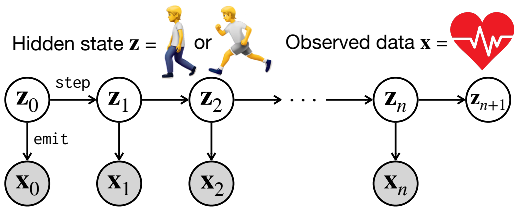
Compiling Probabilistic Programs for Variable Elimination with Information Flow

UNIVERSITY OF WATERLOO
DAVID R. CHERITON SCHOOL OF COMPUTER SCIENCE

Jianlin Li, Eric Wang, and Yizhou Zhang

Hidden Markov model as a probabilistic program

```
z0 = sample Bernoulli(.5)
observe x0 from emit(z0)
z1 = sample step(z0)
observe x1 from emit(z1)
z2 = sample step(z1)
observe x2 from emit(z2)
...
z101 = sample step(z100)
return z101
```



Inference problem: $p\left(\mathbf{z}_{n+1} \mid \mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_n\right) = ?$

SlicStan: For n = 60, compilation takes 17 minutes.

Our Approach

```
# recursive & probabilistic
def hmm(z0, data) =
  case data of
  | nil => return z0
  | cons x xs =>
    y = hmm(z0, xs)
    observe x from emit(y)
    z = sample step(y)
    return z
```

MAPPL

```
# recursive & pure
def hmm(k, z0, data) =
  case data of
  | nil => k(z0)
  | cons x xs =>
    let k' y =
      emit(y).p(x) *
      Σz(step(y).p(z) * k(z))
    in hmm(k', z0, xs)
```

Easy modeling | Correct compilation | Scalable compilation | Scalable inference

Marginalization by enumeration: $O(2^n)$ time complexity

$$p\left(\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_n, \mathbf{z}_{n+1}\right) = \sum_{\mathbf{z}_0}\sum_{\mathbf{z}_1}\sum_{\mathbf{z}_2}\cdots\sum_{\mathbf{z}_n} p\left(\mathbf{z}_0, \mathbf{x}_0, \mathbf{z}_1, \mathbf{x}_1, \cdots, \mathbf{z}_n, \mathbf{x}_n, \mathbf{z}_{n+1}\right)$$

marginalize out $\mathbf{z}_0, \mathbf{z}_1, \mathbf{z}_2, \cdots, \mathbf{z}_n$

$$= \sum_{\mathbf{z}_0}\sum_{\mathbf{z}_1}\sum_{\mathbf{z}_2}\cdots\sum_{\mathbf{z}_n} \underbrace{p(\mathbf{z}_0)\, p(\mathbf{x}_0 \mid \mathbf{z}_0)\, p(\mathbf{z}_1 \mid \mathbf{z}_0)\, p(\mathbf{x}_1 \mid \mathbf{z}_1)\, p(\mathbf{z}_2 \mid \mathbf{z}_1) \cdots p(\mathbf{z}_{n+1} \mid \mathbf{z}_n)}_{\text{table of } 2^{n+1} \text{ rows}}$$

Marginalization by factorization & variable elimination: $O(n)$ time

$$p\left(\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_n, \mathbf{z}_{n+1}\right) = \sum_{\mathbf{z}_n} \underbrace{\left(p(\mathbf{z}_{n+1} \mid \mathbf{z}_n) \sum_{\mathbf{z}_{n-1}} \left(\cdots \sum_{\mathbf{z}_1} \underbrace{\left(p(\mathbf{x}_1 \mid \mathbf{z}_1)\, p(\mathbf{z}_2 \mid \mathbf{z}_1) \sum_{\mathbf{z}_0} \underbrace{p(\mathbf{z}_0)\, p(\mathbf{x}_0 \mid \mathbf{z}_0)\, p(\mathbf{z}_1 \mid \mathbf{z}_0)}_{\text{table of } 2^2 \text{ rows}}\right)}_{\text{table of } 2^2 \text{ rows}} \cdots\right)\right)}_{\text{table of } 2^2 \text{ rows}}$$

