ADEV: Sound Automatic Differentiation of Expected Values of Probabilistic Programs

ALEXANDER K. LEW*, MIT, USA
MATHIEU HUOT*, Oxford University, UK
SAM STATON, Oxford University, UK
VIKASH K. MANSINGHKA, MIT, USA

Optimizing the expected values of probabilistic processes is a central problem in computer science and its applications, arising in fields ranging from artificial intelligence to operations research to statistical computing. Unfortunately, automatic differentiation techniques developed for deterministic programs do not in general compute the correct gradients needed for widely used solutions based on gradient-based optimization.

In this paper, we present ADEV, an extension to forward-mode AD that correctly differentiates the expectations of probabilistic processes represented as programs that make random choices. Our algorithm is a source-to-source program transformation on an expressive, higher-order language for probabilistic computation, with both discrete and continuous probability distributions. The result of our transformation is a new probabilistic program, whose expected return value is the derivative of the original program's expectation. This output program can be run to generate unbiased Monte Carlo estimates of the desired gradient, which can then be used within the inner loop of stochastic gradient descent. We prove ADEV correct using logical relations over the denotations of the source and target probabilistic programs. Because it modularly extends forward-mode AD, our algorithm lends itself to a concise implementation strategy, which we exploit to develop a prototype in just a few dozen lines of Haskell (https://github.com/probcomp/adev).

CCS Concepts: • **Mathematics of computing** → *Statistical software*; • **Theory of computation** → *Denotational semantics*; • **Computing methodologies** → *Symbolic and algebraic manipulation*; *Machine learning*.

Additional Key Words and Phrases: probabilistic programming, automatic differentiation, denotational semantics, logical relations, functional programming, correctness, machine learning theory

ACM Reference Format:
Alexander K. Lew, Mathieu Huot, Sam Staton, and Vikash K. Mansinghka. 2023. ADEV: Sound Automatic Differentiation of Expected Values of Probabilistic Programs. *Proc. ACM Program. Lang.* 7, POPL, Article 5 (January 2023), 33 pages. https://doi.org/10.1145/3571198

1 INTRODUCTION

Specifying and solving optimization problems has never been easier, thanks in large part to the maturation of programming languages and libraries that support *automatic differentiation* (AD). With AD, users can specify objective functions as programs, then automate the construction of programs for computing their derivatives. These derivatives can be fed into optimization algorithms, such as gradient descent or ADAM, to find local minima or maxima of the original objective function.

*Equal contribution

Authors' addresses: Alexander K. Lew, alexlew@mit.edu, MIT, Cambridge, MA, USA; Mathieu Huot, mathieu.huot@cs.ox.ac.uk, Oxford University, Oxford, UK; Sam Staton, sam.staton@cs.ox.ac.uk, Oxford University, Oxford, UK; Vikash K. Mansinghka, vkm@mit.edu, MIT, Cambridge, MA, USA.

2475-1421/2023/1-ART5
https://doi.org/10.1145/3571198

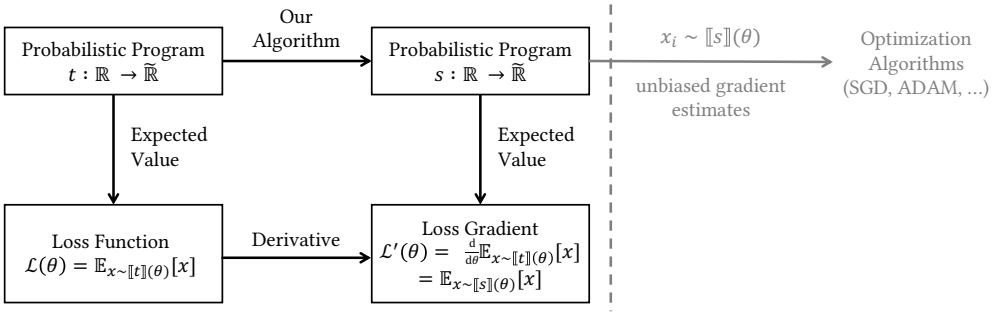

Fig. 1. Our approach to differentiating loss functions defined as expected values. Our algorithm takes as input a probabilistic program t, which, given a parameter of type $\mathbb{R}$ (or a subtype), outputs a value of type $\widetilde{\mathbb{R}}$, which represents *probabilistic estimators* of losses (Def. 3.1). We translate t to a new probabilistic program s, whose expected return value is the *derivative* of t's expected return value. Running s yields provably unbiased estimates x_i of the loss's derivative, which can be used to guide optimization.

