3rd Symposium on Advances in Approximate Bayesian Inference, 2020 1–35

Transforming Worlds: Automated Involutive MCMC for Open-Universe Probabilistic Models

Anonymous Authors
Anonymous Institution

1. Introduction

Open-universe probabilistic models (OUPMs) enable Bayesian inference about the existence and attributes of latent objects underlying observed data, as well as their interconnections. Prominent applications include seismic monitoring for global nuclear safety (Arora et al., 2013; Arora, 2012); information extraction from natural language documents (Russell et al., 2016); generating realistic 3D scenes with varying numbers of objects (Yeh et al., 2012); "inverse graphics" approaches to breaking CAPTCHAs (Mansinghka et al., 2013) and inferring 3D scenes from 2D data (Kulkarni et al., 2015; Zinberg et al., 2019); and simultaneous deduplication, cleaning, and record linkage from real-world databases with millions of records (Lew et al., 2020). Because the state space in OUPMs has a priori unknown dimension, popular black-box inference algorithms such as Hamiltonian Monte Carlo cannot be applied. Instead, applications rely on custom MCMC kernels. These kernels use application-specific, data-driven heuristics to intelligently propose objects to delete, split, or merge, decide when to "birth" new objects, and propose modifications to the properties of and relationships between objects. Although these proposals can be designed and justified in the reversible-jump MCMC framework, this is often challenging. Brooks et al. (2003) noted "the application of reversible jump...has predominantly remained within the domain of the MCMC expert," due to the difficulty of deriving and implementing effective RJMCMC kernels.

This abstract introduces a new framework for OUPM inference that makes it easier to write custom data-driven kernels. It leverages involutive MCMC, a generalization of many MCMC algorithms including reversible-jump MCMC, that was introduced by Cusumano-Towner et al. (2020) and independently by Neklyudov et al. (2020). This paper shows how to adapt the automated formulation from Cusumano-Towner et al. (2020) to automatically generate scalable implementations for inference in OUPMs. Both probabilistic and differentiable programming techniques play key roles. For example, Jacobian determinants are calculated via reverse-mode automatic differentiation. Other factors in the acceptance ratio are calculated via tracing of stochastic choices and dynamic dependency tracking. Given this automation, users can focus on writing inference programs that propose incremental changes to possible worlds, creating, deleting, or modifying objects according to data-driven, application-specific heuristics. These proposals are automatically converted into stationary MCMC kernels via the automatic computation of the accept/reject step.

We report performance data from preliminary experiments on an auditory scene analysis application from Cusimano et al. (2018) and on Gaussian mixture modeling with an unknown number of components (Richardson and Green, 1997). We show that our approach enables data-driven kernels that are faster than generic PPL inference and also faster than generic birth/death RJMCMC kernels without application-specific customizations.

© A. Authors.

Related Work. We build on involutive MCMC, introduced as an automated technique building on probabilistic and differentiable programming by Cusumano-Towner et al. (2020) and (independently) as a mathematical generalization of many classic MCMC methods by Neklyudov et al. (2020). Our main theoretical contribution is a generalization of involutive MCMC to OUPM inference. We compare our approach to generic PPL inference from BLOG (Milch et al., 2005a); note that the use of application-specific MCMC kernels (Milch and Russell, 2012) in BLOG requires modifying its source code to manually calculate correct acceptance ratios for each custom proposal. The Gen (Cusumano-Towner et al., 2019) and Stochaskell (Roberts et al., 2019) PPLs support forms of automated reversible-jump MCMC. Neither Gen nor Stochaskell have built-in high-level OUPM inference constructs; custom kernels in both languges operate on low-level program traces, not high-level OUPM world representations. Stochaskell automatically generates reverse moves, but supports a more restricted class of kernels that are insufficient for our applications.

2. Open Universe Probabilistic Models

Figure 1 shows an open-universe model for inferring the set of seismic events underlying observed detections from seismic monitoring stations. A priori, it is unknown how many events there are, and which detections correspond to the same event (or, in the case of spurious false positives, to no event at all). Standard probabilistic graphical models are ill-suited for encoding this sort of uncertainty, about the *number* of objects reflected in a dataset and their causal relationships. *Open-universe probabilistic models* enable reasoning about such problems, by defining generative processes over entire relational domains.

```
type Event
type Station
type Detection
property function magnitude(e :: Event)
  return mag ~ exponential(1.0)
end
function is_false_positive(d::Detection)
   return length(origin(d)) == 1
end
property function reading(d :: Detection)
  if is_false_positive(d)
     reading ~ normal(1, 0.5)
   else
     (station, event) = origin(d)
     richter_mag = log_10(get(magnitude(event)))
     reading ~ normal(richter_mag, 0.2)
   end
   return reading
end
number function Station()
 return 2
end
number function Event()
  return num ~ poisson(5)
end
number function Detection(:: Station)
   return num ~ poisson(4)
end
number function Detection(:: Station, :: Event)
  return num ~ Int(bernoulli(0.8))
end

observation_model function detections()
 detections = get_object_set(Detection)
 return [get(reading(d)) for d in detections]
end
```

Figure 1: A simple OUPM for seismic monitoring, inspired by Arora et al. (2013), in which `Detections` are received at seismic monitoring `Stations`. Each `Station` receives some number of false-positive `Detections`, and for each `Event`, each `Station` has probability 0.8 of detecting it. Each `Event` has a `magnitude`, and a seismic `reading` is recorded for each `Detection`, which is distributed around the underlying `Event`'s magnitude when the `Detection` is not a false-positive.

Model specification. An OUPM is fully specified by: (1) a finite set $\mathcal{T}$ of object types (in Figure 1, `Event`, `Station`, and `Detection`); (2) a finite set $\mathcal{N}_\tau$ of *possible origins* for each object type τ (in Figure 1, a `Detection` may originate from a (`Station`, `Event`) pair, or from a single (`Station`) that records a false positive); (3) a countable set $\mathcal{P}$ of typed *object properties* $P(\tau_1, \ldots, \tau_n)$ (such as `magnitude(Event)` and `reading(Detection)`); and (4) a *contingent Bayesian network* $\mathcal{C}$ over a set of random variables that encode the number of objects that exist, as well as their properties and relationships (Milch et al., 2005b).

Contingent Bayesian networks. The probabilistic structure of an OUPM is given by a *contingent Bayesian network* $\mathcal{C}$ over an infinite set $\mathcal{V}$ of *possible variables*. These variables are defined in terms of *objects*: we write $\tau((o_1, \ldots, o_n), i)$ to refer to the ith object of type τ with origin $(o_1, \ldots, o_n)$. Then each node $v \in \mathcal{V}$ of the contingent Bayesian network corresponds to either a *number variable* $N_\tau(o_1, \ldots, o_n)$, which represents the number of objects of type τ with origin $(o_1, \ldots, o_n)$, or to a *property variable* $P(o_1, \ldots, o_n)$, which represents the value of the property P for the given tuple of objects. Like an ordinary Bayesian network, a contingent Bayesian network uses a directed graph to describe the probabilistic dependencies among variables. However, the edges in a CBN's graph are contingent: they are labeled with predicates, involving the other possible variables, determining the conditions under which they are *active*. The conditional distribution for a variable v, $p_v(v \mid Pa_\omega(v))$, may depend only on v's *active* parents $Pa_\omega(v)$ in a particular possible world ω.

Induced probability distribution over minimal self-supporting instantiations. Fixing a set $U \subseteq \mathcal{V}$ of *observed* variables (in Figure 1, {`detections()`}), the CBN $\mathcal{C}$ can be used to define a probability distribution over *minimal self-supporting instantiations* (MSSIs) for U: *finite* assignments w to a *subset* of variables $\text{vars}(w) \subseteq \mathcal{V}$, where $U \subseteq \text{vars}(w)$, such that (1) w is *self-supporting*—if $v \in \text{vars}(w)$ and $u \in Pa_w(v)$ then $u \in \text{vars}(w)$; and (2) w is *minimal*—if any variable $v \notin U$ were removed from w it would no longer be self-supporting. Let $\mathcal{W}$ be the set of all possible minimal self-supporting instantiations for a fixed set U of observed variables, and let $\text{vars}(\mathcal{W}) = \{\text{vars}(w) \mid w \in \mathcal{W}\}$ be the set of possible sets-of-variables that can be jointly specified by a minimal self-supporting instantiation. For $\mathbf{v} \in \text{vars}(\mathcal{W})$, let $\mathbf{v}_d$ be the set of all possible joint assignments to the *discrete* variables in $\mathbf{v}$. Then we can define a reference measure μ over the space $\mathcal{W}$ of MSSIs, given by

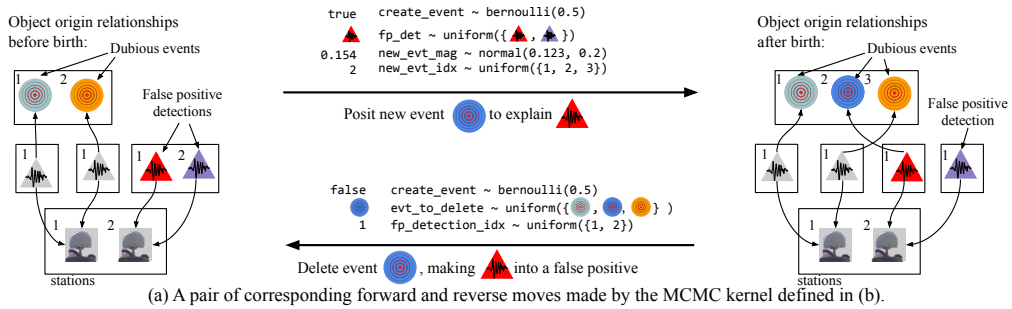
$$\mu(E) = \sum_{\mathbf{v} \in \text{vars}(\mathcal{W})} \sum_{w_d \in \mathbf{v}_d} \lambda^{|\mathbf{v}| - |w_d|}(\{w'_c \mid w' \in E \wedge \text{vars}(w') = \mathbf{v} \wedge \forall v \in \text{vars}(w_d), w'[v] = w_d[v]\}),$$

where w_c is the vector of values of the continuous variables in an MSSI w, and λ^k is the Lebesgue measure on $\mathbb{R}^k$. With respect to μ, the probability density for the OUPM, as a distribution over MSSIs, is given by $p(w) = \prod_{v \in \text{vars}(w)} p_v(w[v] \mid \{w[u] \mid u \in Pa_w(v)\})$.[1]

3. Involutive MCMC for Open Universe Models

An involutive MCMC kernel (Cusumano-Towner et al., 2020; Neklyudov et al., 2020) for a target density p on a measure space (X, Σ_X, μ_X) uses a three-step process to transform a

1. Milch et al. (2005b)'s formulation of contingent Bayesian networks does not support continuous random variables. See Appendix B for a more careful development of CBNs, and an extension (to our knowledge, novel) to the continuous-variable case.



(a) A pair of corresponding forward and reverse moves made by the MCMC kernel defined in (b).

```
is_detection_of(e, d) = !is_false_positive(d) && origin(d)[2] == e
is_dubious(e, ds) = sum(is_detection_of(e, d) for d in ds) == 1
```

```
gen function birth_death_proposal(prev_world)
  dets, events, stations = [get_object_set(prev_world, T)
                            for T in [Detection, Event, Station]]

  # decide whether to create or delete an event
  false_positives = filter(is_false_positive, dets)
  dubious_events = filter(e -> is_dubious_event(e, dets), events)
  create_prob = isempty(false_positives) ? 0.0 :
                    isempty(dubious_events) ? 1.0 : 0.5
  create_event ~ bernoulli(create_prob)

  # make additional choices based on move type
  if create_event
    fp_det      ~ uniform(false_positives)
    # what would the detection of the new event be with no noise?:
    noiseless_det ~ normal(get(prev_world, :reading, fp_det), 0.2)
    new_evt_idx ~ uniform(range(1, length(events) + 1))
  else
    evt_to_delete ~ uniform(dubious_events)
    detection = find(d -> is_detection_of(evt_to_delete, d), dets)
    num_fps = sum(first(origin(d)) == first(origin(detection))
                  for d in fp_detections))
    fp_det_idx ~ uniform(range(1, num_fps + 1))
  end
end
```

```
involution function birth_death_involution()
  set backward[:create_event] = !proposed[:create_event]

  if proposed[:create_event]
    # create the new dubious event
    new_event = create Event with index proposed[:new_evt_idx]
    new_mag = 10^(proposed[:noiseless_det]) # Richter -> amplitude
    set magnitude(new_event)[:mag] = new_mag

    # change the false positive to be a true detection of the new event
    station,   = origin(proposed[:fp_det])
    change proposed[:fp_det] to origin (station, new_event) index 1

    # fill in details of the backward move
    set backward[:evt_to_delete] = new_event
    set backward[:fp_detection_idx] = index(proposed[:fp_detection])
  else
    # delete the dubious event
    dubious_evt = proposed[:evt_to_delete]
    delete dubious_evt

    # find detection of this event and change to a false positive
    dets = get_object_set(Detection)
    det  = find(d -> is_detection_of_event(dubious_evt, d), dets)
    station = (origin(det))[1]
    new_fp = change det to origin (station,) index proposed[:fp_det_idx]

    # fill in details of backward move
    set backward[:fp_det] = new_fp
    set backward[:new_evt_mag] = get(magnitude(dubious_evt))
    set backward[:new_evt_idx] = index(proposed[:evt_to_delete])
  end
end
```

(b) An involutive MCMC kernel consisting of a proposal generative function and an involution.

Figure 2: (a) An execution of a MCMC kernel transforming the state of the seismic monitoring model on the left to that on the right. The move, with random choices shown above the right arrow, posits a new event to explain a false positive detection. The involution sets the new event's magnitude to 10^r, converting Richter-scale value r to an event amplitude, so the acceptance probability's $(\frac{d(\mu \circ f^{-1})}{d\mu}(x, y))$ term is $\log(10)10^r$; our system calculates this via automatic differentiation. The kernel can also make the reverse move ($\leftarrow$). (b) The involutive MCMC kernel visualized in (a).

state x into a next state x'. First, auxiliary variables y are sampled from a state-dependent proposal distribution with density q_x, defined over a measure space (Y, Σ_Y, μ_Y). Second, an *involution*[2] f is applied to (x, y) to obtain (x', y'): x' is the proposed next state, and y' is an auxiliary variable value that would cause the kernel to propose the old state x from the new state x' (since $f(x', y') = (x, y)$). Third, an acceptance probability $\alpha(x, y, x', y')$ is computed. With probability α, x' is the Markov chain's next state; otherwise the state x is repeated. Cusumano-Towner et al. (2020) show that this process yields a kernel stationary for p when

$$\alpha(x, y, x', y') = \frac{p(x')q_{x'}(y')}{p(x)q_x(y)} \times \left(\frac{d(\mu \circ f^{-1})}{d\mu}(x, y)\right), \tag{1}$$

where $\frac{d(\mu \circ f^{-1})}{d\mu}$ is the Radon-Nikodym derivative of the pushforward of $\mu = \mu_X \times \mu_Y$ through f with respect to μ. In general, this does not yield a constructive procedure for *computing*

2. An involution is a function that is its own inverse: $f(f(x, y)) = (x, y)$.

this Radon-Nikodym derivative, but Cusumano-Towner et al. (2020) show that when p and q are distributions over a space $\mathcal{D}$ of finite *dictionaries*, i.e., finite sets of key-value pairs, the Radon-Nikodym derivative can be computed as the determinant of a Jacobian matrix. It turns out (Theorem 1) that a similar result holds when p is an open-universe model. Suppose f is an involution on the space $\mathcal{W} \times \mathcal{D}$, and that there is a countable partition $\{C_i\}$ of $\mathcal{W} \times \mathcal{D}$ such that: (i) $(x, y) \in C_i \implies f(x, y) \in C_i$, (ii) for all $(x, y) \in C_i$, the total number of continuous values in minimal self-supporting instantiation x, plus the number of continuous values in dictionary y, is constant, and (iii) for each C_i, for $(x, y) \in C_i$, there is a differentiable bijection $h_i : \mathbb{R}^{k_i} \to \mathbb{R}^{k_i}$ mapping the continuous values in x and y to continuous values in x' and y' (where $(x', y') = f(x, y)$). Then, we have:

Theorem 1 *For any open universe model with density p, any collection of distributions over finite dictionaries with densities q_x, and any involution satisfying the above conditions, involutive MCMC with $\alpha(x, y, x', y') = \frac{p(x')q_{x'}(y')}{p(x)q_x(y)} \times |\det(Jh_i(x, y))|$ is stationary for p, where $Jh_i(x, y)$ is the Jacobian of h_i at the continuous variables of (x, y), and i is s.t. $(x, y) \in C_i$.*

The proof involves a translation to the finite dictionary case, and exploits the structure of minimal self-supporting instantiations to guarantee the necessary properties of f.

