    - Static Single Assignment
  - With well-defined semantics
- Compiler correctness
  - The compiler must preserve the semantics of the program
  - CompCert: a C compiler verified in Coq

ORACLE

Compiler Engineering: a Principled Trade

Some principles that can inform our work:

- Compiler architecture
  - Pipeline of well-defined transformations and analysis
    - Closure conversion to implement first-class functions
    - Register allocation to use CPU registers
    - Abstract interpretation to cast static analysis
  - Between intermediate representations
    - Abstract syntax trees
    - Static Single Assignment
  - With well-defined semantics
- Compiler correctness
  - The compiler must preserve the semantics of the program
  - CompCert: a C compiler verified in Coq

ORACLE

Compiler Engineering: a Principled Trade

Some principles that can inform our work:

- Compiler architecture
  - Pipeline of well-defined transformations and analysis
    - Closure conversion to implement first-class functions
    - Register allocation to use CPU registers
    - Abstract interpretation to cast static analysis
  - Between intermediate representations
    - Abstract syntax trees
    - Static Single Assignment
  - With well-defined semantics
- Compiler correctness
  - The compiler must preserve the semantics of the program
  - CompCert: a C compiler verified in Coq

ORACLE

Compiler Engineering: a Principled Trade

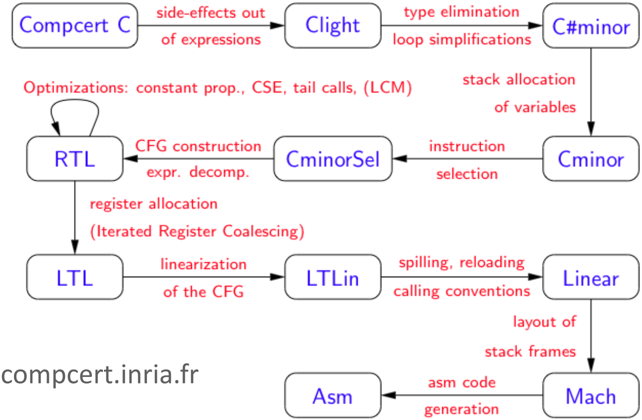
Some principles that can inform our work:

- Compiler architecture
  - Pipeline of well-defined transformations and analysis
    - Closure conversion to implement first-class functions
    - Register allocation to use CPU registers
    - Abstract interpretation to cast static analysis
  - Between intermediate representations
    - Abstract syntax trees
    - Static Single Assignment
  - With well-defined semantics
- Compiler correctness
  - The compiler must preserve the semantics of the program
  - CompCert: a C compiler verified in Coq

ORACLE

Architecture of the Compcert Compiler

Transformation is not one blob from C to Assembly



Source: compcert.inria.fr

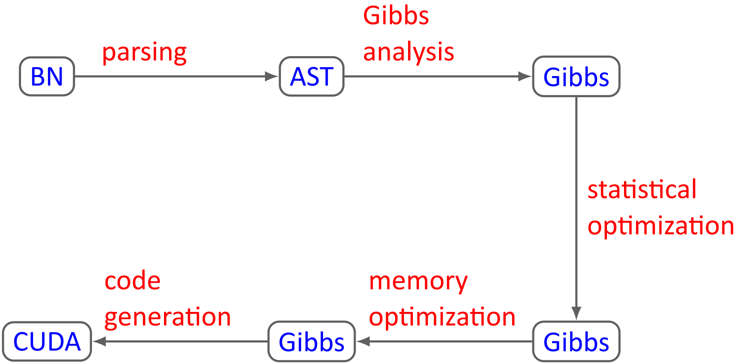
Compilation for Probabilistic Programs

These design principles can guide research in compilation:

- What are the key transformations?
  - *e.g.* How to derive a Gibbs sampler?
- How should we represent intermediate results?
  - Symbolic densities, “Gibbs equations”
- What are the high-level optimization specific to this field?
  - Approximation, estimation
- What is correctness for probabilistic program compilation?
  - *e.g.* Does compiler output converge to the query?

