TURING.JL

A fresh approach to probabilistic programming in Julia

Kai Xu 22 June 2017

Hong Ge & Zoubin Ghahramani

Machine Learning Group, University of Cambridge

OUTLINE

- What is probabilistic programming?
- Why Julia?
- Key features
- How Turing.jl works?
- Live demo
- Q&A

WHAT IS PROBABILISTIC PROGRAMMING?

- Probabilistic programming is general-purpose programming with intrinsic support of non-deterministic statements.
- Such programming languages are called probabilistic programming languages (PPLs).
- One of the main applications of PPLs is probabilistic modelling, a popular field of machine learning.

- Two types of PPLs
  - Standalone:
    - BUGS
    - Stan
  - Embedded-in:
    - Probabilistic C in C
    - Anglican in Clojure
    - WebPPL in JS
    - Turing.jl in ★**Julia**★

```
using Turing

@model gdemo(x) = begin
  s ~ InverseGamma(2, 3)
  m ~ Normal(0, sqrt(s))
  x[1] ~ Normal(m, sqrt(s))
  x[2] ~ Normal(m, sqrt(s))
  s, m
end

modelf = gdemo([1.5, 2])
alg1 = PG(50, 1000)
chn1 = sample(modelf, alg)
alg2 = HMC(1000, 0.2, 3)
chn2 = sample(modelf, alg)
```

Code 1: Simple Turing.jl workflow

Looking closely, we can see the probablistic features we mentioned before in Turing.jl ...

```
using Turing

@model gdemo(x) = begin
  s ~ InverseGamma(2, 3)
  m ~ Normal(0, sqrt(s))
  x[1] ~ Normal(m, sqrt(s))
  x[2] ~ Normal(m, sqrt(s))
  s, m
end

modelf = gdemo([1.5, 2])
alg1 = PG(50, 1000)
chn1 = sample(modelf, alg)
alg2 = HMC(1000, 0.2, 3)
chn2 = sample(modelf, alg)
```

Code 1: Simple Turing.jl workflow

- Non-deterministic?
  - Distributions
- Language?
  - `@model` macro
  - ~ notaion
- Machine learning?
  - Sampling methods
    - SMC, PG, HMC, NUTS, Gibbs ...

WHY JULIA?

- Rich statistical libraries
  - Distributions.jl has a rich distributions
- Meta-programming
  - Turing's compiler, i.e. `@model` and ~
- Coroutines
  - Particle Gibbs (PG) implementation
- Automatic differentiation (AD)
  - Hamiltonian Monte Carlo (HMC) implementation
  - Generic typing help AD work with distributions

KEY FEATURES

- Universal probabilistic programming with an ituitive modelling interface embedded in friendly Julia
- PG sampler for distributions involving discrete variables and for stochastic control flows
- HMC sampler for differentiable distributions
- ❗ Compositional MCMC interface
- ❗ Resumption of MCMC chains
- ☐ More novel samplers, other inference methods, sampling from user-defined models, ...

❗ = new releases, ☐ = next steps

```
using Turing

@model gdemo(x, K) = begin
    m = Vector{Real}(K);
    for i = 1:K
        m[i] ~ Normal(0, 25)
    end
    s ~ InverseGamma(2, 3)
    N = length(x); z = tzeros(Int, N)
    for i = 1:N
        z[i] ~ Categorical(1/K*ones(K))
        x[i] ~ Normal(m[z[i]], sqrt(s))
    end
    z, s, m
end

modelf = gdemo([1.1, 1.0, 0.9, 2.1, 2.2], 2)
alg = Gibbs(1000, PG(50, 1, :z), HMC(1, 0.2, 3, :m, :s))
chn = sample(modelf, alg)
```

Code 2: Compositional MCMC interface

```
using Turing

@model gdemo(...) = begin
  ...
end

modelf = gdemo(...)
chn1 = sample(modelf, HMC(1000, 1, 0.2, 3; save_state=true))
chn2 = sample(modelf, NUTS(1000, 0.65; resume_from=chn1))
```

Code 3: Resumption of MCMC chains

HOW TURING.JL WORKS?