4. Automation of Acceptance Probability via PPLs and AD

Algorithm 1 Automated Involutive MCMC for OUPMs

procedure OUPM-IMCMC(p, q, f, x)
  $(y, q_x(y)) \leftarrow$ SampleAndScore(q, x)
  $(U, M, y', J) \leftarrow$ ProcessInvolution$(f, (x, y))$
  $(x', \frac{p(x')}{p(x)}) \leftarrow$ UpdateWorld(x, U, M)
  $q_{x'}(y') \leftarrow$ Score(q, x', y')
  with probability $\frac{p(x')}{p(x)} \times q_{x'}(y') \times \frac{1}{q_x(y)} \times J$ **return** x' **else return** x
end procedure
procedure ProcessInvolution$(f, (x, y))$
  ▷ Track continuous reads and writes, manipulation moves M, and property updates U
  $(\mathrm{Rd}, \mathrm{Wr}, M, U, y') \leftarrow \{\}, \{\}, [], \{\}, \{\}$
  Execute f, but with
    each manipulation command m (`create`, `delete`, `change`, `split`, `merge`) $\equiv (M \leftarrow M \cup \{m\})$
    `"proposed[`k`]"` $\equiv$ (**if** $y[k]$ is continuous: $\mathrm{Rd} \leftarrow \mathrm{Rd} \cup \{y[k]\}$; $y[k]$)
    `"set backward[`k`] =` v`"` $\equiv$ (**if** v is continuous: $\mathrm{Wr} \leftarrow \mathrm{Wr} \cup \{y[k]\}$; $y' \leftarrow y' \cup \{k \mapsto v\}$)
    `"get(`$P(o_1, \ldots, o_n)$`)"` $\equiv (v \leftarrow x[P(o_1, \ldots, o_n)]$; **if** v is continuous: $\mathrm{Rd} \leftarrow \mathrm{Rd} \cup \{v\}$; $v)$
    `"set` $P(o_1, \ldots, o_n) = v$`"` $\equiv$ (**if** v is continuous: $\mathrm{Wr} \leftarrow \mathrm{Wr} \cup \{v\}$; $U[P(o_1, \ldots, o_n)] \leftarrow v$)
  $J \leftarrow$ uninitialized $|\mathrm{Rd}| \times |\mathrm{Rd}|$ matrix
  for i in $\{1, \ldots, |\mathrm{Wr}|\}$:
    ▷ Execute f with reverse-mode AD to compute $\frac{\partial \mathrm{Wr}[i]}{\partial \mathrm{Rd}[j]}$ for each $j \in \{1, \ldots, |\mathrm{Rd}|\}$
    $J[:, i] \leftarrow \nabla_{\mathrm{Rd}}(\mathrm{Wr}[i])$
  return $(U, M, y', |\det J|)$
end procedure

We have extended the Gen probabilistic programming system (Cusumano-Towner et al., 2019) with new constructs for OUPMs (sample code in Figure 1) and automated involutive MCMC for these models (Figure 2, Algorithm 1). The contributions are: (1) a new modeling language for open-universe models, capable of efficiently simulating MSSIs, evaluating densities, and modifying world states; (2) a new DSL for defining object-based involutive MCMC proposals (Appendix C); and (3) an *automated* implementation of involutive MCMC for open-universe models, which automatically acceptance ratios via probabilistic and differentiable programming techniques (Algorithm 1). See Appendix D for more details.

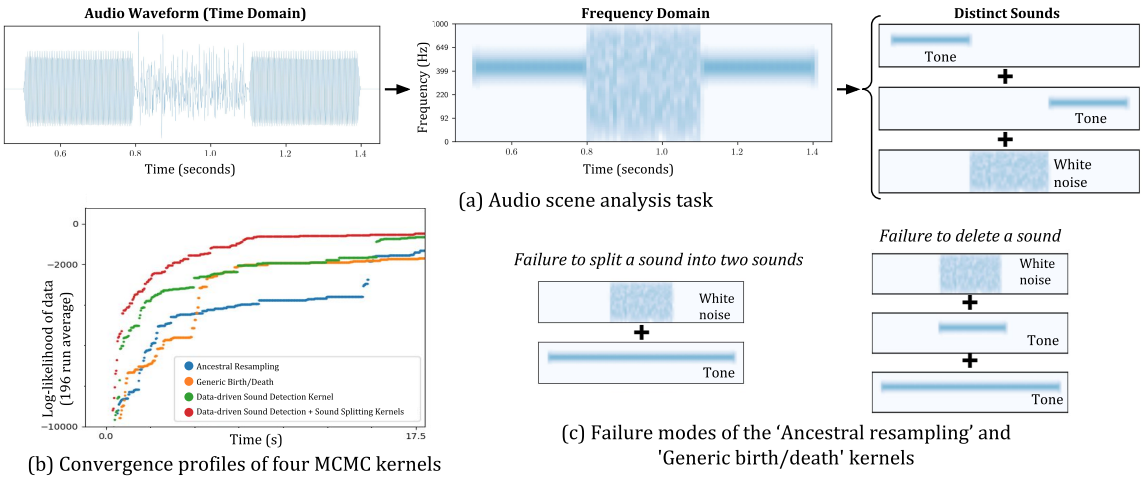


Figure 3: An OUPM for auditory perception. (a) A waveform (left) is observed as a spectrogram in the frequency domain (center), from which we infer the number and type of constituent sound sources that together explain the observed data (right). (b) Convergence profiles of 4 MCMC kernels; the two data-driven iMCMC kernels outperform generic PPL and generic RJMCMC kernels. (c) Approximate posterior samples reflecting typical failure modes of non-data-driven kernels.

5. Experiments

Auditory scene analysis. We implemented a variant of Cusimano et al. (2018)'s model of human auditory scene perception, which is based on an OUPM of sound sources with properties including amplitude, type, pitch, and duration. See Figure 3 for an example problem, and Appendix A.2 for details. We implemented two custom, data-driven kernels: a **data-driven sound detection kernel** (Figure 10) that applies image-filtering and edge-detection to frequency-domain spectrograms and proposes new audio sources to explain sounds it detects in the data but not in the currently-hypothesized world, and a **data-driven sound splitting kernel** (Figure 11) that splits long sounds into shorter ones at endpoints detected in the observed soundwaves. We also implemented two baseline inference strategies: an **ancestral resampling** kernel (Figure 8) based on generic PPL inference in BLOG; and a **generic birth/death** RJMCMC kernel (Figure 9) that proposes to create a sound with parameters generated from the prior, or delete a sound. Figure Figure 3(b) shows quantitative performance comparisons indicating superior performance for custom data-driven kernels. Common failure modes of the generic kernels are illustrated in Figure 3(c).

Open-universe Gaussian mixture modeling. We also implemented Richardson and Green (1997)'s data-driven inference algorithm for clustering in a Gaussian mixture model with an unknown number of components. See Appendix A.1 for details, including an illustration in Figure 4. We have found that BLOG inference typically fails to accept moves to world states with different numbers of clusters, whereas the data-driven algorithm rapidly mixes. Note that Richardson and Green (1997)'s algorithm includes a split-merge kernel with a complex Jacobian term that our approach calculates automatically.

References

Nimar Arora, Stuart Russell, and Erik Sudderth. Net-visa: Network processing vertically integrated seismic analysis. *The Bulletin of the Seismological Society of America*, 103: 709–729, 04 2013. doi: 10.1785/0120120107.

Nimar S Arora. *Model-based Bayesian Seismic Monitoring*. PhD thesis, EECS Department, University of California, Berkeley, May 2012. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-125.html.

Albert S Bregman. *Auditory scene analysis: The perceptual organization of sound*. MIT press, 1994.

Stephen Brooks, Paolo Giudici, and G.O. Roberts. Efficient construction of reversible jump proposal distributions (with discussion). *J. R. Statist. Soc. B*, 65:3–55, 01 2003.

Maddie Cusimano, Luke B Hewitt, Josh Tenenbaum, and Josh H McDermott. Auditory scene analysis as bayesian inference in sound source models. In *CogSci*, 2018.

Marco Cusumano-Towner, Alexander K. Lew, and Vikash K. Mansinghka. Automating involutive mcmc using probabilistic and differentiable programming. 2020.

Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 221–236. ACM, 2019.

T. D. Kulkarni, P. Kohli, J. B. Tenenbaum, and V. Mansinghka. Picture: A probabilistic programming language for scene perception. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4390–4399, 2015. doi: 10.1109/CVPR.2015.7299068.

Alexander K. Lew, Monica Agrawal, David Sontag, and Vikash K. Mansinghka. Pclean: Bayesian data cleaning at scale with domain-specific probabilistic programming, 2020.

Vikash K Mansinghka, Tejas D Kulkarni, Yura N Perov, and Josh Tenenbaum. Approximate bayesian image interpretation using generative probabilistic graphics programs. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 1520–1528. Curran Associates, Inc., 2013. URL http://papers.nips.cc/paper/4881-approximate-bayesian-image-interpretation-using-generative-probabilistic-graphics-pr pdf.

Brian Milch and Stuart J. Russell. General-purpose MCMC inference over relational structures. *CoRR*, abs/1206.6849, 2012. URL http://arxiv.org/abs/1206.6849.

Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG: Probabilistic models with unknown objects. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, IJCAI 2005, pages 1352–1359. Morgan Kaufmann Publishers Inc., 2005a.

Brian Milch, Bhaskara Marthi, David Sontag, Stuart Russell, Daniel L. Ong, and Andrey Kolobov. Approximate inference for infinite contingent bayesian networks. In *(AISTATS'05) 10th International Workshop on Artificial Intelligence and Statistics*, January 2005b. URL https://www.microsoft.com/en-us/research/publication/approximate-inference-infinite-contingent-bayesian-networks/.

Brian Christopher Milch. *Probabilistic Models with Unknown Objects*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2006. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-174.html.

Kirill Neklyudov, Max Welling, Evgenii Egorov, and Dmitry Vetrov. Involutive MCMC: A Unifying Framework. *arXiv preprint arXiv:2006.16653*, 2020.

Sylvia. Richardson and Peter J. Green. On bayesian analysis of mixtures with an unknown number of components (with discussion). *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 59(4):731–792, 1997. doi: 10.1111/1467-9868.00095. URL https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/1467-9868.00095.

David A. Roberts, Marcus Gallagher, and Thomas Taimre. Reversible jump probabilistic programming. volume 89 of *Proceedings of Machine Learning Research*, pages 634–643. PMLR, 16–18 Apr 2019. URL http://proceedings.mlr.press/v89/roberts19a.html.

S. Russell, O. Lassen, J. Uang, and W. Wang. The physics of text: Ontological realism in information extraction. In *AKBC@NAACL-HLT*, 2016.

Wei Wang and Stuart Russell. A smart-dumb/dumb-smart algorithm for efficient split-merge mcmc. In *Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence*, UAI'15, page 902–911, Arlington, Virginia, USA, 2015. AUAI Press. ISBN 9780996643108.

Yi-Ting Yeh, Lingfeng Yang, Matthew Watson, Noah D. Goodman, and Pat Hanrahan. Synthesizing open worlds with constraints using locally annealed reversible jump mcmc. *ACM Trans. Graph.*, 31(4), July 2012. ISSN 0730-0301. doi: 10.1145/2185520.2185552. URL https://doi.org/10.1145/2185520.2185552.

Ben Zinberg, Marco Cusumano-Towner, and Vikash K. Mansinghka. Structured differentiable models of 3D scenes via generative scene graphs. In *Workshop on Perception as Generative Reasoning*, 2019. URL https://pgr-workshop.github.io/img/PGR025.pdf.

Appendix A. Experimental Details

A.1. Open-universe Gaussian Mixture Model.

Richardson and Green (1997) develop a series of highly data-driven MCMC kernels for a Gaussian mixture model with a unknown number of mixture components. Here we compare their algorithm, implemented using our system for involutive MCMC in OUPMs, to the generic inference engine in BLOG.

