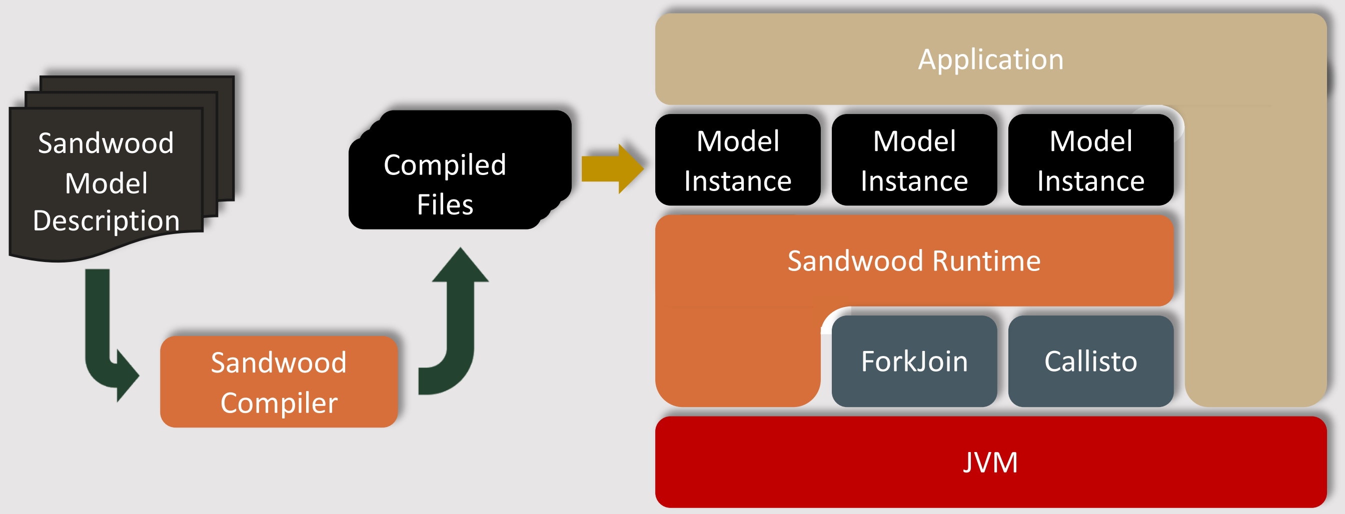
Sandwood: Runtime Adaptable Probabilistic Programming for Java

Daniel Goodman, Adam Pocock, Jason Peck, Guy Steele

Project Aims

- Create a JVM based probabilistic programming language that will be familiar to Java developers for inclusion in Java applications.
- Create a compiler and runtime for efficient encapsulated models allowing them to be distinct components of a system.
- Construct a range of backend implementations for high performance and scalability that can adjust to different runtime systems: Multi-CPU, Multi-GPU, Java Vector API (JEP 338), ….