ORACLE

Augur Compilation Pipeline

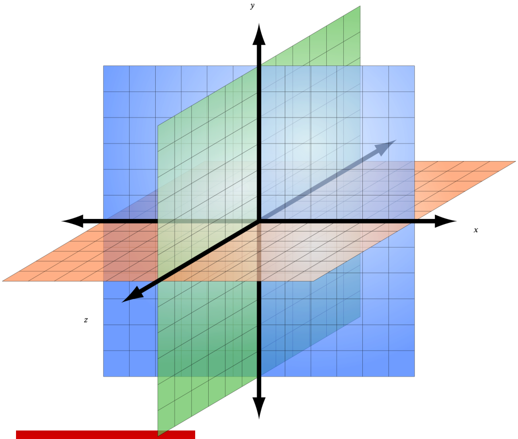


Augur Compilation Pipeline

BN

Language Classification

We need to specify what language we're compiling



3 dimensions of expressiveness

1. Model (CRFs, BNs)
2. Query (MAP, sampling)
3. Inference (MCMC, EM)

Modeling Language

- Bayesian Networks
- Predefined family of distributions with known pdf/pmf
- Only bounded recursion, no conditionals, no functions

```
val phi = Dirichlet(V,beta).sample(K)
val theta = Dirichlet(K,alpha).sample(M)
val w = for(i <- 1 to M)
          for(j <- 1 to N(i))
            val z: Int = Categorical(K,theta(i)).sample()
            Categorical(V,phi(z)).sample()
```

ORACLE

Query Language

- Fixed:

  Maximum a Posteriori ($\arg\max_{\theta,\phi} \mathbb{P}(\theta, \phi \mid W)$)
  sampling

- Assign constants to subset of variables

```
observe(w,data)
```

Inference

- Fixed:
  - Gibbs sampling
  - Metropolis-Hasting
  - Hamiltonian MC

We want to sample from $p(X_1, X_2 | X_3 = x_3)$

1. Choose x_1^0, x_2^0 at random
2. Repeat
   - $x_1^{t+1} \sim p(X_1 | X_2 = x_2^t, X_3 = x_3)$
   - $x_2^{t+1} \sim p(X_2 | X_1 = x_1^{t+1}, X_3 = x_3)$

These are called “Gibbs equations”

ORACLE

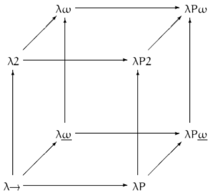
We Want a Refund!

That's not *really* probabilistic programming!

Maybe, maybe not, but...

Can write *many, many* useful models

Incredibly useful to learn about probabilistic program compilation



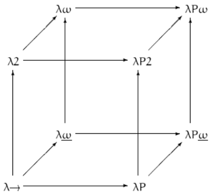
We Want a Refund!

That's not *really* probabilistic programming!

Maybe, maybe not, but...

Can write *many, many* useful models

Incredibly useful to learn about probabilistic program compilation



Augur Compilation Pipeline

Representation: Abstract Syntax

How should we initially represent the program?

- We choose to symbolically represent the density

$$p ::= cat \mid dir \mid bern$$

$$P ::= App(p, \vec{X}) \mid Cond(p, \vec{X}, \vec{X}) \mid Prod(P, P) \mid Mul(i, N, P)$$

For our program, we have:

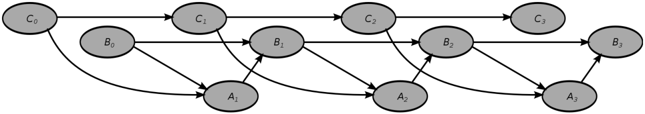
$$Prod(Mul(k = 1, K, App(dir, \phi_k)), Mul(m = 1, M, Prod(..., ...)))$$

Compiler vs Interpreter

Compiler:

$$Prod(Mul(k = 1, K, App(dir, \phi_k)), Mul(m = 1, M, Prod(..., ...)))$$

Interpreter:



ORACLE

Semantics

- π: abstract syntax of a probabilistic program
- q: query (assignment of variables to values)

The semantics of π_q is a function $[\![\pi_q]\!]$
from unobserved variables to probabilities