Factorization as program partitioning via information-flow typing

Variable elimination amounts to incrementally compiling away probabilistic effects

```
H  z0 = sample Bernoulli(.5)
   observe x0 from emit(z0)     H
L  z1 = sample step(z0)         H
   observe x1 from emit(z1)     L
L  z2 = sample step(z1)         L
   observe x2 from emit(z2)     L
   ...
L  z101 = sample step(z100)     L
```

partition over $\mathbf{z}_0$

probabilistic | High partition

```
z0 = sample Bernoulli(.5)
observe x0 from emit(z0)
z1 = sample step(z0)
```

pure

$$\sum_{\mathbf{z}_0} e_0$$

```
observe x1 from emit(z1)
z2 = sample step(z1)
observe x2 from emit(z2)
...
z101 = sample step(z100)
```

Low partition

Doesn't information flow from `step(z0)` to `z1`?

Crucial for program partitioning

Information flows from `step(z0)` to the distribution over `z1`'s values.

But `z1`'s value in an execution trace contributes zero information to the semantics of the program.

Information-flow typing of distributions

Typing rule for variable bindings in a usual information-flow type system

$$\frac{\Gamma \vdash t_1 : \tau_1^{l_1} \qquad \Gamma, x : \tau_1^{l_1} \vdash t_2 : \tau_2^{l_2}}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 \;:\; \tau_2^{l_2}}$$

$[\![t_1]\!]$ is a distribution, and l_1 classifies $[\![t_1]\!]$

Typing rule for variable bindings in MAPPL

$$\frac{\Gamma \vdash t_1 : \tau_1^{l_1} \qquad \Gamma, x : \tau_1^{\mathrm{L}} \vdash t_2 : \tau_2^{l_2}}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 \;:\; \tau_2^{l_1 \sqcup l_2}}$$

Distribution $[\![\text{let } x = t_1 \text{ in } t_2]\!]$ is obtained by marginalizing out x w.r.t. $[\![t_1]\!]$

Soundness of information-flow typing via a logical-relations argument

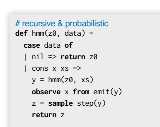
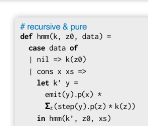
Low-labeled computation behaves irrespective of high-labeled input

Noninterference

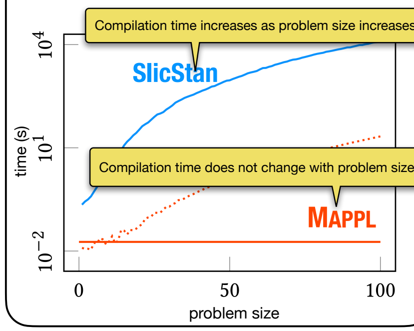
$$x : \tau^{\mathrm{H}} \vdash t : \tau'^{\mathrm{L}} \implies \int f(u)\, [\![t]\!]_{x \mapsto v_1}(\mathrm{d}u) = \int f(u)\, [\![t]\!]_{x \mapsto v_2}(\mathrm{d}u)$$

Compiler correctness

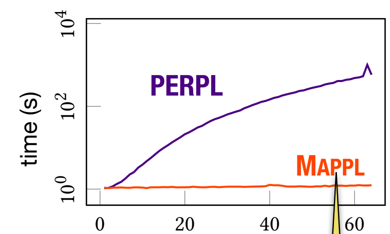
Compilation preserves semantics

$\int_v [\![$  $]\!](\mathrm{d}v) = [\![$  $]\!]$

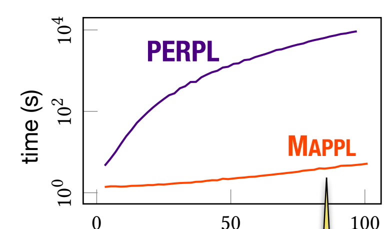
Scalability of compilation



Scalability of exact inference



Recovers the linear-time "forward algorithm" (Rabiner & Huang 1986)



Recovers the cubic-time "inside algorithm" (Baker 1979)