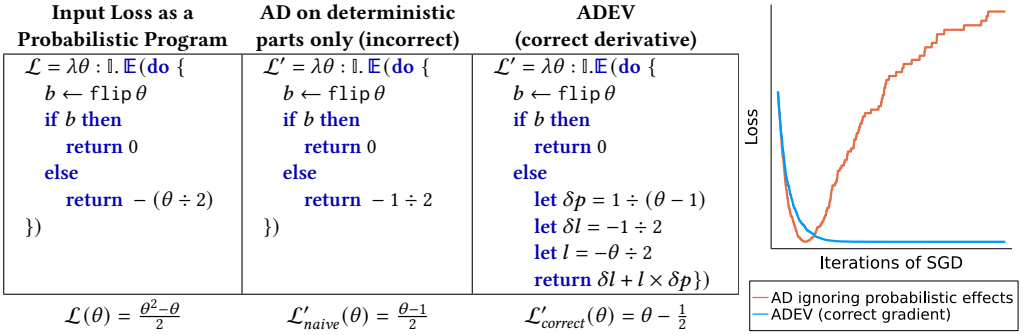

Fig. 2. If probabilistic constructs are ignored, AD may produce incorrect results. In this case, standard AD fails to account for θ's effect on the *probability* of entering each branch. ADEV, by contrast, correctly accounts for the probabilistic effects, generating similar code to what a practitioner might hand-derive. *Right:* Correct gradients are often crucial for downstream applications, e.g. optimization via stochastic gradient descent.

Unfortunately, there is an important class of functions that today's AD systems *cannot* differentiate correctly: those defined as *expected values* of probabilistic processes. Consider, for example, the reinforcement learning problem of optimizing the parameters of a robot's algorithm, based on simulations of its behavior in random environments. The practitioner hopes maximize the *expected* (i.e., average) reward across all possible runs of the simulator. But obtaining gradients of this objective is not straight-forward; naively applying AD to the stochastic reward simulator will in general give incorrect results. Instead, practitioners often resort to hand-derived gradient estimators that they must manually prove correct. And this dilemma is hardly unique to robotics: the optimization of expected values is a ubiquitous problem, as the diverse examples in Table 1 attest.

In this paper, we present ADEV, a new AD algorithm that correctly computes derivatives of the expected values of probabilistic programs. Our general approach is sketched in Figure 1:

- The user provides a program t encoding a probabilistic process dependent on a parameter θ.
- The user's goal is to find $\theta^* = \text{argmin}_\theta \mathcal{L}(\theta)$, where the *loss function* $\mathcal{L}$ maps a parameter value θ to the *expected return value* of t, run on input θ.
- Applying ADEV to t yields a new probabilistic program s. Our algorithm is *correct* in that the expected return value of s at input θ is exactly the derivative $\mathcal{L}'(\theta)$ of the loss.

Table 1. The need to differentiate expected values of probabilistic processes is ubiquitous in many fields, including machine learning, operations research, and finance [Mohamed et al. 2020].

Application	Probabilistic Process	Expected Value	Use of Gradients
Supervised learning	Evaluate loss on random minibatch	Loss on all data	Minimize total loss
Reinforcement learning	Measure reward in simulated environment	Average reward	Maximize reward
Variational Bayes	Sample variational family, estimate ELBO	ELBO objective	Minimize $KL(q\|\|p)$
Train on synthetic data	Generate synthetic data and evaluate loss	Expected loss under simulator	Minimize average loss
Sensitivity analysis in computational finance	Simulate future option prices, to assess investment risk	Expected risk	Analyze risk assessment's sensitivity to pricing assumptions
Operations research	Simulate efficiency of a customer queue	Average efficiency	Maximize efficiency
Bayesian optimization	Sample current belief distribution about a function's value at a candidate point, and evaluate whether the point would be a new 'best parameter value'	Probability of improvement over current best parameter value	Choose next sample point to maximize probability of improvement

- Even if $\mathcal{L}'(\theta)$ cannot be evaluated exactly, users can *run* the probabilistic program s to simulate provably unbiased estimates of $\mathcal{L}'(\theta)$, which can be used for stochastic optimization.

