Incremental Computation for Efficient Programmable Inference in Probabilistic Programs

FABIAN ZAISER, Massachusetts Institute of Technology, USA
JACK CZENSZAK, Yale University, USA
MARTIN C. RINARD, Massachusetts Institute of Technology, USA
VIKASH K. MANSINGHKA, Massachusetts Institute of Technology, USA
ALEXANDER K. LEW, Yale University, USA

Inference in probabilistic programs generally requires evaluating many possible program executions to find those of high posterior density. To scale inference to large datasets, it is crucial that expensive intermediate results are shared across these many evaluations, rather than recomputed from scratch. This paper presents a new approach to realizing this sharing, based on *incremental computation*, a technique for efficiently recomputing (deterministic) program outputs when program inputs change. First, we show how expressive probabilistic programs can be compiled to deterministic ones that compute their density functions. Then, building on the incremental λ-calculus, we develop a general technique for compositionally incrementalizing expressive functional programs, and apply it to these densities. The resulting incremental densities can be used to accelerate a broad range of Monte Carlo inference algorithms, including for nonparametric models not well supported by existing systems. Furthermore, our decomposition of incremental density computation into separate density and incrementalization steps allows for modular reasoning about correctness—a key pain point in existing systems, where ad-hoc incrementalization features are a known source of soundness bugs. We develop denotational logical relations arguments for the correctness of each step independently, and implement the approach in a Julia prototype, finding that it leads to asymptotic runtime improvements in the size of the dataset on a range of models and inference algorithms.

CCS Concepts: • **Mathematics of computing** → **Probabilistic reasoning algorithms**; **Bayesian computation**; • **Theory of computation** → *Denotational semantics.*

Additional Key Words and Phrases: incremental computation, probabilistic programming, Bayesian inference

ACM Reference Format:
Fabian Zaiser, Jack Czenszak, Martin C. Rinard, Vikash K. Mansinghka, and Alexander K. Lew. 2026. Incremental Computation for Efficient Programmable Inference in Probabilistic Programs. *Proc. ACM Program. Lang.* 10, PLDI, Article 238 (June 2026), 25 pages. https://doi.org/10.1145/3808316

1 Introduction

Probabilistic programming systems promise to accelerate the practice of probabilistic modeling and inference, by providing languages for expressing rich probabilistic models, and automation for aspects of inference. Unfortunately, for many models, probabilistic programming systems produce slow inference code—in some cases, asymptotically slower than expert implementations of the same algorithms. One culprit behind this efficiency gap is insufficient *incrementalization*: inference

Authors' Contact Information: Fabian Zaiser, Massachusetts Institute of Technology, USA, fzaiser@mit.edu; Jack Czenszak, Yale University, USA, jack.czenszak@yale.edu; Martin C. Rinard, Massachusetts Institute of Technology, USA, rinard@csail.mit.edu; Vikash K. Mansinghka, Massachusetts Institute of Technology, USA, vkm@mit.edu; Alexander K. Lew, Yale University, USA, alexander.lew@yale.edu.

ACM 2475-1421/2026/6-ART238
https://doi.org/10.1145/3808316

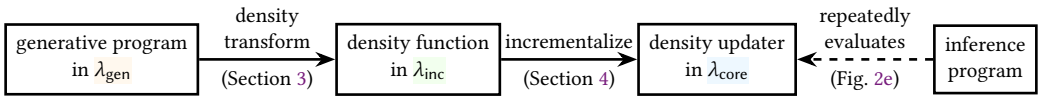


Fig. 1. Overview of our staged approach to incremental density computation for programmable inference.

can require thousands or millions of queries to the probability density function of a model, and it is crucial for scalability that expensive intermediate results are shared across these queries.

In this paper, we propose a new approach to closing this efficiency gap for a broad class of models and Monte Carlo inference algorithms, based on *incremental computation*—a technique for efficiently recomputing (deterministic) program outputs when program inputs change. As illustrated in Fig. 1, we first show how expressive probabilistic programs can be compiled into deterministic programs that compute their density functions. Then, building on the incremental λ-calculus [9, 18], we develop a general technique for compositionally incrementalizing expressive higher-order functional programs, and apply it to these density functions.

Our approach addresses two key weaknesses of existing incremental features in probabilistic programming systems [10, 14, 28, 31, 39, 45]. First, existing incrementalization is often limited to particular inference algorithms (e.g., single-site Metropolis-Hastings) or model classes (e.g., parametric models with a fixed number of random variables). By contrast, our approach treats densities as ordinary deterministic programs, and incrementalizes them with respect to arbitrary input changes. As a result, we support the incrementalization needs of a broad range of inference algorithms (Fig. 2e), including for nonparametric models such as Dirichlet process mixtures. Second, incrementalization in the presence of probability is notoriously difficult to get right, and existing implementations are brittle and prone to soundness bugs (see, e.g., Fig. 2g). Our staged approach isolates all probabilistic reasoning in the density compilation step, so that incrementalization need only be proven correct for a deterministic language. We develop new techniques for carrying out this proof: a new approach to typed incrementalization of closures, and a coinductive *updater* abstraction that encapsulates cached intermediate values and supports iterated sequences of updates, enabling a novel denotational correctness argument for caching-based higher-order incremental computation. Our approach is implemented in a Julia prototype that achieves asymptotic speedups over Gen, a state-of-the-art incrementalizing PPL, across a range of models and inference algorithms.

1.1 Motivating Example

To illustrate our approach, we consider the problem of inferring a *clustering* for a dataset of points $\{(x_i, y_i)\}_{i=1}^{n}$ in the plane (Fig. 2a). Our first step is to write a probabilistic program, gmm (Fig. 2b), encoding a *generative model*: a hypothesized data-generating process by which we imagine our dataset might have been constructed. The program gmm (for "Gaussian mixture model") accepts as input a number n of datapoints to generate, and outputs a synthetic dataset: a list of n pairs. As the simulated dataset in Fig. 2c illustrates, the key feature of this program is that it generates datasets in which points tend to cluster into a handful of ellipse-shaped components. Indeed, in each simulation, the program explicitly chooses a random number of components to generate, samples random parameters for each component, and randomly chooses a component from which to generate each datapoint. Thus, we can reframe the problem of *inferring a clustering* for our dataset as the problem of finding *executions* of the gmm program that could plausibly have generated it.

Probabilistic Programs and Traces. The gmm program consists of a sequence of *random assignment* statements x ~ t, with an identifier x on the left and a probabilistic expression t on the right (i.e., an expression whose meaning is a probability distribution that the system can sample). A *trace*

(a) Inferring clusters for given data.

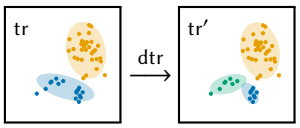
```
def gmm := λn.
  # Generate unknown number of 2D Gaussian clusters
  clusterNames ~ fresh(geometric(0.5))
  params ~ for name in clusterNames {
    cov ~ inverseWishart(NU0, PSI0)
    mean ~ normal(MU0, cov/KAPPA0)
    ret (mean, cov)   }
  # Assign each datapoint to a cluster
  weights ~ dirichlet(clusterNames, ALPHA)
  assignments ~ multinomial(weights, n)
  # Draw data points from the clusters
  data ~ for name in assignments
          using (mean, cov) := params[name] {
    sample normal(mean, cov)   }
  ret data
```

(b) Generative model as a probabilistic program in λ_{gen}. Uppercase identifiers denote numeric constants.

Density:
$9.07 \cdot 10^{-84}$

```
{clusterNames: {id73, id64, id12, id30},
 params: {id73: {mean: (1.4, 0.6), cov: ...}
          id64: ..., id12: ..., id30: ...},
 weights: {id73: 0.34, id64: 0.51,
           id12: 0.11, id30: 0.04},
 assignments: [id73, id73, id64, ...],
 data: [(1.4, 0.2), (0.7, −0.1), ...] }
```

(c) Example trace sampled from gmm and its density.




```
w, u ← incr(
      density(gmm(n)), tr)
# w = 1.36 · 10^−112
dw, u′ ← u.apply(dtr)
# dw.new = 8.39 · 10^−110
```

Trace change dtr:

```
{clusterNames: {add: {id59}},
 params: {add: {id59: {mean: (−4.4, −1.6), cov: ...}}
    change: {id85: {mean: ..., cov: ...}},
 weights: {add: {id59: 0.12},
    change: {id85: {new: 0.14, same: false}}},
 assignments: {change: [(4, {new: id59, same: false}), ...]}}
```

(d) A trace change dtr from tr (left) to tr′ (right), splitting the bottom cluster id_{85} into two (adding id_{59}). The density change dw is computed incrementally by the updater u.

Metropolis-Hastings (MH)

```
tr ← arbitrary initial trace
w, u ← incr(density(gmm(n)), tr)
while not stopped do
    dtr, h ~ proposal for changes to trace tr
    # h: Hastings correction ratio for the trace change
    dw, u′ ← u.apply(dtr)
    With probability min(1, h · dw.new/w):
      # Accept the proposal:
      tr ← apply(tr, dtr); w ← dw.new; u ← u′
```

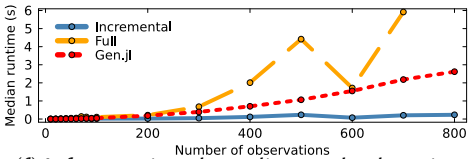
Gibbs Sampling

```
tr ← arbitrary initial trace
w, u ← incr(density(gmm(n)), tr)
while not stopped do
    Pick a variable v to resample
    Initialize maps dtr, dw, and us
    for each candidate value c do
        dtr[c] ← set v to c
        dw[c], us[c] ← u.apply(dtr[c])
    Sample c* with prob. ∝ dw[c*].new
    u ← us[c*]; w ← dw[c*].new
    tr ← apply(tr, dtr[c*])
```

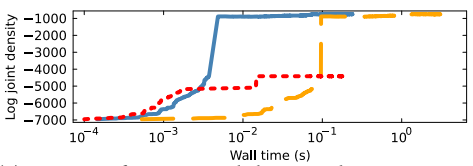
Sequential Monte Carlo (SMC)

```
tr_1, ..., tr_K ← initial traces
Initialize lists w and u
for k = 1, ..., K do
    w_k, u_k ← incr(uncurry(density(gmm)),
                    (n, tr_k))
for each data point y_j do
    Initialize lists dtr, dw, and u′
    for k = 1, ..., K do
        dn, dtr_k ← (add 1, add y_j to trace tr_k)
        dw_k, u′_k ← u_k.apply((dn, dtr_k))
    Resample u′_k, dw_k, dtr_k with prob. ∝ dw_k.new / w_k
    for k = 1, ..., K do
        w_k, tr_k, u_k ← dw_k.new, apply(tr_k, dtr_k), u′_k
    Run MH for each particle tr_k (using u_k)
```

(e) Inference algorithms using trace changes, incrementalized density functions, and updaters.



(f) Inference time depending on the data size.



(g) Wrong inference result by Gen due to an incrementalization bug (on a binary mixture model).

Fig. 2. Example: modeling and inference for a 2D Gaussian mixture model with incremental densities.

of a probabilistic program is a record of a particular execution: for each random assignment x ~ t, it stores an entry x: v, recording the choice(s) v made by t during that execution. For example, consider the trace in Fig. 2c. The first line of gmm draws a random number from a geometric distribution and generates a set of that many fresh *names*; the trace records that this set, clusterNames, was {id_{73}, id_{64}, id_{12}, id_{30}}, of size 4. Next, gmm loops over this set and for each name inside, generates a *covariance* and *mean* from particular *prior distributions*. Because this loop is over an unordered *set* of names, its trace (stored under the label "params") is an unordered *map* from names to the values sampled during the corresponding iteration of the loop. Next, the gmm program uses the dirichlet primitive to sample random *mixing weights*, resulting in a map from cluster names to real-valued weights, summing to 1. The program then uses the multinomial primitive to draw a list of n *cluster assignments* according to the mixing weights. Finally, the program loops through this list of assignments to generate the n datapoints. Each datapoint is generated using the normal primitive, with parameters determined by the component to which we have assigned the point. Because this loop is over an (ordered) list of n elements, its trace is also a list of n elements, recording the values sampled in each iteration of the loop.

Density Functions. The *density function* of a probabilistic program maps an execution trace to a non-negative score, which intuitively captures how likely the trace is. In Section 3, we present a program transformation that automatically compiles probabilistic programs to deterministic ones that compute their density functions. Roughly speaking, the density of a trace can be computed by running the program as usual, except instead of sampling from a primitive distribution, one looks up the recorded value in the trace and multiplies a running accumulator (the *weight*) by the (known) probability mass or density of the recorded value under the primitive distribution.[1] For the gmm model, the value of the density function is that it helps distinguish good clusterings from bad ones: if we fix the data entry to match our observed dataset, and vary the number of clusters, their parameters, and their assignments, the density will be low when the trace is incoherent, and high when it provides a plausible explanation of the data.

Incremental Density Computation. Inferring a plausible trace consistent with observed data typically requires the density function to be evaluated at many related traces. The goal of this paper is to accelerate inference by performing these density queries *incrementally*. In Section 4, we present a second program transformation, which compiles the density program from Section 3 into an *incremental version*. The incremental version can be run on an *initial* trace tr, to yield a density w and an *updater* u. The updater caches certain intermediate values from the computation, and thus helps to calculate the density at a slightly modified trace tr′ without full recomputation. In many cases, incrementalization delivers asymptotic speedups: in the gmm model, for example, we reduce the cost of some density queries from $O(n)$ to $O(1)$, by skipping unnecessary loop iterations.

Concretely, an updater u is applied to a *change description* dtr, which precisely encodes the delta from tr to tr′. The type of the change description is based on the type of tr (Fig. 2d). For example,

$$\mathrm{dtr} = \{\text{assignments: } \{\text{change: } [(1, \{\text{new: } \mathrm{id}_{64}, \text{same: } \textbf{false}\})]\}\}$$

is one possible change to a trace of gmm, encoding a reassignment of datapoint 1 to cluster id_{64}. Because the entry at the label assignments in tr is a list, we supply a list-change, which may specify insertions, deletions, and element modifications; in this case, we only modify element 1 of the list. Running u.**apply**(dtr) efficiently recomputes the density given this change, which only requires re-evaluating a single loop iteration (for the reassigned datapoint). The updater also returns a new updater u′, which can be used to efficiently compute densities for traces similar to tr′.

[1]This is a standard technique. However, since our language supports fresh name generation and unordered collections, there are several subtleties that complicate the derivation of density functions. We address these challenges in Section 3.