In our case, we can define $[\![\pi]\!]$ by induction on the AST

$$
\begin{aligned}
[\![bern(x, \mu)]\!] &= \mu^x (1 - \mu)^{(1-x)} \\
[\![Prod(t_1, t_2)]\!] &= [\![t_1]\!] \times [\![t_2]\!]
\end{aligned}
$$

Semantics

For LDA, $[\![\pi]\!] =$

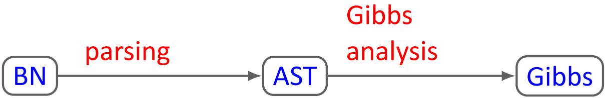
$$\lambda \phi, \boldsymbol{\theta}, \boldsymbol{z}, \boldsymbol{w}. \left[\prod_{k=1}^{K} dir(\phi_k)\right] \prod_{m=1}^{M} dir(\theta_m) \prod_{n=1}^{N_m} cat(w_{mn} | \phi_{z_{mn}}) \; cat(z_{mn} | \theta_m)$$

and

$$[\![\pi_{\boldsymbol{w}}]\!] = \lambda \phi . \lambda \boldsymbol{\theta} . \lambda \boldsymbol{z} \; \frac{[\![\pi]\!](\phi, \boldsymbol{\theta}, \boldsymbol{z}, \boldsymbol{w})}{\sum_{\boldsymbol{w}} [\![\pi]\!](\phi, \boldsymbol{\theta}, \boldsymbol{z}, \boldsymbol{w})}$$

ORACLE

Augur Compilation Pipeline



Gibbs Sampling

We want to sample from $p(X_1, X_2 | X_3 = x_3)$

1. Choose x_1^0, x_2^0 at random
2. Repeat
    - $x_1^{t+1} \sim p(X_1 | X_2 = x_2^t, X_3 = x_3)$
    - $x_2^{t+1} \sim p(X_2 | X_1 = x_1^{t+1}, X_3 = x_3)$

These are called "Gibbs equations"

ORACLE

Deriving Gibbs Equations by Rewriting

$$p(\phi_k|\boldsymbol{w}, \boldsymbol{z}, \boldsymbol{\theta}, \boldsymbol{\phi}^{-k}) = ?$$
$$p(\theta_m|\boldsymbol{w}, \boldsymbol{z}, \boldsymbol{\theta}^{-m}, \boldsymbol{\phi}) = ?$$
$$p(z_{mn} = k|w_{mn}, \theta_m, \phi_k) = ?$$

We derive the equations using a *term rewriting system*

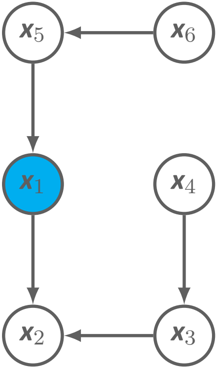
Deriving Gibbs Equations by Rewriting

The "Markov blanket" rule

Goal: get rid of conditionally independent variables

Example rule:

$$P(x \mid y) \Rightarrow \frac{P(x,y)}{\int P(x,y)\,\mathrm{d}x}$$

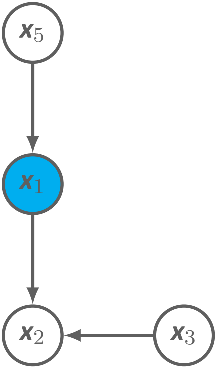


ORACLE

Deriving Gibbs Equations by Rewriting

The "Markov blanket" rule

Goal: get rid of conditionally independent variables



Example rule:

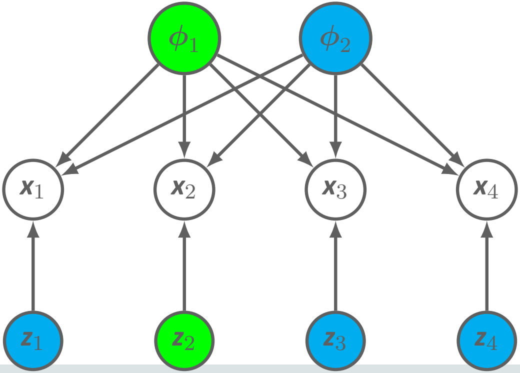
$$P(x \mid y) \Rightarrow \frac{P(x,y)}{\int P(x,y)\,\mathrm{d}x}$$