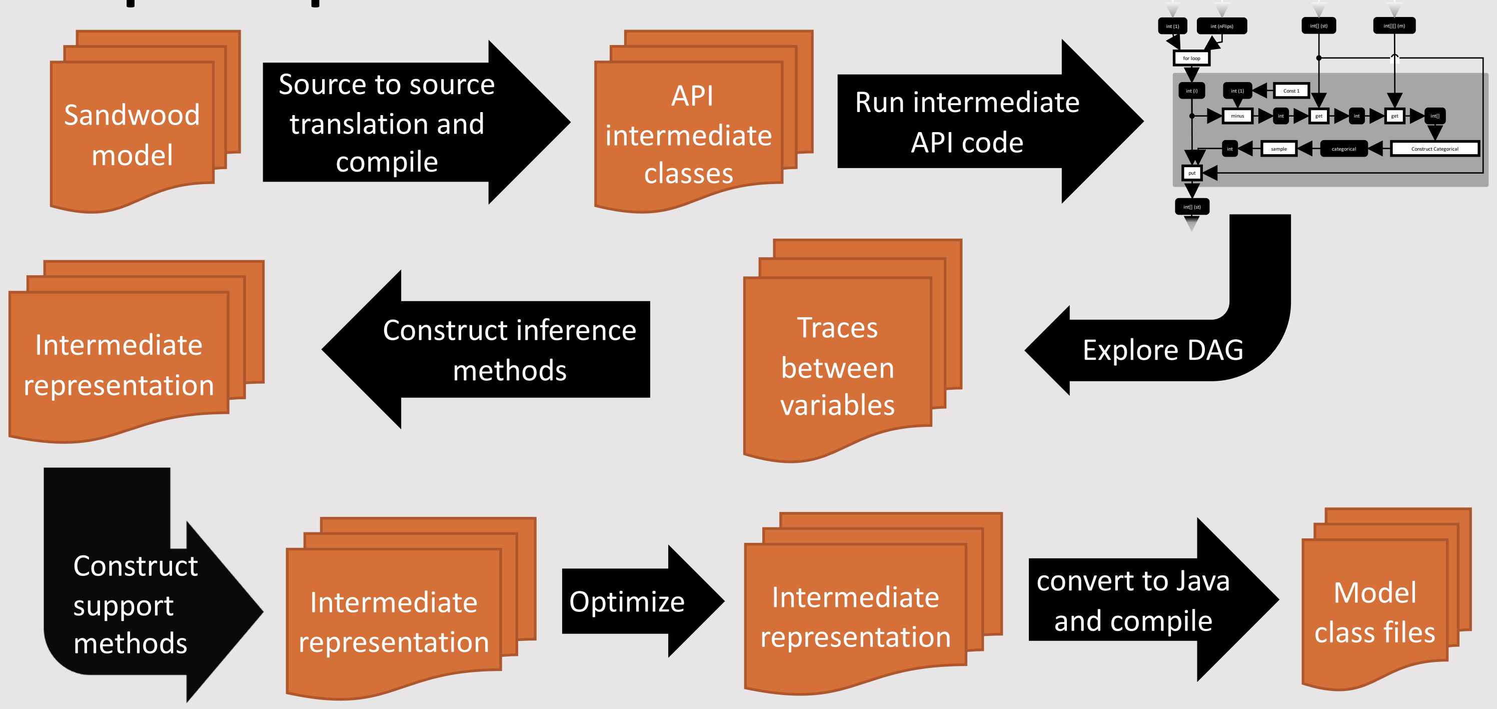
Sandwood Components



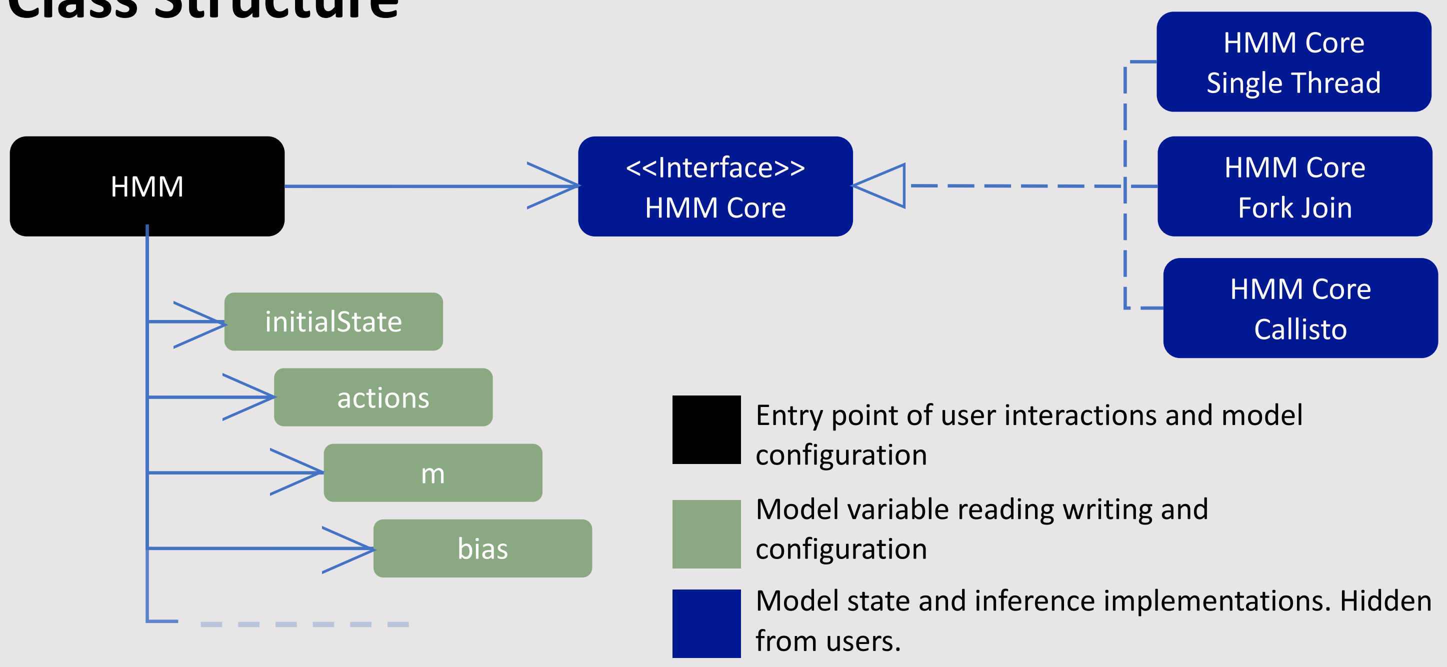
Example HHM Model

```
package examples.hmm;

model HMM(boolean[] measured, int nCoins) {
  //Construct a transition matrix m.
  double[] v = new double[nCoins] <~ 0.1;
  double[][] m = dirichlet(v).sample(nCoins);

  //Construct a weighting for the first
  //coin to flip.
  double[] initialCoin = dirichlet(v).sample();

  //Construct a bias for each coin
  double[] bias = beta(1.0, 1.0).sample(nCoins);

  //Allocate space to record which coin is flipped.
  int nFlips = measured.length;
  int[] st = new int[nFlips];

  //Calculate the movements between coins.
  st[0] = categorical(initialCoin).sampleDistribution();
  for (int i: [1..nFlips) )
    st[i] = categorical(m[st[i - 1]]).sampleDistribution();

  //Flip the coins.
  boolean[] flips = new int[nFlips];
  for (int j: [0..nFlips) )
    flips[j] = bernoulli(bias[st[j]]).sample();

  //Assert that the flips match the measured data.
  flips.observe(measured);
}
```