The model is as follows. The number of mixture components, k, is a Poisson random variable. The mixture weights and the parameters of each mixture component are drawn from the appropriate conjugate priors:

$$w_1, \ldots, w_k \sim \text{Dirichlet}(\delta \mathbf{1}_k) \qquad \mu_j \sim \mathcal{N}(\xi, 1/\kappa) \qquad \sigma_j^2 \sim \text{inverse-gamma}(\alpha, \beta),$$

where $\mathbf{1}_k$ denotes a vector of ones in $\mathbb{R}^k$; w_j, μ_j, and σ_j^2 denote the weight, mean, and variance of the j-th mixture component respectively; and α, β, δ, κ, and ξ are fixed hyper-parameters. Lastly, for the i-th data point we sample a cluster allocation, and given the cluster allocation, we sample a value from the assigned cluster; ie.

$$z_i \sim \text{Categorical}(w_1, \ldots, w_k) \qquad y_i \sim \mathcal{N}(\mu_{z_i}, \sigma_{z_i}^2)$$

Our involutive MCMC proposal consists of first updating the weights, means, variances, and allocations using Gibbs updates, and then performing a split or merge move to change the number of clusters. The Gibbs updates are given by

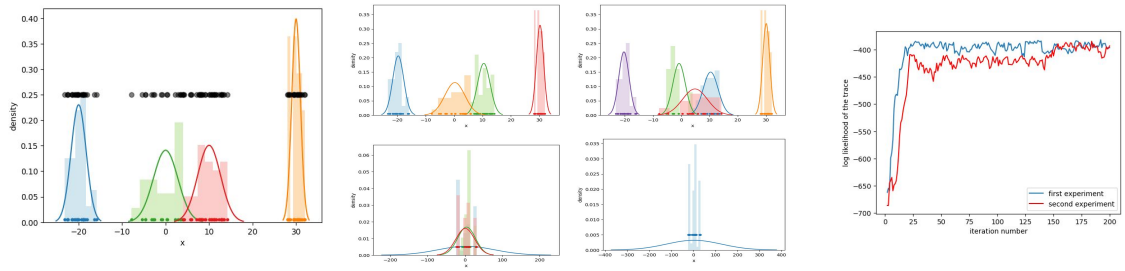
$$\begin{aligned}
w_1, \ldots, w_k | \ldots &\sim \text{Dirichlet}(w_1 + n_1, \ldots, w_k + n_k) \\
\mu_j | \ldots &\sim \mathcal{N}\left(\frac{\kappa\xi + \sigma_j^{-2} \sum_{z_i = j} y_i}{\sigma_j^{-2} n_j + \kappa}, \left(\sigma_j^{-2} n_j + \kappa\right)^{-1} \right) \\
\sigma_j^2 | \ldots &\sim \text{inverse-gamma}(\alpha + \frac{1}{2} n_j, \beta + \sum_{z_i = j} (y_i - \mu_i)^2) \\
p(z_i = j | \ldots) &\propto \mathcal{N}(\mu_j, \sigma_j^2),
\end{aligned}$$

where n_j denotes the number of data points allocated to the j-th cluster. Our merge moves consist of choosing indices $j_1 \neq j_2$ and j_* uniformly at random and merging the j_1-th and j_2-th clusters into the j_*-th cluster. We ensure that the zeroth, first, and second moments of the newly generated cluster match the corresponding moments in the mixture of the original cluster; i.e.

$$\begin{aligned}
w_* &= w_1 + w_2 \\
w_* \mu_* &= w_1 \mu_1 + w_2 \mu_2 \\
w_* (\mu_* + \sigma_*^2) &= w_1 (\mu_1 + \sigma_1^2) + w_2 (\mu_2 + \sigma_2^2).
\end{aligned} \tag{2}$$

To ensure ergodicity, our split procedure relies on three auxiliary variables

$$u_1 \sim \text{beta}(2, 2) \qquad u_2 \sim \text{beta}(2, 2) \qquad u_3 \sim \text{beta}(1, 1),$$



(a) Ground Truth Clusterings (colored) and Data (black)

(b) Two Samples from the Inferred Posterior: Richardson & Green's Data-driven MCMC (top), BLOG Ancestral Sampling (bottom)

(c) Trace Likelihoods vs. Sample number for Richardson & Green's algorithm

Figure 4: Gaussian Mixture Model Inference Experiment Results

and generates new clusters according to

$$
\begin{aligned}
w_1 &= w_* u_1 & w_2 &= w_*(1-u_1) \\
\mu_1 &= \mu_* - u_2 \sigma_* \sqrt{w_2/w_1} & \mu_2 &= \mu_* + u_2 \sigma_* \sqrt{w_1/w_2} \\
\sigma_1^2 &= u_3(1-u_2^2)\sigma_*^2 w_*/w_1 & \sigma_2^2 &= (1-u_3)(1-u_2^2)\sigma_*^2 w_*/w_2.
\end{aligned} \tag{3}
$$

Using (2) and (3) we can derive that our involution's Jacobian matrix (leaving out rows and columns which merely copy a value) is

$$
J = \begin{array}{c} \\ w_1 \\ \mu_1 \\ \sigma_1^2 \\ w_2 \\ \mu_2 \\ \sigma_2^2 \end{array}
\begin{array}{c}
\begin{array}{cccccc} \mu_* & \sigma_*^2 & w_* & u_1 & u_2 & u_3 \end{array} \\
\begin{bmatrix}
0 & 0 & u_1 & w_* & 0 & 0 \\
1 & -\frac{u_2\sqrt{\frac{w_2}{w_1}}}{2\sigma_*} & 0 & -\frac{u_2\sigma_*\sqrt{-\frac{u_1-1}{u_1}}}{2u_1(u_1-1)} & -\sigma_*\sqrt{\frac{w_2}{w_1}} & 0 \\
0 & -\frac{u_3 w_*(u_2^2-1)}{w_1} & 0 & \frac{u_3\sigma_*^2(u_2^2-1)}{u_1^2} & -\frac{2u_2u_3w_*\sigma_*^2}{w_1} & -\frac{w_*\sigma_*^2(u_2^2-1)}{w_1} \\
0 & 0 & 1-u_1 & -w_* & 0 & 0 \\
1 & \frac{u_2\sqrt{\frac{w_1}{w_2}}}{2\sigma_*} & 0 & -\frac{u_2\sigma_*\sqrt{-\frac{u_1}{u_1-1}}}{2u_1(u_1-1)} & \sigma_*\sqrt{\frac{w_1}{w_2}} & 0 \\
0 & \frac{w_*(u_2^2u_3-u_2^2-u_3+1)}{w_2} & 0 & \frac{\sigma_*^2(u_2^2u_3-u_2^2-u_3+1)}{(u_1-1)^2} & \frac{2u_2w_*\sigma_*^2(u_3-1)}{w_2} & \frac{w_*\sigma_*^2(u_2^2-1)}{w_2}
\end{bmatrix}
\end{array}
$$

Given a specification of the inference kernel, our system automatically calculates this Jacobian matrix and its determinant

$$
|\det J| = \frac{w_*|\mu_1-\mu_2|\sigma_1^2\sigma_2^2}{u_2(1-u_2^2)u_3(1-u_3)\sigma_*^2}.
$$

without the user needing to work through this derivation.

Figure 4 shows representative results, in which the involutive MCMC proposal accurately recovers the true latent clusters, whereas the ancestral sampling proposal only roughly narrows down the component parameters.

```
type Cluster
type Datapoint
number Cluster() ~ poisson(λ) + 1
number Datapoint() = NUM_DATAPOINTS # global constant
property mean(::Cluster) ~ normal(ξ,1/κ)
property var(::Cluster) ~ inverse_gamma(α_v, β_v)
property unnormalized_weight(::Cluster) ~ gamma(α_w, β_w)

property function cluster(::Datapoint)
  clust_to_un_weight = Dict(c => get(unnormalized_weight(c))
                            for c in get_object_set(Cluster()))
  cluster ~ unnormalized_categorical(clust_to_un_weight)
  return cluster
end
property function value(d::Datapoint)
  cluster = get(cluster(d))
  value ~ normal(get(mean(cluster)), get(var(cluster)))
  return value
end

observation_model function get_datapoints()
  return [get(value(Datapoint with index i))
                            for i=1:get_number(Datapoint())]
end
```

Figure 5: Gaussian mixture model with an unknown number of components. Instead of imperatively sampling cluster weights $w_1, \ldots, w_n \sim \text{Dirichlet}(\delta \mathbf{1}_k)$, in our model, each cluster has an *unnormalized cluster weight* property sampled from a Gamma distribution; normalizing these then sampling clusters from the resulting distribution is equivalent to sampling clusters according to weights sampled from a Dirichlet.

```
gen function splitmerge_proposal(prev_world)
  prob_merge = get_number(prev_world, Cluster()) <= 1 ? 0.0 : 0.5
  do_merge ~ bernoulli(prob_merge)
  if is_merge
    {*} ~ merge_proposal(prev_world)
  else
    {*} ~ split_proposal(prev_world)
  end
end
gen function merge_proposal(prev_world)
  clusters = get_object_set(prev_world, Cluster())
  merge1 ~ uniform_choice(clusters)
  merge2 ~ uniform_from_list([c for c in clusters if c != merge1])
  new_idx ~ uniform_discrete(1, length(clusters) - 1)
end
gen function split_proposal(prev_world)
  clusters = get_object_set(prev_world, Cluster())
  to_split ~ uniform_choice(clusters)
  idx1 ~ uniform_discrete(1, length(clusters) + 1)
  idx2 ~ uniform_from_list([i for i=1:(length(clusters)+1) if i != idx1])
  u1 ~ beta(2, 2)
  u2 ~ beta(2, 2)
  u3 ~ beta(1, 1)
  mean, var = get(prev_world, :mean, to_split), get(prev_world, :var, to_split)
  mean1, mean2 = mean - u2*sqrt(var * (1-u1)/u1), mean + u2*sqrt(var+u1/(1-u1))
  var1, var2 = u3*(1 - u2^2)*var/u1, (1 - u3)*(1-u2^2)*var/(1-u1)

  for dp in get_object_set(prev_world, Datapoint())
    if get(prev_world, :cluster, dp) == to_split
      y = get(prev_world, :value, dp)
      p = u1 * exp(logpdf(gaussian, y, mean1, var1))
      q = (1-u1) * exp(logpdf(gaussian, y, mean2, var2))
      {:to_first => dp} ~ bernoulli(p/(p+q))
    end
  end
end
```

```
function merged_component_params(w1, mean1, var1, w2, mean2, var2)
   w = w1 + w2
   mean = (w1*mean1 + w2*mean2) / w
   var = -mean^2 + (w1*(mean1^2 + var1) + w2*(mean2^2 + var2)) / w
   return (w, mean, var)
end
function reverse_split_params(w, mean, var, w1, mean1, var1, w2, mean2, var2)
   u1 = w1/w
   u2 = (mean - mean1) / sqrt(var * w2/w1)
   u3 = var1/var * u1 / (1 - u2^2)
   Return (u1, u2, u3)
end
```

```
involution function splitmerge_involution()
  backward[:do_merge] = !proposed[:do_merge]
  if proposed[:do_merge]
    c1, c2 = proposed[:merge1], proposed[:merge2]
    new = merge c1, c2 to index proposed[:new_idx]
    backward[:idx1], backward[:idx2] = index(c1), index(c2)
    backward[:to_split] = new

    mean1, var1, w1 = get(mean(c1)), get(var(c1)), get(unnormalized_weight(c1))
    mean2, var2, w2 = get(mean(c2))), get(var(c2)) get(unnormalized_weight(c2))

    unnorm_weight, mean, var = merged_component_params(mean1, var1, w1,
                                                       mean2, var2, w2)
    set unnormalized_weight(new) = unnorm_weight
    set mean(new) = mean; set var(new) = var

    u1, u2, u3 = reverse_split_params(unnorm_weight, mean, var,
                                      w1, mean1, var1, w2, mean2, var2)
    backward[:u1], backward[:u2], backward[:u3] = u1, u2, u3

    # reassign datapoints to merged cluster
    for dp in get_object_set(Datapoint())
      if get(cluster(dp)) in [c1, c2]
        set cluster(dp) = new
        backward[:to_first => dp] = (get(cluster(dp)) == c1) ? true : false
      end
    end
  else
    old = backward[:to_split]
    new1, new2 = split old to indices proposed[:idx1], proposed[:idx2]
    backward[:merge1], backward[:merge2] = new1, new2
    backward[:new_idx] = index(old)

    u1, u2, u3 = proposed[:u1], proposed[:u2], proposed[:u3]
    mean, var = get(mean(old)), get(var(old))
    mean1, mean2 = mean - u2*sqrt(var * (1-u1)/u1), mean + u2*sqrt(var+u1/(1-u1))
    var1, var2 = u3*(1 - u2^2)*var/u1, (1 - u3)*(1-u2^2)*var/(1-u1)
    set mean(new1) = mean1
    set var(new1) = var1
    set mean(new2) = mean2
    set var(new3) = var2
    un_weight = get(unnormalized_weight(old))
    set unnormalized_weight(new1) = un_weight * u1
    set unnormalized_weight(new3) = un_weight * (1 - u1)

    # reassign datapoints to split clusters
    for dp in get_object_set(Datapoint())
      if get(cluster(dp)) == old
        set cluster(dp) = backward[:to_first => dp] ? new1 : new2
      end
    end
  end
end
```

Figure 6: Data-driven Gaussian mixture model split/merge kernel from Richardson and Green (1997).

A.2. Auditory Scene Interpretation.

Auditory scene analysis Bregman (1994) is a perception task that involves separating an audio waveform into individual sound sources when the number and properties of the sources are unknown a-priori (Figure 3a). Recently, Cusimano et al. (2018) proposed a computational model of human auditory scene analysis based on an open-universe probabilistic model of sound sources, and showed that probabilistic inference in this model based on MCMC is able to qualitatively reproduce human inferences about auditory scenes, including reporting ambiguity in the same situations as humans.

The model. We developed a simplified version of the model of Cusimano et al. (2018) that uses a prior distribution on the number of sounds and the type of each sound ('tone' or 'white noise'). Tones have an unknown frequency, onset time, and duration, and white noise sounds have an unknown amplitude, onset time, and duration.

```
type Source
number Source() ~ uniform_discrete(1, 4)
property is_noise(::Source) ~ bernoulli(0.4)
property freq_in_erb(::Source) ~ uniform(0.4, 37)
property amplitude(::Source) ~ normal(8.0, 10.0)
property onset(::Source) ~ uniform(0.1, SCENE_LENGTH)
property duration(::Source) ~ uniform(0.1, SCENE_LENGTH)
property function soundwaves(s::Source)
  ons = get(onset(s)); dur = get(duration(s))
  if get(is_noise(s))
    return render_noise(get(amplitude(s)), ons, dur)
  else
    return render_tone(get(freq(s)), ons, dur)
  end
end
observation_model function observed_sound()
  waves = [get(soundwaves(s)) for s in
                        get_object_set(Source))]
  true_sound = sum(waves)
  return observed_sound ~ noisy_matrix(true_sound, 1.0)
end
```

Figure 7: Our OUPM for audio inference.

Inference Experiments. We implemented four involutive MCMC kernels for this model in our extended version of Gen. We performed experiments appplying these kernels to the audio waveform shown in Figure 3a, which consists of a short tone, followed by a short period of white noise, followed by a short tone at the same frequency as the first tone. When the noise is loud, it becomes ambiguous to humans whether there are two short tones, or one continuous tone that is temporarily obscured by loud noise; resultantly, this example is of interest when investigating the behavior of inference programs in ambiguous situations (this was one of the sounds investigated by Cusimano et al. (2018)). The amplitude of the noise we used was low so that humans unambiguously infer two separate tones, and a successful inference program should make the same inference.

Generic Kernels The first two kernels we implemented are not application specific or data driven. Our **ancestral resampling** kernel (Figure 8) resimulates a single variable from the prior distribution (also ancestrally generating any additional variable values which

need to be added to the world due to the resimulated variable). Our **generic birth-death** reversible-jump kernel (Figure 9) randomly proposes to create or delete individual sounds, proposing new sounds' properties from the prior. Figure 3(b) shows that in our experiments, these two kernels converge to inferences of latent waveforms under which the observed sounds are substantially less likely than under the latent waveforms our data-driven kernels recover. Figure 3(c) visualizes two common failure mode these kernels exhibit due to their inability to efficiently incorporate evidence from the data, and their inability to make proposals to split long sounds into short ones (which require application-specific logic to calculate endpoints of the new sounds based on the position of the sound being split). We found that the generic birth-death kernel often failed to exit a local minimum in which a single continuous tone is inferred instead of two temporally separated tones of the same frequency (Figure 3c, left). The generic kernel performs worst—it usually infers one continuous tone, but often also infers spurious tones.

Aside: proposing from the prior in involutive MCMC. To facilitate proposing values from the prior in involutive MCMC kernels, we have implemented a `regenerate` command which can be called from within the involution DSL. (For examples, see Figures 8, 9, and 10.) The command `regenerate properties of object` samples a value from the prior for each property of the given object (using the prior as it will appear in x'). This complicates our picture of involutive MCMC slightly, as there are now random values being sampled during the execution of the involution function, instead of only in the proposal generative function. We thus must view the generative function as only implementing part of the q distribution and sampling part of y, with the rest of q implemented by the involution DSL's `regenerate` commands.
To ensure involutiveness, when a transition $(x, y) \rightarrow (x', y')$ uses a `regenerate` command, the reversing move $(x', y') \rightarrow (x, y)$ must specify a value in y for regeneration. Since users do not specify a dictionary key for the values in y which are regenerated (unlike values in y which are sampled in the proposal generative function), we provide a special syntax, `backward[properties of object] = regenerated`, which automatically sets the regenerated values in y for each property of `object` to equal the property's current value in x. See Section E for more details.