Accelerating Inference. We can exploit incremental density computation to speed up our inference algorithm of choice. Figure 2e gives pseudocode for three families of popular Monte Carlo algorithms (Metropolis-Hastings, Gibbs sampling, and sequential Monte Carlo), illustrating the role that incremental densities can play in each. We apply Metropolis-Hastings to our gmm model. The algorithm first constructs an (arbitrary) initial trace whose data field is fixed to our observed dataset. It then repeatedly proposes small changes to the trace, accepting or rejecting each change according to the resulting change in density. As Fig. 2f illustrates, incremental density computation leads to asymptotically faster evaluation of each proposal, and thus dramatically reduces the wall-clock time for inference to converge. Although existing systems, such as Gen [14], support incrementality to some extent, our approach exploits some opportunities for asymptotic speedups that existing systems fail to capture. Furthermore, due to the complexity of reasoning about incremental computation and probability, we have found some systems to have incrementalization bugs that can lead inference to silently fail (see Fig. 2g).[2] A key aim of this work is to provide a framework, based on modular program transformations and logical relations, for reasoning clearly about the correctness of incremental densities.

1.2 Our Approach

Our approach overcomes several key challenges to deliver sound, efficient incremental densities:

(C1) Combining Probabilities and Incrementality. Existing support for incrementalization in probabilistic programming systems is generally based on a direct consideration of the probabilistic language's semantics. Incremental computation in the presence of probabilities is nontrivial: even the existence of program variables can depend on random choices, which complicates the propagation of changes through the program. As a result, many existing systems rely on complex dependency tracking based on data structures inspired by graphical models [14, 29, 31, 45], which can be difficult to reason about (and empirically, is a source of several soundness bugs). Our approach isolates all probabilistic reasoning from incrementalization, by first compiling a probabilistic program to a deterministic density function. Incrementalization is then implemented (and proven correct) for only the deterministic language.

(C2) Incrementalizing Open-Universe Models. Probabilistic programs often generate latent collections of *objects* whose number is not known in advance (e.g., the clusters in the gmm program). To support efficient recomputation when objects are added and deleted (or, e.g., split and merged), traces and caches must use data structures that make such operations efficient. For example, it is crucial that gmm uses dictionaries with IDs as keys (rather than, e.g., vectors with consecutive integer indices) to store cluster information; otherwise, deleting a latent cluster would require expensive re-indexing to maintain the invariant of consecutive component IDs, inducing $O(n)$ changes to cluster assignments. To overcome this challenge, we designed our probabilistic language to support features such as fresh name generation and unordered collections. Using these features, it is possible to express nonparametric, open-universe models in a way that makes them amenable to efficient incrementalization.

(C3) Densities for Open-Universe Models. Unfortunately, the unique names and unordered collections we use to efficiently encode open-universe models violate two key assumptions behind the density calculations in most PPLs: (1) that all choices of a given type are *either* discrete *or* continuous; and (2) that any given execution trace of a program could have been generated in exactly one way by that program. The first assumption is violated because *names* are sometimes drawn from continuous distributions (to ensure that they will, with probability 1, be fresh [40]),

[2] See, e.g., https://github.com/probcomp/Gen.jl/issues/512 or https://github.com/probcomp/Gen.jl/issues/193.

and sometimes from discrete distributions (when we are choosing randomly to refer to one of a finite set of previously generated IDs). The second assumption is violated because, when an unordered set is stored in a trace, we erase information about the order in which the elements were generated. In Section 3, we develop a family of reference measures that allow us to formulate a working notion of density in this setting. We show how to compute correct densities by including (1) factorial corrections counting the number of ways in which an unordered collection could have been generated, and (2) additional "freshness checks" for names recorded in the trace.

(C4) Representing Changes. The probabilistic programming system must be able to handle a broad range of user-defined changes to capture trace updates that arise in real-world inference algorithms. We address this with a type-directed approach: the kinds of changes to a trace depend on the kinds of data in the trace. First, we use trace typing [23] to synthesize a rich static type for the traces of a probabilistic program, based on its control flow. Second, inspired by the incremental λ-calculus [9, 18, 37], we define change representations by induction on the trace type. This approach enables concise representations of several common changes that are cumbersome to express (and lead to inefficient incrementalization) in state-of-the-art PPLs.

(C5) Processing Changes Efficiently. A core technical contribution of this paper is a new approach to typed higher-order incremental computation with caches. Given a trace change, our goal is to compute a density update with the same asymptotic runtime complexity as an expert manual implementation. To our knowledge, no existing PPL achieves the best asymptotic performance for updates to nonparametric models such as Dirichlet process mixtures. The key source of high asymptotic complexity is the unnecessary re-execution of loops. Thus our algorithm is designed to precisely propagate change information about the input trace (dtr) through the density program, so that before entering a loop, we have enough information to compute precisely which loop iterations need to be re-executed. This change propagation is realized by per-primitive change propagators, and rules inspired by the caching incremental λ-calculus [18] for composing them. We develop a new approach to implementing this design in a typed, higher-order setting, which allows us to view loops themselves as combinator-like, higher-order primitives with their own built-in logic for caching, change propagation, and incrementalization.

(C6) Reasoning about Correctness. Our two-stage approach allows us to reason separately about the correctness of density computation and incrementalization; we prove each transformation correct using independent logical relations arguments over the denotational semantics of our languages. Interestingly, previous work on cache-based incrementalization [18] was formalized only in an untyped setting, because the types of caches of intermediate values quickly become unwieldy. We address this challenge by encapsulating caches in an opaque updater type $\mathsf{Upd}(\tau;\tau')$, which hides the cache type after construction. The semantics treats updaters as coinductive data types (infinite trees of all possible future update sequences) and can ignore caching concerns. This enables a modular logical relations proof over a standard, set-theoretic denotational semantics.

Contributions. In summary, we make the following contributions.

- **A new approach to typed higher-order incremental computation (Section 4):** We present a new program transformation for turning higher-order deterministic programs into *updaters* that efficiently compute changes to their outputs given changes to their inputs. Our updaters support caches of intermediate results that are encapsulated via coinductive *updater types*, giving the first typed treatment of caching incremental computation while staying within the simply-typed setting. Each update also returns a new updater to process subsequent changes, which is essential for iterative applications such as Markov chain Monte Carlo inference.

- **Incremental densities for open-universe models (Sections 2 and 3):** We present a program transformation that compiles probabilistic programs into deterministic, incrementalizable density functions. These densities are defined on typed execution traces, based on a novel extension of *trace typing* [5, 23, 24, 48] to support unordered collections and named objects. In combination with our incrementalization transformation, this enables efficient trace updates when inferring unknown collections of entities in open-universe and nonparametric models [31, 33].
- **Denotational correctness arguments (Sections 3.3 and 4.3):** We present a denotational account of the correctness of both stages of our pipeline. For the density transformation, we develop novel semantic techniques for reasoning about the densities of probabilistic programs that may generate unordered collections of names. For the incrementalization transformation, we present the first denotational argument for the correctness of caching incremental computation, which relies on a new coinduction-based reasoning principle for the correctness of updaters.
- **Empirical validation (Section 5):** We implement our approach for an embedded probabilistic DSL in Julia and compare it with Gen [14], a state-of-the-art programmable inference system implemented in the same host language. We show that our approach offers significant speedups compared to naive density evaluation and its asymptotic runtime is the same or better than Gen.

2 Probabilistic and Deterministic Languages

This section introduces the syntax and semantics of three related programming languages. As illustrated in Fig. 1, the transformations we develop in Sections 3 and 4 translate between these languages: user probabilistic programs in λ_{gen} are translated to deterministic density functions in λ_{inc}, which are further translated into incremental density functions in λ_{core}. The grammars and typing rules for all three languages are presented in Figs. 3 and 4.

2.1 Probabilistic Language

Our probabilistic language, λ_{gen}, is the user-facing language for encoding probabilistic models.

Ground Types. λ_{gen} is a variant of the simply-typed λ-calculus with the following ground types:

$$\sigma ::= \mathbb{B} \mid \mathbb{R} \mid \mathbb{N} \mid \mathsf{Name} \mid \sigma_1 \times \sigma_2 \mid \mathsf{List}\ \sigma \mid \mathsf{NameMap}\ \sigma \mid \{\ell_1 : \sigma_1, \ldots, \ell_n : \sigma_n\} \qquad (\ell_i \in \mathcal{L})$$

Each type σ is taken to denote a space of values $[\![\sigma]\!]$, and a term $\Gamma \vdash t : \tau$ denotes a map $[\![t]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$.[3] The types $\mathbb{B}$, $\mathbb{N}$, and $\mathbb{R}$ denote the standard spaces of Booleans, natural numbers, and reals respectively. To model *names*, we take $[\![\mathsf{Name}]\!] = [0, 1]$, exploiting an insight of Sabok et al. [40] that if names are interpreted as real numbers, fresh name generation can be modeled as continuous sampling (as, with probability 1, two independent samples will not be equal). Record types $\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}$, with field labels drawn from a countably infinite set $\mathcal{L}$, denote indexed products, with projections $\pi_{\ell_1}, \ldots, \pi_{\ell_n}$. The empty record $\{\}$ serves as a unit type for λ_{gen}. $\mathsf{List}\ \sigma$ denotes $\sqcup_{i \in \mathbb{N}} [\![\sigma]\!]^i$. The type $\mathsf{NameMap}\ \sigma$ denotes finite dictionaries with keys in $[\![\mathsf{Name}]\!]$ and values in $[\![\sigma]\!]$, which we represent concretely as *sorted* lists of $[\![\mathsf{Name}]\!] \times [\![\sigma]\!]$ pairs without duplicate keys (sorted by the usual order on $[\![\mathsf{Name}]\!] = [0, 1]$). We write $\mathsf{NameSet}$ as sugar for $\mathsf{NameMap}\ \{\}$. We use the syntactic sugar **if** s **then** t **else** t' for a primitive operation $\mathrm{ite}_\sigma(s, t, t')$ where t and t' have ground type σ, and $s[t]$ for a primitive $\mathrm{get}_\sigma(t, s)$ that indexes lists and maps.

Function Types. Function types in λ_{gen} are written $\sigma \to_\kappa \tau$. The annotation κ on the arrow specifies the type of data in a closure's captured environment, but does not affect the semantics:

[3] As is standard in the semantics of higher-order probabilistic programming languages, these spaces are *quasi-Borel spaces* [21], a drop-in replacement for measurable spaces with better support for function types.

$$\text{Types } \sigma, \tau ::= \mathbb{B} \mid \mathbb{N} \mid \mathbb{R} \mid \text{Name} \mid \sigma \times \tau \mid \text{List } \sigma \mid \text{NameMap } \sigma \mid \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}$$
$$\underbrace{\mid \text{Dist}_\kappa\ \tau \mid \text{P}_\kappa\ \sigma\ \tau \mid \tau \to_\sigma \tau'}_{\lambda_{\text{gen}}} \quad \underbrace{\mid \tau \to_\sigma \tau'}_{\lambda_{\text{inc}}} \quad \underbrace{\mid \tau \to \tau' \mid \text{Sub}\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \mid \text{Upd}(\tau; \tau')}_{\lambda_{\text{core}}}$$

$$\text{Terms: } s, t ::= x \mid c \mid x := s; t \mid \boldsymbol{\lambda} x.\, t \mid s\ t \mid (s, t) \mid t.1 \mid t.2$$

$$\text{in } \lambda_{\text{gen}}: \quad \mid \textbf{ret}\ t \mid x \leftarrow s; t \mid \textbf{sample}\ t \mid t\ @\ \ell$$
$$\mid \textbf{for}\ x\ \textbf{inSet}\ xs\ \textbf{using}\ (v_1 := y_1[u_1], \ldots, v_n := y_n[u_n])\ \{\ t\ \}$$
$$\mid \textbf{for}\ x\ \textbf{inRange}\ w\ \textbf{with}\ z := s\ \textbf{using}\ (v_1 := y_1[u_1], \ldots, v_n := y_n[u_n])\ \{\ t\ \}$$
$$\text{in } \lambda_{\text{inc}}: \quad \mid t.\ell \mid t|_{\{\ell_1,\ldots,\ell_n\}} \mid \textbf{for}\ x\ \textbf{in}\ xs\ \textbf{using}\ (v_1 := y_1[u_1], \ldots, v_n := y_n[u_n])\ \{\ t\ \}$$
$$\mid \textbf{for}\ x\ \textbf{in}\ xs\ \textbf{with}\ z := s\ \textbf{using}\ (v_1 := y_1[u_1], \ldots, v_n := y_n[u_n])\ \{\ t\ \}$$
$$\text{in } \lambda_{\text{core}}: \quad \mid t.\ell \mid t|_{\{\ell_1,\ldots,\ell_n\}} \mid \textbf{has}_\ell\ t \mid \textbf{mkUpd}(s; t) \mid s.\textbf{apply}(t)$$

Fig. 3. Syntax of our languages $\lambda_{\text{gen}}, \lambda_{\text{inc}}, \lambda_{\text{core}}$. Labels ℓ are from a countable set $\mathcal{L}$.

$[\![\sigma \to_\kappa \tau]\!] = [\![\sigma]\!] \Rightarrow [\![\tau]\!]$ (where $\Rightarrow$ constructs the semantic function space). For example, we have

$$y : \mathbb{R}, z : \mathbb{N} \vdash \lambda x.\ x < y : \mathbb{R} \to_{\mathbb{R}} \mathbb{B}$$

because the function $\lambda x.\ x < y$ closes over the variable y of type $\mathbb{R}$.[4] (It is admittedly unusual to track this information in the type of a function. More details will be provided in Section 4, but the intuition is that our incrementalization transformation will represent *changes* to functions of type $\sigma \to_\kappa \tau$ as changes to the data they close over, and so the type κ of this data must be explicit.)

Probabilistic Programs. Probabilistic programs in λ_{gen} have types of the form $\text{P}_\kappa\ \sigma\ \tau$: τ is the return type of the probabilistic program, and σ is its *trace type*, i.e., the data type used to represent reified traces of its execution [23]. As with function types, κ is an environment annotation, reflecting the types of any free variables captured by the probabilistic computation. (We omit it when none are captured.) For example, the generative model of Fig. 2b has type $\text{P}\ \sigma\ (\text{List}\ \mathbb{R}^2)$, where

$$\sigma = \{\text{clusterNames} : \text{NameSet}, \text{params} : \text{NameMap}\ \{\text{mean} : \mathbb{R}^2, \text{cov} : \mathbb{R}^4\},$$
$$\text{weights} : \text{NameMap}\ \mathbb{R}, \text{assignments} : \text{List Name}, \text{data} : \text{List}\ \mathbb{R}^2\}$$

Semantically, $[\![\text{P}_\kappa\ \sigma\ \tau]\!] = \text{Prob}\ [\![\sigma]\!] \times ([\![\sigma]\!] \Rightarrow [\![\tau]\!])$. That is, a probabilistic program denotes a pair (μ, f), where $\mu \in \text{Prob}\ [\![\sigma]\!]$ is a distribution on traces, and $f : [\![\sigma]\!] \to [\![\tau]\!]$ is a deterministic function mapping traces to corresponding return values. Formally, Prob X is a subset of the function space $(X \Rightarrow [0, \infty]) \Rightarrow [0, \infty]$: we identify a probability distribution μ with its *expectation operator*, which sends functions $f : X \to [0, \infty]$ to their expected values $\mathbb{E}_\mu[f]$. For example, the uniform distribution on $[0, 1]$ is $\Lambda_{[0,1]} = f \mapsto \int_0^1 f(x)\text{d}x$, and the *Dirac delta* distribution at a point x is $\delta_x = f \mapsto f(x)$. For $k : X \to \text{Prob}\ Y$, we write $\mu; k \in \text{Prob}\ Y$ for the composition $f \mapsto \mu(x \mapsto k(x)(f))$. We write $x \leftsquigarrow \mu; \nu$, where x occurs free in ν, to mean $\mu; (x \mapsto \nu)$.