Deriving Gibbs Equations by Rewriting

The "mixture" rule

Goal: handling mixed and mixed-membership models

$$\prod_i^N P(x_i) \Rightarrow \prod_i^N \{P(x_i)\}_{q(i)=T} \prod_i^N \{P(x_i)\}_{q(i)=F}$$

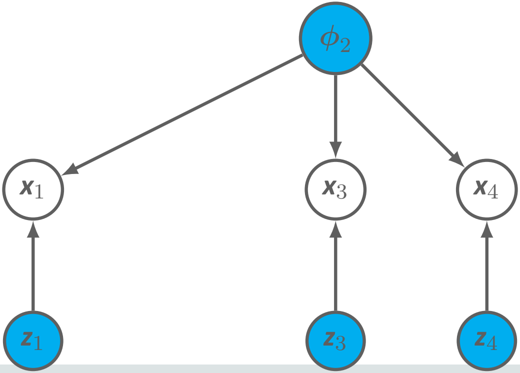


Deriving Gibbs Equations by Rewriting

The "mixture" rule

Goal: handling mixed and mixed-membership models

$$\prod_i^N P(x_i) \Rightarrow \prod_i^N \{P(x_i)\}_{q(i)=T} \prod_i^N \{P(x_i)\}_{q(i)=F}$$



Deriving Gibbs Equations by Rewriting

The "conjugacy" rule
Goal: closed form solutions for some continuous variables

Model: sequence $x_1 \ldots x_n$ of coin flips $\prod_i \mu^{x_i}(1-\mu)^{(1-x_i)}$
Query: $p(\mu|x_1 \ldots x_n)$?

$$p(\mu|x_1 \ldots x_n) \Rightarrow p(\mu) \prod_i \mu^{x_i}(1-\mu)^{(1-x_i)}$$

$$\Rightarrow ?$$

If p is Beta distribution then:

$$\Rightarrow Beta(\mu; \alpha + \sum_i \mathbb{1}[x_i = 1], \beta + \sum_i \mathbb{1}[x_i = 0])$$

Deriving Gibbs Equations by Rewriting

The "conjugacy" rule
Goal: closed form solutions for some continuous variables

Model: sequence $x_1 \ldots x_n$ of coin flips $\prod_i \mu^{x_i}(1-\mu)^{(1-x_i)}$
Query: $p(\mu | x_1 \ldots x_n)$?

$$p(\mu | x_1 \ldots x_n) \Rightarrow p(\mu) \prod_i \mu^{x_i}(1-\mu)^{(1-x_i)}$$

$$\Rightarrow ?$$

If p is Beta distribution then:

$$\Rightarrow Beta(\mu; \alpha + \sum_i \mathbb{1}[x_i = 1], \beta + \sum_i \mathbb{1}[x_i = 0])$$

ORACLE

Deriving Gibbs Equations by Rewriting

- Conjugate relation: sample from known distribution
- Non conjugate relation: sample using, *e.g.* rejection sampling
- Discovering these relations is critical
- Pointing out when they are available is useful
  - Something an IDE could do

ORACLE

The Gibbs Equations

$$p(\phi_k|\mathbf{w},\mathbf{z},\boldsymbol{\theta},\boldsymbol{\phi}^{-k}) = dir(\phi_k|\langle \mathtt{wpt}[k][v] + \beta : v \in [1..V]\rangle)$$
$$p(\theta_m|\mathbf{w},\mathbf{z},\boldsymbol{\theta}^{-m},\boldsymbol{\phi}) = dir(\theta_m|\langle \mathtt{tpd}[m][k] + \alpha : k \in [1..K]\rangle)$$
$$p(z_{mn} = k|w_{mn},\theta_m,\phi_k) = cat(w_{mn}|\phi_k)\ cat(k|\theta_m)$$

$\mathtt{tpd}[m][k]$: count occurences of topic k in document m

$\mathtt{wpt}[k][n]$: count occurences of word n assigned to topic k

Independence $\Rightarrow$ parallelism

The Final Code

In parallel: $\phi_k \sim Dir(\langle \texttt{wpt}[k][v] + \beta : v \in [1..V] \rangle)$

