Delayed sampling via Barriers and Funsors

Fritz Obermeyer, Eli Bingham @ Broad Institute

Summary: Funsor is a powerful low-level technology for implementing PPLs. Using Funsor, we can implement a **delayed sampling** PPL on this little poster.

Example: consider a model of a stochastic control system, attempting to keep a latent state z within [-10,10].

```
def model():
    z = sample("z_init",Normal(0,1))    # latent state
    k = 0 * z                           # control
    cost = 0 * z                        # cumulative cost of controlle
    for t in range(1000):
        z,k,cost = barrier([z,k,cost])  # inference may resample here
        k = where(z > 10, k + 1, k)
        k = where(z < 10, k + 1, k)
        k = where(-10 <= z & z <= 10, 0 * k, k)
        z = sample(f"z_{t}",Normal(z+k,1))
        x = sample(f"x_{t}",Normal(z,1))
    return cost
```

Our embedded PPL extends Python with two primitives: **sample()** and **barrier()**.

Barrier returns ground versions of its input arguments, eliminating any free / delayed variables.

```
def sample(name, dist):
    return HANDLER.sample(name, dist)

def barrier(state):
    return HANDLER.barrier(state)
```

1

2

We'll implement inference algorithms as **effect handlers**, i.e. contexts for nonstandard interpretation.

```
with MyInferenceAlgorithm(observations=observations) as inference:
    output = model()  # executes with nonstandard interpretation

posterior = inference.get_posterior(output)
```

The standard interpretation will draw samples from **sample()** and simply treat **barrier()** as the identity function.

Nonstandard interpretations will later inherit from this base class, and we will set them as the global HANDLER.

Effect handlers like this are used by Pyro and Edward2.

```
class StandardHandler:
    def __enter__(self):
        # install this handler at the beginning of each with statement
        global HANDLER
        self.old_handler = HANDLER
        HANDLER = self
        return self

    def __exit__(self, type, value, traceback):
        # revert this handler at the end of each with statement
        global HANDLER
        HANDLER = self.old_handler

    def sample(self, name, dist):
        return dist.sample()  # by default, draw a random sample

    def barrier(self, state):
        return state  # by default do nothing

HANDLER = StandardHandler()
```

Effect handlers for inference via Sequential Monte Carlo

3

We implement **sequential importance resampling** by updating a vector log_joint of particle log weights.

At **sample()** statements we sample each particle independently.

At **barrier()** statements we resample particles.

```
class SMC(StandardHandler):
    def __init__(self, observations, num_particles=100):
        self.observations = observations
        self.log_joint = zeros(num_particles)
        self.num_particles = num_particles

    def sample(self, name, dist):
        if name in self.observations:
            value = self.observations[name]
        else:
            value = dist.sample(sample_shape=self.log_joint.shape)
        self.log_joint += dist.log_prob(self.observations[name])
        return value

    def barrier(self, state):
        index = Categorical(logits=self.log_joint).sample()
        self.log_joint[:] = 0
        state = [x[index] for x in state]
       return state

    def get_posterior(self, value):
        probs = exp(self.log_joint)
        probs /= probs.sum()
        return {"samples": value, "probs": probs}
```

4

Effect handlers for inference via Variable Elimination & Delayed Sampling

We implement **variable elimination** inference via Funsor's lazy compute graphs and dynamic programming.

This handler ignores **barrier()** statements, so inference is exact and completely lazy.

```
class VariableElimination(StandardHandler):
    def __init__(self, observations, num_particles=100):
        self.observations = observations
        self.log_joint = funsor.Number(0)
        self.num_particles = num_particles

    def sample(self, name, dist):
        if name in self.observations:
            value = self.observations[name]
        else:
            value = funsor.Variable(name)  # create a delayed sample
        self.log_joint += dist.log_prob(self.observations[name])
        return value

    def get_posterior(self, value):
        return funsor.Expectation(self.log_joint, value)
```

Finally we implement delayed sampling by extending **VariableElimination** to eagerly eliminate variables at **barrier()** statements.

```
class DelayedSampling(VariableElimination):
    def barrier(self, state):
        subs = self.log_joint.sample(state.inputs, self.num_samples)
        self.log_joint = self.log_joint(**subs)
        state = [x(**subs) for x in state]
        return state
```