Probabilistic programs are built by composing *primitive distributions* of type $\text{Dist}_\kappa\ \sigma$, where $[\![\text{Dist}_\kappa\ \sigma]\!] = \text{Prob}\ [\![\sigma]\!]$. The key constructs are:

(1) *Deterministic computation.* A probabilistic program can return a deterministically computed value using the construct **ret** t. The execution traces of deterministic programs are empty, because there are no random choices to record. For example, we have the judgments

$$\vdash \textbf{ret}\ (5 + 3, \textbf{true}) : \text{P}\ \{\}\ (\mathbb{N} \times \mathbb{B}) \qquad x : \mathbb{R} \vdash \textbf{ret}\ x < 3 : \text{P}_{\mathbb{R}}\ \{\}\ \mathbb{B}$$

[4] Technically, the typing rules compute the equivalent type $\mathbb{R} \to_{\{\}\times\mathbb{R}} \mathbb{B}$.

where on the right, the environment annotation $\kappa = \mathbb{R}$ indicates that the program closes over a variable of type $\mathbb{R}$. For $\Gamma \vdash t : \tau$ and $\gamma \in [\![\Gamma]\!]$, we have $[\![\textbf{ret}\ t]\!](\gamma) = (\delta_{\{\}}, \_ \mapsto [\![t]\!](\gamma))$.

(2) *Sampling primitive distributions.* The construct **sample** t generates and returns a sample from the primitive distribution specified by t : Dist σ. For example, **sample** normal$(0,1)$: P $\mathbb{R}\,\mathbb{R}$ samples a standard Gaussian and returns it. The trace simply records the sampled value:

$$[\![\textbf{sample}\ t]\!](\gamma) = ([\![t]\!](\gamma), \mathrm{id})$$

In addition to the standard discrete and continuous primitive distributions, λ_{gen} has a primitive fresh : Dist $\mathbb{N}$ $\to$ Dist NameSet. This primitive generates a random number n from its argument distribution,[5] then returns a set of n freshly generated names (i.e., n independent samples from the uniform distribution on $[0,1]$).

(3) *Labeling random choices.* The construct t **@** ℓ *labels* the random choice(s) made by t with $\ell \in \mathcal{L}$; the trace type of t **@** ℓ is the singleton record $\{\ell : \sigma\}$, where σ is the trace type of t. For example, we have **sample** normal$(0,1)$ **@** val : P $\{\mathrm{val} : \mathbb{R}\}\,\mathbb{R}$. Semantically,

$$[\![t\ \textbf{@}\ \ell]\!](\gamma) = (x \leftsquigarrow \mu; \delta_{\{\ell \mapsto x\}}, f \circ \pi_\ell) \qquad \text{where } (\mu, f) := [\![t]\!](\gamma)$$

(4) *Sequencing.* The construct $x \leftarrow t; s$ runs t and binds its return value to x, then runs s. The terms t and s must both be probabilistic programs with disjoint record trace types; the trace type of $x \leftarrow t; s$ is the merge of the two record types. We write $x \sim t$ as syntactic sugar for $x \leftarrow t$ **@** x (or for $x \leftarrow$ **sample** t **@** x if t : Dist σ). For example, we have the judgment

$$x : \mathbb{R} \vdash y \sim \mathrm{normal}(x,1); z \sim \mathrm{beta}(1,2); \textbf{ret}\ z < y : \mathsf{P}_{\mathbb{R}}\ \{y : \mathbb{R}, z : \mathbb{R}\}\ \mathbb{B}$$

where the trace type $\{y : \mathbb{R}, z : \mathbb{R}\}$ is the concatenation of $\{y : \mathbb{R}\}$, $\{z : \mathbb{R}\}$, and $\{\}$. Semantically,

$$[\![x \leftarrow t; s]\!](\gamma) = (\rho_1 \leftsquigarrow \mu; \rho_2 \leftsquigarrow k(f(\rho_1)); \delta_{\rho_1 +\!\!+ \rho_2}, \rho \mapsto g(f(\rho|_{L_1}))(\rho|_{L_2}))$$

where $(\mu, f) = [\![t]\!](\gamma)$, $\langle k, g\rangle = [\![s]\!](\gamma[x \mapsto -])$, and L_1 and L_2 are the label sets of the trace types of t and s. Our probabilistic program types form a *graded monad* with grade σ.[6]

(5) *Loops.* The loop **for** x **inRange** d { t }, where d : Dist $\mathbb{N}$, draws $n \sim d$, then iterates from $x := 1$ to n.[7] Its traces are lists, where each element is a trace of t, and n is implicitly recorded as the length of the list. Its return value is also a list, of the values returned by t at each iteration. Optionally, an accumulator may be initialized and updated each iteration:

for i **inRange** 5 **with** z := 0 { z′ := z + 1; y ∼ normal(z′, 0.1); **ret** (y, z′) } : P (List {y : $\mathbb{R}$}) (List $\mathbb{R}$)

The construct **for** x **inSet** d { t } loops over an unordered set of names (no accumulator). This loop draws *names* $\sim d$, and runs t (of type P $\sigma\ \tau$) once for each name, collecting the results into a NameMap τ. Its trace type is NameMap σ, as its traces store a trace of t for each iteration. The clause **using** $(v_1 := y_1[u_1]; \ldots; v_n := y_n[u_n])$ may also be added before a loop's body. This is equivalent to adding $v_1 := y_1[u_1]; \ldots; v_n := y_n[u_n];$ to the loop body, but tells the system to track the indices of the collections y_i accessed at each iteration for finer-grained incrementality.

2.2 Deterministic Languages

In addition to λ_{gen}, we have two deterministic languages: λ_{inc} (the *source language* of our incrementalization transformation) and λ_{core} (the *target language* of our incrementalization transformation). Both λ_{inc} and λ_{core} have set-theoretic denotational semantics (Figs. 5 and 6): each type τ denotes a set $[\![\tau]\!]$, and each term $\Gamma \vdash t : \tau$ denotes a function $[\![t]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$. The deterministic languages inherit the ground types of λ_{gen}.

[5] When we wish to generate a fixed number of fresh names, we write fresh(n) as sugar for fresh(dirac(n)).
[6] However, trace types σ form only a *partial* monoid, because only disjoint record types σ_1 and σ_2 can be merged.
[7] We write **for** x **inRange** n { t }, where $n : \mathbb{N}$, as shorthand for **for** x **inRange** dirac(n) { t }.

$$\frac{}{\mathsf{CtxType}(\bullet) = \{\}} \qquad \frac{\mathsf{CtxType}(\Gamma) = \tau}{\mathsf{CtxType}(\Gamma, x : \tau') = \tau \times \tau'} \qquad \frac{\Gamma, x : \tau \vdash t : \tau'}{\Gamma \vdash \boldsymbol{\lambda} x.\, t : \tau \to_{\mathsf{CtxType}(\Gamma|_{\mathsf{FV}(\lambda x.t)})} \tau'}\ (\lambda_{\mathsf{gen}}, \lambda_{\mathsf{inc}})$$

$$\frac{\Gamma \vdash s : \tau \to_\sigma \tau' \qquad \Gamma \vdash t : \tau}{\Gamma \vdash s\ t : \tau'}\ (\lambda_{\mathsf{gen}}, \lambda_{\mathsf{inc}}) \qquad \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathbf{ret}\ t : \mathsf{P}_{\mathsf{CtxType}(\Gamma|_{\mathsf{FV}(t)})}\ \{\}\ \tau}\ (\lambda_{\mathsf{gen}}) \qquad \frac{\Gamma \vdash t : \mathsf{P}_\kappa\ \sigma\ \tau}{\Gamma \vdash t\ \mathbf{@}\ \ell : \mathsf{P}_\kappa\ \{\ell : \sigma\}\ \tau}\ (\lambda_{\mathsf{gen}})$$

$$\frac{\Gamma \vdash t : \mathsf{Dist}_\kappa\ \sigma}{\Gamma \vdash \mathbf{sample}\ t : \mathsf{P}_\kappa\ \sigma\ \sigma}\ (\lambda_{\mathsf{gen}}) \qquad \frac{\begin{array}{c}\Gamma \vdash s : \mathsf{P}_\kappa\ \{\ell_1 : \sigma_1, \ldots, \ell_m : \sigma_m\}\ \tau \\ \Gamma, x : \tau \vdash t : \mathsf{P}_{\kappa'}\ \{\ell'_1 : \sigma'_1, \ldots, \ell'_n : \sigma'_n\}\ \tau' \qquad \{\ell_1, \ldots, \ell_m\} \cap \{\ell'_1, \ldots, \ell'_n\} = \emptyset\end{array}}{\Gamma \vdash x \leftarrow s; t : \mathsf{P}_{\mathsf{CtxType}(\Gamma|_{\mathsf{FV}(x \leftarrow s; t)})}\ \{\ell_1 : \sigma_1, \ldots, \ell_m : \sigma_m, \ell'_1 : \sigma'_1, \ldots, \ell'_n : \sigma'_n\}\ \tau'}\ (\lambda_{\mathsf{gen}})$$

$$\frac{\begin{array}{c}\Gamma \vdash w : \mathsf{Dist}_\kappa\ \mathbb{N} \\ \Gamma \vdash s : \rho \qquad \Gamma, x : \mathbb{N}, z : \rho \vdash y_i[u_i] : \theta_i \qquad \Gamma, x : \mathbb{N}, z : \rho, v_1 : \theta_1, \ldots, v_n : \theta_n \vdash t : \mathsf{P}_{\kappa'}\ \sigma\ (\tau \times \rho)\end{array}}{\Gamma \vdash \mathbf{for}\ x\ \mathbf{inRange}\ w\ \mathbf{with}\ z := s\ \mathbf{using}\ (v_1 := y_1[u_1], \ldots, v_n := y_n[u_n])\ \{\ t\ \} : \mathsf{P}_{\mathsf{CtxType}(\Gamma|_{\mathsf{FV}(\ldots)})}\ (\mathsf{List}\ \sigma)\ (\mathsf{List}\ \tau)}\ (\lambda_{\mathsf{gen}})$$

$$\frac{\Gamma \vdash xs : \mathsf{Dist}_\kappa\ \mathsf{NameSet} \qquad \Gamma, x : \mathsf{Name} \vdash y_i[u_i] : \theta_i \qquad \Gamma, x : \mathsf{Name}, v_1 : \theta_1, \ldots, v_n : \theta_n \vdash t : \mathsf{P}_{\kappa'}\ \sigma\ \tau'}{\Gamma \vdash \mathbf{for}\ x\ \mathbf{inSet}\ \mathsf{xs}\ \mathbf{using}\ (v_1 := y_1[u_1], \ldots, v_n := y_n[u_n])\ \{\ t\ \} : \mathsf{P}_{\mathsf{CtxType}(\Gamma|_{\mathsf{FV}(\ldots)})}\ (\mathsf{NameMap}\ \sigma)\ (\mathsf{NameMap}\ \tau')}\ (\lambda_{\mathsf{gen}})$$

$$\frac{\Gamma \vdash xs : \mathsf{List}\ \tau \qquad \Gamma \vdash s : \rho \qquad \Gamma, x : \tau, z : \rho \vdash y_i[u_i] : \theta_i \qquad \Gamma, x : \tau, z : \rho, v_1 : \theta_1, \ldots, v_n : \theta_n \vdash t : \tau' \times \rho}{\Gamma \vdash \mathbf{for}\ x\ \mathbf{in}\ \mathsf{xs}\ \mathbf{with}\ z := s\ \mathbf{using}\ (v_1 := y_1[u_1], \ldots, v_n := y_n[u_n])\ \{\ t\ \} : \mathsf{List}\ \tau'}\ (\lambda_{\mathsf{inc}})$$

$$\frac{\Gamma \vdash xs : \mathsf{NameSet} \qquad \Gamma, x : \mathsf{Name} \vdash y_i[u_i] : \theta_i \qquad \Gamma, x : \mathsf{Name}, v_1 : \theta_1, \ldots, v_n : \theta_n \vdash t : \tau'}{\Gamma \vdash \mathbf{for}\ x\ \mathbf{in}\ \mathsf{xs}\ \mathbf{using}\ (v_1 := y_1[u_1], \ldots, v_n := y_n[u_n])\ \{\ t\ \} : \mathsf{NameMap}\ \tau'}\ (\lambda_{\mathsf{inc}})$$

$$\frac{\Gamma \vdash t : \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}}{\Gamma \vdash t.\ell_i : \tau_i}\ (\lambda_{\mathsf{inc}}, \lambda_{\mathsf{core}}) \qquad \frac{\Gamma \vdash t : \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \qquad \{i_1 < \cdots < i_k\} \subseteq \{1, \ldots, n\}}{\Gamma \vdash t|_{\{\ell_{i_1}, \ldots, \ell_{i_k}\}} : \{\ell_{i_1} : \tau_{i_1}, \ldots, \ell_{i_k} : \tau_{i_k}\}}\ (\lambda_{\mathsf{inc}}, \lambda_{\mathsf{core}})$$

$$\frac{\Gamma \vdash s : \sigma \qquad \Gamma \vdash t : \tau \times \sigma \to \tau' \times \sigma}{\Gamma \vdash \mathbf{mkUpd}(s; t) : \mathsf{Upd}(\tau; \tau')}\ (\lambda_{\mathsf{core}}) \qquad \frac{\Gamma \vdash s : \mathsf{Upd}(\tau; \tau') \qquad \Gamma \vdash t : \tau}{\Gamma \vdash s.\mathbf{apply}(t) : \tau' \times \mathsf{Upd}(\tau; \tau')}\ (\lambda_{\mathsf{core}}) \qquad \frac{\Gamma \vdash t : \mathsf{Sub}\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}}{\Gamma \vdash \mathbf{has}_{\ell_i}\ t : \mathbb{B}}\ (\lambda_{\mathsf{core}})$$

Fig. 4. Selected Typing Rules. We write $\Gamma|_{\mathsf{FV}(t)}$ for the restriction of Γ to the free variables of t.

The λ_{inc} Calculus. As in λ_{gen}, λ_{inc} function types $\sigma \to_\kappa \tau$ are annotated with captured environment types κ. But in λ_{inc}, the *semantics* of a function tracks how it depends on its closed-over data:

$$[\![\sigma \to_\kappa \tau]\!] = [\![\kappa]\!] \times ([\![\kappa]\!] \times [\![\sigma]\!] \Rightarrow [\![\tau]\!])$$

The semantics of function abstraction and application explicitly manipulate these closures.