Custom kernels. Some specific domains, like data-clustering, have seen standard MCMC kernels developed for data-driven inference, which are fairly conceptually simple but previously required fairly deriving fairly involved calculations (Richardson and Green, 1997; Wang and Russell, 2015). In addition to facilitating the implementation of these types of kernels and removing the need to manually derive acceptance probabilities, involutive MCMC facilitates the development of data-driven kernels in more complicated domains which involve complicated logic to use information from data effectively. For instance, Cusimano et al. (2018)'s experiments used MCMC inference with a proposal based neural-network transformations of observed soundwaves, implemented in an early version of our inference programming system. For our experiments in this paper, we developed a roughly 200-LOC image-filtering and edge-detection algorithm to analyze observed soundwaves and detect possible errors in the currently inferred latent sounds. We called this sound-analysis program from within 2 involutive MCMC kernels we developed, which our system auto-

```
gen function ancestral_resampling_proposal(prev_world)
  var_to_resample ~ uniform_choice(variables(prev_world))
  if is_number_variable(var_to_resample)
    new_num ~ uniform_discrete(1, 4)
  end
end
```

```
involution function ancestral_resampling_involution()
  backward[:var_to_resample] = proposed[:var_to_resample]
  if is_number_variable(proposed[:var_to_resample])
    set number[Source()] = proposed[:new_num]
    backward[:new_num] = get_number(Source())
    for i=(get_number(Source())+1):proposed[:new_num]
      regenerate properties of (Source with index i)
    end
    for i=(proposed[:new_num]+1):get_number(Source())
      backward[properties of (Source with index i)] = regenerated
    end
  else
    regenerate proposed[:var_to_resample]
    backward[proposed[:var_to_resample]] = regenerated
  end
end
```

Figure 8: Our **ancestral resampling** kernel, which selects a random variable and samples a new value for it from the prior. The only complication is that when we sample the number variable (changing the number of audio sources in the world), we must also generate new properties for any newly created audio sources. (These properties are also generated from the prior.) For details on generating values from the prior in involutive MCMC, see Section E.

```
gen function generic_birthdeath_proposal(prev_world)
  do_birth ~ bernoulli(0.5)
  sources = get_object_set(prev_world, Source)
  if do_birth
    idx ~ uniform_discrete(1, length(sources) + 1)
  else
    to_delete ~ uniform_choice(sources)
  end
end
```

```
involution function generic_birthdeath_involution()
  backward[:do_birth] = !proposed[:do_birth]
  if proposed[:do_birth]
    new = create Source at index proposed[:idx]
    backward[:to_delete] = new
    regenerate properties of new
  end
    old = delete proposed[:to_delete]
    backward[:idx] = index(old)
    backward[properties of old] = regenerated
  end
end
```

Figure 9: The **generic birth/death** kernel either creates or deletes a random sound. When a new sound is created, its properties are all generated from the prior. For details on generating values from the prior in involutive MCMC, see Section E.

mated the implementation of, including efficiently updating world states and calculating the Jacobian correction term.

Our **data-driven sound detection kernel** (Figure 10) either proposes to delete a sound, or to create a new sound which with high probability is proposed at time and frequency where our analysis program detected a sound in the data but not the currently inferred latents. Figure 3(b) shows that this kernel both initially performs better than the generic kernels, and ultimately converges to a better state.

We also add a **data-driven sound splitting kernel** (Figure 11), which either proposes to merge two short tones/noises into a long one, or splits a long sound into two short ones, proposing endpoints for the new sounds based on the analysis of the sounds. Figure 3(b) shows that the addition of this kernel substantially speeds up convergence, which is to be expected since this kernel can integrate information from the data about another class of possible mistakes in our latents: when one sound has been mistakenly interpreted as two, or vice versa.

```
gen function propose_detected_sound_proposal(prev_world)
  num_sounds = get_number(prev_world, Source())
  birth_prior = num_sounds == 4 ? 0. : (num_sounds == 0 ? 1. : 0.5)
  do_birth ~ bernoulli(birth_prior)
  if do_birth
    # use edge-detection and image filtering to analyze the
    # observed sounds and the currently inferred latents;
    # return sounds detected in the observation but not latents
    # represented as DetectedSound objects
    detected_noises_and_tones =
        analyze_observed_and_latent_sounds(prev_world)

    # if we did not detect any sounds, we will just do a generic
    # birth; otherwise, with high probability, do a smart birth
    smartmove_prior = length(detected_noises_and_tones) == 0 ? 0. : 0.95
    do_smart ~ bernoulli(smartmove_prior)

    if do_smart
        {*} ~ propose_smart_birth(prev_world, detected_noise_and_tones)
    end
    new_idx ~ uniform_discrete(1, get_number(prev_world, Source()) + 1)
  else
    # delete a sound
    sound_to_delete ~ uniform_choice(get_object_set(prev_world, Source()))
    {*} ~ sample_reversibility_parameters(prev_world, sound_to_delete)
  end
end

gen function sample_reversibility_parameters(prev_world, sound_to_delete)
  # get the world which will arise after deleting this sound:
  world_without_sound = delete_object(prev_world, sound_to_delete)

  # what will the reversing birth move proposal detect?:
  detected_noises_and_tones =
        analyze_observed_and_latent_sounds(world_without_sound)

  noises = filter(s -> s.is_noise, detected_noises_and_tones)
  tones = filter(s -> !s.is_noise, detected_noises_and_tones)
  if get(prev_world, :is_noise, sound_to_delete)
    bwd_move_detects_this_type = length(noises) > 0
  else
    bwd_move_detects_this_type = length(tones) > 0
  end

  # if the bwd move (y') could birth this, probably it is smart
  rev_smart_prior = bwd_move_detects_this_type ? 0.95 : 0.
  reverse_move_is_smart ~ bernoulli(rev_smart_prior)

  if reverse_move_is_smart
    # in this case, for reversibility, we must select what the reverse move
    # detected to delete the sound we are currently deleting
    if get(prev_world, :is_noise, sound_to_delete)
      possible_rev_selected = noises
    else
      possible_rev_selected = tones
    end
    start = get(prev_world, :starttime, sound_to_delete)
    dur = get(prev_world, :starttime, sound_to_delete)
    overlap_with_region(x, y) = interval_overlap((x, y), (start, start + dur))
    intervals = map(s -> (s.starttime, s.starttime + s.duration),
                                                possible_rev_selected)
    overlaps_plus_1 = map(r -> overlap_with_region(r) + 1, interval)
    reverse_selected ~ list_categorical(normalize(overlaps_plus_1),
                                                possible_rev_selected)
  end
end
```

```
struct DetectedSound
  is_noise::Boolean
  length::Real
  starttime::Real
  mean_freq::Real
  freq_range::Real # range of frequencies this sound is above a certain
volume
  volume::Real
end
function interval_overlap((a, b), (c, d))
  If a > c; return overlap((c, d), (a, b)); end
  If b < c; return 0; end
  If d < b; return d - c; end
  Return c - b
end
score_possible_sound(s) = s.length * s.freq_range * s.volume
```

```
gen function propose_smart_birth(prev_world, detected_noises_and_tones)
  scores = normalize(map(score_possible_sound, detected_noises_and_tones))
  sound_to_birth ~ categorical_from_list(scores, detected_noises_and_tones)

  starttime ~ normal(sound_to_birth.starttime, 0.05)
  duration ~ normal(sound_to_birth.length, 0.05)
  if is_noise(sound_to_birth)
    amplitude ~ normal(sound_to_birth.volume, 4.)
  else
    freq ~ normal(sound_to_birth.freq, 1.)
  end
end
```

```
involution function propose_detected_sound_involution()
  set backward[:do_birth] = !proposed[:do_birth]
  if proposed[:do_birth]
    set backward[:death_idx] = proposed[:new_idx]
    source = create Source with index proposed[:new_idx]
    set backward[:reverse_move_is_smart] = proposed[:is_smart]
    if proposed[:is_smart]
      set backward[:reverse_selected] = proposed[:sound_to_birth]
      set duration(source) to proposed[:duration]
      set onset(source) to proposed[:onset]
      if proposed[:sound_to_birth].is_noise
        set is_noise(source) = true
        set amplitude(source) = proposed[:amplitude]
      else
        set is_noise(source) = false
        set freq_in_erb(source) = proposed[:freq]
      end
    else
      regenerate properties of source
    end
  else
    set backward[:new_idx] = proposed[:death_idx]
    source = delete Source with index proposed[:death_idx]
    set backward[:is_smart] = proposed[:reverse_move_is_smart]
    if proposed[:reverse_move_is_smart]
      set backward[:sound_to_birth] = proposed[:reverse_selected]
      set backward[:onset] = get(onset(source))
      set backward[:duration] = get(duration(source))
      if get(is_noise(source))
        set backward[:amplitude] = get(amplitude(source))
      else
        set backward[:freq] = get(freq(source))
      end
    else
      # specify that the reverse move will regenerate all properties
      # of this source from the prior
      backward[properties of source] = regenerated
    end
  end
end
```

Figure 10: The **data-driven sound detection kernel** inference kernel used in the audio example. This kernel either deletes an existing audio source or creates a new one. When creating a new audio source, it uses an image-filtering edge-detection algorithm `analyze_observed_and_latent_sounds` (not shown) to try to detect noises and tones in the observed sound waves which are not explained by the currently hypothesized latent audio sources. With high probability, the kernel will choose to sample one of the detected sounds, and sample properties for the new sounds which have similar values to detected one. To ensure this birth move is reversible, the death move which deletes a sound must check whether the reversing birth move will detect a sound similar to the sound being deleted. If so, it arranges for the reversing birth move to have sampled this sound. If the death move detects that no tones/noises will be detected during the reversing birth move (e.g. if we are deleting a suprious latent sound not corresponding to anything in the data), it must specify that the reversing birth move should be "dumb" (non-data-driven), and propose all values from the prior, rather than proposing them to be close to the properties of a detected sound in the data.

```
gen function split_at_detected_endpoints_proposal(prev_world)
  sources = get_object_set(prev_world, Source())
  tones = filter(s -> !get(prev_world, :is_noise, s), sources)
  noises = filter(s -> get(prev_world, :is_noise, s), sources)
  merge_possible = length(tones) > 1 || length(noises) > 1
  splitprior = merge_possible ? 0.5 : 1.
  do_split ~ bernoulli(splitprior)

  if do_split
    to_split ~ uniform_choice(sources)
    idx1 ~ uniform_discrete(1, length(sources) + 1)
    idx2 ~ uniform_from_list([i for i=1:length(sources) + 1 if i != idx1])

    if get(prev_world, :is_noise, to_split)
      old_amp = get(prev_world, :amplitude, to_split)
      amp1 ~ normal(old_amp, 0.5); Amp2 ~ normal(old_amp, 0.5)
    else
      old_freq = get(prev_world, :freq_in_erb, to_split)
      freq1 ~ normal(old_freq, 0.5); freq2 ~ normal(old_freq, 0.5)
    end
    {*} ~ smart_sample_endpoints(prev_world, to_split)
  else
    to_idx ~ uniform_discrete(1, length(sources) - 1)
    mergeable = (length(tones) > 1 && length(noises) > 1) ?
          union(noises, tones) : (length(tones) > 1 ? tones : noises)
    # sample 2 sources to merge, where the second's beginning
    # and ending is after the first's
    function is_before(x, y)
      startx = get(prev_world, :starttime, x)
      starty = get(prev_world, :starttime, y)
      durx = get(prev_world, :duration, x)
      dury = get(prev_world, :duration, y)
      return startx < starty && startx + durx < starty + dury
    end
    function is_not_last_mergeable(x)
      return length(filter(y -> is_before(y, x), mergeables)) > 1
    end
    not_last_mergeables = filter(is_not_last_mergeable, mergeables)
    merge1 ~ uniform_choice(not_last_mergeables)
    merge2 ~ uniform_choice(filter(x -> is_before(x, merge1), mergeables))

    if get(prev_world, :is_noise, merge1)
      amp1 = get(prev_world, :amplitude, merge1)
      amp2 = get(prev_world, :amplitude, merge2)
      amplitude ~ normal((amp1 + amp2)/2, 0.5)
    else
      freq1 = get(prev_world, :freq_in_erb, merge1)
      freq2 = get(prev_world, :freq_in_erb, merge2)
      freq ~ normal((freq1 + freq2)/2, 0.5)
    end
  end
end
# one of the 2 new sounds we split `to_split` to will start
# at the time when `to_split` begins; the other will end at the
# time `to_split` ends.  This function samples how long each will be.
gen function smart_sample_durations(prev_world, to_split)
  # analyze the region of the spectrogram in the time and frequency
  # range of the sound we are splitting; use a Sobel edge-detection
  # filter to find breaks in the sound in this region
  likely_endpoints = get_detected_endpoints(prev_world, to_split)
  if likely_endpoints != nothing # if breakpoints were detected
    likely_start, likely_end = likely_endpoints/SCENE_LENGTH

    sound_start = get(prev_world, :starttime, to_split)
    sound_duration = get(prev_world, :duration, to_split)
    likely_duration1 = likely_start - sound_start
    likely_duration2 = sound_start + sound_duration - likely_end

    dur1 ~ normal(likely_duration1, 0.05)
    dur2 ~ normal(likely_duration2, 0.05)
  else
    dur1 ~ uniform(1, SCENE_LENGTH)
    dur2 ~ uniform(1, SCENE_LENGTH)
  end
end
```

```
involution function splitmerge_involution()
  backward[:do_split] = !proposed[:do_split]
  if proposed[:do_split]
    old = proposed[:to_split]
    (new1, new2) = split old to indices
                                  proposed[:idx1], proposed[:idx2]
    backward[:merge1] = new1
    backward[:merge2] = new2
    backward[:to_idx] = index(old)
    set is_noise(new1) = get(is_noise(old))
    set is_noise(new2) = get(is_noise(old))

    # set starttimes and durations of new sounds
    set starttime(new1) = get(starttime(old))
    set duration(new1) = proposed[:duration1]

    # the second new sound will have the same end as the old sound.
    # the model specifies in terms of starttime, duration, so we
    # must do some math to calculate the new startpoint:
    start2 = (get(starttime(old)) + get(duration(old))
                                      - proposed[:duration2])
    set starttime(new2) = start2
    set duration(new2) = proposed[:duration2]

    if get(is_noise(old))
      set amplitude(new1) = proposed[:amp1]
      set amplitude(new2) = proposed[:amp2]
      backward[:amplitude] = get(amplitude(old))
    else
      set freq_in_erb(new1) = proposed[:freq1]
      set freq_in_erb(new2) = proposed[:freq2]
      backward[:freq] = get(freq_in_erb(old))
    end
  else
    old1, old2 = proposed[:merge1], proposed[:merge2]
    new = merge old1, old2 to index proposed[:to_idx]
    backward[:idx1] = index(old1)
    backward[:idx2] = index(old2)
    backward[:to_split] = new

    set is_noise(new) = get(is_noise(old1))
    set starttime(new) = get(starttime(old1))
    endpoint = get(starttime(old2)) + get(duration(old2))
    set duration(new) = endpoint - get(starttime(old1))
    backward[:duration1] = get(duration(old1))
    backward[:duration2] = get(duration(old2))

    if get(is_noise(old1))
      set amplitude(new) = proposed[:amplitude]
      backward[:amp1] = get(amplitude(old1))
      backward[:amp2] = get(amplitude(old2))
    else
      set freq_in_erb(new) = proposed[:freq]
      backward[:freq] = get(freq_in_erb(old1))
      backward[:freq] = get(freq_in_erb(old2))
    end
  end
end
```

Figure 11: The **data-driven sound splitting kernel** either chooses to split one long sound into two short ones, or merge two short tones into a longer one. In the split move, we use a similar edge-detection algorithm to what was used in the **create detected sound** kernel to detect changepoints in the observed sound during the time at which the sound being split occurs. The kernel proposes an endpoint for the first new sound and a startpoint for the second new sound to be close to the detected changepoints.