Figure 2 illustrates our method on a toy example. The loss function $\mathcal{L}$ is defined as the expectation of a program that flips a biased coin, with probability-of-heads θ. Depending on the outcome, we receive either 0 loss (the 'heads' case), or a *negative* loss of $-\frac{\theta}{2}$ (indicating a positive reward). The problem is to find the θ that minimizes expected loss. Intuitively, the optimal strategy must trade off the *benefits* of increasing θ (higher payoff in the 'tails' case) with its *drawbacks* (lower probability of entering the 'tails' case in the first place). The expected loss $\mathcal{L}(\theta) = \frac{\theta^2-\theta}{2}$ is minimized at $\theta = 0.5$.

Applying AD to only the deterministic parts of f fails to account for the effect of increasing θ on the *probability* of entering the high-reward branch. The resulting (incorrect) gradient is negative for all $\theta \in (0, 1)$; optimizing with it significantly overshoots the optimal value of 0.5. By contrast, ADEV automatically introduces additional terms to account for the dependence of b on θ, leading to a gradient that can be soundly used to optimize the loss.

Our translation of $\mathcal{L}$ into $\mathcal{L}'$ may appear complex and non-local, but in fact, we arrived at our algorithm by modularly extending a standard 'dual-number' forward-mode AD macro (e.g., as presented by Huot et al. [2020]) to handle probabilistic types and terms. As in standard forward-mode AD, our translation is mostly structure-preserving, with almost all the action happening in the translation of primitives, like `flip` in this example. (The term we display for $\mathcal{L}'$ in Figure 2 has been further simplified for clarity, via monad laws and β-reductions; see Figure 15.)

Contributions. We present ADEV, a new AD algorithm for correctly automating the derivatives of the expectations of expressive probabilistic programs. It has the following desirable properties:

(1) **Provably correct:** It comes with guarantees relating the output program's expectation to the input program's expectation's derivative (Theorem 6.1).
(2) **Modular:** ADEV is a modular extension to traditional forward-mode AD, and can be modularly extended to support new gradient estimators and probabilistic primitives (Table 2).
(3) **Compositional:** ADEV's translation is local, in that all the action happens in the translation of primitives (as in standard forward-mode AD).
(4) **Flexible:** ADEV provides levers for navigating trade-offs between the variance and computational cost of the output program, viewed as an unbiased gradient estimator.
(5) **Easy to implement:** It is easy to modify existing forward-mode implementations to support ADEV — our Haskell prototype is just a few dozen lines (Appx. A, github.com/probcomp/adev).

Recipe for New ADEV Modules

- Add new types, constructs, or primitives
- Extend macro $\mathcal{D}\{\cdot\}$ to new constructs
- For new types τ, define specification $\mathscr{R}_\tau$
- Prove new constructs preserve correctness

Table 2. ADEV is implemented modularly and admits modular extensions.

Modular language extension	**Reference**
Real-valued probabilistic primitives + combinators	Sec. 3
Discrete prob. prog. + enumeration + REINFORCE [Ranganath et al. 2014]	Sec. 4
Continuous prob. prog. + REPARAM [Kingma and Welling 2014]	Sec. 5
Discontinuous operations (e.g. $\leq$)	Sec. 6
Control variates (baselines) for variance reduction [Mnih and Gregor 2014]	Appx. B.1
Variance reduction via dependency tracking [Schulman et al. 2015]	Appx. B.2
Storchastic [van Krieken et al. 2021] multi-sample estimators	Appx. B.4
Higher-order primitive for differentiable particle filter [Ścibior et al. 2021]	Appx. B.5
Implicit reparameterization gradients [Figurnov et al. 2018]	Appx. B.7
Weak or measure-valued derivatives [Heidergott and Vázquez-Abad 2000]	Appx. B.8
Reparameterized rejection gradients [Naesseth et al. 2017]	Appx. B.9
SBSA with common random numbers [Kleinman et al. 1999]	Appx. B.10

Key Challenges. To develop our algorithm, we had to overcome four key technical challenges:

(1) **Challenge: Reasoning about correctness compositionally.** Our correctness criterion makes sense for the main program, but not necessarily for subterms, hindering compositional reasoning. **Solution: Logical relations.** We adapt the *logical relations* technique of Huot et al. [2020] (Sec. 2) to define extended correctness criteria that apply to any type in our language.