The λ_{inc} language also features a *deterministic* looping construct, **for** x **in** xs $\{\ t\ \}$, where xs may either be of type $\mathsf{List}\ \sigma$ (in which case $x : \sigma \vdash t : \tau$ and the loop returns a $\mathsf{List}\ \tau$) or of type NameSet (in which case $x : \mathsf{Name} \vdash t : \tau$ and the loop returns a $\mathsf{NameMap}\ \tau$). As in λ_{gen}, the loop supports a **using** $(v_1 := y_1[u_1], \ldots, v_n := y_n[u_n])$ clause for fine-grained tracking of the indices at which each iteration accesses the collections y_i. Iterating over lists (but not unordered collections) also supports a variant with an accumulator, introduced by the clause **with** $z := s$ before the loop body.

The λ_{core} Calculus. The semantics of λ_{core} has standard function types (without environment annotations κ). In addition to the ground types of λ_{gen}, it has *subrecord types* $\mathsf{Sub}\{\ell_1 : \sigma_1, \ldots, \ell_n : \sigma_n\}$. Values of a subrecord may assign data to only a subset of the record's labels. When accessing a label that is not present in a subrecord, we return a default value $\mathsf{default}_\tau \in [\![\tau]\!]$, chosen by induction on τ (e.g., 0 for $\mathbb{N}$, the empty map for $\mathsf{NameMap}\ \tau$, $x \mapsto \mathsf{default}_{\tau'}$ for functions $\tau \to \tau'$ etc.). The core language also has an *updater* type $\mathsf{Upd}(\tau; \tau')$. Updaters are constructed using **mkUpd**$(s; t)$ and applied using $s.$**apply**(t). We defer the discussion of these constructs to Section 4.

$$[\![\mathbb{B}]\!] = \{0,1\} \quad [\![\mathsf{Name}]\!] = [0,1] \quad [\![\mathsf{List}\ \sigma]\!] = [\![\sigma]\!]^* := \bigcup_{n\in\mathbb{N}} [\![\sigma]\!]^n \quad [\![\mathsf{NameMap}\ \tau]\!] = \{f : A \to [\![\tau]\!] \mid A \subset_{\mathsf{fin}} [\![\mathsf{Name}]\!]\}$$
$$[\![\tau \to_\kappa \tau']\!] = [\![\kappa]\!] \times ([\![\kappa]\!] \times [\![\tau]\!] \Rightarrow [\![\tau']\!]) \qquad [\![\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}]\!] = \{r : \{\ell_1, \ldots, \ell_n\} \to \bigcup_{i=1}^n [\![\tau_i]\!] \mid r(\ell_i) \in [\![\tau_i]\!]\}$$
$$[\![\Gamma]\!] = \{\gamma : \{x_1, \ldots, x_n\} \to \bigcup_{i=1}^n [\![\tau_i]\!] \mid \gamma(x_i) \in [\![\tau_i]\!]\} \text{ for contexts } \Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$$

$$[\![\lambda x.\, t]\!](\gamma) = \left((\gamma[x_1], \ldots, \gamma[x_n]), ((v_1, \ldots, v_n), v) \mapsto [\![t]\!](\gamma[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, x \mapsto v])\right)$$
$$\text{where } \Gamma|_{\mathsf{FV}(\lambda x.t)} = x_1 : \tau_1, \ldots, x_n : \tau_n$$
$$[\![s\ t]\!](\gamma) = f(env, [\![t]\!](\gamma)) \text{ where } [\![s]\!](\gamma) = (env, f)$$
$$\left[\!\!\left[\begin{array}{l}\textbf{for } x \textbf{ in } xs \textbf{ with } z := s \\ \textbf{using } (v_1 := y_1[u_1], \\ \quad \ldots, v_n := y_n[u_n])\ \{\ t\ \}\end{array}\right]\!\!\right](\gamma) = (t_1, \ldots, t_N) \text{ where } [\![xs]\!](\gamma) = (a_1, \ldots, a_N), z_0 = [\![s]\!](\gamma), \text{ and for } j = 1, \ldots, N,$$
$$(t_j, z_j) = [\![v_1 := y_1[u_1]; \ldots; v_n := y_n[u_n]; t]\!](\gamma[x \mapsto a_j, z \mapsto z_{j-1}])$$

Fig. 5. Semantics of λ_{inc} (selected rules).

$$[\![\tau \to \tau']\!] = [\![\tau]\!] \Rightarrow [\![\tau']\!] \qquad [\![\mathsf{Upd}(\tau; \tau')]\!] = [\![\tau]\!]^+ \Rightarrow [\![\tau']\!] \text{ where } S^+ = \bigcup_{n=1}^\infty S^n$$
$$[\![\mathsf{Sub}\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}]\!] = \{r : S \to \bigcup_{i=1}^n [\![\tau_i]\!] \mid S \subseteq \{\ell_1, \ldots, \ell_n\}, r(\ell_i) \in [\![\tau_i]\!]\}$$

$$[\![\textbf{mkUpd}(s; t)]\!](\gamma) = ((x_1, \ldots, x_n) \in [\![\tau]\!]^n) \mapsto y_n \text{ where } cache_1 = [\![s]\!](\gamma), f = [\![t]\!](\gamma)$$
$$\text{and } (y_i, cache_{i+1}) = f(x_i, cache_i) \text{ for } i = 1, \ldots, n$$
$$[\![s.\textbf{apply}(t)]\!](\gamma) = (f(x \in [\![\tau]\!]^1), (xs \in [\![\tau]\!]^n) \mapsto f((x, xs) \in [\![\tau]\!]^{n+1}))$$
$$\text{where } s : \mathsf{Upd}(\tau; \tau') \text{ and } f = [\![s]\!](\gamma) : [\![\tau]\!]^+ \to [\![\tau']\!] \text{ and } x = [\![t]\!](\gamma) \in [\![\tau]\!]$$

Fig. 6. Semantics of λ_{core} (selected rules).

3 Density Transformation

The first step in our pipeline is to translate a probabilistic λ_{gen} program into a deterministic λ_{inc} program that computes its trace distribution's *density function*. This isolates all probabilistic reasoning from incrementalization, which we discuss in Section 4.

3.1 Densities of Traced Programs with Name Generation

Given a program of type P σ τ, we wish to compute its *density function* of type $\sigma \to \mathbb{R}$. This function maps an execution trace, specifying values for all random choices, to a nonnegative real number, which quantifies the likelihood of the given trace. A trace is likely if the individual random choices are; the overall density is a product of the densities of each recorded primitive choice. To make these ideas precise, we require the following definition from measure theory.

Definition 3.1 (Radon-Nikodym derivative). Let μ be a probability distribution on a space X, and ν a measure on X, called the *reference measure*. A function $\frac{\mathrm{d}\mu}{\mathrm{d}\nu} : X \to \mathbb{R}$ is called a *density* or *Radon-Nikodym derivative* of μ with respect to ν if for all $f : X \to \mathbb{R}$, $\mathbb{E}_\mu[f] = \int f(x) \frac{\mathrm{d}\mu}{\mathrm{d}\nu}(x) \nu(\mathrm{d}x)$.

Here, $\mathbb{E}_\mu[f]$ denotes the expected value of f with inputs sampled from μ. Radon-Nikodym derivatives generalize the standard notions of *probability mass function* and *probability density function*: when ν is the counting measure $\#_X$ on a set X, $\frac{\mathrm{d}\mu}{\mathrm{d}\nu}$ is μ's mass function, and when ν is the Lebesgue measure $\Lambda_\mathbb{R}$, it is μ's density function. Probabilistic programs generally make both discrete and continuous choices, so their traces are typically heterogeneous products. The standard approach to thinking about densities of such programs is to use inductively defined, type-indexed reference measures ν_σ, designed so that the overall density will be a product of the mass functions of all discrete choices and the density functions of all continuous choices:

$$\nu_\mathbb{R} = \Lambda_\mathbb{R} \qquad \nu_\mathbb{B} = \#_\mathbb{B} \qquad \nu_\mathbb{N} = \#_\mathbb{N} \qquad \nu_{\sigma\times\tau} = \nu_\sigma \otimes \nu_\tau \qquad \ldots$$

```
def program :=
  n ~ poisson(3);
  opts ~ fresh(n + 1);

  xs ~ for _ inSet opts {
    sample normal(0, 1)
  };
  chosen ~ unif(opts);
  ret xs[chosen]
```

```
def density := λ(t, U).
  w1 := poissonD(3)(t.n, U);
  w2 := freshD(t.n + 1)(t.opts, U);
  U := U ∪ t.opts;
  w3 := product(for o in t.opts using (x := t.xs[o]) {
    normalD(0, 1)(x, U)
  });
  w4 := unifD(t.opts)(t.chosen, U);
  (t.xs[t.chosen], w1 · w2 · w3 · w4)
```

$$\frac{\mathrm{d}\mu}{\mathrm{d}\nu_\emptyset^\sigma}(t) = \mathrm{Pois}(t.\mathrm{n};3) \cdot \underbrace{(t.\mathrm{n}+1)! \cdot \mathbf{1}[|t.\mathrm{opts}| = t.\mathrm{n}+1]}_{\mathrm{freshD}(t.\mathrm{n}+1)(t.\mathrm{opts},\emptyset)} \cdot \prod_{o \in t.\mathrm{opts}} \mathcal{N}(t.\mathrm{xs}[o];0,1) \cdot \underbrace{\frac{\mathbf{1}[t.\mathrm{chosen} \in t.\mathrm{opts}]}{t.\mathrm{n}+1}}_{\mathrm{unifD}(t.\mathrm{opts})(t.\mathrm{chosen},t.\mathrm{opts})}$$

Fig. 7. Computing the density of a probabilistic program. In this figure, distD denotes the density function of a distribution dist, and σ (used in $\nu_\emptyset^\sigma$) is the trace type of program.

$$\begin{array}{lcl@{\qquad}lcl}
\nu_U^\sigma & = & \#_{[\![\sigma]\!]} \text{ for } \sigma \in \{\{\}, \mathbb{B}, \mathbb{N}\} & \nu_U^{\mathbb{R}} & = & \Lambda_{\mathbb{R}} \\
\nu_U^{\mathsf{Name}} & = & \#_U + \Lambda_{[0,1]} & \nu_U^{\sigma\times\tau} & = & a \leftsquigarrow \nu_U^\sigma; b \leftsquigarrow \nu_{U\cup\mathrm{names}(a)}^\tau; \mathsf{ret}\ (a,b) \\
\nu_U^{\mathsf{List}\ \sigma} & = & \sum_{n=0}^\infty \nu_U^{\sigma^n} & \nu_U^{\mathsf{NameMap}\ \sigma} & = & \sum_{n=0}^\infty \nu_U^{(\mathsf{Name}\times\sigma)^n}
\end{array}$$

Fig. 8. Indexed families of reference measures, defined inductively on the ground types of λ_{gen}.

Our setting, however, is complicated by the presence of *names*. Semantically, $[\![\mathsf{Name}]\!]$ is the real interval $[0,1]$. However, whether a randomly chosen name is a *continuous* or a *discrete* choice (i.e., whether we should use a *density* or *mass* function for its likelihood) varies from program to program. For example, consider the program in Fig. 7, which generates several fresh names and later chooses one of them uniformly at random. The fresh names (at address opts) are continuous samples, but the selected name (at address chosen) is a discrete choice. Thus, the trace density should have as factors both the *density* of the fresh primitive,[8] and the *mass* of the unif primitive.

Our inductively defined reference measures, in Fig. 8, formalize this intuition. The reference measures are indexed by both a type σ and a *support* U, which is a countable subset of $[\![\mathsf{Name}]\!]$. The reference measure ν_U^{Name} is a *sum* of $\Lambda_{[0,1]}$ and $\#_U$, indicating that randomly chosen names may either be continuous choices (i.e., freshly generated), or discrete choices from the support set U of available names. The reference measure for product types makes the names that appear in the first component ($\mathrm{names}(a)$) available as new atoms for discrete choices in the second component.[9] These definitions give us a sufficiently general notion of *density* to handle all programs in λ_{gen}:

Lemma 3.2. *Let* $\vdash t : \mathsf{P}\ \sigma\ \tau$ *in* λ_{gen}. *Then* $[\![t]\!]_1$ *has a density* $\frac{\mathrm{d}[\![t]\!]_1}{\mathrm{d}\nu_\emptyset^\sigma} : [\![\sigma]\!] \to \mathbb{R}$ *with respect to* $\nu_\emptyset^\sigma$.

3.2 Density Transformation

We now present a program transformation, $\mathcal{W}\{-\}$, which mechanically translates probabilistic programs into deterministic ones computing their density functions. More precisely, we translate a program of type $\mathsf{P}\ \sigma\ \tau$ into a deterministic program with two inputs and two outputs, as illustrated in Fig. 7. The inputs are a *trace* t and a set of previously generated names U. The outputs are a return value—the value returned by the program when its random choices are fixed to those recorded in t—and a nonnegative real, the density of the program with respect to ν_{U}^σ at the trace t.

[8] The distribution over sets of n fresh names is the uniform distribution on $\{(x_1,\ldots,x_n) \in [0,1]^n \mid x_1 < \cdots < x_n\}$, a set with volume $\frac{1}{n!}$. Thus the density at any point within this set is $1/\frac{1}{n!} = n!$, and the density at other points is 0.