A.2.1. The automated acceptance probability for the **data-driven sound detection kernel**

To illustrate the calculations our system automates, we give the acceptance probability calculation performed when running the **data-driven sound detection kernel**, including calculating the Jacobian of the transformation.

Model density. We can factor the density of the model, $p(x)$, as $p(x) = p(x_{\mathcal{L}})p(x_{\mathcal{O}}|x_{\mathcal{L}})$, where $x_{\mathcal{L}}$ is the collection of latent variable values in x (the number of underlying sound sources, and the properties of each one), and $x_{\mathcal{O}}$ is the observed sound waves variable in x. For any x where each variable is within its support, the probability of the latents is:

$$p(x_{\mathcal{L}}) = \frac{1}{4}\prod_{i=1}^{x_{N_S}} (p(x_{D,i}) \times p(x_{\tau,i}) \times p(x_{(\alpha|f),i}) \times p(x_{n,y_s}))$$

$$= \frac{1}{4}\prod_{i=1}^{x_{N_S}} ((\frac{1}{L-0.1}) \times (\frac{1}{L-0.1}) \times (\begin{cases} \mathcal{N}(x_{(\alpha|f),i}, 8, 10) & x_{n,i} = 1 \\ \frac{1}{37-0.4} & \text{otherwise} \end{cases}) \times 0.4^{x_{n,i}}(1-0.4)^{1-x_{n,i}})$$

Here, x_{N_S} is the number of sound sources in MSSI x (the value of the number variable `number Source()`); $x_{D,i}$ is the duration of the ith audio source in x; $x_{\tau,i}$ is the starttime of the ith audio source in x; $x_{n,i}$ is an indicator which is 1 if the ith source in x is a noise and 0 otherwise; $x_{(\alpha|f),i}$ denotes the amplitude of the ith sound in x if it is a noise, and the frequency of this sound if it is a tone. We let L denote the length of the rendered audio tones (which is a global constant).

The probability of the observation given the latents is:

$$p(x_{\mathcal{O}} \,|\, x_{\mathcal{L}}) = \mathcal{N}(x_{\mathcal{O}}, r(x_{\mathcal{L}}), I_{|r(x_{\mathcal{L}})|})$$

where x_O denotes the observed soundwaves in x (from the `observed_sound` property in Figure 7) represented as a vector, $r(x_{\mathcal{L}})$ denotes the *rendering* of the latent audio sources in x into sound-waves, and I_k denotes the k-dimensional identity matrix.

State-dependent proposal density. The state-dependent proposal distribution $q_x(y)$ is parameterized by a current MSSI x. It is a distribution over finite dictionaries, implemented as a Gen program. Using standard techniques from probabilistic programming, Gen can automate the computation of densities of dictionaries sampled by the proposal:

$$q_x(y) = \begin{cases} \frac{(1-\beta(x))}{x_{N_S}} \cdot (\beta_\rho(x, y_i) \cdot \theta_\rho(x, y_i)_{y_{s'}})^{y_\rho} \cdot (1-\beta_\rho(x))^{1-y_\rho} & y_b = 0 \\ \frac{\beta(x)\cdot(1-\beta_\sigma(x))}{x_{N_S}+1} \cdot \frac{(0.4\cdot\mathcal{N}(y_\alpha;8,10))^{y_n}\cdot(0.6\cdot(37-0.4)^{-1})^{1-y_n}}{(L-0.1)^2} & y_b = 1 \wedge y_\sigma = 0 \\ \frac{\beta(x)\cdot\beta_\sigma(x)}{x_{N_S}+1} \cdot \theta(x)_{y_s} \cdot \mathcal{N}\left(\begin{bmatrix} y_D \\ y_\tau \\ y_{v_D(x,y_s)} \end{bmatrix}; \begin{bmatrix} D_D(x,y_s) \\ D_\tau(x,y_s) \\ D_{v_D(x,y_s)}(x,y_s) \end{bmatrix}, \begin{bmatrix} 0.05 & 0 & 0 \\ 0 & 0.05 & 0 \\ 0 & 0 & \sigma_{v_D(x,y_s)} \end{bmatrix}\right) & y_b = 1 \wedge y_\sigma = 1 \end{cases}$$

Here, x_{N_S} is the number of sound sources in MSSI x, and y_b determines whether a new sound source will be proposed; if $y_b = 1$, y_σ determines whether the new source will be proposed using the data-driven edge-detection proposal or from the prior. (In Figure 10, y_b corresponds to `do_birth`; y_σ to `is_smart`.) When $y_\sigma = 1$, an index y_s is computed

determining *which* of the sounds from the image-filtering and edge-detection algorithm will be proposed (the index of sample `sound_to_birth` in the `propose_smart_birth` function in Figure 10). Based on y_s, new sound onset and duration parameters y_τ and y_D, as well as either a frequency y_f or an amplitude y_α. When the data-driven proposal is not used ($y_b = 1 \wedge y_\sigma = 0$), y_α (or y_f), y_n (whether this sound source is a white noise source, as opposed to a tone), and other parameters are proposed from the model's prior over sound sources. If $y_b = 0$, a death move is proposed, to delete sound y_i. The variable y_ρ is introduced to enable the implementation of an involution: it represents whether the move *reversing* a proposed death move will be data-driven. If $y_\rho = 1$, $y_{s'}$ represents which heuristically detected sound source will be used as the basis to repropose the deleted audio source, in the reverse proposal.

The proposal is based on functions β, β_ρ, β_σ, θ, θ_ρ, D_D, D_τ, D_α, D_f, and v_D, as well as the scene length parameter L and the standard deviations $\sigma_\alpha = 4$ and $\sigma_f = 1$. These functions are described below:

- $\beta(x) = [x_{N_S} < 4] \cdot 0.5 + [x_{N_S} = 0] \cdot 0.5$ is the probability of y_b given x.
- $\beta_\sigma(x) = 1 - 0.05^{[|\theta(x)|>0]}$ is the probability of $y_\sigma = 1$ given that $y_b = 1$ for MSSI x.
- $\beta_\rho(x, i)$ is 0, if removing sound i from x and running image filtering and edge detection does not find any sounds of the same type as sound i in x, and 0.95 otherwise.
- $\theta(x)$ is a vector-valued function, taking values in the $(k(x)-1)$-simplex $\Delta^{k(x)-1}$, where $k(x)$ is the number of detected possible sound sources from the image filtering and edge detection algorithm applied to state x. It represents a distribution over more or less promising detections.
- $\theta_\rho(x, i)$ is a vector-valued function over the $(k_\rho(x, i) - 1$-simplex, where $k_\rho(x, i)$ is the number of detected sounds from the edge detection routine, applied to x without sound source i, that have the same sound source type as sound source i in x.
- $D_D(x, i)$ is the predicted duration of the ith detected sound source from the image filtering and edge detection algorithm applied to x. Similarly, D_τ gives the predicted onset, D_α the predicted amplitude (if applicable), and D_f the predicted frequency (if applicable).
- $v_D(x, i)$ is the symbol α (amplitude) if the ith detected sound source is labelled as a noise, and the symbol f (frequency) if is labelled as a tone.

The Jacobian. The **create detected sound** kernel always either creates or deletes a sound; we show the Jacobian of the matrix for a move which creates a sound (the transpose of which is the Jacobian of the matrix for the corresponding deletion). The transition $(x, y) \to (x', y')$ during a creation copies over every continuous value in x and y into x'. These values come from the properties `starttime` (τ) and `duration` (D) of each source, and the `amplitude` (D) property of each noise or `frequency` (f) property of each tone. Depending on the index of the newly-created sound, the indices of each object may change, as is described in . In particular, when we birth a new source with index i, we must increment the index of each existing source with index $\geq i$. We define the following identifiers for

whether the object at index a is moved to index b, and whether the birth index i equals a given index a:

$$C_{a\to b} = \begin{cases} 1 & (i > a \wedge a = b) \vee (i \leq a \wedge b = a+1) \\ 0 & \text{otherwise} \end{cases}$$

$$B_a = \begin{cases} 1 & i = a \\ 0 & \text{otherwise} \end{cases}$$

Using these definitions, we can write the Jacobian of the transformation which copies starttime (τ), duration (D), and amplitude/frequency ($\alpha|f$) values for each object as:

$$J = \begin{array}{c} \\ y_{(\alpha|f)} \\ y_\tau \\ y_D \\ x_{(\alpha|f),1} \\ x_{\tau,1} \\ x_{D,1} \\ x_{(\alpha|f),2} \\ x_{\tau,2} \\ x_{D,2} \\ \vdots \\ x_{D,N_S} \end{array} \begin{array}{c} \begin{array}{cccccccc} x_{(\alpha|f),1} & x_{\tau,1} & x_{D,1} & x_{(\alpha|f),2} & x_{\tau,2} & x_{D,2} & \dots & x_{D,N_S+1} \end{array} \\ \begin{pmatrix} B_1 & 0 & 0 & B_2 & 0 & 0 & \dots & 0 \\ 0 & B_1 & 0 & 0 & B_2 & 0 & \dots & 0 \\ 0 & 0 & B_1 & 0 & 0 & B_2 & \dots & B_{N_S+1} \\ C_{1\to 1} & 0 & 0 & C_{1\to 2} & 0 & 0 & \dots & 0 \\ 0 & C_{1\to 1} & 0 & 0 & C_{1\to 2} & 0 & \dots & 0 \\ 0 & 0 & C_{1\to 1} & 0 & 0 & C_{1\to 2} & \dots & C_{1\to N_S+1} \\ C_{2\to 1} & 0 & 0 & C_{2\to 2} & 0 & 0 & \dots & 0 \\ 0 & C_{2\to 1} & 0 & 0 & C_{2\to 2} & 0 & \dots & 0 \\ 0 & 0 & C_{2\to 1} & 0 & 0 & C_{2\to 2} & \dots & C_{2\to N_S+1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \\ 0 & 0 & C_{2\to N_S} & 0 & 0 & C_{2\to N_S} & \dots & C_{N_S\to N_S+1} \end{pmatrix} \end{array}$$

where

$$J_{a,b} = \begin{cases} 0 & a \pmod 3 \neq b \pmod 3 \\ B_b & a \leq 3 \\ C_{a\to b} & a > 3 \end{cases}$$

Since this is a permutation matrix, it always has determinant -1 or 1. Our system is capable of computing this Jacobian using automatic differentiation and discovering this. As an optimization, in practice, our system can identify that this transformation only copies values, and thus will have a permutation-matrix Jacobian which it does not need to take the determinant of, since it will certainly have magnitude 1, as we describe in the **Jacobian sparsity** paragraph in Section D. For an description of an inference kernel we have implemented which has a Jacobian whose determinant's magnitude is (usually) not 1, see Section A.1.

Acceptance probability. When using the involutive MCMC kernel, our system automatically calculates $q_x(y)$, $q_{x'}(y')$, $p(x)$, $p(x')$, and J, to compute the acceptance probability of the move $(x, y) \to (x', y')$,

$$\frac{p(x')}{p(x)} \times \frac{q_{x'}(y')}{q_x(y)} \times |\det J|$$

These terms are all equivalent to the above expressions for $q_x(y)$, $p(x)$, and J (sometimes substituting x' for x or y' for y). In practice, the q terms are automatically calculated using the given expressions, while computational optimizations let us avoid calculating the

determinant of the Jacobian matrix (which is just a permutation matrix), and let us avoid fully calculating $p(x')$ and $p(x)$, instead directly calculating

$$\frac{p(x')}{p(x)} = \frac{p(x'_{\mathcal{O}}|x'_{\mathcal{L}}) \times \frac{1}{4}\prod_{i=1}^{x'_{N_S}} p(x'_{D,i}) \times p(x'_{\tau,i}) \times p(x'_{(\alpha|f),i}) \times p(x'_{n,i})}{p(x_{\mathcal{O}}|x_{\mathcal{L}}) \times \frac{1}{4}\prod_{i=1}^{x_{N_S}} p(x_{D,i}) \times p(x_{\tau,i}) \times p(x_{(\alpha|f),i}) \times p(x_{n,i})}$$

$$= \frac{p(x'_{\mathcal{O}}|x'_{\mathcal{L}})}{p(x_{\mathcal{O}}|x_{\mathcal{L}})} \times \begin{cases} p(x'_{D,y_s}) \times p(x'_{\tau,y_s}) \times p(x'_{(\alpha|f),y_s}) \times p(x'_{n,y_s}) & y_b = 1 \\ \frac{1}{p(x_{D,y_s}) \times p(x_{\tau,y_s}) \times p(x_{(\alpha|f),y_s}) \times p(x_{n,y_s})} & y_b = 0 \end{cases}$$

where, as above, y_b is 1 when y proposes to create a new source and 0 otherwise, and y_s is the index of the source being created/deleted. We can avoid evaluating the product expressions because the majority of terms in the numerator and denominator cancel out, since all audio sources except the one being created/deleted are unaffected by the update.

Appendix B. Open-Universe Models and Contingent Bayesian Networks

In this section, we give a more complete definition of open-universe probabilistic models and contingent Bayesian networks (outlined in Section 2).

Open-universe probabilistic models are models of relational domains in which the number of objects, and their link structure, is not known a priori. The usual representations for structured probability distributions—e.g., probabilistic graphical models—are insufficient for specifying OUPMs, which typically define an unbounded number of random variables, with conditional independence relationships that are themselves random. OUPMs can be specified formally, however, as programs in sufficiently expressive probabilistic programming languages (Milch et al., 2005a), or as *contingent Bayesian networks* (CBNs) with infinitely many nodes (Milch et al., 2005b). Here, we review the standard definition of OUPMs as CBNs, and extend it to allow for objects with real-valued random properties.