(2) **Challenge: Compositional differentiation of probability kernels.** ML researchers often build gradient estimators for whole models [Mohamed et al. 2020]. But to differentiate *compositionally* we need a notion of 'probability kernel derivative,' and rules for composition. **Solution: Higher-order semantics of probabilistic programs and AD.** Recent PPL semantics view probability as a submonad of the continuation monad [Vákár et al. 2019]. In this light, probabilistic primitives are really higher-order primitives, averaging a continuation's value over all possible sampled inputs. This gives a blueprint for a notion of derivative at probabilistic types, based on existing theory of higher-order AD [Huot et al. 2020] (Sec. 4).

(3) **Challenge: Commuting limits.** Differentiating expectations requires swapping integrals and derivatives, which may not be sound. The dominated convergence theorem gives sufficient regularity conditions, but they are hard to formulate or enforce compositionally. **Solution: Lightweight static analysis to surface regularity conditions.** Our macro optionally outputs a *verification condition* (presented to the user as syntax) making explicit every regularity assumption that ADEV makes while translating a program (Sec. 5). These regularity assumptions are often ignored (i.e., not even stated) in the ML literature on gradient estimation.

(4) **Challenge: Safely exposing non-differentiable primitives.** Probabilistic programs that use non-differentiable primitives, like $\leq$ or *ReLU*, may have differentiable expectations. But dominated convergence requires integrands to be continuously differentiable w.r.t. the parameter. **Solution: Static typing for fine-grained differentiability tracking.** To ensure we only swap integrals and derivatives when it is sound to do so, we use static typing to track the smoothness of deterministic subterms with respect to each of their free variables (Sec. 6).

2 BACKGROUND: FORWARD-MODE AD FOR DETERMINISTIC PROGRAMS

In this section, we review standard forward-mode AD, a well-established technique for automating the derivatives of *deterministic* programs [Director and Rohrer 1969; Griewank and Walther 2008; Rall 1981]. Our presentation is based on Huot et al. [2020]'s formalization of forward-mode AD in a pure, higher-order functional language. The simplicity of the algorithm, and the modularity of Huot et al. [2020]'s correctness argument via logical relations, makes it well-suited to extensions, like those we introduce in Sections 3-6 and in Appendix B (see Table 2).

$$\begin{aligned}
\text{Smooth base types } \kappa &::= \mathbb{R} \mid \mathbb{R}_{>0} \mid \mathbb{I} \\
\text{Types } \tau &::= \mathbf{1} \mid \mathbb{N} \mid \kappa \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \\
\text{Terms } t &::= () \mid r\ (\in \mathbb{R}) \mid c \mid \boxed{c_{\mathcal{D}}} \mid x \mid (t_1, t_2) \mid \lambda x : \tau . t \mid \textbf{let}\ x = t_1\ \textbf{in}\ t_2 \\
&\quad \mid \textbf{fst}\ t \mid \textbf{snd}\ t \mid t_1\ t_2 \\
\text{Primitives } c &::= + \mid - \mid \times \mid \div \mid \texttt{exp} \mid \texttt{log} \mid \texttt{sin} \mid \texttt{cos} \mid \texttt{pow}
\end{aligned}$$

We write $\textbf{let}\ (x, y) = t_1\ \textbf{in}\ t_2$ as sugar for $\textbf{let}\ x = \textbf{fst}\ t_1\ \textbf{in let}\ y = \textbf{snd}\ t_1\ \textbf{in}\ t_2$

Fig. 3. Syntax of Sec. 2's *deterministic* simply-typed λ-calculus, where r ranges over real numeric constants, and c over source-language primitive functions, to each of which is associated a target-language dual-number derivative $c_{\mathcal{D}}$ (Fig. 5). Gray highlights indicate syntax only present in the AD macro's target language.

2.1 Source Language for AD

The grammar of our starting language is given in Figure 3. Our types, terms, typing rules, and semantics are standard, but we recall them here to fix notation:

Types and Terms. Our language includes numeric types[1] ($\mathbb{R}$, $\mathbb{R}_{>0}$, $\mathbb{I} = (0, 1)$, $\mathbb{N}$), tuples $A \times B$, and function types $A \to B$. For terms, it features the standard constructs for building and accessing tuples, creating abstractions, and applying functions. We also provide primitives for smooth numerical operations, like $\texttt{log} : \mathbb{R}_{>0} \to \mathbb{R}$. Technically, we need multiple versions of each primitive ($+_{\mathbb{N}}, +_{\mathbb{R}}$), but we will suppress these subscripts when clear from context.