Key ML techniques

- Bayesian inference
  - General framework for probabilistic modelling
- Sampling
  - Particle filtering
  - Markov chain Monte Carlo

Key Julia techniques

- Coroutines
  - Particle based samplers
- Automatic differentiation
  - Hamiltonian based samplers

Key system components

- Model *defined by* users
  - Normal Julia program with modelling operations
- Sampler *specified by* users
  - Need to interact with model
- `VarInfo`
  - Key data structure
  - Enable interactions between models and samplers
  - Users don't see it
- Samples *returned to* users
  - Embedded in `Chain` type

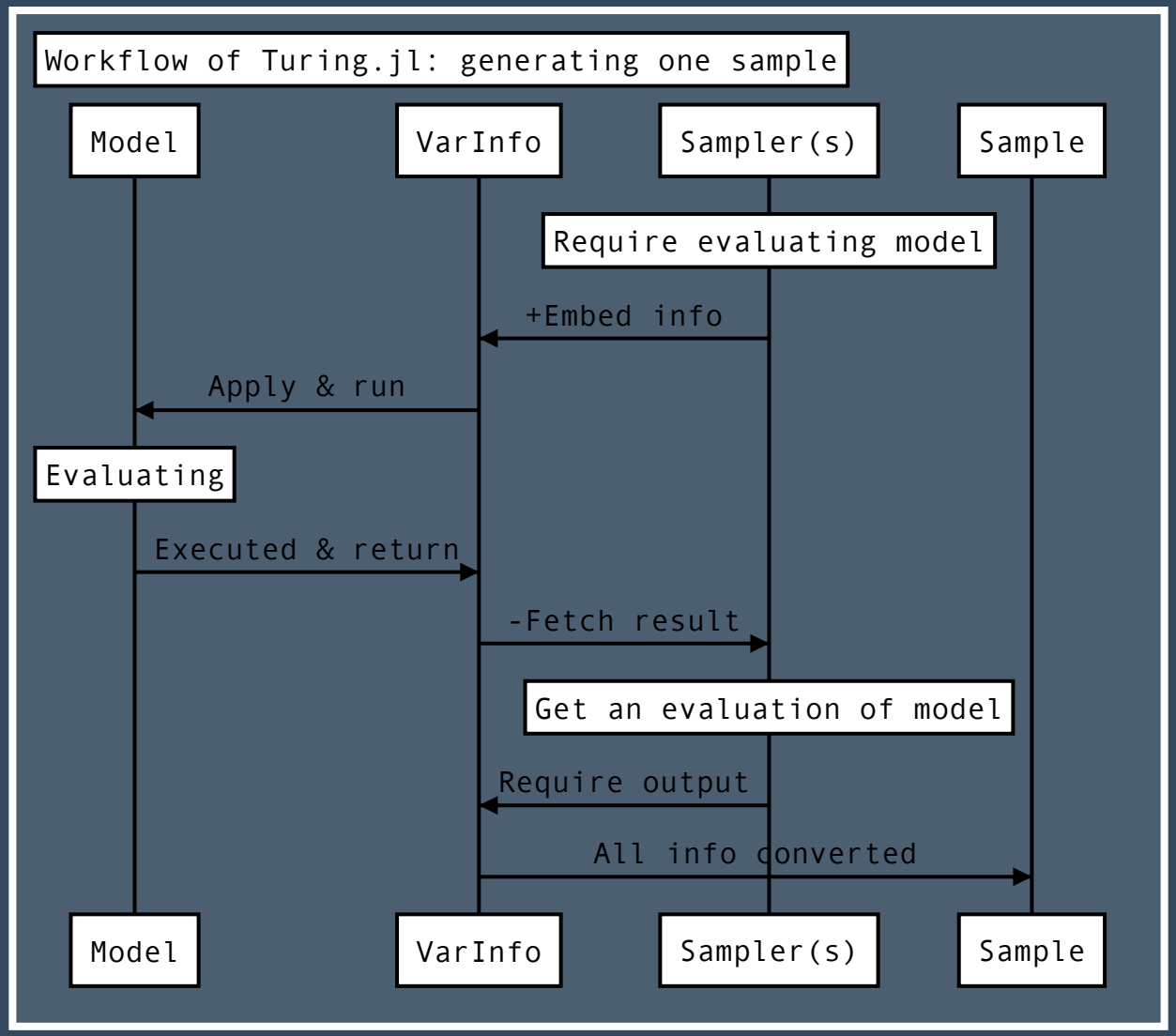

Figure 1: Workflow of Turing.jl: generating one sample

HOW TURING.JL USES COROUTINES

- IS, SMC and PG represent each particle running in parallel using a coroutine
- Each particle is essentially a copy of the model, i.e. a block of normal Julia code
- Duplicating or killing a particle is cheap
- Continuation and pausing of coroutine is frequently used because SMC and PG do model evaluation sequentially for each observation
- Lead to state-of-the-art SMC and PG performance

HOW AD WORKS IN TURING.JL

1. Amend variables that need gradient information into `Dual` numbers
2. Set dual parts to `1` for each dimension
3. Update variables in `VarInfo`
4. Do model evalutation
5. Fetch gradient information from returns
    - Returned log-likelihood is in `Dual` type with gradient as its dual parts

LIVE DEMO

Time to see a live Turing.jl program ...

Q&A

Any question?

THANK YOU FOR LISTENING 😄