[9] The order of products is arbitrary, in that for all ground types σ, τ, we have $\nu_U^{\sigma\times\tau} = \mathrm{swap}_*\, \nu_U^{\tau\times\sigma}$ (proven in Appendix B.1).

$$\begin{array}{rcl@{\qquad}rcl} \mathcal{W}\{\sigma\} & = & \sigma \text{ for ground types } \sigma & \mathcal{W}\{\sigma \times \tau\} & = & \mathcal{W}\{\sigma\} \times \mathcal{W}\{\tau\} \\ \mathcal{W}\{\mathsf{P}_\kappa\, \sigma\, \tau\} & = & \mathcal{W}\{\sigma\} \times \mathsf{NameSet} \to_{\mathcal{W}\{\kappa\}} \mathcal{W}\{\tau\} \times \mathbb{R} & \mathcal{W}\{\sigma \to_\kappa \tau\} & = & \mathcal{W}\{\sigma\} \to_{\mathcal{W}\{\kappa\}} \mathcal{W}\{\tau\} \end{array}$$

Fig. 9. Density transformation from λ_{gen} to λ_{inc} on types (selected rules).

$$\mathcal{W}\{c\} = c_{\mathsf{inc}} \qquad \mathcal{W}\{\mathbf{ret}\ t\} = \boldsymbol{\lambda}(\mathsf{tr}, \mathsf{U}).\, (\mathcal{W}\{t\}, 1) \qquad \mathcal{W}\{\mathbf{sample}\ t\} = \boldsymbol{\lambda}(\mathsf{tr}, \mathsf{U}).\, (\mathsf{tr}, (\mathcal{W}\{t\}\ (\mathsf{tr}, \mathsf{U})).2)$$

$$\mathcal{W}\{t\ @\ \ell\} = \boldsymbol{\lambda}(\mathsf{tr}, \mathsf{U}).\, \mathcal{W}\{t\}\ (\mathsf{tr}.\ell, \mathsf{U})$$

$$\mathcal{W}\{x \leftarrow t_1; t_2\} = \boldsymbol{\lambda}(\mathsf{tr}, \mathsf{U}).\, (x, w_1) := \mathcal{W}\{t_1\}\ (\mathsf{tr}|_{L_1}, \mathsf{U}); (y, w_2) := \mathcal{W}\{t_2\}\ (\mathsf{tr}|_{L_2}, \mathsf{U} \cup \mathsf{names}(\mathsf{tr}|_{L_1})); (y, w_1 \cdot w_2)$$

where $t_1 : \mathsf{P}\ \{\ell : \sigma_\ell \mid \ell \in L_1\}\ \tau_1$ and $t_2 : \mathsf{P}\ \{\ell : \sigma'_\ell \mid \ell \in L_2\}\ \tau_2$

$$\mathcal{W}\left\{\begin{array}{l}\mathbf{for}\ x\ \mathbf{inRange}\ d : \mathsf{Dist}\ \mathbb{N}\ \mathbf{with}\ z := s \\ \mathbf{using}\ (v_1 := y_1[u_1], \ldots, v_n := y_n[u_n])\ \{\ t\ \}\end{array}\right\} = \left(\begin{array}{l}\boldsymbol{\lambda}(\mathsf{tr}, \mathsf{U}).\, \mathsf{N} := \mathsf{length}\ \mathsf{tr};\ (\_, \mathsf{w_N}) := \mathcal{W}\{d\}\ (\mathsf{N}, \mathsf{U}); \\ \quad \mathsf{ys} := \mathbf{for}\ x\ \mathbf{in}\ \mathsf{range}\ \mathsf{N}\ \mathbf{with}\ (z, \mathsf{V}) := (\mathcal{W}\{s\}, \mathsf{U}) \\ \quad\quad \mathbf{using}\ (v_1 := y_1[\mathcal{W}\{u_1\}], \ldots, v_n := y_n[\mathcal{W}\{u_n\}], \\ \quad\quad\quad \mathsf{tr}' := \mathsf{tr}[x])\ \{\ ((r, z), w) := \mathcal{W}\{t\}\ (\mathsf{tr}', \mathsf{V}); \\ \quad\quad\quad ((r, w), (z, \mathsf{V} \cup \mathsf{names}(\mathsf{tr}')))\ \}; \\ \quad (\mathsf{rs}, \mathsf{ws}) := \mathsf{unzip}\ \mathsf{ys};\ (\mathsf{rs}, \mathsf{w_N} \cdot \mathsf{product}\ \mathsf{ws})\end{array}\right)$$

Fig. 10. Density transformation from λ_{gen} to λ_{inc} on terms (selected rules).

Intuitively, the density program works by executing the same logic as the original probabilistic program. When a primitive random choice is encountered, we look up the corresponding value in the trace. We use the primitive distribution's built-in density (passing in the choice's value and the current support set U of names) to compute its likelihood. When we find new names in the trace, we add them to U.[10] Then we compute the return value and the product of all primitive densities.

This algorithm is formalized as a program transformation in Figs. 9 and 10. Given an open term $\Gamma \vdash t : \tau$ in λ_{gen}, we translate it to a term $\mathcal{W}\{\Gamma\} \vdash \mathcal{W}\{t\} : \mathcal{W}\{\tau\}$ in the target language, λ_{inc}. Because the density function largely follows the same logic as the original probabilistic program, $\mathcal{W}\{-\}$ is defined to "pass through" many constructs, sending tuples to tuples, projections to projections, abstractions to abstractions, and applications to applications (preserving the environment annotations κ on higher types). The interesting cases are how it handles probabilistic program terms. Deterministic programs **ret** t have constant density 1 at the empty trace. The density of a program that makes a single primitive random sample, **sample** t, is just the density of the primitive distribution t. Compound programs $x \leftarrow t_1; t_2$ are handled by separately computing the density of t_1 (on the subtrace containing the choices made by t_1) and the density of t_2 (on the subtrace containing the choices made by t_2), then taking their product. (Any new names appearing in the trace of t_1 are also unioned into the running support set U, which is passed to the density for t_2. This reflects that fresh names generated by t_1 may occur non-fresh, as discrete choices, in t_2.)

Note that loops in λ_{gen} are translated into more elaborate loops in λ_{inc}. The target-language loops iterate over a collection of subtraces, computing the density of each subtrace under the loop body while maintaining a growing set of used names V. These densities are then multiplied together—along with the density of the distribution used to decide how many iterations to run, if this was a random choice—to yield a density for the overall loop.

3.3 Correctness

The density transformation is correct in the following sense:

[10]Tracking the support U allows the density function to return 0 when multiple purportedly fresh names in a trace are identical. In particular, the built-in density $\mathsf{freshD}(n)(\mathsf{names}, \mathsf{U})$ is either $n!$ (if $|\mathsf{names}| = n$ and $\mathsf{names} \cap \mathsf{U} = \emptyset$), or 0 otherwise. This logic is necessary for the density to be technically correct, but properly implemented inference algorithms should never consider traces that trigger freshness violations. As such, much like array bounds checking, support tracking is a feature that can be turned off for performance after the user is confident that inference is correctly implemented.

Theorem 3.3. *Let t be a closed λ_{gen} term of type* $\mathsf{P}\,\sigma\,\tau$*, with trace distribution* $\mu = [\![t]\!]_1$*. Then for all traces* $\rho \in [\![\sigma]\!]$*,* $\frac{\mathrm{d}\mu}{\mathrm{d}\nu_\emptyset^\sigma}(\rho) = [\![(\mathcal{W}\{t\}\ (x, \emptyset)).2]\!](x \mapsto \rho)$.

That is, running the density program on a trace ρ and the empty name set $\emptyset$, then extracting the second component of the output, does yield the correct density of the program's trace distribution, evaluated at ρ. This result is established by a logical relations argument, developed in Appendix B.

4 Incrementalization

We now describe a program transformation $\mathcal{I}\{-\}$ for incrementalizing a deterministic program. The incremental version of a function $f : \sigma \to \tau$ (on ground types σ and τ) has type

$$f' : \sigma \to \tau \times \mathsf{Upd}(\Delta(\sigma); \Delta(\tau))$$

Like f, it accepts an argument of type σ, and computes an initial output of type τ. Additionally, it returns an *updater*, which encapsulates a cache of intermediate results and can thus efficiently compute output changes given changes to the input. Concretely, let $(y_1, u_1) := f'(x_1)$ and dx_1 an input change from x_1 to x_2. Then $u_1.\mathbf{apply}(dx_1) =: (dy_1, u_2)$ yields an output change dy_1 from $y_1 := f(x_1)$ to $y_2 := f(x_2)$, and a new updater u_2 for changes on top of x_2. This allows us the flexibility to freely stack input changes and compute the corresponding output changes.

The key idea of the transformation is that each primitive operation (e.g., arithmetic operations and list/map operations) has an incremental version as above. Our program transformation composes these incremental primitives and wires all the change information and updaters together. Even if a given primitive does not benefit from incrementalization itself, it is useful to propagate information about its output changes to downstream computations. This is most valuable for loops, enabling us to rerun as few loop iterations as possible, which can yield asymptotic speedups.

4.1 Change Types and Updaters

We associate every type τ with a change type $\Delta(\tau)$, which represents changes to values of type τ (Fig. 11). For basic types like $\mathbb{R}$ or $\mathbb{B}$, a change is simply a new value together with a flag indicating whether the value has changed. For product types, we make a distinction between pairs and records: a change to a pair is a pair of changes to the components, whereas a change to a record is a subrecord of changes to the fields (omitting unchanged fields). Changes to collections (lists and maps) store changes to the elements at specific indices or keys and insertions/deletions of elements. For closures, we do not allow the function itself ("its code") to change, only its captured environment, to keep updates efficient. Finally, changes to contexts are simply changes to the variables in the context. The change types are designed to strike a balance between keeping change representations small and including enough information to avoid caching. For instance, changes to atomic ground types include the new value, so operations do not need to cache their arguments.

Updaters. Updaters encapsulate cached intermediate values and the logic for using them to efficiently recompute an output. We define $\mathsf{Upd}(\Delta(\tau); \Delta(\tau'))$ as an *opaque* type with a constructor $\mathbf{mkUpd}(\mathsf{cache}; \mathsf{update})$ accepting an initial cache $: \sigma$ and a function $\mathsf{update} : \Delta(\tau) \times \sigma \to \Delta(\tau') \times \sigma$, which takes an input change and a cache and returns an output change and a new cache. Note that updater types hide the cache type σ after construction, like existential types. This is an innovation over previous work, and enables type-directed reasoning about updaters.

To use an updater u, one applies it to a change dx, written $\mathsf{u}.\mathbf{apply}(\mathsf{dx}) =: (\mathsf{dy}, \mathsf{u}')$, which yields an output change dy and a new updater u' that can be used to compute output changes on top of dx. Updater construction and application satisfy the following program equivalence:

$$\mathbf{mkUpd}(\mathsf{cache}; \mathsf{upd}).\mathbf{apply}(\mathsf{dx}) = \{(\mathsf{dy}, \mathsf{cache}') := \mathsf{upd}\ (\mathsf{dx}, \mathsf{cache}); (\mathsf{dy}, \mathbf{mkUpd}(\mathsf{cache}'; \mathsf{upd}))\}$$

Semantics of Updaters. Semantically, updaters are treated as coinductive data types:

$$\textit{Intuition:}\quad \textbf{codata}\ \mathsf{Upd}(\Delta(\tau);\Delta(\tau')) \approx \Delta(\tau) \to \Delta(\tau') \times \mathsf{Upd}(\Delta(\tau);\Delta(\tau'))$$

Thus its denotation is an infinite-depth tree whose edges are labeled by values of $\Delta(\tau)$ and nodes by values of $\Delta(\tau')$ (see Fig. 6). (Category theoretically, this is the final coalgebra of the functor $F(X) = [\![\Delta(\tau)]\!] \Rightarrow [\![\Delta(\tau')]\!] \times X$.) Formally, we represent such trees as functions $[\![\Delta(\tau)]\!]^+ \to [\![\Delta(\tau')]\!]$ from non-empty lists of $\Delta(\tau)$ (representing the path in the tree) to $\Delta(\tau')$ (the value at the node reached at the end of that path). Note that the idea of recursive updaters was mentioned by Morihata [37], but to our knowledge has not been formalized before.

The constructor **mkUpd**(cache; update) builds a tree where the value of the node y_n reached along the path $(x_1, \ldots, x_n)$ results from iteratively applying update to the inputs $x_1, \ldots, x_n$, together with the most recent cache, starting from the given cache (see Fig. 6). The application u.**apply**(dx) returns the node value in u at the path with single edge dx, and the updater rooted at that node.

4.2 Program Transformation

Type Transformations. Our program transformation $\mathcal{I}\{-\}$ behaves like the identity on types (see Fig. 11), except for closure types, which are translated into incremental versions

$$\mathcal{I}\{\sigma \to_\kappa \tau\} := \mathcal{I}\{\sigma\} \to \mathcal{I}\{\tau\} \times \mathsf{Upd}(\Delta(\kappa \times \sigma); \Delta(\tau))$$

with updaters that can process both input changes and changes to captured variables in κ.

Primitives. For every constant $c : \tau$ in λ_{inc}, we assume a corresponding incrementalized version $c_{\mathsf{core}} : \mathcal{I}\{\tau\}$ in λ_{core}. For instance, consider the function length : List $\mathbb{N} \to \mathbb{N}$ in λ_{inc}. Its incremental version caches the length and constructs an updater with an update function updateLength:[11]

```
def length_core := λxs. len := length(xs); (len, mkUpd(len; updateLength))
def updateLength := λ(dxs, cachedLen).
    len := cachedLen + length(dxs.insert) − length(dxs.remove);
    ({new: len, same: len == cachedLen}, len)
```

The function updateLength computes the new length len taking insertions and deletions into account. It returns the length change of type $\Delta(\mathbb{N})$ together with the new length to be cached.

Composition. Suppose we have a function evens : List $\mathbb{N} \to$ List $\mathbb{N}$ that filters out odd numbers from a list and we compose it with the length function: countEvens(xs) := length(evens(xs)). Intuitively, to incrementalize it, we can take the incremental versions of the two functions and create a new updater that uses the component updaters as its cache:[12]

```
def countEvens_core := λ(xs).
    (ys, u_evens) := evens_core(xs);
    (len, u_length) := length_core(ys);
    (len, mkUpd((u_evens, u_length); updateCE))

def updateCE := λ(dxs, (u_evens, u_length)).
    (dy, u'_evens) := u_evens.apply(dxs);
    (dlen, u'_length) := u_length.apply(dy);
    (dlen, (u'_evens, u'_length))
```

Our program transformation automates this logic. A term $\Gamma \vdash t : \tau$ in λ_{inc} is translated to a term $\mathcal{I}\{\Gamma\} \vdash \mathcal{I}_\Gamma\{t\} : \mathcal{I}\{\tau\} \times \mathsf{Upd}(\Delta(\mathsf{CtxType}(\Gamma)); \Delta(\tau))$ in λ_{core} as defined in Fig. 14. The idea is that we must not only be able to evaluate a given term, but also to obtain changes to its evaluation depending on environment changes. To handle such changes in an updater, we reify them as values of type $\Delta(\mathsf{CtxType}(\Gamma))$: tuples of changes to the variables in Γ (using CtxType(−) from Fig. 4).