Judgments. A *context* Γ is a list associating variable names with their types (e.g., $\Gamma = x : \tau, y : \sigma$). The typing judgment $\Gamma \vdash t : \tau$ indicates that, in context Γ, t is a well-typed term of type τ. If $\vdash t : \tau$ (i.e., if t is well-typed in an empty context), we call t a closed term. The typing rules are standard.

Semantics. To each type τ we assign a set of values $[\![\tau]\!]$. To numeric types, we assign the corresponding sets of numbers. We interpret product and function types as products and functions on the interpretations of their arguments: $[\![A \times B]\!] = [\![A]\!] \times [\![B]\!]$, and $[\![A \to B]\!] = [\![A]\!] \to [\![B]\!]$. Then, for any term in context $\Gamma \vdash t : \tau$, we assign a meaning $[\![\Gamma \vdash t : \tau]\!] \in [\![\Gamma]\!] \to [\![\tau]\!]$, where $[\![\Gamma]\!]$ is the space of *environments* mapping the variable names in Γ to values of their corresponding types. For example, the meaning of a variable is the function that looks up that variable in the environment: $[\![\Gamma \vdash x : \tau]\!](\rho) = \rho[x]$. When the context or the type is clear, we may omit them, writing $[\![t]\!]$ or $[\![t : \tau]\!]$. Using this shorthand, we give some more examples of term interpretations:

$$[\![t_1\ t_2]\!](\rho) = [\![t_1]\!](\rho)([\![t_2]\!](\rho)) \quad [\![(t_1, t_2)]\!](\rho) = ([\![t_1]\!](\rho), [\![t_2]\!](\rho)) \quad [\![\lambda x.t]\!](\rho) = \lambda v.[\![t]\!](\rho[x \mapsto v])$$

Notation 2.1. *For closed terms t, we write $[\![t]\!]$ instead of $[\![t]\!](\rho)$, where ρ is the empty environment.*

2.2 Forward-Mode AD

We assume the user has written a program $\vdash t : \mathbb{R} \to \mathbb{R}$, and wishes to automate the construction of a program $\vdash s : \mathbb{R} \to \mathbb{R}$ computing its (denotation's) derivative $\theta \mapsto [\![t]\!]'(\theta)$. Forward-mode AD translates the source program into a program representing the derivative in two steps:

- First, we apply a macro, $\mathcal{D}\{\cdot\}$, to the user's program, yielding a new program $\vdash \mathcal{D}\{t\} : \mathbb{R} \times \mathbb{R} \to \mathbb{R} \times \mathbb{R}$. This new program operates on *dual numbers*, pairs of numbers representing the *value* and *derivative* of a computation. For any differentiable h, applying $[\![\mathcal{D}\{t\}]\!]$ to the dual number $(h(\theta), h'(\theta))$ should yield $(([\![t]\!] \circ h)(\theta), ([\![t]\!] \circ h)'(\theta))$.
- Second, we output the program $s = \lambda\theta : \mathbb{R}.\textbf{snd}(\mathcal{D}\{t\}\ (\theta, 1))$. Since $(\theta, 1) = (id(\theta), id'(\theta))$, we know $[\![\mathcal{D}\{t\}]\!](\theta, 1)$ returns a dual number representing $(([\![t]\!] \circ id)(\theta), ([\![t]\!] \circ id)'(\theta)) = ([\![t]\!](\theta), [\![t]\!]'(\theta))$, whose second component we extract to return $[\![t]\!]$'s derivative.

[1] In our implementation, reals are represented by floating-point numbers, but we note that our correctness results do not account for any error introduced by floating-point approximations.

AD on contexts: $\mathcal{D}\{\bullet\} = \bullet \qquad \mathcal{D}\{\Gamma, x : \tau\} = \mathcal{D}\{\Gamma\}, x : \mathcal{D}\{\tau\}$

AD on types:

$$\begin{array}{ll} \mathcal{D}\{\kappa\} & = \kappa \times \mathbb{R} \\ \mathcal{D}\{\mathbb{N}\} & = \mathbb{N} \end{array} \qquad \begin{array}{ll} \mathcal{D}\{\mathbf{1}\} & = \mathbf{1} \\ \mathcal{D}\{\tau_1 \times \tau_2\} & = \mathcal{D}\{\tau_1\} \times \mathcal{D}\{\tau_2\} \\ \mathcal{D}\{\tau_1 \to \tau_2\} & = \mathcal{D}\{\tau_1\} \to \mathcal{D}\{\tau_2\} \end{array}$$