Compiler Pipeline


Sandwood model
Source to source translation and compile
API intermediate classes
Run intermediate API code
Explore DAG
Traces between variables
Construct inference methods
Intermediate representation
Construct support methods
Intermediate representation
Optimize
Intermediate representation
convert to Java and compile
Model class files

Class Structure



■ Entry point of user interactions and model configuration

🟩 Model variable reading writing and configuration

🟦 Model state and inference implementations. Hidden from users.

Application Code

```
//Construct the model
int nCoins = 3;
boolean[] flips = loadObservedFlips(....);
HMM model = new HMM(flips, nCoins);

//Set the retention policies
model.setDefaultRetentionPolicy(RetentionPolicy.MAP);
model.st.setRetentionPolicy(RetentionPolicy.NONE);

//Run 2000 inference steps to infer model values
model.inferValues(2000);

//Gather the results.
double[] bias = model.bias.getMAP();
double[][] transitions = model.m.getMAP();
```

Single Threaded Performance

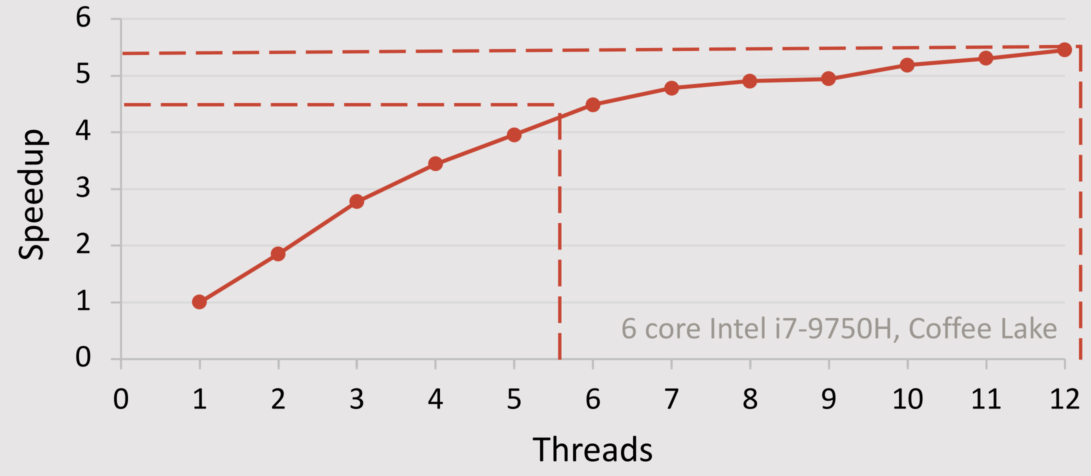
Comparing the example HMM model in Sandwood (Single threaded) with the same model in PyMC3.

Model and input length	Iterations				
	1000	2000	4000	8000	16000
PyMC3, 1k	201.8s	388.4s	769.4s	1523s	3021s
Sandwood, 1k	0.174s	0.367s	0.760s	1.450s	2.809s
Sandwood, 10k	1.764s	3.499s	7.500s	14.34s	25.77s

Speedups are in excess of 1000X

Multi-Threaded Performance

Measuring speedup with a more complex HMM model on a 6 core Intel machine with hyperthreading.



4.5 times speedup with 6 threads rising to 5.45 with hyper-threading.

Conclusions

- Sandwood is a fast scalable probabilistic programming language for the JVM.
- Sandwood is designed to be familiar to Java developers to prevent models becoming black boxes to the people responsible for maintaining the system.
- Compiled models are structured in an intuitive Object-Oriented style enabling a clean separation between the model and the application.
- Common parts of models can be described in functions that can be shared between models.
- Supports a subset of Javadoc allowing models to be self documenting.

daniel.goodman@oracle.com