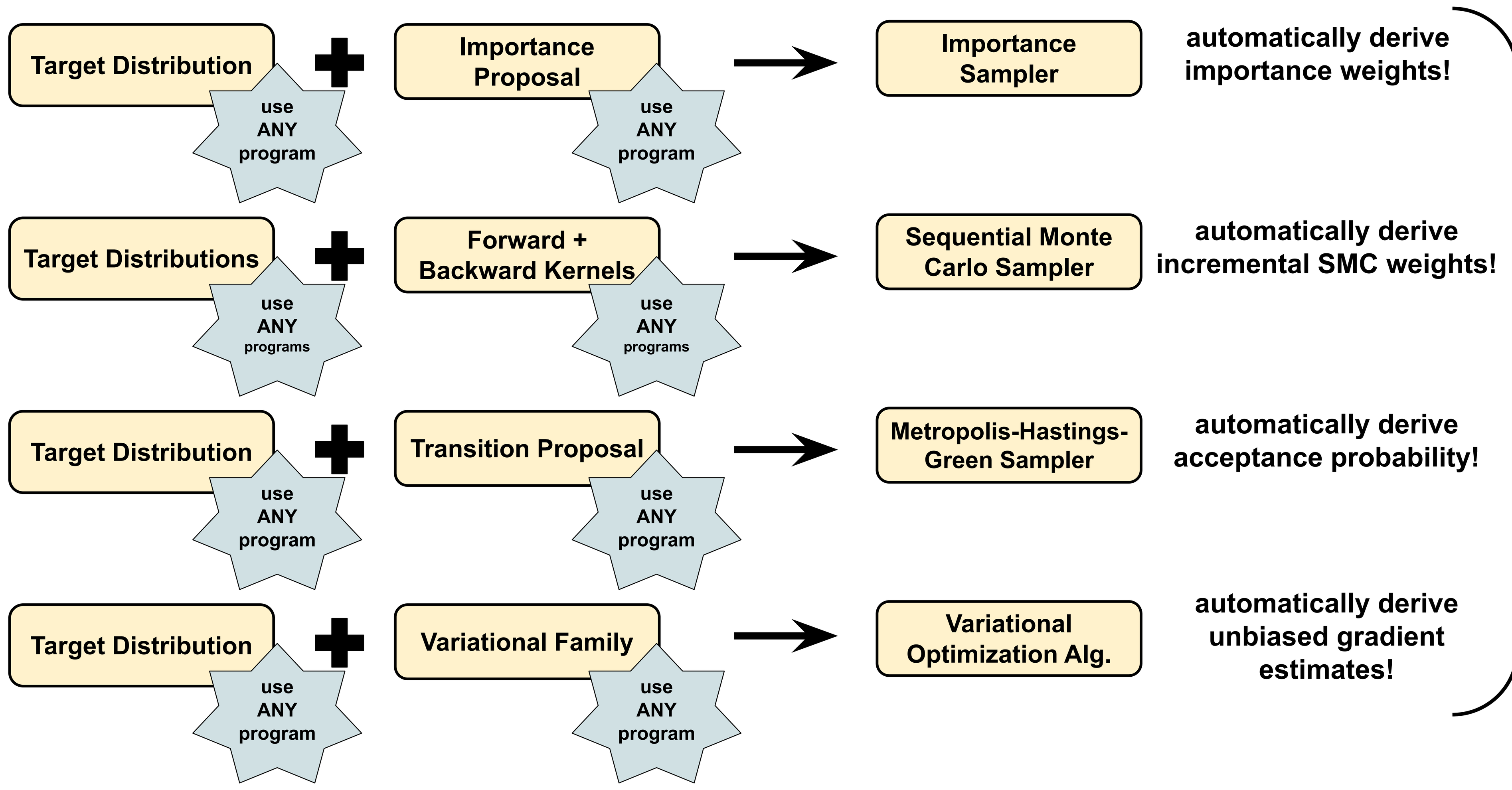
Auxiliary-Variable Programmable Inference

Alexander K. Lew[1] Benjamin Sherman[2] Marco Cusumano-Towner[1] Michael Carbin[2] Vikash K. Mansinghka[1]
[1]MIT Probabilistic Computing Project [2]MIT Programming Systems Group