Open-Universe Probabilistic Models. An *open universe probabilistic model* is a tuple $(\mathcal{T}, \mathcal{N}, \mathcal{P}, \mathcal{C})$, where

- $\mathcal{T}$ is a finite [3] set of *object types* over which the model is defined (e.g., *Person*).
- $\mathcal{N}$ is a finite [3] set of *object origin declarations* $(\tau, (\tau_1, \ldots, \tau_n))$, with $\tau, \tau_j \in \mathcal{T}$. An object origin declaration specifies that a tuple of objects, of types $(\tau_1, \ldots, \tau_n)$ respectively, can be the *origin* of a separate object of type τ. For example, in a model of genealogy, we might declare $(Person, (Person, Person)) \in \mathcal{N}$, encoding that a pair of people can be the origin (i.e., parents) of other people. A model's object origin declarations determines sets of *possible objects* $\mathcal{O}_\tau(\mathcal{N})$ for each type τ, which are finite expressions of the form $(\tau, (o_1, \ldots, o_n), i)$, where $(\tau, (\tau_1, \ldots, \tau_n)) \in \mathcal{N}$, $i \in \mathbb{N}$, and $o_j \in \mathcal{O}_{\tau_j}(\mathcal{N})$ for each j. We call $(\tau, (o_1, \ldots, o_n), i)$ the ith object of type τ with origin $(o_1, \ldots, o_n)$.
- $\mathcal{P}$ is a countable set of typed *object property declarations* $(P, (\tau_1, \ldots, \tau_n))$, with $\tau_j \in \mathcal{T}$, together with value domains $\mathrm{dom}(P)$ for each property. A property represents data concerning zero or more objects, of types $(\tau_1, \ldots, \tau_n)$ respectively. For example, a model might declare a property representing the number of flights between two places.
- $\mathcal{C}$ is a *continuously-extended contingent Bayesian network* on the set of *possible variables* for $(\mathcal{T}, \mathcal{N}, \mathcal{P})$, $\{(x, (o_1, \ldots, o_n)) \mid (x, (\tau_1, \ldots, \tau_n)) \in \mathcal{P} \cup \mathcal{N} \wedge \forall j, o_j \in \mathcal{O}_{\tau_j}(\mathcal{N})\}$. For every origin declaration $(\tau, (\tau_1, \ldots, \tau_n)) \in \mathcal{N}$, and every tuple $(o_1, \ldots, o_n) \in \bigotimes_j \mathcal{O}_{\tau_j}(\mathcal{N})$ of possible objects of the correct types, there is a possible *number variable* $(\tau, (o_1, \ldots, o_n))$, with domain $\mathbb{Z}_{\geq 0}$, representing the number of objects of type τ that have objects $(o_1, \ldots, o_n)$ as their origin. For every property declaration $(P, (\tau_1, \ldots, \tau_n))$ and object tuple $(o_1, \ldots, o_n)$ of the correct types, there is a possible *property variable* specifying the value of the property P (within $\mathrm{dom}(P)$) of the object tuple $(o_1, \ldots, o_n)$. The contingent Bayesian network itself defines a distribution over assignments to these infinitely many possible variables, as we describe in the following section.

3. The set of types and the set of object origin declarations can be countable without compromising any results, but existing formal languages for defining OUPMs, including ours, do not provide a way to declare models where either of these are infinite. It would be interesting to consider whether any natural applications would require this flexibility, and how languages might be designed to expose it.

Contingent Bayesian Networks. For a set of possible variables $\mathcal{V}$, let $\Omega = \bigotimes_{v \in \mathcal{V}} \mathrm{dom}(v)$ be the space of all possible assignments of the (possibly infinitely many) variables to values within their domains. When the domains are discrete, we can use a *contingent Bayesian network* (CBN) (Milch et al., 2005b) to define a distribution over such assignments. Analogously to a Bayesian network on a finite number of variables, a CBN on an infinite set of variables $\mathcal{V}$ specifies an infinite directed graph with $\mathcal{V}$ as the nodes. But whereas traditional Bayesian networks fix the dependency structure, in a CBN the edges are *contingent*: each edge $u \to v$ is labeled with an event $\Omega_{u \to v} \subseteq \Omega$ within which the edge is *active*. As in Bayesian networks, to each variable v we associate a conditional probability distribution (CPD) p_v. But this conditional probability distribution is allowed to depend only on the *active* parents of v, which can be formalized as follows: for each variable $v \in \mathcal{V}$, the contingent Bayesian network associates a partition Λ_v of the space Ω, such that if ω_1 and ω_2 differ only in their assignment to u, and the edge $u \to v$ either does not exist or is *inactive* in ω_1, then ω_1 and ω_2 belong to the same component of Λ_v. Then $p_v(\cdot \mid \omega \in \lambda)$ is allowed to depend on *which component* λ of the partition ω is in, but nothing else, preventing dependencies on the values of inactive parents. We write $Pa_\omega(v)$ for the active parents of the variable v in state ω. These conditional probabilities together define a probability measure Π on the space Ω, equipped with a non-standard σ-algebra; we refer the reader to Milch et al. (2005b) for a detailed treatment.

Minimal Self-Supporting Instantiations. In open-universe models, possible worlds $\omega \in \Omega$ are generally infinite, because they specify assignments to all of the possible variables $v \in \mathcal{V}$. But for inference, we must work with finite representations. If we can partition Ω into disjoint equivalence classes Ω_i, then we can work with the equivalence classes, using the distribution $p(\Omega_i) = \Pi(\Omega_i)$. One useful strategy for partitioning Ω is to consider the *minimal self-supporting instantiations* of a finite set of variables $U \subseteq \mathcal{V}$. Each component w of the partition is labeled by a set of variables $\mathrm{vars}(w) \subseteq \mathcal{V}$, with $U \subseteq \mathrm{vars}(w)$, as well as an assignment $w[v] \in \mathrm{dom}(v)$ for each $v \in \mathrm{vars}(w)$. The component w contains all full possible worlds $\omega \in \Omega$ that agree with w on the values of each $v \in \mathrm{vars}(w)$. To ensure that the components w are indeed disjoint, we only include w as a component if: (1) it is *self-supporting*, meaning that for all $\omega \in w$, if $v \in \mathrm{vars(w)}$ and $u \in Pa_\omega(v)$, then $u \in \mathrm{vars}(w)$; and (2) it is *minimal*, meaning that removing any variable $v \notin U$ from $\mathrm{vars}(w)$ would cause it to fail to be self-supporting. Together, these criteria ensure that for any $U \subseteq \mathcal{V}$, the components w are disjoint, and so $p(w) = \Pi(w)$ is a valid probability distribution over minimal self-supporting instantiations w. Conveniently, the probability of each component w is a simple product of the conditional distributions p_v, as in a typical Bayesian network:

$$p(w) = \Pi(w) = \prod p_v(w[v] \mid \{w[u] \mid u \in Pa_w(v)\}). \tag{4}$$

B.1. Continuously-Exetended Contingent Bayesian Networks.

We now extend the CBN framework to support models with some continuous object property domains, so long as no discrete variables count continuous variables as parents.[4] As

4. See the following section for a similar but more involved extension to a more general case, where discrete variables can depend on continuous variables.

before, let $\mathcal{V}$ be a collection of variables, but now suppose some variables $\mathcal{V}_c \subseteq \mathcal{V}$ have continuous domains. Then define the surrogate discrete domain $\mathrm{dom}_D(v)$ of a variable v to be $\mathrm{dom}(v)$ if $v \notin \mathcal{V}_c$, and $\{\star\}$ otherwise. A *continuously extended* CBN defines (1) an ordinary CBN on the variables $\mathcal{V}$, treating them as having surrogate discrete domains dom_D, and (2) conditional probability densities $p_v(\cdot \mid \{w[u] \mid u \in Pa_w(v)\})$ for each continuous variable $v \in \mathcal{V}_c$. The CBN induces a distribution Π on minimal self-supporting worlds, which we modify to define a probability measure on the space $\mathcal{W}$ of *continuously extended minimal self-supporting worlds*: pairs (w_d, w_c) of a minimal self-supporting world w_d for the discrete CBN, plus a real-valued assignment w_c to all variables in $\mathrm{vars}(w_d) \cap \mathcal{V}_c$. We let $w[v]$ denote $w_c[v]$ if $v \in \mathcal{V}_c$ and $w_d[v]$ otherwise. The distribution has density

$$p(w) = \Pi(w_d) \times \prod_{v \in \mathcal{V}_c \cap \mathrm{vars}(w)} p_v(w_c[v] \mid \{w[u] \mid u \in Pa_w(v)\}) \tag{5}$$

with respect to the reference measure

$$\mu(E) = \sum_w \lambda^{|w_c|}(\{w'_c \mid w' \in E \wedge \mathrm{vars}(w') = \mathrm{vars}(w) \wedge \forall v \in \mathrm{vars}(w') \setminus \mathcal{V}_c, w'[v] = w[v]\}),$$

where λ^k denotes the k-dimensional Lebesgue measure. Using Equation 4 for the first term of $p(w)$, we see that in the continuous case, we still have the ordinary Bayes net factorization $p(w) = \prod_{v \in \mathrm{vars}(w)} p_v(w[v] \mid \{w[u] \mid u \in Pa_w(v)\})$.

B.1.1. Continuously-Extended CBNs with Continuous Parents of Discrete Variables

In the previous section, we define continuously-extended contingent Bayesian networks with the restriction that no discrete variables have continuous variables as parents. We now give a more flexible development.

Let $\mathcal{V}$ be a collection of variables, where $\mathcal{V}_c \subseteq \mathcal{V}$ have continuous domains. Define surrogate discrete domain $\mathrm{dom}_D(v)$ to be $\mathrm{dom}(v)$ if $v \notin \mathcal{V}_c$, and $\{\star\}$ otherwise. Now define a continuously extended CBN to comprise: (1) an ordinary CBN on the variables $\mathcal{V}$, treating them as having surrogate discrete domains dom_D, and (2) a conditional *joint* probability density $\mathcal{P}_c(w_c \mid w)$ on the values of the continuous variables in a minimal self-supporting instantiation w given the values of all w's discrete variables. This CBN induces distribution Π on minimal self-supporting worlds; as before, we modify this to define the space $\mathcal{W}$ of *continuously extended minimal self-supporting worlds*, consisting of a minimal self-supporting world w of the CBN, and an assignment $w[v]$ to each continuous variable in $\mathrm{vars}(v) \cap \mathcal{V}_c$. We use w_c to denote the assignment of values $w[v]$ to the continuous variables $v \in \mathcal{V}_c \cap \mathrm{vars}(w)$ in w. The distribution then has density

$$p(w) = \Pi(\{\omega \mid \forall v \in \mathrm{vars}(w) \setminus \mathcal{V}_c, \omega[v] = w[v]\}) \times \mathcal{P}_c(w_c|w)$$

with respect to the same reference measure from Section 2:

$$\mu(E) = \sum_w \lambda^{|w_c|}(\{w'_c \mid w' \in E \wedge \mathrm{vars}(w') = \mathrm{vars}(w) \wedge \forall v \in \mathrm{vars}(w') \setminus \mathcal{V}_c, w'[v] = w[v]\}),$$

where λ^k denotes the k-dimensional Lebesgue measure. We note that the density we give ultimately factorizes, as before, as a marginal probability for the discrete variables and a conditional density for the continuous variables. (In principle, this alone is not a restriction, because any distribution over minimal self-supporting instantiations can be factored in this way.) But in the previous formulation (from Section 2), this factorization was embedded into the discrete CBN itself, affecting which worlds w counted as minimal and self-supporting in the first place. The key difference between this section's formulation and the one in Section 2 is that here we divorce the factorization of the continuous density from the CBN's structure; the edges involving continuous variables in the CBN serve only to determine which continuous variables are required in a minimal, self-supporting instantiation for an observed set of variables U.

Appendix C. OUPM Inference Programming Language

In this section, we briefly outline our implementation of open-universe modeling constructs in the Gen probabilistic programming system, and we describe the DSL we have developed for writing involutions for involutive MCMC on OUPMs.

OUPM Constructs in Gen We have implemented open universe modeling constructs in the Gen probabilistic programming system, inspired by BLOG (Milch, 2006); for examples, refer to Figure 1 or Figure 7. We add `type` statements for declaring object types, `property` declarations for adding properties, and `number` declarations for adding possible origins. When writing world models with our system, users must define a special property $O()$ with respect to which the model will take minimal self-supporting instantiations; they do this using an `observation_model` statement. Property and number declarations are attached to *function bodies* which together define the contingent Bayesian network.

In particular, a property statement declares the existence of a property $P(\tau_1, \ldots, \tau_n)$, and defines a probabilistic generative process for the property's value. A number statement $(\tau, (\tau_1, \ldots, \tau_n))$ declares that $(\tau_1, \ldots, \tau_n)$ is a permissible origin signature for objects of type τ, and declares the distribution over how many objects of type τ there are with each origin matching $(\tau_1, \ldots, \tau_n)$. (So each property/number statement simultaneously serves as an object property declaration/object origin declaration, and declares the distribution over the corresponding property/number variables in the OUPM's CBN.) The bodies of number and property declarations are specified as *generative functions*, imperative probabilistic programs that induce distributions over finite dictionaries, called *traces*. A generative function is implemented using ordinary Julia syntax for defining functions, but may sample random variable values using $x \sim d$ statements. A generative function defines a distribution over traces mapping each key x to the value sampled from the corresponding d in an execution of the function. We augment generative functions to additionally support `get(`$P(o_1, \ldots, o_n)$`)` statements to access the value of $P(o_1, \ldots, o_n)$ in the world; this introduces an edge in the contingent Bayes net representing the model.

Represented as generative functions, number variables must always have 0 or 1 keys in their traces, corresponding to either deterministically returning a number, or sampling a single random number. Properties may have more than 1 key in their trace (so long as the set of possible keys is countable), but when they do, the the property statement P must be understood as defining a separate object property declaration P `=>` k for each k which can appear as a key in the trace. For simplicity, in what follows, we will assume that each $P(o_1, \ldots, o_n)$ is itself variable with only 0 or 1 keys in its generative function's traces; however, the algorithms we give also hold when there can be multiple keys. We represent MSSIs as sets of pairs $(P(o_1, \ldots, o_n), t)$, where P is a number or property generative function whose type signature matches $(o_1, \ldots, o_n)$, and t is a trace for this variable.

OUPM Involution Programming Language We implement a new OUPM involution programming language to implement involutions on the space of pairs (x, y), where x is a world represented as described above, and y is a trace from a generative function. Involutions are written as imperative functions which may call special commands to modify state x in order to produce x', to access values in x and y, and to specify the new values for y'. Specifically, the DSL provides support for the following.

- **Querying the current model state (x).** Users may query the current model state using the syntax `get(`$P(o_1, \ldots, o_n)[a]$`)`, which retrieves the sample at address a of the trace for the property $P(o_1, \ldots, o_n)$. Similarly, `get_number(`$\tau(o_1, \ldots, o_n)$`))` retrieves the number variable value in x defining how many objects of type τ exist with origin $(o_1, \ldots, o_n)$. Several higher-level commands are also provided to access higher-level discrtete information about the current state, including `get_object_set(`τ`)`, which gets the set of all existing objects of type τ in x.

- **Querying the auxiliary randomness (y).** The syntax `proposed[`a`]` returns $y[a]$, the value at address a in the random trace y of the proposal program $\mathcal{Q}$.

- **Proposing the new model state (x').** The new model state is proposed via commands that imperatively modify the old state x. They can do this either by changing the *structure* of x (by adding or removing objects, and changing the origins or indices of objects), or by writing new values to object properties.

  The commands **create**, **delete**, **split**, **merge**, and **change** add, remove, and change the relationships of objects in the possible world, updating the values of number variables as necessary. The behavior of each detailed below, and illustrated there in Figure 12. Commands that remove objects have the effect of deleting any traces for applications $G(o_1, \ldots, o_i, \ldots o_n)$ that apply to an object o_i slated for deletion. If the deleted object is the origin of other objects (e.g., if an `Event` with `Detection`s is deleted), those objects are not automatically deleted: users may either manually remove them or change their origins using **change**.

  The command `set` $G(o_1, \ldots, o_n)[a] = v$ sets the sample at address a of a trace $G(o_1, \ldots, o_n)$ to v in the proposed state x'.