AD on pure expressions:

$$\begin{array}{ll} \mathcal{D}\{\lambda x : \tau.t\} & = \lambda x : \mathcal{D}\{\tau\}.\mathcal{D}\{t\} \\ \mathcal{D}\{t_1 t_2\} & = \mathcal{D}\{t_1\}\mathcal{D}\{t_2\} \\ \mathcal{D}\{\textbf{let } x = t_1 \textbf{ in } t_2\} & = \textbf{let } x = \mathcal{D}\{t_1\} \textbf{ in } \mathcal{D}\{t_2\} \\ \mathcal{D}\{(t_1, t_2)\} & = (\mathcal{D}\{t_1\}, \mathcal{D}\{t_2\}) \end{array} \qquad \begin{array}{ll} \mathcal{D}\{\textbf{fst } t\} & = \textbf{fst } \mathcal{D}\{t\} \\ \mathcal{D}\{\textbf{snd } t\} & = \textbf{snd } \mathcal{D}\{t\} \\ \mathcal{D}\{r : \kappa\} & = (r, 0) \\ \mathcal{D}\{r : \mathbb{N}\} & = r \end{array} \qquad \begin{array}{ll} \mathcal{D}\{x\} & = x \\ \mathcal{D}\{()\} & = () \\ \mathcal{D}\{c\} & = c_{\mathcal{D}} \end{array}$$

Fig. 4. The standard forward-mode AD translation as a whole program transformation. Note that the types of variables $x : \tau$ (both free and bound) are changed to $x : \mathcal{D}\{\tau\}$. For every primitive $c : \tau$ of the source language, $c_{\mathcal{D}} : \mathcal{D}\{\tau\}$ is its built-in derivative. $\mathcal{D}\{-\}$ is a typed-translation: if $\Gamma \vdash t : \tau$, then $\mathcal{D}\{\Gamma\} \vdash \mathcal{D}\{t\} : \mathcal{D}\{\tau\}$.

If the first step is done correctly, the correctness of the second step should be clear. Therefore, the content of the forward-mode AD algorithm mostly lives in the definition of the $\mathcal{D}\{\cdot\}$ macro, and in the proof of its correctness. To emphasize this, we restate the property we need $[\![\mathcal{D}\{t\}]\!]$ to satisfy if we want the second step above to follow:

Definition 2.1 (correct dual-number derivative at $\mathbb{R}$). *Let $f : \mathbb{R} \to \mathbb{R}$ be a differentiable function. Then $f_D : \mathbb{R} \times \mathbb{R} \to \mathbb{R} \times \mathbb{R}$ is a correct dual-number derivative of f if for all differentiable $h : \mathbb{R} \to \mathbb{R}$, $f_D(h(\theta), h'(\theta)) = ((f \circ h)(\theta), (f \circ h)'(\theta))$.*

Then the AD macro is correct if it computes these dual-number derivatives:

Definition 2.2 (correctness of $\mathcal{D}\{\cdot\}$). *The AD macro $\mathcal{D}\{\cdot\}$ is correct if, for all closed terms $\vdash t : \mathbb{R} \to \mathbb{R}$, $[\![\vdash \mathcal{D}\{t\} : \mathbb{R} \times \mathbb{R} \to \mathbb{R} \times \mathbb{R}]\!]$ is a correct dual-number derivative of $[\![\vdash t : \mathbb{R} \to \mathbb{R}]\!]$.*

Defining the AD Macro. The AD macro $\mathcal{D}\{\cdot\}$ itself is given in Figure 4. In every place that real numbers (of type $\mathbb{R}$) appeared in the original program, they are now replaced by dual numbers (of type $\mathbb{R} \times \mathbb{R}$). This affects the type of every term in the program, and the assumed types of any free variables in the context; we write $\mathcal{D}\{\tau\}$ for the type that terms of type τ have after translation to use dual-numbers. Since reals are replaced by pairs of reals, we have $\mathcal{D}\{\mathbb{R}\} = \mathbb{R} \times \mathbb{R}$ (and more generally, $\mathcal{D}\{\kappa\} = \kappa \times \mathbb{R}$ for all smooth base types κ). Because functions into $\mathbb{N}$ have no derivative information to track, $\mathcal{D}\{\mathbb{N}\} = \mathbb{N}$. On product and function types, $\mathcal{D}\{\cdot\}$ is defined recursively: $\mathcal{D}\{\tau_1 \times \tau_2\} = \mathcal{D}\{\tau_1\} \times \mathcal{D}\{\tau_2\}$ and $\mathcal{D}\{\tau_1 \to \tau_2\} = \mathcal{D}\{\tau_1\} \to \mathcal{D}\{\tau_2\}$.