1. (One View of) Programmable Inference: the Goal and the Challenge

The Goal: Automate Inference Algorithms from Declarative Specs



The Challenge: Densities

Simple enough if each program supports:

- Simulation
- Density Evaluation

But densities can:

- Be intractable

```
sum([exp(normal(i,1))
     for i in range(100)])
```

- Fail to exist (with respect to the usual reference measures)

```
let x ~ normal(0,1) in (x, 2x)
min(normal(0,1), 0)
```

Existing Approaches to Density in Probabilistic Programming

Trace-Based (Gen, Pyro, ProbTorch, WebPPL)

- Compute joint densities of *traces* of all primitive random choices made by prob. progs — easy multiplications
- Proposal/variational family primitive choices must be in 1-1 correspondence with target distribution choices
- Expressiveness rests on which primitives are available

Symbolic (Hakaru, Stochaskell, PSI, Bhat et al, Mattinson & Ong)

- Transform prob. progs into (unbiased estimators of) densities with respect to certain reference measures
- Must "total" (exactly evaluate) possibly intractable densities that appear in denominators (e.g., proposals)
- Supports some loops, but not general recursion

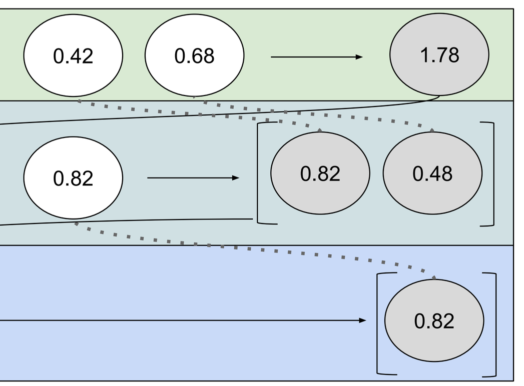
2. Our Approach: Inference Towers

We do not require densities, but *do* require programs to be equipped with internal proposals

```
def sum_uniforms():
  return sample(rand()) + 2*sample(rand())
```

```
with proposal(y):
  lo, hi = max(y - 2, 0), min(1, y)
  u = sample(uniform(lo, hi))
  return [u, (y - u1) / 2]
```

```
with proposal(choices):
  return [first(choices)]
```



Equip every probabilistic function *P*, at definition time, with a *helper* probabilistic function *Q* (its **internal proposal**), which takes as **input** the return value of *P*, and generates as **output** a possible **list of return values of *P*'s** callees.

(If Q makes probabilistic calls, it needs its own internal proposal—creating an inference *tower*.)

Why? Given any *two* valid tower-equipped programs over the same output space (F and G, with F << G), **we'll show how to automatically derive a valid importance sampler.**

The technique **supports recursion**, **does not require** absolute continuity w.r.t. **any particular base measure**, **does not require evaluating integrals** or large sums.

More Examples:

uniform(a,b)

```
def uniform(a, b):
  u = sample(rand())
  return u * (b - a) + a

with proposal(y):
  return [(y - a) / (b - a)]
```

geometric(p)

```
def geometric(p):
  if sample(flip(p)):
    return 0
  return 1 + sample(geometric(p))

with proposal(n):
  return n == 0 ? [t] : [f, n-1]
```

rejection(p, pred)

```
def rejection(p, pred):
  x = sample(p)
  return pred(x) ? x : sample(rejection(p, pred))

with proposal(accepted):
  x = sample(p)
  if pred(x) || sample(flip(0.1)):
    return [accepted]
  return [x, accepted]

with proposal(choices):
  if len(choices) == 2:
    return [first(choices), f]
  x = sample(p)
  return pred(x) ? [x] : [x, t]

with proposal(q_choices):
  if len(q_choices) == 1:
    return q_choices
  return second(q_choices) ? [first(q_choices)] : []
```

flip(p)

```
def flip(p):
  return sample(rand()) < p

with proposal(b):
  return [sample(b ?
             uniform(0, p) :
             uniform(p, 1))]

with proposal(choices):
  return choices
```