[11] We omit the empty environment change $\Delta(\kappa) = \Delta(\{\}) = \mathsf{Sub}\{\}$ for simplicity.
[12] This is a simplified version of what our program transformation produces; we again ignore empty environments.

$$\mathcal{I}\{\sigma\} = \sigma \text{ for } \sigma \text{ ground} \qquad \mathcal{I}\{\sigma \times \tau\} = \mathcal{I}\{\sigma\} \times \mathcal{I}\{\tau\} \qquad \mathcal{I}\{\sigma \to_\kappa \tau\} = \mathcal{I}\{\sigma\} \to \mathcal{I}\{\tau\} \times \mathsf{Upd}(\Delta(\kappa \times \sigma); \Delta(\tau))$$
$$\mathcal{I}\{\Gamma\} = (x_1 : \mathcal{I}\{\sigma_1\}, \ldots, x_n : \mathcal{I}\{\sigma_n\}) \text{ for contexts } \Gamma = (x_1 : \sigma_1, \ldots, x_n : \sigma_n)$$
$$\Delta(\sigma) = \{\mathsf{new} : \sigma, \mathsf{same} : \mathbb{B}\} \quad \text{for } \sigma = \mathbb{B}, \mathbb{N}, \mathbb{R}, \mathsf{Name} \qquad \Delta(\sigma \times \tau) = \Delta(\sigma) \times \Delta(\tau)$$
$$\Delta(\mathsf{List}\ \sigma) = \{\mathsf{insert} : \mathsf{List}\ (\mathbb{N} \times \mathcal{I}\{\sigma\}), \mathsf{remove} : \mathsf{List}\ \mathbb{N}, \mathsf{change} : \mathsf{List}\ (\mathbb{N} \times \Delta(\sigma))\}$$
$$\Delta(\mathsf{NameMap}\ \sigma) = \{\mathsf{add} : \mathsf{NameMap}\ \mathcal{I}\{\sigma\}, \mathsf{remove} : \mathsf{NameSet}, \mathsf{change} : \mathsf{NameMap}\ \Delta(\sigma)\}$$
$$\Delta(\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}) = \mathsf{Sub}\{\ell_1 : \Delta(\tau_1), \ldots, \ell_n : \Delta(\tau_n)\} \qquad \Delta(\sigma \to_\kappa \tau) = \Delta(\kappa)$$
$$\Delta(\Gamma) = (x_1 : \Delta(\sigma_1), \ldots, x_n : \Delta(\sigma_n)) \text{ for contexts } \Gamma = (x_1 : \sigma_1, \ldots, x_n : \sigma_n)$$

Fig. 11. Incrementalizing transformation $\mathcal{I}\{-\}$ from λ_{inc} to λ_{core} on types and change types $\Delta(-)$.

$$\pi_{\Gamma \mapsto x} : \mathsf{CtxType}(\Gamma) \to \sigma \quad (\text{assuming } (x : \sigma) \in \Gamma) \qquad \pi_{\Gamma \mapsto xs} : \mathsf{CtxType}(\Gamma) \to \mathsf{CtxType}(\Gamma|_{xs})$$
$$\pi_{\Gamma, x:\tau \mapsto x} = \boldsymbol{\lambda}(\_, x).\, x \qquad \pi_{\bullet \mapsto \emptyset} = \boldsymbol{\lambda}\_.\, \{\}$$
$$\pi_{\Gamma, y:\tau \mapsto x} = \boldsymbol{\lambda}(\gamma, \_).\, \pi_{\Gamma \mapsto x}(\gamma) \text{ if } y \neq x \qquad \pi_{(\Gamma, x:\tau) \mapsto xs} = \begin{cases} \boldsymbol{\lambda}(\gamma, v).\, (\pi_{\Gamma \mapsto xs \setminus \{x\}}(\gamma), v) & \text{if } x \in xs \\ \boldsymbol{\lambda}(\gamma, \_).\, \pi_{\Gamma \mapsto xs}(\gamma) & \text{otherwise} \end{cases}$$

Fig. 12. Functions to project out variables from contexts and tuples.

$\mathsf{same}_\tau : \Delta(\tau)$	Trivial change	(only supported for some τ)
$\mathsf{sameAs}_\tau : \tau \to \Delta(\tau)$	Trivial change at a particular value	(for ground types τ)
$\mathsf{apply}_\tau : \tau \times \Delta(\tau) \to \tau$	Apply a change to a value	(for ground types τ)

Fig. 13. Additional primitives involving changes, needed for incrementalization.

$$\mathcal{I}_\Gamma\{x\} = (x, \mathbf{mkUpd}(\{\}; \boldsymbol{\lambda}(d\gamma, \_).\, (\pi_{\Delta(\Gamma) \mapsto x}\ d\gamma, \{\})))$$
$$\mathcal{I}_\Gamma\{\boldsymbol{\lambda}x.\, t\} = (\boldsymbol{\lambda}x.\, \mathcal{I}_{\Gamma|_{\mathsf{FV}(\boldsymbol{\lambda}x.t)}, x:\sigma}\{t\},\ \mathbf{mkUpd}(\{\}; \boldsymbol{\lambda}(d\gamma, \_).\, (\pi_{\Delta(\Gamma) \mapsto \mathsf{FV}(\boldsymbol{\lambda}x.t)}\ d\gamma, \{\})))$$
$$\mathcal{I}_\Gamma\{s\ t\} = (f, u_f) := \mathcal{I}_\Gamma\{s\};\ \ (x, u_x) := \mathcal{I}_\Gamma\{t\};\ \ (y, u_y) := f\ x;$$
$$\big(y,\ \mathbf{mkUpd}(\ (u_f, u_x, u_y);\ \boldsymbol{\lambda}(d\gamma, (u_f, u_x, u_y)).\ \ (df, u'_f) := u_f.\mathbf{apply}(d\gamma);$$
$$(dx, u'_x) := u_x.\mathbf{apply}(d\gamma);\ \ (dy, u'_y) := u_y.\mathbf{apply}((df, dx));\ \ (dy, (u'_f, u'_x, u'_y))))\big)$$
$$\mathcal{I}_\Gamma\{t.\ell\} = (r, u_r) := \mathcal{I}_\Gamma\{t\};\ (r.\ell, \mathbf{mkUpd}((r.\ell, u_r); \boldsymbol{\lambda}(d\gamma, (r_l, u_r)).\ \ (dr, u'_r) := u_r.\mathbf{apply}(d\gamma);$$
$$\mathbf{if\ has}_\ell\ \mathsf{dr}\ \mathbf{then}\ (\mathsf{dr}.\ell, (\mathsf{apply}(r_l, \mathsf{dr}.\ell), u'_r))\ \mathbf{else}\ (\mathsf{sameAs}(r_l), (r_l, u'_r))))$$

Fig. 14. Incrementalizing transformation from λ_{inc} to λ_{core} on terms (selected rules).

The transformation rule for variables (Fig. 14) yields a cache-less updater that simply projects out the variable change from the environment change (using $\pi_{\Gamma \mapsto x}$ from Fig. 12). For abstractions $\boldsymbol{\lambda}x.\, t : \sigma \to_\kappa \tau$, we translate the body t in the environment that includes exactly the free variables, which yields an updater $\mathsf{Upd}(\Delta(\kappa \times \sigma); \Delta(\tau))$ for the closure change $\Delta(\kappa)$ and the input change $\Delta(\sigma)$. For applications $s\ t : \tau$, the translation of s yields an updater for closure changes and t for input changes, which are composed to yield an updater for the application of the closure. For record access $t.\ell : \tau$, we translate the term t to an updater for changes to the record. However, such a change is a subrecord that may not contain the field ℓ. If a trivial change same_τ exists (e.g., $\{\}$ for record types), we can use it to obtain the change to the field. Otherwise, we have to cache the previous value r_l and use the trivial change $\mathsf{sameAs}(r_l)$ if ℓ is not in the record change (see Fig. 13).

Incrementalizing Loops. A loop **for** x **in** xs { t } can be viewed as a sophisticated, higher-order primitive $\mathsf{loopCombinator}(xs, \boldsymbol{\lambda}x.\, t)$ that takes a collection xs and a loop body t and returns a loop result. Such a combinator-like primitive can be equipped with its own (complex) incremental version. To minimize recomputation, the loop combinator caches the values produced at each iteration and maintains mappings between collection indices and the iterations that accessed

```
function ℐ_Γ{for x : τ in xs { t }}
    results ← [ ];
    upds ← [ ];
    xs ← xs
    for i ← 1 to length(xs) do
        x ← xs[i]
        (y, u_y) ← ℐ_{Γ,x:τ}{t}
        push(results, y)
        push(upds, u_y)
    cache ← {inputs: xs, upds: upds}
    return (results, mkUpd(cache; Update))

            Cache type:
    {   inputs: List τ,
        upds: List Upd(τ; τ')    }
```

```
function Update(dγ, cache)
    dxs ← π_{Δ(Γ)↦xs} dγ
    Extract inputs, upds from cache
    toVisit ← indices modified by dxs
    if dγ changes any free variable of t besides xs then
        toVisit ← {1, ..., length(inputs)}
    changed ← ∅
    for i ← 1 to length(inputs) do
        if i ∉ toVisit then continue
        dx ← dxs[i] or sameAs(inputs[i]) if i ∉ dxs
        (dy, upds[i]) ← upds[i].apply((dγ, dx))
        inputs[i] ← apply(inputs[i], dx)
        if ¬isSame(dy) then
            changed[i] ← dy
    cache ← {inputs: inputs, upds: upds}
    return ({changed: changed}, cache)
```

Fig. 15. Pseudocode for incrementalizing simplified for-expressions (without accumulators or **using** $(\cdot)$ clauses). It only handles modifications to list elements (no insertions or deletions) and assumes *xs* is a variable.

$$C_\sigma = \{(x, dx, x') \in [\![\sigma]\!] \times [\![\Delta(\sigma)]\!] \times [\![\sigma]\!] \mid dx(\text{new}) = x' \wedge (dx(\text{same}) = \textbf{true} \implies x = x')\}$$
$$\text{for } \sigma = \mathbb{B}, \mathbb{N}, \mathbb{R}, \text{Name}$$
$$C_{\sigma\times\tau} = \{((x, y), (dx, dy), (x', y')) \in [\![\sigma \times \tau]\!] \times [\![\Delta(\sigma \times \tau)]\!] \times [\![\sigma \times \tau]\!] \mid (x, dx, x') \in C_\sigma \wedge (y, dy, y') \in C_\tau\}$$
$$C_{\sigma\to_\kappa\tau} = \{((e, f), de, (e', f')) \in [\![\sigma \to_\kappa \tau]\!] \times [\![\Delta(\kappa)]\!] \times [\![\sigma \to_\kappa \tau]\!] \mid (e, de, e') \in C_\kappa, f = f'\}$$

Fig. 16. Relation of valid changes (selected rules).

them. The collections that are accessed by the loop body are communicated to the system via the **using** (element := collection[index], ...) clause. It uses this information, together with incoming change descriptions, to determine which iterations need to be rerun, and thus which entries of the loop's output must change. The incrementalization of simple loops (**for** *x* **in** *xs* { *t* }) is presented in Fig. 15. For fully featured loops, this is more complicated and relegated to the appendix.

4.3 Correctness

Change Validity. In order to reason about the correctness of the incrementalization, we first need to define when a change is valid. For this we introduce a type-indexed relation $C_\tau \subseteq [\![\tau]\!] \times [\![\Delta(\tau)]\!] \times [\![\tau]\!]$ where $(x, dx, x') \in C_\tau$ means that dx is a valid change from x to x'. The relation is defined inductively on types in Fig. 16. For instance, dx is a valid change from x to x' for basic types if its "new" field is x' and its "same" field being **true** implies $x = x'$. Changes to tuples are valid if the individual changes to their components are valid. Finally, a change to a closure is valid if it is a valid change to its captured environment and the function itself is unchanged.

Updater Validity. Similarly, we need to define when an updater $\text{Upd}(\Delta(\sigma); \Delta(\tau))$ is valid for a function $f : [\![\sigma]\!] \to [\![\tau]\!]$, previously evaluated at an input $x \in [\![\sigma]\!]$. To this end, we introduce the validity predicate $\mathcal{U}^{f,x}_{\sigma\rightsquigarrow\tau} \subseteq [\![\text{Upd}(\Delta(\sigma); \Delta(\tau))]\!]$, defined *coinductively* (as the greatest fixed point):

$$\mathcal{U}^{f,x}_{\sigma\rightsquigarrow\tau} = \{u \in [\![\text{Upd}(\Delta(\sigma); \Delta(\tau))]\!] \mid \forall dx, x' \text{ with } (x, dx, x') \in C_\sigma : (y, dy, y') \in C_\tau \wedge u' \in \mathcal{U}^{f,x'}_{\sigma\rightsquigarrow\tau}$$
$$\text{where } y = f(x), y' = f(x'), \text{ and applying } u \text{ to } dx \text{ yields } (dy, u')\}$$

In words, u is a valid updater for the function f at the input x if for any valid change dx from x to x', applying the updater to dx yields a valid output change dy from $f(x)$ to $f(x')$ and a new updater u' that is again valid for f, but now at input x'.

$$\mathcal{R}_\sigma = \{(x, x') \mid x = x'\} \text{ (for ground types } \sigma) \qquad \mathcal{R}_{\sigma\times\tau} = \{((x,y),(x',y')) \mid (x,x') \in \mathcal{R}_\sigma \wedge (y,y') \in \mathcal{R}_\tau\}$$
$$\mathcal{R}_{\sigma\to_\kappa\tau} = \{((e,f),f') \mid \forall (x,x') \in \mathcal{R}_\sigma.\, (y,y') \in \mathcal{R}_\tau \wedge u \in \mathcal{U}^{f,(e,x)}_{\kappa\times\sigma\rightsquigarrow\tau} \text{ where } y = f(e,x), (y',u) = f'(x')\}$$
$$\mathcal{R}_\Gamma = \{(\gamma,\gamma') \mid \forall (x:\sigma) \in \Gamma.\, (\gamma(x),\gamma'(x)) \in \mathcal{R}_\sigma\}$$

Fig. 17. Logical relation between λ_{inc} and λ_{core} (selected rules).

$$\text{tpl}_\Gamma : [\![\Gamma]\!] \to [\![\text{CtxType}(\Gamma)]\!] \qquad \text{ctx}_\Gamma : [\![\text{CtxType}(\Gamma)]\!] \to [\![\Gamma]\!]$$
$$\text{tpl}_\bullet(\emptyset) = \{\}, \quad \text{tpl}_{\Gamma,x:\tau}(\gamma) = (\text{tpl}_\Gamma(\gamma \setminus \{x\}), \gamma(x)) \qquad \text{ctx}_\bullet(\{\}) = \emptyset, \quad \text{ctx}_{\Gamma,x:\tau}(\gamma) = \text{ctx}_\Gamma(\pi_1(\gamma))[x \mapsto \pi_2(\gamma)]$$

Fig. 18. Functions to convert between contexts and tuples.

Logical Relation. With this machinery in place, we define a logical relation $\mathcal{R}_\tau \subseteq [\![\tau]\!] \times [\![\mathcal{I}\{\tau\}]\!]$ between terms in λ_{inc} and λ_{core}, which characterizes when the λ_{core} term is a correct incremental version of the λ_{inc} term (Fig. 17). The relation is the identity at ground types and at composite types, it is defined in the standard way: two values are related if all their components are.

The interesting case is closure types $\sigma \to_\kappa \tau$ because they have incremental versions. The semantics of such a closure is $(env, f) \in [\![\kappa]\!] \times ([\![\kappa]\!] \times [\![\sigma]\!] \to [\![\tau]\!])$, consisting of the captured environment and a "function pointer" for computing outputs from inputs and the environment. Such a closure is translated to an ordinary λ_{core} function f' of type $[\![\mathcal{I}\{\sigma \to_\kappa \tau\}]\!] = [\![\mathcal{I}\{\sigma\}]\!] \to [\![\mathcal{I}\{\tau\}]\!] \times [\![\text{Upd}(\Delta(\kappa\times\sigma);\Delta(\tau))]\!]$. The translation sends transformed inputs to transformed outputs, together with an updater from environment and input changes to output changes. Now, (env_0, f) and f' are related if for all related inputs $x_0 \in [\![\sigma]\!], x_0' \in [\![\mathcal{I}\{\sigma\}]\!]$, letting $y_0 = f(env_0, x_0)$ and $(y_0', u_0) = f'(x_0')$, we have: (1) the outputs $y_0 \in [\![\tau]\!], y_0' \in [\![\mathcal{I}\{\tau\}]\!]$ are related, and (2) the resulting updater $u_0 \in [\![\text{Upd}(\Delta(\kappa\times\sigma);\Delta(\tau))]\!]$ is a valid updater of f for the input (env_0, x_0).