When applied to a term $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t : \tau$, AD produces a new term $x_1 : \mathcal{D}\{\tau_1\}, \ldots, x_n : \mathcal{D}\{\tau_n\} \vdash \mathcal{D}\{t\} : \mathcal{D}\{\tau\}$. The new term is mostly the same as the old term—only two things change:

- Constant real numbers r are replaced with constant *dual* numbers $(r, 0)$ with 0 derivative.
- Primitives $c : \tau \to \sigma$ are translated into new, target-language primitives $c_D : \mathcal{D}\{\tau\} \to \mathcal{D}\{\sigma\}$. For each c, c_D is a built-in dual-number derivative for the primitive c (though we have yet to make this precise, except when $\tau = \sigma = \mathbb{R}$).

Notation 2.2. *In examples, we use the variable naming convention $dx = (x, \delta x)$ for dual numbers.*

The semantics of the new primitives c_D are given in Figure 5. When $c : \mathbb{R} \to \mathbb{R}$, $[\![c_D]\!]$ has the form

$$[\![c_D]\!] = \lambda(x, \delta x).([\![c]\!](x), [\![c]\!]'(x) \cdot \delta x),$$

which ensures that when run on a pair $(h(\theta), h'(\theta))$, it computes $([\![c]\!](h(\theta)), [\![c]\!]'(h(\theta)) \cdot h'(\theta))$, the second component of which uses the familiar chain rule from calculus to compute $([\![c]\!] \circ h)'(\theta)$.

$$
\begin{array}{ll}
[\![\mathsf{exp}_{\mathcal{D}}]\!](x,\delta x) & = \textbf{let } y = \exp\ x \\
 & \quad \textbf{in } (y, y \times \delta x) \\
[\![\mathsf{pow}_{\mathcal{D}}]\!](dx, 0) & = (0,0)
\end{array}
\qquad
\begin{array}{ll}
[\![(+_{\mathbb{R}})_{\mathcal{D}}]\!]((x,\delta x),(y,\delta y)) & = (x+y, \delta x + \delta y) \\
[\![(\times_{\mathbb{R}})_{\mathcal{D}}]\!]((x,\delta x),(y,\delta y)) & = (x \times y, \delta x \times y + x \times \delta y) \\
[\![\mathsf{pow}_{\mathcal{D}}]\!]((x,\delta x), n+1) & = \textbf{let } y = \mathsf{pow}(x,n) \textbf{ in} \\
 & \quad (x \times y, (n+1) \times y \times \delta x)
\end{array}
$$

Fig. 5. Dual number interpretation of deterministic primitives. On the right we use syntax for simplicity, but it should be understood as metalanguage syntax. We write $dx = (x, \delta x)$ for dual numbers. Note that each primitive implements the chain rule, multiplying the derivative with respect to x by δx.

2.3 Proof Technique: Reasoning about Correctness with Logical Relations

We now review a powerful proof technique for reasoning about AD and establishing its correctness, based on logical relations [Ahmed 2006; Barthe et al. 2020; Huot et al. 2020; Katsumata 2013; Krawiec et al. 2022]. It may seem like overkill for such a simple algorithm, but the technique will shine when we try to make sense of highly non-standard extensions to forward-mode AD, in Secs. 3-6.

The Challenge with Simple Proof by Induction. We might hope we could establish AD's correctness with a simple proof by induction: if AD is correct for each subterm in a program, it is correct for the whole program.[2] The challenge is that the notion of *correctness* we gave in Definition 2.1 applies only to translations of closed $\mathbb{R} \to \mathbb{R}$ programs. The meaning of an open subterm, $[\![\Gamma \vdash t : \tau]\!]$, will in general be a function from environments to values of type τ (which may not be $\mathbb{R}$). The meaning of its translation, $[\![\mathcal{D}\{\Gamma\} \vdash \mathcal{D}\{t\} : \mathcal{D}\{\tau\}]\!]$, will also be a function, from dual-number environments to values of type $\mathcal{D}\{\tau\}$. Our simple correctness criterion about differentiable $\mathbb{R} \to \mathbb{R}$ functions cannot be applied here, and so it is unclear what inductive hypothesis a proof by induction would use.