- **Transforming the auxiliary randomness (y').** The user's code is responsible for computing a new trace y' of $\mathcal{Q}$ that makes f into an involution. Intuitively, y' must specify the proposal choices that would *reverse* the current move; in Figure 2, the forward move that posits a new `Event` to explain a false positive is reversed by a move that deletes the newly created `Event` and reclassifies its sole detection as a false positive. Users can write to y' using the syntax `backward[`a`] =` v.

C.1. Object manipulation moves

The involution programming language includes commands to create, delete, split, merge, and change the origins of objects in the world. It is clear that these moves should increment or decrements number variables in the world to change the number of objects hypothesized to exist. However, to enable us to delete any object with any index, or create an object at any index (needed for reversibility), we need some scheme for reindexing the other existing objects. (For example, when creating a new object 1, the system must "make room" for the new object 1, for example by incrementing the index of every existing object.) We describe the reindexing scheme we use.

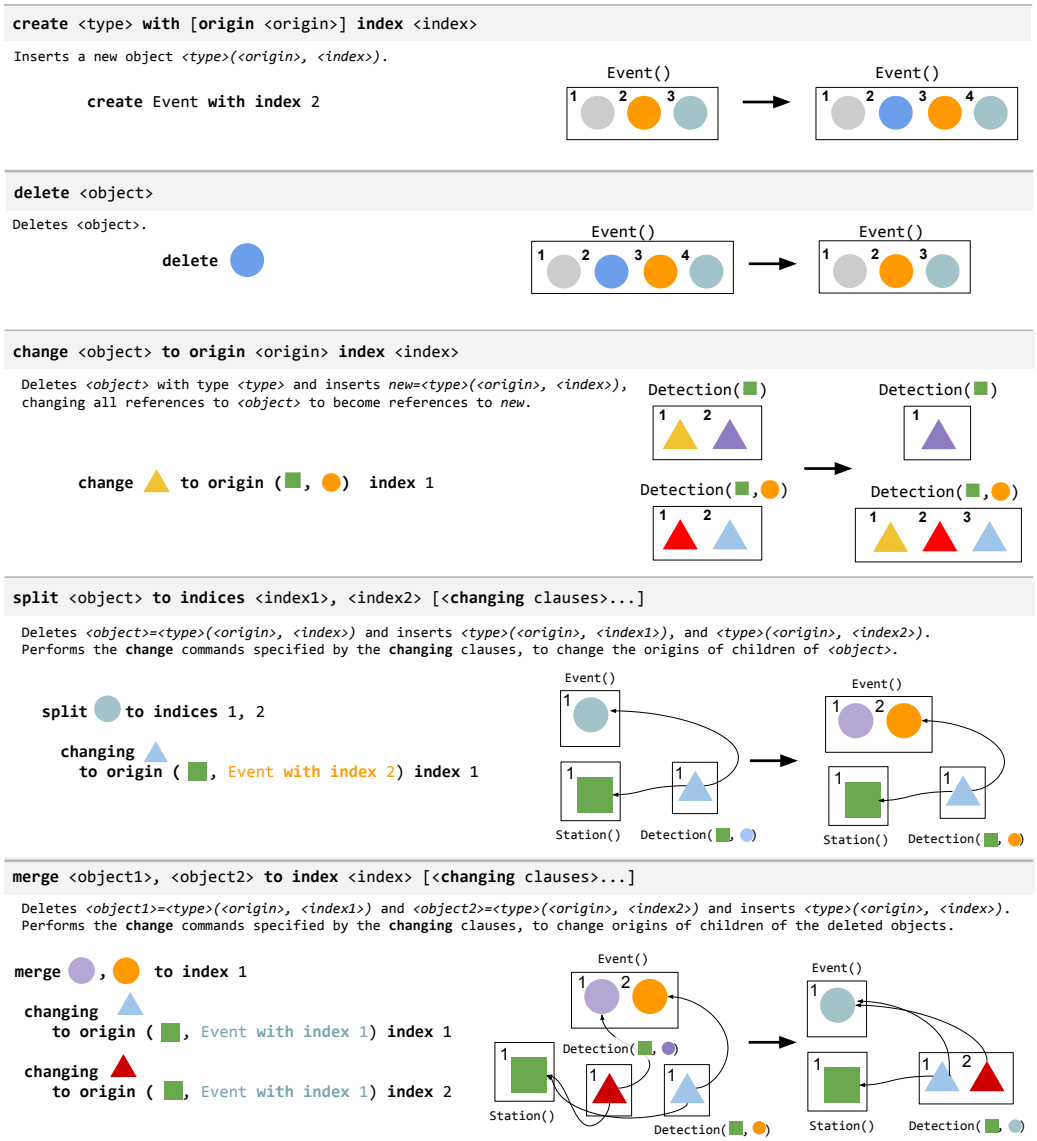


Figure 12: The object manipulation moves, with examples illustrating how they change the space of objects in the seismic monitoring example. To construct references objects, for use in the commands, illustrated with the shapes in this diagram, users use the syntax `<type> with origin <origin> index <index>`.

Reindexing. For any given origin tuple of objects, `origin`, a world contains $N_T^{\texttt{origin}}$ objects of type T: $\{T(\texttt{origin}, 1), ..., T(\texttt{origin}, N_T^{\texttt{origin}})\}$. Deleting the object $T(\texttt{origin}, i)$ corresponds to a "vector deletion" which decrements the value of $N_T^{\texttt{origin}}$ and decrements the index of every object $T(\texttt{origin}, j)$ for $j > i$. Likewise, adding an object $T(\texttt{origin}, i)$ is a "vector insertion" which increments $N_T^{\texttt{origin}}$ and increments the index of every object $T(\texttt{origin}, j)$ for $j \geq i$. In a world model, "changing the index" of an object $T(\texttt{origin}, j)$ corresponds to updating the world so that the choicemap for any property or number variable $P[o_1, ..., T(\texttt{origin}, j), ..., o_n]$ becomes the choicemap for $P[o_1, ..., T(\texttt{origin}, j +$

$1), ..., o_n]$ or $P[o_1, ..., T(\texttt{origin}, j-1), ..., o_n]$. Likewise, the update must change any object which has $T(\texttt{origin}, j)$ in its origin to instead have $T(\texttt{origin}, j+1)$ or $T(\texttt{origin}, j-1)$ in its origin. While in a naive implementation, simple reindexing operations require copying many choicemaps and greatly perturbing the model state, which causes extreme inefficiency when implementing split/merge and birth/death moves in PPLs like vanilla Gen or Stochaskell, the use of explicit open universe modeling constructs encodes enough structure for us to implement these moves very efficiently without copying data, using the object aliasing technique presented in Appendix D.

Appendix D. Automatically Implementing Inference Kernels Efficiently

In this section, we describe in more detail the technique by which we automate the implementation of inference kernels, and the optimizations we use to produce implementations performant enough to scale to real-world data sizes. In particular, we are able to exploit sparsity in the calculated Jacobian to avoid calculating the determinant of a large matrix, and exploit the conditional-independence relationships encoded in the contingent Bayesian network underlying the OUPM to efficiently calculate the state ratio $p(x')/p(x)$. The performance enabled by the latter is one of the key benefits of introducing first-class support for involutive MCMC on OUPMs.

To implement the inference kernel specified by a given proposal q and involution f, we must be able to sample and score auxiliary randomness $y \sim q_x$, produce transitions $(x, y) \to (x', y')$, and calculate acceptance probabilities by calculating $q_x(y)$, $q_{x'}(y')$, $\frac{p(x')}{p(x)}$, and $|\det \frac{\partial(x',y')}{\partial(x,y)}|$[5]. Since q is a generative function, scoring and sampling are simple; furthermore, y only contains the minimal number of random choices needed to specify the transition $x \to x'$, and thus is cheap to score. On the other hand, naive calculation of the Jacobian term and state ratio may be extremely expensive, since x may be large, for instance containing a number of variables polynomial in dataset size. As shown in algorithm 3, after sampling $y \sim q_x$, we factorize the move into 2 parts, both of which we provide efficient algorithms for:

1. Running f in an instrumented manner to compute the Jacobian term, y', and an update specification U for x
2. Computing x' and the state ratio from x and U

D.1. Efficiently calculating the Jacobian term

The involutive acceptance probability must include the absolute value of the determinant of the Jacobian matrix for the transformation of continuous values in (x, y) into those in (x', y'). Naively, this can be extremely expensive to calculate, since $|x|$ may be very large, and determinant calculation is $O(n^3)$ in matrix dimension.

Jacobian sparsity. In practice, most transformations $(x, y) \to (x', y')$ only modify the values of a small number of continuous values; all other continuous values are simply copied from (x, y) to (x', y') (either at the same address or a new one). Such a copy introduces a column into the Jacobian matrix which contains all zeros except for one 1 in the position the value was copied to. Such a column cannot effect the absolute value of the matrix's determinant, so we do not need to include rows and columns corresponding to copying in the matrix. That this means in practice is that in algorithm 2, we only track transformations of the values which are explicitly read from x or y into values explicitly written into the next state x' or reversing randomness y'; all unmentioned continuous values are simply copied without transformation. This enables us to calculate the determinant of the Jacobian matrix in $O(1)$ in the size of x for many updates, rather than $O(|x|^3)$.

5. In this section, we switch to using the notation $|\det \frac{\partial(x',y')}{\partial(x,y)}|$ to mean the same thing as $Jh_i(x, y))$ from Theorem 1, to avoid the dependency on the symbol h_i, and emphasize the idea of differentiating continuous values in (x', y') with respect to those in (x, y).

Algorithm 2 Non-standard interpretation of f for Jacobian calculation

procedure ProcessInvolution$(f, (x, y))$
  ▷ Track continuous reads and writes, manipulation moves M, and property updates U
  $(\text{Rd}, \text{Wr}, M, U, y') \leftarrow \{\}, \{\}, [], \{\}, \{\}$
  Execute f, but with
    each manipulation command m (`create`, `delete`, `change`, `split`, `merge`) $\equiv (M \leftarrow M \cup \{m\})$
    `"proposed[`k`]"` $\equiv$ (**if** $y[k]$ is continuous: $\text{Rd} \leftarrow \text{Rd} \cup \{y[k]\}$; $y[k]$)
    `"set backward[`k`] =` v`"` $\equiv$ (**if** v is continuous: $\text{Wr} \leftarrow \text{Wr} \cup \{y[k]\}$; $y' \leftarrow y' \cup \{k \mapsto v\}$)
    `"get(`$P(o_1, \ldots, o_n)[k]$`)"` $\equiv (v \leftarrow x[P(o_1, \ldots, o_n)][k]$; **if** v is continuous: $\text{Rd} \leftarrow \text{Rd} \cup \{v\}$; $v)$
    `"set` $P(o_1, \ldots, o_n)[k] = v$`"` $\equiv$ (**if** v is continuous: $\text{Wr} \leftarrow \text{Wr} \cup \{v\}$; $U[P(o_1, \ldots, o_n)][k] \leftarrow v)$
  $J \leftarrow$ uninitialized $|\text{Rd}| \times |\text{Rd}|$ matrix
  for i in $\{1, \ldots, |\text{Wr}|\}$:
    ▷ Execute f with reverse-mode AD to compute $\frac{\partial \text{Wr}[i]}{\partial \text{Rd}[j]}$ for each $j \in \{1, \ldots, |\text{Rd}|\}$
    $J[:, i] \leftarrow \nabla_{\text{Rd}}(\text{Wr}[i])$
  $U \leftarrow$ CompileManipulationMoves(U, M, x) ▷ Get variable updates to implement manipulation moves.
  return $(U, y', |\det J|)$
end procedure

Implementation via nonstandard interpretation. Algorithm 2 shows how to use nonstandard interpretation of a world involution f to calculate the Jacobian correction term. Additionally, this interpretation tracks all object manipulation moves which are called, and all `set` commands. These are compiled into an object U which fully specifies the update $x \to x'$. U is a mapping from variable names $G(o_1, \ldots, o_n)$ to partial traces $U[G(o_1, \ldots, o_n)]$, which map addresses to values. U specifies to update each trace $x[G(o_1, \ldots, o_n)]$ by overwriting addresses a in this trace with $U[G(o_1, \ldots, o_n)][a]$. We do not detail the process of compiling object manipulation moves into update specifications; it is a fairly straightforward process of determining how much each number variable must be incremented, and changing object alias associations (which are described below).

D.2. Efficiently calculating the state ratio

Because we use world models written using declarative `get` commands to mediate dependencies among variables in each world, our backend has access to information about which variables are conditionally independent in each state for any world model. Unlike in existing systems for programmable inference, which can only track dependency information in very simple models, this in principle enables us to efficiently calculate the state ratio

$$\frac{p(x')}{p(x)} = \frac{\prod_{v \in \text{vars}(x')} p(x'[v] | x'_{Pa_{x'}(v)})}{\prod_{v \in \text{vars}(x)} p(x[v] | x_{Pa_x(v)})} \tag{6}$$

To do this, we only need to consider variables which introduce terms to this ratio that do not cancel out. These are the variables which have been added to or deleted from x', variables whose traces have changed between x and x', and variables whose dependencies' values have changed.

Complications vs updating regular Bayesian networks. While world models are similar to Bayesian networks in that they contain a dependency graph among variables, they are far more general, and present novel challenges for producing efficient updates. In OUPMs, dependencies among variables are not fixed, and may change from state to state in a model, because the underlying Bayesian networks are contingent. While Milch et al. (2005b) introduced the contingent Bayesian network formalism to handle such models, they did not develop algorithms for performing arbitrary updates to these networks. Another

Algorithm 3 Efficient Involutive MCMC for World Models

procedure OUPM-IMCMC(p, q, f, x)
  $(y, q_x(y)) \leftarrow$ SampleAndScore(q, x)
  $(U, y', J) \leftarrow$ ProcessInvolution$(f, (x, y))$
  $(x', \frac{p(x')}{p(x)}) \leftarrow$ UpdateWorld(x, U)
  $q_{x'}(y') \leftarrow$ Score(q, x', y')
  $\alpha \leftarrow \frac{p(x')}{p(x)} \times q_{x'}(y') \times \frac{1}{q_x(y)} \times J$
  with probability α **return** x' **else return** x
end procedure

▷ Update a trace by executing the corresponding generative function.
procedure Execute$(v = G(o_1, \ldots, o_n), \tilde{x}, U, x)$
  $r \leftarrow 0$, $C \leftarrow$ get$(U, v, \{\})$
  $t \leftarrow$ Trace $G(o_1, \ldots, o_n)$, but with
  "$a \sim d$" $\equiv$ **if** $a \in$ vars(C): $C[a]$; **else**: $x[v][a]$
  "`get` $G'(o'_1, \ldots, o'_m) \equiv$ (
   if $(u = G'(o'_1, \ldots o'_m)) \notin$ vars$(\tilde{x})$:
    ▷ If we might have to update u, we must do it before accessing its value in v.
    $(\tilde{x}, r') \leftarrow$ Execute$(G'(o'_1, \ldots o'_m), \tilde{x}, U, x)$
    $r \leftarrow r \times r'$
   $\tilde{x}[u]$)
   if $p_{\tilde{x}}(t) = 0$: ▷ In this case, v will be pruned.
   $\tilde{x} \leftarrow x \cup \{x[v]\}$
   else:
   $\tilde{x} \leftarrow \tilde{x} \cup \{v \mapsto t\}$
   if $v \in$ vars(x):
    $r \leftarrow r \times \frac{p_{\tilde{x}}(t)}{p_x(x[v])}$
   else:
    $r \leftarrow r \times p_{\tilde{x}}(t)$
  return $(\tilde{x}, r)$
end procedure