To state correctness of $\mathcal{I}\{-\}$, we need a few helper functions operating on the tuple representation of contexts (Fig. 18). Using this, we obtain the following result, proven in Appendix C.

Lemma 4.1 (Fundamental Lemma). *Assume $\Gamma \vdash t : \tau$ in λ_{inc}, and thus $\mathcal{I}\{\Gamma\} \vdash \mathcal{I}_\Gamma\{t\} : \mathcal{I}\{\tau\} \times \text{Upd}(\Delta(\text{CtxType}(\Gamma));\Delta(\tau))$ in λ_{core}. Then for all $(\gamma,\gamma') \in \mathcal{R}_\Gamma$, we have*

$$([\![t]\!](\gamma), \pi_1([\![\mathcal{I}\{t\}]\!](\gamma'))) \in \mathcal{R}_\tau \quad \text{and} \quad \pi_2([\![\mathcal{I}\{t\}]\!](\gamma')) \in \mathcal{U}^{[\![t]\!]\circ\text{ctx}_\Gamma,\text{tpl}_\Gamma(\gamma)}_{\text{CtxType}(\Gamma)\rightsquigarrow\tau}$$

Considering the special case of one-element contexts and unfolding the definition of the validity predicate for updaters to handle sequences of input changes, one obtains the following corollary:

Corollary 4.2. *Let $x : \sigma \vdash t : \tau$ be a λ_{inc} term denoting a function $f : [\![\sigma]\!] \to [\![\tau]\!]$, for ground types σ, τ. Let $g = [\![\mathcal{I}\{t\}]\!]$ and $x_0, x_1, x_2, \ldots \in [\![\sigma]\!]$ be a sequence of inputs. Let $dx_0, dx_1, dx_2, \ldots \in [\![\Delta(\sigma)]\!]$ be valid change descriptions: $(x_i, dx_i, x_{i+1}) \in C_\sigma$. Finally, let $(y_0, u_0) = g(x_0)$, and for all i, let (dy_i, u_{i+1}) be the result of applying u_i to dx_i. Then we have $y_0 = f(x_0)$ and $(f(x_i), dy_i, f(x_{i+1})) \in C_\tau$ for all i.*

5 Case Studies

We have implemented a prototype of our approach by embedding the λ_{gen} and λ_{inc} DSLs in Julia. We investigate the impact of incrementalization across a variety of models and custom MCMC kernels. We also compare to Gen [13, 14], a state-of-the-art PPL with incrementalization.

Models and Kernels. To test our approach, we implement 16 MCMC kernels in 6 Bayesian models. (1) *Robust regression [8]:* Bayesian linear regression with outliers. We implement Gaussian drift moves on the parameters and Metropolis-Hastings moves on outlier indicators. (2) *Binary Gaussian mixture [16]:* 2D Gaussian mixture model with inferred parameters, mixing weight, and cluster assignments. Kernels update parameters and mixing weights, or one cluster assignment. (3) *Hidden Markov Model with Inferred Parameters [42]:* A hidden Markov model with an a priori unknown

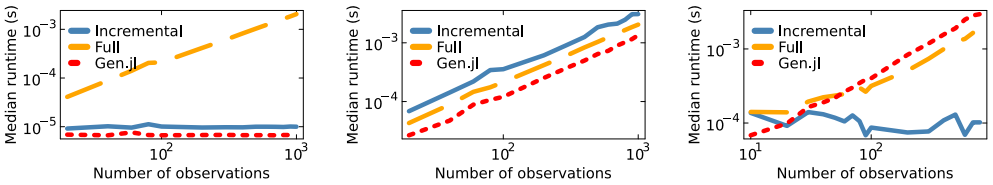
Table 1. Fixed-size benchmarks: median time to complete one iteration of a given MCMC kernel.

Model	**Absolute Runtime**			**Relative Runtime**	
MCMC Kernel	Incremental	Full	Gen	$\frac{\text{Full}}{\text{Inc}}$	$\frac{\text{Gen}}{\text{Inc}}$
Robust Regression *(Box and Tiao [8])*, $N = 1000$					
Parameter change	5.93 ms	3.05 ms	1.77 ms	0.51×	0.30×
Outlier status change	0.017 ms	3.06 ms	0.010 ms	176×	0.59×
Binary Gaussian Mixture *(Diebolt and Robert [16])*, $N = 500$					
Parameters change	3.91 ms	1.85 ms	0.78 ms	0.47×	0.20×
Cluster assignment change	0.028 ms	1.68 ms	0.013 ms	61×	0.46×
Hidden Markov Model with Inferred Parameters *(Scott [42])*, $T = 1000$					
Transition matrix row update	0.211 ms	3.88 ms	1.24 ms	18×	5.88×
Latent state update	0.034 ms	3.81 ms	0.014 ms	112×	0.41×
Mixture of Finite Mixtures *(Miller and Harrison [34])*, $N = 200$					
Birth/death move (empty cluster)	0.238 ms	2.35 ms	3.61 ms	9.9×	15.17×
Birth/death move (singleton cluster)	0.219 ms	2.32 ms	4.10 ms	10.6×	18.72×
Cluster parameter change	2.13 ms	2.19 ms	0.761 ms	1.03×	0.36×
Mixing weights change	0.148 ms	2.37 ms	1.37 ms	16×	9.26×
Cluster assignment change	0.225 ms	8.76 ms	0.032 ms	39×	0.14×
Latent Dirichlet Allocation *(Blei et al. [6])*, $N = 900$ (30×30)					
Document theme change	0.206 ms	9.07 ms	0.031 ms	44×	0.15×
Topic change	0.716 ms	9.42 ms	0.835 ms	13×	1.17×
Topic assignment change (MH)	0.069 ms	9.07 ms	0.019 ms	131×	0.28×
Dirichlet Process Mixture Model *(Neal [38])*, $N = 200$					
Cluster assignment update	0.183 ms	6.64 ms	5.14 ms	36×	28.09×
Cluster parameter update	0.733 ms	1.28 ms	0.421 ms	1.75×	0.57×

transition matrix in $\mathbb{R}^{100\times100}$, and continuous observations. Our MCMC kernels update a single row of the transition matrix or a single latent state. (4) *Mixture of Finite Mixtures [34]:* A 2D Gaussian mixture model where the number of clusters is unknown with Poisson prior. Empty birth/death moves create new empty clusters or remove unoccupied clusters. Singleton birth/death moves propose to split a datapoint into its own singleton cluster, or to merge a singleton cluster into an existing cluster. Cluster parameter changes propose updates to a single latent cluster's parameters. Mixing weight changes re-propose the mixing proportions of the current latent clusters. Cluster assignment changes reassign a single datapoint to a different (pre-existing) cluster. (5) *Latent Dirichlet Allocation [6]:* A topic model. We consider MCMC moves that change the theme (topic mixture) of a single document, the lexicon (word mixture) of a single topic, or the topic assignment for a single word. (6) *Dirichlet Process Mixture Model [38]:* A nonparametric mixture model with infinitely many clusters, marginalized via a Chinese Restaurant Process. We consider Gibbs updates to cluster assignments and parameters as described in Algorithm 8 of Neal [38].

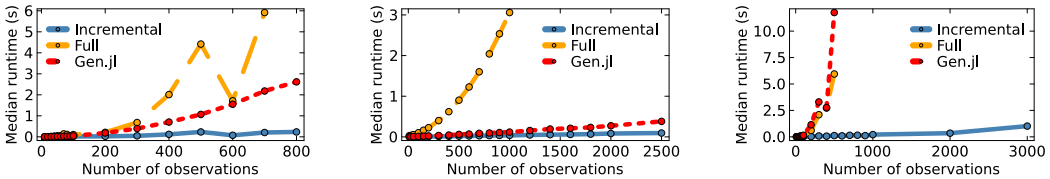
Systems. We implement each model both in our Julia embedding of λ_{gen}, and in Gen's static modeling language, with Gen's experimental support for fine-grained incremental computation turned on (using the `@gen (static, diffs) function` syntax). We implement each inference kernel in Julia, using three different strategies for computing the necessary model densities: (1) our own implementation of incremental density computation, (2) our own implementation of full density recomputation, and (3) Gen's implementation of incremental density computation.

Results. Table 1 presents absolute and relative runtimes of each MCMC kernel, for each approach to density (re)computation. Timings were recorded for each application of each kernel during 100 steps of MCMC inference on datasets of fixed size (chosen separately per model), and the median across all applications of a given kernel is reported. The log-log scaling plots in Fig. 19 show how median times for executing a single kernel scale with dataset size. Each iteration of MCMC inference may execute $O(N)$ kernels (e.g., in a mixture model, to consider reassigning each



(a) Outlier Status Change (b) Parameter Drift Update (c) Singleton Birth/Death

Fig. 19. Representative log-log scaling plots for individual MCMC moves. Although our approach incurs constant-factor overhead, it is the only one to achieve the correct asymptotic scaling across moves.



(a) Mixture of Finite Mixtures (b) Hidden Markov Model (c) Dirichlet Process Mixture

Fig. 20. Linear scaling plots for entire MCMC parameter sweeps.

datapoint's cluster assignment, then updating cluster parameters); Figure 20 shows how these aggregate MCMC moves (and thus end-to-end inference) scale with the number of datapoints.

Findings. The log-log scaling plots in Fig. 19 are representative of three patterns across our experiments. (a) In many cases, our approach successfully reduces the asymptotic complexity of an update (e.g., from linear-time to constant-time), but so does Gen's state-of-the-art incrementalization, often with better constant factors. (b) In rare cases (see red entries in Table 1), the density must be fully recomputed by all three systems, and our approach to incrementalization adds overhead relative to the non-incremental version. (c) In some cases (see green entries in Table 1), our approach delivers asymptotic savings not captured by Gen's state-of-the-art incrementalization. Our approach is the only one that achieves the desired asymptotic complexity for all updates. Figure 20 illustrates how savings on individual updates affect the runtime of an MCMC algorithm that composes these updates into realistic schedules (e.g., "run one parameter update and N cluster assignment updates per iteration"). The plots again surface several interesting patterns. (a) In some cases, our approach reduces the cost of an update from linear to constant-time, enabling it to be efficiently run on every datapoint at every iteration. The failure of Gen to incrementalize these updates leads to $O(N^2)$ runtime overall. (b) When a single row of an HMM's transition matrix is changed, our approach (but not Gen's) efficiently revisits only the time steps that executed the changed transition. This leads to an $O(K)$ reduction in runtime per step. But this is just a constant-factor speed-up with respect to the sequence length. (c) In a Dirichlet process mixture, the expected number of latent clusters grows logarithmically with N, so each Gibbs cluster assignment move requires more density queries as N increases. This means even our approach scales super-linearly, despite reducing the cost of each density evaluation to a constant. This is even more noticeable without incrementalization.

Soundness. We found that Gen sometimes crashed, silently computed incorrect densities, or resorted to full recomputation, requiring various workarounds. For example, our initial GMM program sampled cluster assignments and observations in separate loops.[13] We had to merge the loops to achieve $O(1)$ assignment updates in Gen. Conversely, we had to sample the latent means and covariances in *separate* loops, because Gen did not correctly track changes to tuples.

[13]By "loop" we mean an application of Gen's `Map` or `Unfold` combinator to a kernel implementing the loop body.

Implementation Details. Our implementation faithfully realizes the algorithmic choices formalized in Sections 2 to 4, but differs in implementation details due to our choice of Julia as a host language (chosen for its scientific computing ecosystem and for fair comparison with Gen). While our formalism uses explicit program transformations, in Julia we use a combination of macros, nonstandard interpretation, and function overloading via multiple dispatch to achieve the same effect. This is a common way of embedding DSLs in Julia, also used in Gen [14] and Turing [17]. Our implementation features an abstract `Updater` class with separate subclasses for each primitive operation, where each subclass stores an operation-specific cache and has an `update` method corresponding to applying the updater to an input change. In alignment with our theoretical presentation, the implementation uses immutable, persistent collections (including for its caches).

6 Related Work and Discussion

Incrementalizing λ-Calculi. Our work is inspired by the incremental λ-calculus of Cai et al. [9], which introduces the notion of change types and an incrementalizing program transformation. Their change type for functions $\sigma \to \tau$ is $\sigma \to \Delta(\sigma) \to \Delta(\tau)$, which is more general than our closure-based approach. However, their work does not support caching.

Giarrusso et al. [18] add caching of intermediate results in what they call *cache-transfer style*. They work in the untyped λ-calculus in order to avoid tracking cache types. They require programs to be preprocessed into A'-normal form and lambda-lifted. Matsuda et al. [32] solve the typing issues in the first-order setting using existential types for the caches. Morihata [37] instead incrementalizes functions $\sigma \to \tau$ to *caching derivative-associated* functions $\sigma \to \tau \times (\Delta\sigma \to \Delta\tau)$, where intermediate results are stored in the inner closure's environment, so the caches do not need to be represented in the types. He also mentions that a series of modifications can be supported via "recursively-caching" functions, but does not elaborate. This was inspiration for our updater types.

Dynamic Dependency Tracking. An influential *dynamic* incrementalization technique is self-adjusting computation [1], where the initial run produces a computation trace memoizing intermediate results. Changed inputs require only the recomputation of affected parts of the trace (see [2]). Adapton [20] instead constructs a demand-driven dependency graph at runtime. Follow-up work [19] also uses first-class names, recognizing them as a "critical linguistic feature for efficient incremental computation". Whereas they use unique names to identify subcomputations, we use them to identify domain entities in probabilistic models.

Incrementalization in Probabilistic Programming. Several program transformations aim to speed up single-site Metropolis-Hastings specifically. Shred [49] optimizes changes to random choices that preserve the control flow by compiling to a straight-line program and only recomputing its transitive dependencies. C3 [39] avoids recomputation by combining continuation-passing style with call-site caching. Similarly, Castellan and Paquet [10] describe a translation from a probabilistic program to an event structure, tracking dependencies and avoiding unnecessary sampling.

For more complex inference algorithms, many systems analyze probabilistic programs using graphical models because changing a variable only affects its *Markov blanket*. An early instance of this is BLOG [33], a declarative PPL for open-universe models, whose Metropolis-Hastings inference maintains a dynamic dependency graph to avoid recomputing factors in the acceptance ratio. As changing a variable can change the dependency graph, proposals require recomputing the dependency structure. The Bean Machine [45] extends this with programmable proposal distributions that can be composed and blocked across variables. Venture [28, 29] was perhaps the first to introduce dynamic dependency tracking for a higher-order generative language with programmable inference, enabling optimal asymptotic scaling for a broad range of inference algorithms. Its dependency tracking also handles aliasing from memoization and was the first to be

extended to sublinear-time inference updates, by subsampling dependencies. Böck and Cito [7] present a program analysis to factorize the density function in the presence of loops, which can speed up single-site MH, black-box variational inference, and sequential Monte Carlo.