In parallel: $\theta_m \sim Dir(\langle \texttt{tpd}[m][k] + \alpha : k \in [1..K] \rangle)$

In parallel: $z_{mn} \sim \phi'_{w_{mn}} \odot \theta_m$

$\texttt{tpd}[m][k]$: count occurences of topic k in document m

$\texttt{wpt}[k][n]$: count occurences of word n assigned to topic k

Compiler vs Interpreter

Compiler:

Term rewriting

- \- Difficult to do, balancing ad hoc and impossible
- \- Does the rewrite process terminate?
- \- Can we characterize when it fails?
- \+ Having the equations is highly valuable
- \+ Can be parallelized very effectively for GPU

Interpreter:

At runtime, for each variable, "read off" the blanket

- \+ Trivial to do
- \- Expensive, "local" view of the program

ORACLE

Compiler Correctness

What would it mean for this compiler to be correct?

- Program π + query q
- Initial state sampled from μ
- The stochastic matrix for π_q is P_{π_q}
- δ: total variation distance

$$\forall \pi_q,\ \exists \tau,\ \forall \mu,\ \delta(\mu P^t_{\pi_q}, \tau) \to 0 \text{ as } t \to \infty$$

Not sufficient.

Compiler Correctness

What would it mean for this compiler to be correct?

- Program π + query q
- Initial state sampled from μ
- The stochastic matrix for π_q is P_{π_q}
- δ: total variation distance

$$\forall \pi_q,\ \exists \tau,\ \forall \mu,\ \delta(\mu P^t_{\pi_q}, \tau) \to 0 \text{ as } t \to \infty$$

Not sufficient.

ORACLE

Compiler Correctness

- Program π + query q
- Initial state sampled from μ
- The stochastic matrix for π_q is P_{π_q}
- δ: total variation distance

$$\forall \pi_q,\ \delta(\mu P_{\pi_q}^t, [\![\pi_q]\!]) \to 0 \text{ as } t \to \infty$$

Sufficient. Unfortunately, not necessary... but this is another talk

ORACLE

Compiler Correctness

- Program π + query q
- Initial state sampled from μ
- The stochastic matrix for π_q is P_{π_q}
- δ: total variation distance

$$\forall \pi_q,\ \delta(\mu P^t_{\pi_q}, [\![\pi_q]\!]) \to 0 \text{ as } t \to \infty$$

Sufficient. Unfortunately, not necessary... but this is another talk

ORACLE

Augur: Conclusion

- Some successes
  - We can generate very efficient distributed/GPU code for non-trivial models/inference
  - It's important to carefully represent the intermediate program
- Some remaining challenges
  - The term rewriting system is very unprincipled
  - Compiler code-base explosion to support more inference methods

Check out this work: Hakaru, Shuffle, Bayonet

Augur: Conclusion

- Some successes
  - We can generate very efficient distributed/GPU code for non-trivial models/inference
  - It's important to carefully represent the intermediate program
- Some remaining challenges
  - The term rewriting system is very unprincipled
  - Compiler code-base explosion to support more inference methods

Check out this work: Hakaru, Shuffle, Bayonet

ORACLE

Conclusion

For machine learning people:

Think in terms of:

- Narrowly-scoped transformations
- Intermediate representations
- Transformation correctness

For compiler people:

- Separation of concerns more difficult for probabilistic programming
- Some transformations (like Gibbs) deserve new ideas
- Verification milestone: prove correctness formally

ORACLE

More Info

Collaborators:

Oracle Labs ML: **Adam Pocock**, Steve Green
Oracle Labs PL: Guy Steele
Harvard: **Daniel Huang**, Greg Morrisett
CMU: Joseph Tassarotti

Papers:

NIPS'14 Augur: Data-Parallel Probabilistic Modeling
PLDI'17 Compiling Markov chain Monte Carlo algorithms for probabilistic modeling

Code: Augur v2, by Daniel Huang

- https://github.com/danehuang/augurv2

Integrated Cloud

Applications & Platform Services

ORACLE

ORACLE®