▷ Update x using U given ord = topologicalorder$_x$.
procedure UpdateWorld$(x, U, \text{ord}, \text{children}_x)$
  $\tilde{x} \leftarrow \{\}$ ▷ New world.
  $r \leftarrow 1$ ▷ State ratio.
  pos $\leftarrow 0$ ▷ Invariant: variables topologically before pos have been updated.
  $Q \leftarrow$ priorityqueue(ord, default $= \infty$)
   for $v \in$ vars(U):
    enqueue!(Q, v)
   while $\neg$isempty(Q):
    $v \leftarrow$ pop!(Q)
    if $v \notin$ vars$(\tilde{x})$:
     ▷ Copy variables between pos and ord$[v]$ to $\tilde{x}$.
     $\tilde{x} \leftarrow \tilde{x} \cup \{u \mapsto x[u] : \text{ord}[u] \in [\text{pos}, \text{ord}[v]) \wedge u \notin \text{vars}(\tilde{x})\}$
     pos $\leftarrow$ ord$[v]$
     $(\tilde{x}, r') \leftarrow$ Execute$(v, \tilde{x}, U, x)$ ▷ Update v.
     $r \leftarrow r \times r'$
     if $v \in$ vars(x) **and** $\text{ret}_{\tilde{x}}(\tilde{x}[v]) \neq \text{ret}_x(x[v])$:
      for $u \in \text{children}_x(v)$:
       enqueue!(Q, u)
   ▷ Remove variables the observation model no longer depends on.
   $(x', \text{deleted}) \leftarrow$ prune$(\tilde{x}, U)$
   for $v \in$ deleted:
    $r \leftarrow r / p_{\tilde{x}}(\tilde{x}[v])$
    $\tilde{x} \leftarrow \tilde{x}|_{\text{vars}(\tilde{x}) \setminus v}$
   return (x', r)
end procedure

complication is that in a world model, the values of variables in the world may deterministically depend on other variables. For instance, it would be valid for the `reading` of a seismic `Detection` to be deterministically equivalent to the `magnitude` of the corresponding `Event`. In these situations, the values of variables (`readings`) may change even if no update is specified for them, since their values are deterministic functions of other updated variables (`magnitudes`). Hence, we cannot compile an update which only visit variables in the Markov blankets of variables updates are explicitly specified for, unlike in Bayesian networks. Instead, we introduce algorithm 3, which dynamically tracks changes to the dependency graph and changes to variables which require new variables to be updated. One important insight is that when we update a variable a, we must visit every variable b which depends on a to correctly calculate the state ratio. At this time, we can check whether b's value is deterministically changed by the update to a, and if so, enqueue the dependencies of b to be updated. This is implemented by procedure `UpdateWorld` , which pulls variables relevant to the state ratio or update from a queue, adding new variables to the queue as needed. However, complications arise due to the necessity of updating variables in a topological order of the dependency graph.

Update order. The order in which variables are updated is critical for the correctness of `UpdateWorld`. For instance, say $b \in Pa_{x'}(a)$ and $a, b \in \text{vars}(U)$. If we update b before we have updated a, it is impossible to calculate the factor $p(x'[b]|x'_{Pa_{x'}(b)}) = p(x'[b]|x'[a])$

in the state ratio, because the value of $x'[a]$ is not yet known. Thus, variables must be updated in the topological order induced by the graph of dependencies in the state x'. However, we cannot know the topological order of x' before the update, so we instead use the topological ordering of the old state x, with special-case logic to handle places where this order differs from that in x'. This is justified by the fact that efficient updates are only helpful in transitions $x \to x'$ producing fairly local changes, and thus any update we could implement efficiently would only change the world's topological order minorly. To implement this, the update queue Q in `UpdateWorld` is a priority queue indexed by the topological order of x. In practice, `UpdateWorld` must also update the topological order of x to one for x'; we elide these details, along with details of maintaining the children_x lookup table (which gives the children of each variable in the CBN graph) and the details of implementing the `prune` function which removes variables which are not needed for a minimal self-supporting instantiation.

We handle changes in variable order between x and x' in the `Execute` procedure, which updates variable traces. When `Execute` observes that a variable v `gets` the value of a variable u which has not yet been updated, this means that in x, u was topologically later than v, but in x', it is topologically before v. In this situation, we recursively call $\text{Execute}(u)$ to ensure that we have updated u to the proper value before it is used to update v.

Runtime. The full algorithm which accounts for topological order updating runs in $O((N + M)(\log(N + M) + T))$, where N in the number of variables which are updated or impact the state ratio, M is the number of variables whose indices in the topological ordering change during the update, and T is the maximum runtime of updating a single variable in the world. [6] So long as $N << |\text{vars}(x')|$ and M is small (both of which are true for most local updates), this algorithm is much faster than generating a new state x' from scratch, which takes $\theta(|\text{vars}(x')|T)$.

Aliased object representation. To efficiently implement object manipulation moves, which may change the origins and indices of objects, we use a special internal representation of worlds. If we stored a mapping from $G(o_1, \ldots, o_n)$ to traces in the world, then whenever an object $o_k = T(\text{origin}, j)$ had its origin or index change, we would need to update the entry $G(o_1, \ldots, T(\text{origin}, j), \ldots, o_n)$ for every property of this object. Instead, we assign each object a unique identifier by storing a lookup table from identifier i_k to object $o_k = T(\text{origin}, j)$ with each world state. We then store traces in a dictionary from keys $G(i_1, \ldots, i_n)$, where each i_k is an identifier. Index and origin changes can thus be implemented in $O(1)$ by updating entries in the lookup table, rather than $O(n)$ in the number of properties of the updated object.

6. The log terms come from the need to sort variables to update in the priority queue. Future work on static analysis of inference programs may be able to perform this sorting at compile-time in some cases.

Appendix E. Involutive MCMC with Ancestral Regeneration.

A key benefit of involutive MCMC is that it makes it feasible to rapidly prototype inference kernels, because the acceptance-probability calculations are automated. A useful development pattern is to begin with a kernel which uses generic inference techniques, like ancestral regeneration, for proposal values for most variables, and then iterate on this kernel by adding more data-driven proposals for variables for which the generic kernels are ineffective.

To facilitate this, in the involution programming language presented in Section C, we add a command `regenerate` $P(o_1, \ldots, o_n)$ to resample property P of $(o_1, \ldots, o_n)$ from the prior distribution

$$p_{P(o_1,\ldots,o_n)}(\cdot | x_{Pa_x(P(o_1,\ldots,o_n))}) \tag{7}$$

where $Pa_x(u)$ denotes the parents of variable u in the contingent Bayesian network for the open universe model the kernel operates on (with density p), in world x.

To support this command, we must modify our formulation of automated involutive MCMC. The reason for this is that in involutive MCMC, all randomness in an inference kernel must come from the proposal $y \sim q_x$; f must be a deterministic function. Therefore, we must understand the execution of `regenerate` lines in the involution program as in fact being part of the q_x proposal which produces auxiliary randomness y.

The ancestral sampling distribution. Given a partial world P containing values for some of the variables over which a contingent Bayesian network is defined, we define the *ancestral sampling* distribution r_P over the space $\mathcal{W}$ of MSSIs, which completes P into a full MSSI w. If w and P disagree on any variable v (so $w[v] \neq P[v]$), then $r_P(w) = 0$, and if they agree on all variables,

$$r_P(w) = \prod_{v \in \text{vars}(w) \setminus \text{vars}(P)} p_v(w[v] | w_{Pa_w(v)}). \tag{8}$$

An extended proposal and involution for ancestral regeneration. Say we have an involutive MCMC kernel defined by a proposal distribution $\tilde{q}_x$ and an involution f, where additional randomness is incorporated by regenerating the values of some variables. To explain this as involutive MCMC, we now define a distribution q_x using $\tilde{q}_x$ which includes the randomness from regeneration. We will describe q_x as producing tuples $(\tilde{y}, w)$, where $\tilde{y} \sim \tilde{q}_x$, and w is sampled from the ancestral resampling distribution r. Say there exists a function $\tilde{f}_1(x, \tilde{y}) = x'$, where $\tilde{x}'$, where $\tilde{x}'$ is a partial world. (This represents the user-written "involution" $\tilde{f}$ applying changes to world x, based on y, to produce a partial world, in which other variables may then be ancestrally sampled.) We can then say $w \sim r_{\tilde{f}_1(x,\tilde{y})}$. We define:

$$q_x((\tilde{y}, w)) = \tilde{q}_x(\tilde{y}) \times r_{\tilde{f}_1(x,\tilde{y})}(w) \tag{9}$$

We now define an involution $f(x, (\tilde{y}, w)) = (x', (\tilde{y}', w'))$. To do this, we must assume that in addition to having a function $\tilde{f}_1(x, \tilde{y}) = x'$, we in fact have a function $\tilde{f}(x, \tilde{y}) = (\tilde{f}_1(x, \tilde{y}), \tilde{y}')$, which also outputs reversing randomness $\tilde{y}'$. (Users write this $\tilde{f}$ in the "involution" DSL which supports regeneration.) This must have the property that

for any $x' \in \mathcal{W}$ which agrees with $\tilde{x}'$, $\tilde{f}(x', \tilde{y}') = (\tilde{x}, \tilde{y}')$ for some $\tilde{x}$ agreeing with x. (This is an involution in the case where $\tilde{x}, \tilde{x}' \in \mathcal{W}$.) Given $\tilde{f}$, we define

$$f(x, (\tilde{y}, w)) = (w, (\tilde{y}', w')) \tag{10}$$

where

$$(\tilde{x}', \tilde{y}') = \tilde{f}(x, \tilde{y}); \qquad w' = x\big|_{\text{vars}(x) \setminus \text{vars}(\tilde{f}_1(\tilde{x}', \tilde{y}'))}$$

The acceptance ratio. Using the above distribution, for $\tilde{f}$ satisfying similar partitioning properties as we require for involutions f in Theorem 1, the acceptance probability $\alpha(x, (\tilde{y}, w), x', (\tilde{y}', w'))$ is

$$\frac{p(x')}{p(x)} \times \frac{\tilde{q_{x'}}(\tilde{y}') \times r_{\tilde{f}_1(x', \tilde{y}')}(x)}{\tilde{q_x}(\tilde{y}) \times r_{\tilde{f}_1(x, \tilde{y})}(x')} \times |\det \frac{\partial \tilde{f}(x, \tilde{y})}{\partial (x, \tilde{y})}| \tag{11}$$

where $|\det \frac{\partial \tilde{f}(x, \tilde{y})}{\partial (x, \tilde{y})}|$ is the determinant of the Jacobian of the transformation of continuous values under $\tilde{f}$ at $(x, \tilde{y})$.

E.1. Automating involutive MCMC with ancestral resampling.

In section C, we introduced a language for specifying involutions on open-universe models. An involution in the language takes as arguments the current world state x and auxiliary randomness $\tilde{y} \sim \tilde{q_x}$, and produces a sequence M of object manipulation moves, and a set U of tuples (v, u), where v is a variable to update the trace for, and u specifies the updated value. It also produces the reversing randomness $\tilde{y}'$ and the Jacobian correction term J. A new state x' is constructed by applying M to the current state x, then applying all the specified updates from U. However, it is possible that the produced state x' is not a valid world model state, either because

1. x' is not minimal, in which case variables in x' are automatically deleted by our system to ensure minimality; or

2. x' is not self-supporting, in which case additional variables needed for it to be self supporting are automatically generated from the prior by our system.

Automatic deletions impose no semantic difficulties; however, when adding variables automatically, the above formulation of involutive MCMC with ancestral regeneration is needed to correctly calculate acceptance ratios.

Accumulating the state ratio. To implement an MCMC kernel, after sampling $\tilde{y} \sim \tilde{q_x}$ and calculating $(M, U, \tilde{y}', J)$, we construct state x' using a function update(x, M, U) (Algorithm 3). To calculate the acceptance ratio, we can calculate $\tilde{q}_{x'}(\tilde{y})$ and $\tilde{q}_x(\tilde{y}')$ directly, and we know J. Thus, what remains is to calculate $\frac{p(x')}{p(x)} \times \frac{r_{\tilde{f}_1(x', \tilde{y}')}(x)}{r_{\tilde{f}_1(x, \tilde{y})}(x')}$. When no values are generated via ancestral resampling, this reduces to the term $\frac{p(x')}{p(x)}$. Our implementation has the update procedure return this term as well, as it can be calculated by accumulating values while updating x to x'. We accumulate $\frac{p(x'[v]:x'_{Pa_{x'}(v)})}{p(x[v]:x_{Pa_x(v)})}$ every time the trace for variable v is updated from t_v to t'_v, $\frac{1}{p(x[v]:x_{Pa_x(v)})}$ for every variable v which is deleted, and $p(x'[v]$:

$x'_{Pa_{x'}(v)}$) for every variable which is added. In the case where some values are produced via ancestral sampling rather than the external proposal, we must account for the $\frac{r_{\tilde{f}_1(x',\tilde{y}')}(x)}{r_{\tilde{f}_1(x,\tilde{y})}(x')}$ term. By equation 8, the denominator of this is the product of terms $p_v(x'[v] : x'_{Pa_{x'}(v)})$ for variables v which are ancestral sampled in this move from x to x'. This is straightforward for the update procedure to accumulate as it generates each of these values. The numerator of this fraction is the product of $p_v(x[v] : x_{Pa_x(v)})$ terms for each variable v which is ancestral-sampled in the reverse move. To properly account for this, then, the update procedure needs to know which properties will be ancestral-sampled in the reversing move. We can thus define this procedure's signature as $(x', \frac{r_{\tilde{f}(x',\tilde{y}')}(x)}{r_{\tilde{f}(x,\tilde{y})}(x')}) = \text{update}(x, M, U, R)$ where R is the set of variables which will be regenerated in the reversing move.

Involution language extension. To provide access to R, we extend the language from section C, adding the syntax `backward[`$P(o_1, \ldots, o_n)$`] = regenerated` to specify that the reverse-direction proposal ancestral samples a value for $P(o_1, \ldots, o_n)$. One can also write `backward[properties of` $(o_1, \ldots, o_n)$`] = regenerated` to specify that all properties for any object tuple including o are regenerated by the reverse move. We also add a command `regenerate` P `of` o, to specify that the value for $P(o)$ should be ancestral-sampled to produce x', rather than copying the value for this property currently in x. (Copying from x is the default behavior when $P(o_1, \ldots, o_n)$ is present in x; ancestral sampling is the default behavior when it is not present in x.)

One way to think about the need to specify the values ancestrally sampled in the reverse move is as follows. Although users do not specify a dictionary key for values in the full auxiliary randomness y which are generated by ancestral resampling, to ensure that the full f is a valid involution, f must still write the values of these variables to y' in the reversing move. Because users don't specify a dictionary key, we provide a special syntax, `set backward[`$P(o_1, \ldots, o_n)$`] = regenerated` to automatically set the regenerated value in y' of variable $P(o_1, \ldots, o_n)$ to equal this variable's current value in x.

While the need to specify when a reverse move uses ancestral sampling adds a small cognitive burden to the user of our inference programming language, we provide dynamic error checks to ensure it is not a source of incorrectness. If users forget to include the proper `backward[...] = regenerated` statements, we can dynamically detect this error by running the reverse move and seeing which values are actually regenerated. It is thus simple to report when users have forgotten to include these commands, making it a fairly painless debugging procedure to add the proper reverse-regenerate specifications.