Gen [14] allows users to customize incrementalization via an *update* function in its generative function interface. By default, its static modeling language (for straight-line probabilistic programs) automates incremental updates via dependency tracking and partial evaluation. It also has undocumented, experimental support for incrementalization based on "diffs" [11], disabled by default due to bugs and limited support. Our work drew inspiration from Gen, and could inform future extensions to Gen, e.g., to handle features such as names, tuples, and higher-order functions.

Lim et al. [25] also aim to improve the efficiency of density computations in probabilistic programs, but via auto-vectorization based on tensor operations rather than incremental computation; notably, incrementalization gives asymptotic speedups even on a single CPU, without parallel hardware. Instead of incremental density evaluation, Cusumano-Towner et al. [12] tackle the related problem of incrementally generating *samples* for changes to the input *program* by translating their traces.

Density Semantics. Lee et al. [22] define both measure and density semantics for an imperative, first-order probabilistic language, but their reference measure only supports densities when all primitive distributions are continuous, whereas our traces may contain heterogeneous types. Tassarotti and Tristan [44] present a verified compilation pipeline from a simplified version of Stan to optimized density code in C, but their language and approach (operational semantics, imperative) are very different from ours. Becker et al. [5] use a similar semantics to ours, but their language does not feature name generation or unordered collections—key contributions of ours.

Nonparametric Models. Existing systems struggle with exchangeable sequences and nonparametric models. Gen [14] can express nonparametric models, but its traces introduce artificial orderings on exchangeable sequences, making incrementalization less effective because insertions and deletions shift indices. Matheos et al. [31] automate involutive MCMC (which can express many MCMC algorithms) for open-universe models, but have to deal with complicated and error-prone bookkeeping of indices into variable-length sequences. By contrast, our approach of unordered collections with name-based access offers stable identifiers for elements of exchangeable sequences, simplifying the implementation and improving the effectiveness of incrementalization.

Limitations. Long, straight-line programs made of many individually inexpensive operations—e.g., many large Bayes nets—do not benefit from incrementalization in our approach, which must still execute an incrementalized version of each line of such a program. As it stands, our approach is also limited to loops over finite data structures. Extending our implementation to handle general **while** loops would be straightforward, requiring only small changes to our existing density and incrementalization transformations. General recursion would be more challenging, as traces of recursive probabilistic programs would need to be assigned *recursive trace types*, for which we would need to define new change representations and incrementalization rules. Whether via recursion or while loops, the possibility of non-termination would complicate our semantics and correctness proofs (requiring, e.g., the use of quasi-Borel *predomains* [46]).

Extensions. We have focused on MCMC, but incremental computation could also benefit other Monte Carlo inference algorithms (see, e.g., the SMC pseudocode in Fig. 2e). For variational inference, the fit is less natural: traces are independently sampled rather than incrementally modified. Our pipeline works by generating *deterministic* density programs and incrementalizing them. However, it is common to *stochastically estimate* densities that would be intractable to compute exactly [3]. It would be interesting to develop versions of incrementalization that work for stochastic computations, perhaps building on ideas from Cusumano-Towner et al. [12].

Data Availability Statement

The artifact for this paper (consisting of the Julia implementation and benchmarks) is archived on Zenodo [50].

Acknowledgments

We would like to thank McCoy Becker, Marco Cusumano-Towner, Mathieu Huot, George Matheos, and Tan Zhi-Xuan for helpful conversations at various stages of this project. We are also grateful to Cameron Freer and the anonymous reviewers for their helpful feedback, which improved the presentation of the paper.

References

[1] Umut A. Acar. 2005. *Self-Adjusting Computation.* Ph. D. Dissertation. Carnegie Mellon University.

[2] Umut A. Acar. 2009. Self-adjusting computation: (an overview). In *Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2009, Savannah, GA, USA, January 19-20, 2009*, Germán Puebla and Germán Vidal (Eds.). ACM, 1–6. https://doi.org/10.1145/1480945.1480946

[3] Christophe Andrieu and Gareth O. Roberts. 2009. The pseudo-marginal approach for efficient Monte Carlo computations. *The Annals of Statistics* 37, 2 (2009), 697–725. https://doi.org/10.1214/07-AOS574

[4] John Baez and Alexander Hoffnung. 2011. Convenient categories of smooth spaces. *Trans. Amer. Math. Soc.* 363, 11 (2011), 5789–5825.

[5] McCoy R. Becker, Mathieu Huot, George Matheos, Xiaoyan Wang, Karen Chung, Colin Smith, Sam Ritchie, Rif A. Saurous, Alexander K. Lew, Martin C. Rinard, and Vikash K. Mansinghka. 2026. Probabilistic Programming with Vectorized Programmable Inference. *Proc. ACM Program. Lang.* 10, POPL (2026), 2523–2554. https://doi.org/10.1145/3776729

[6] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet allocation. *Journal of Machine Learning Research* 3, Jan (2003), 993–1022.

[7] Markus Böck and Jürgen Cito. 2025. Static Factorisation of Probabilistic Programs With User-Labelled Sample Statements and While Loops. *CoRR* abs/2508.20922 (2025). https://doi.org/10.48550/ARXIV.2508.20922 arXiv:2508.20922

[8] George E. P. Box and George C. Tiao. 1968. A Bayesian approach to some outlier problems. *Biometrika* 55, 1 (1968), 119–129.

[9] Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. 2014. A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 145–155. https://doi.org/10.1145/2594291.2594304

[10] Simon Castellan and Hugo Paquet. 2019. Probabilistic Programming Inference via Intensional Semantics. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 322–349. https://doi.org/10.1007/978-3-030-17184-1_12

[11] Marco F. Cusumano-Towner. 2020. *Gen: a high-level programming platform for probabilistic inference.* Ph. D. Dissertation. Massachusetts Institute of Technology.

[12] Marco F. Cusumano-Towner, Benjamin Bichsel, Timon Gehr, Martin T. Vechev, and Vikash K. Mansinghka. 2018. Incremental inference for probabilistic programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018*. ACM, 571–585. https://doi.org/10.1145/3192366.3192399

[13] Marco F. Cusumano-Towner, Alexander K. Lew, and Vikash K. Mansinghka. 2020. Automating Involutive MCMC using Probabilistic and Differentiable Programming. arXiv:2007.09871 [stat.CO] https://arxiv.org/abs/2007.09871

[14] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 221–236. https://doi.org/10.1145/3314221.3314642

[15] Swaraj Dash, Younesse Kaddar, Hugo Paquet, and Sam Staton. 2023. Affine monads and lazy structures for Bayesian programming. *Proc. ACM Program. Lang.* 7, POPL (2023), 1338–1368. https://doi.org/10.1145/3571239

[16] Jean Diebolt and Christian P. Robert. 1994. Estimation of finite mixture distributions through Bayesian sampling. *Journal of the Royal Statistical Society: Series B (Methodological)* 56, 2 (1994), 363–375.

[17] Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: A Language for Flexible Probabilistic Inference. In *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics (AISTATS) (Proceedings of Machine*

Learning Research, Vol. 84). PMLR, 1682–1690.

[18] Paolo G. Giarrusso, Yann Régis-Gianas, and Philipp Schuster. 2019. Incremental λ-Calculus in Cache-Transfer Style - Static Memoization by Program Transformation. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 553–580. https://doi.org/10.1007/978-3-030-17184-1_20

[19] Matthew A. Hammer, Jana Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael W. Hicks, and David Van Horn. 2015. Incremental computation with names. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 748–766. https://doi.org/10.1145/2814270.2814305

[20] Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: composable, demand-driven incremental computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 156–166. https://doi.org/10.1145/2594291.2594324

[21] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability theory. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017*. IEEE Computer Society, 1–12. https://doi.org/10.1109/LICS.2017.8005137

[22] Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. 2020. Towards Verified Stochastic Variational Inference for Probabilistic Programs. *Proc. ACM Program. Lang.* 4, POPL (2020), 16:1–16:33. https://doi.org/10.1145/3371084

[23] Alexander K. Lew, Marco F. Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K. Mansinghka. 2020. Trace types and denotational semantics for sound programmable inference in probabilistic languages. *Proc. ACM Program. Lang.* 4, POPL (2020), 19:1–19:32. https://doi.org/10.1145/3371087

[24] Jianlin Li, Leni Aniva, Pengyuan Shi, and Yizhou Zhang. 2023. Type-Preserving, Dependence-Aware Guide Generation for Sound, Effective Amortized Probabilistic Inference. *Proc. ACM Program. Lang.* 7, POPL (2023), 1454–1482. https://doi.org/10.1145/3571243

[25] Sangho Lim, Hyoungjin Lim, Wonyeol Lee, Xavier Rival, and Hongseok Yang. 2026. Optimising Density Computations in Probabilistic Programs via Automatic Loop Vectorisation. *Proc. ACM Program. Lang.* 10, POPL (2026), 597–627. https://doi.org/10.1145/3776663

[26] Saunders Mac Lane. 1998. *Categories for the Working Mathematician* (2nd ed.). Graduate Texts in Mathematics, Vol. 5. Springer.

[27] Saunders Mac Lane and Ieke Moerdijk. 1994. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Springer.

[28] Vikash Mansinghka, Daniel Selsam, and Yura N. Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR* abs/1404.0099 (2014). arXiv:1404.0099 http://arxiv.org/abs/1404.0099

[29] Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin C. Rinard. 2018. Probabilistic programming with programmable inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 603–616. https://doi.org/10.1145/3192366.3192409

[30] Cristina Matache, Sean K. Moss, and Sam Staton. 2022. Concrete categories and higher-order recursion. In *37th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2022*. ACM, 57:1–57:14. https://doi.org/10.1145/3531130.3533370

[31] George Matheos, Alexander K. Lew, Matin Ghavamizadeh, Stuart Russell, Marco F. Cusumano-Towner, and Vikash Mansinghka. 2021. Transforming Worlds: Automated Involutive MCMC for Open-Universe Probabilistic Models. In *Third Symposium on Advances in Approximate Bayesian Inference*. https://openreview.net/forum?id=8Itm8dQnJRc

[32] Kazutaka Matsuda, Samantha Frohlich, Meng Wang, and Nicolas Wu. 2023. Embedding by Unembedding. *Proc. ACM Program. Lang.* 7, ICFP (2023), 1–47. https://doi.org/10.1145/3607830

[33] Brian Milch and Stuart Russell. 2006. General-Purpose MCMC inference over relational structures. In *Proceedings of the Twenty-Second Conference on Uncertainty in Artificial Intelligence (UAI'06)*. AUAI Press, 349–358.

[34] Jeffrey W. Miller and Matthew T. Harrison. 2018. Mixture models with a prior on the number of components. *J. Amer. Statist. Assoc.* 113, 521 (2018), 340–356.

[35] Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS 1989)*. IEEE Computer Society, 14–23. https://doi.org/10.1109/LICS.1989.39155

[36] Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.

[37] Akimasa Morihata. 2020. Short Cut to Incremental Typed Functional Programs (Extended Abstract). In *WPTE 2020: 7th International Workshop on Rewriting Techniques for Program Transformations and Evaluation*. http://maude.ucm.es/wpte20/papers/WPTE_2020_morihata.pdf Informal proceedings.

[38] Radford M. Neal. 2000. Markov chain sampling methods for Dirichlet process mixture models. *Journal of computational and graphical statistics* 9, 2 (2000), 249–265.

[39] Daniel Ritchie, Andreas Stuhlmüller, and Noah D. Goodman. 2016. C3: Lightweight Incrementalized MCMC for Probabilistic Programs using Continuations and Callsite Caching. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016, Cadiz, Spain, May 9-11, 2016 (JMLR Workshop and Conference Proceedings, Vol. 51)*, Arthur Gretton and Christian C. Robert (Eds.). JMLR.org, 28–37. http://proceedings.mlr.press/v51/ritchie16.html

[40] Marcin Sabok, Sam Staton, Dario Stein, and Michael Wolman. 2021. Probabilistic programming semantics for name generation. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434292

[41] Adam Ścibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K. Moss, Chris Heunen, and Zoubin Ghahramani. 2018. Denotational validation of higher-order Bayesian inference. *Proc. ACM Program. Lang.* 2, POPL (2018), 60:1–60:29. https://doi.org/10.1145/3158148

[42] Steven L. Scott. 2002. Bayesian methods for Hidden Markov Models: Recursive computing in the 21st century. *J. Amer. Statist. Assoc.* 97, 457 (2002), 337–351.

[43] Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017 (Lecture Notes in Computer Science, Vol. 10201)*. Springer, 855–879. https://doi.org/10.1007/978-3-662-54434-1_32

[44] Joseph Tassarotti and Jean-Baptiste Tristan. 2023. Verified Density Compilation for a Probabilistic Programming Language. *Proc. ACM Program. Lang.* 7, PLDI (2023), 615–637. https://doi.org/10.1145/3591245

[45] Nazanin Khosravani Tehrani, Nimar S. Arora, Yucen Lily Li, Kinjal Divesh Shah, David Noursi, Michael Tingley, Narjes Torabi, Sepehr Masouleh, Eric Lippert, and Erik Meijer. 2020. Bean Machine: A Declarative Probabilistic Programming Language For Efficient Programmable Inference. In *International Conference on Probabilistic Graphical Models, PGM 2020, 23-25 September 2020, Aalborg, Hotel Comwell Rebild Bakker, Skørping, Denmark (Proceedings of Machine Learning Research, Vol. 138)*, Manfred Jaeger and Thomas Dyhre Nielsen (Eds.). PMLR, 485–496. http://proceedings.mlr.press/v138/tehrani20a.html

[46] Matthijs Vákár, Ohad Kammar, and Sam Staton. 2019. A domain theory for statistical probabilistic programming. *Proc. ACM Program. Lang.* 3, POPL (2019), 36:1–36:29. https://doi.org/10.1145/3290349

[47] Matthijs Vákár and Luke Ong. 2018. On s-finite measures and kernels. arXiv:1810.01837 https://arxiv.org/abs/1810.01837

[48] Di Wang, Jan Hoffmann, and Thomas Reps. 2021. Sound Probabilistic Inference via Guide Types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021*. ACM, 788–803. https://doi.org/10.1145/3453483.3454077

[49] Lingfeng Yang, Pat Hanrahan, and Noah D. Goodman. 2014. Generating Efficient MCMC Kernels from Probabilistic Programs. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics, AISTATS 2014, Reykjavik, Iceland, April 22-25, 2014 (JMLR Workshop and Conference Proceedings)*. JMLR.org, 1068–1076. http://proceedings.mlr.press/v33/yang14d.html

[50] Fabian Zaiser, Jack Czenszak, Martin Rinard, Vikash Mansinghka, and Alexander Lew. 2026. *Artifact for: "Incremental Computation for Efficient Programmable Inference in Probabilistic Programs" (PLDI 2026)*. https://doi.org/10.5281/zenodo.19080766

Received 2025-11-14; accepted 2026-04-03