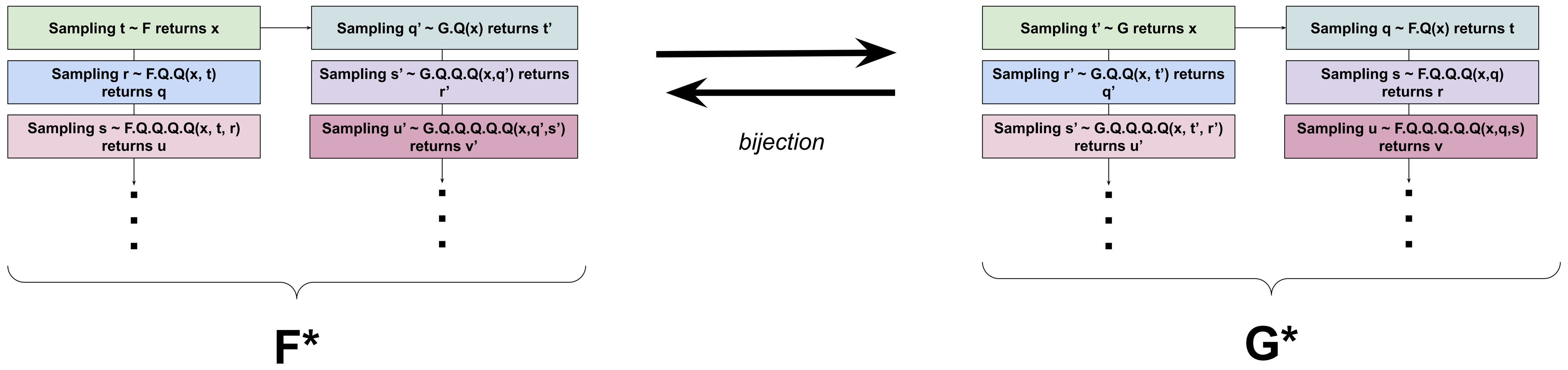
flip_and_flip(p)

```
def flip_and_flip(p):
  return sample(flip(p)) && sample(flip(p))

with proposal(b):
  return b ? [t, t] : sample(flip(0.5)) ? [f] : [t, f]

with proposal(flips):
  return b ? [] : [len(flips) == 1]
```

Compare to Parametric Inversion (Tavares et al.) **or** Gen Internal Proposals (Cusumano-Towner et al.) **or** Importance-Weighting Combinator (Sennesh et al.)

3. Automatic Differentiation for Density Ratios of Tower-Equipped Probabilistic Programs

Given two probabilistic programs, F and G, their inference towers define a bijection between all rand() calls made by F* and G*, which are marginally equal to F and G:



The absolute value of the determinant of the Jacobian of this bijection is the Radon-Nikodym derivative of F* w.r.t. G*, an unbiased estimate of dF/dG(x) when (x, ...) ~ G*.

Furthermore, it can be computed **compositionally**, using the **Cauchy-Binet Theorem**:

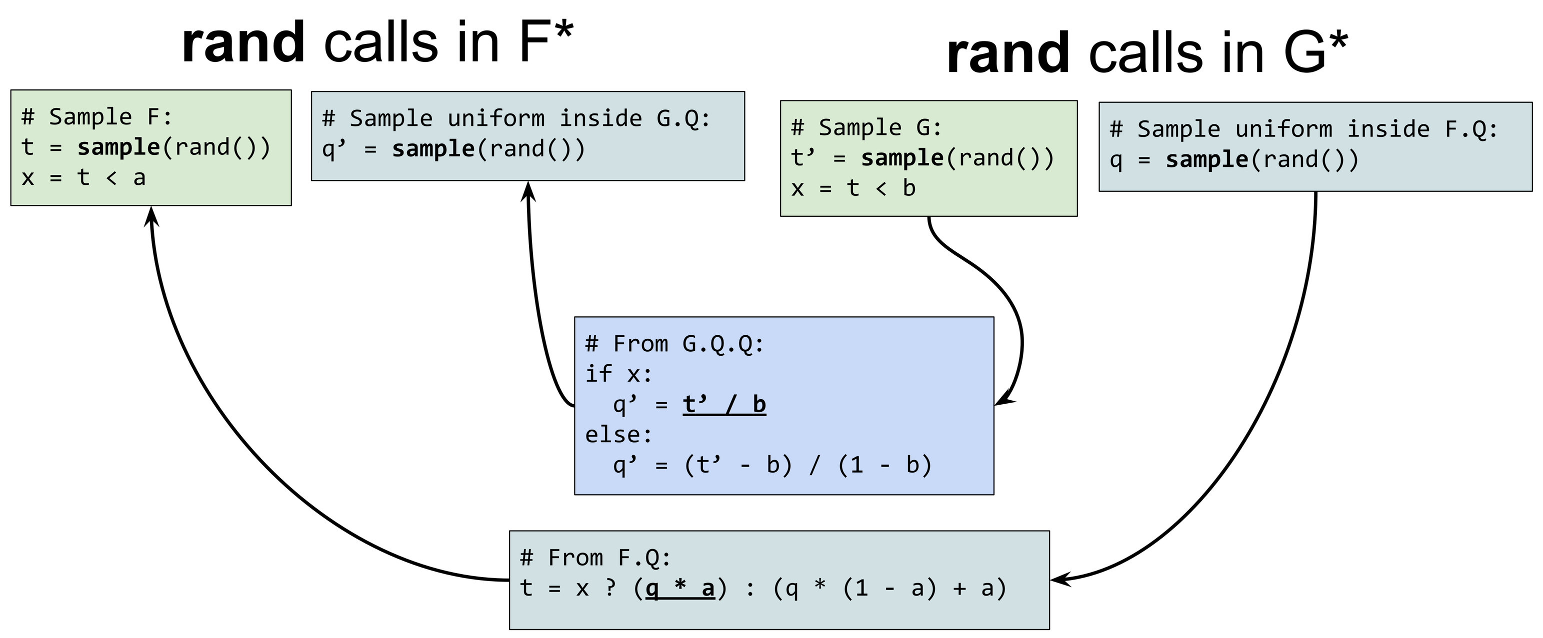
$$\det(AB) = \sum_{S \in \binom{[n]}{m}} \det(A_{[m],S}) \det(B_{S,[m]})$$

Each prob. prog.'s tower can be encapsulated behind **estimate** and **propose**, an alternative to the common **logpdf** / **sample** interface.
Further, **estimate** & **propose** for a program F can be implemented in terms of **estimate** & **propose** for its callees.

Compare to Gen's propose and assess interface methods

Simple example:
F = flip(a); G = flip(b)

	t'_G	q_F
t_F	**0**	**a**
q'_G	**1/b**	**0**

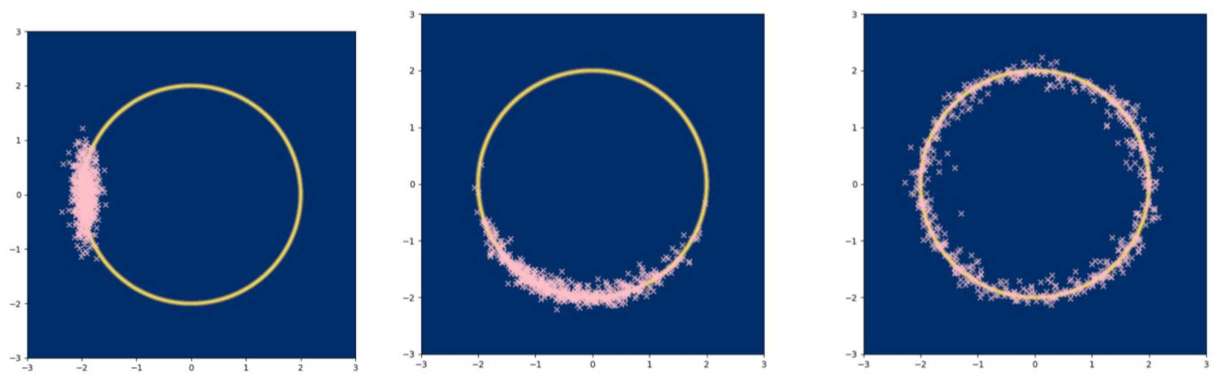


Bijection from G* → F*

4. Beyond Importance Sampling: SMC, MCMC, and Variational Inference

Possible to develop versions of SMC, MCMC, and SVI that use these towers, enabling expressive proposals and variational families.

Right: variational inference with a variational family that itself calls a recursive importance resampling procedure.



Basic SVI IS SVI (n=10, t=500) IS SVI (n=20, t=500)

Cf. Burda et al., Importance-Weighted Auto-Encoders