The Logical Relations Approach. The logical relations proof technique circumvents this issue by defining a *different* inductive hypothesis for each type. In proofs about AD, what this means is that we ultimately define a different notion of *correct dual-number derivative* for functions between *any* two types in our language. Once we've done this, we can then proceed with an ordinary proof by induction: if the translation of each subterm $\Gamma \vdash t : \tau$ yields a correct dual-number derivative of the $[\![\Gamma]\!] \to [\![\tau]\!]$ function that t denotes, then the translation of the enclosing term is also correct.

But how can we define correct dual-number derivatives for functions $f : [\![\tau_1]\!] \to [\![\tau_2]\!]$ between arbitrary types? Looking more closely at Definition 2.1, we can see that it phrases correctness for $f : \mathbb{R} \to \mathbb{R}$ functions in a slightly non-standard way: f_D is a correct derivative if, when composed with a function h's derivative, it *preserves the relationship* that h and h' enjoyed, of "being a derivative." This motivates a more general approach to defining correctness based on the idea of preserving the relationship between a function and its derivative. We proceed in two stages:

- First, for each type τ, we define a notion of derivative for $\mathbb{R} \to [\![\tau]\!]$ functions: a relation between an $\mathbb{R} \to [\![\tau]\!]$ function and an $\mathbb{R} \to [\![\mathcal{D}\{\tau\}]\!]$ function encoding what it means to be a derivative.
- Then, for arbitrary functions $f : [\![\tau_1]\!] \to [\![\tau_2]\!]$, we define correctness as the *preservation* of this relationship: we look at what happens when f (and its translation) are composed with $\mathbb{R} \to [\![\tau_1]\!]$ functions (and their $\mathbb{R} \to [\![\mathcal{D}\{\tau_1\}]\!]$ derivatives), and check that what we get out are related $\mathbb{R} \to [\![\tau_2]\!]$ and $\mathbb{R} \to [\![\mathcal{D}\{\tau_2\}]\!]$ functions.

More precisely, in step 1, we define for each type τ a *dual-number relation* $\mathcal{R}_\tau$ encoding what it means to be a derivative of an $\mathbb{R} \to [\![\tau]\!]$ function:

[2] Technically, the induction is usually over the typing derivation of the term, and what we call "subterms" are really subtrees of the typing derivations corresponding to the premises of the bottom-most inference rule in the typing derivation.

Dual-Number Logical Relations $\mathscr{R}_\tau$ for the Deterministic Language (§2)

$$\mathscr{R}_{\mathbb{R}} = \left\{(f : \mathbb{R} \to \mathbb{R}, g : \mathbb{R} \to \mathbb{R} \times \mathbb{R}) \mid f \text{ differentiable} \wedge \forall \theta \in \mathbb{R}. g(\theta) = (f(\theta), f'(\theta))\right\}$$

$$\mathscr{R}_{\mathbb{R}_{>0}} = \left\{(f : \mathbb{R} \to \mathbb{R}_{>0}, g : \mathbb{R} \to \mathbb{R}_{>0} \times \mathbb{R}) \mid (\iota_{\mathbb{R}_{>0}} \circ f, \langle \iota_{\mathbb{R}_{>0}}, id\rangle \circ g) \in \mathscr{R}_{\mathbb{R}}\right\}$$

$$\mathscr{R}_{\mathbb{I}} = \left\{(f : \mathbb{R} \to \mathbb{I}, g : \mathbb{R} \to \mathbb{I} \times \mathbb{R}) \mid (\iota_{\mathbb{I}} \circ f, \langle \iota_{\mathbb{I}}, id\rangle \circ g) \in \mathscr{R}_{\mathbb{R}}\right\}$$

$$\mathscr{R}_{\mathbb{N}} = \left\{(f : \mathbb{R} \to \mathbb{N}, g : \mathbb{R} \to \mathbb{N}) \mid f \text{ is constant } \wedge f = g\right\}$$

$$\mathscr{R}_{\tau_1 \times \tau_2} = \left\{(f : \mathbb{R} \to [\![\tau_1 \times \tau_2]\!], g : \mathbb{R} \to [\![\mathcal{D}\{\tau_1\} \times \mathcal{D}\{\tau_2\}]\!]) \mid (\pi_1 \circ f, \pi_1 \circ g) \in \mathscr{R}_{\tau_1} \wedge (\pi_2 \circ f, \pi_2 \circ g) \in \mathscr{R}_{\tau_2}\right\}$$

$$\mathscr{R}_{\tau_1 \to \tau_2} = \left\{(f : \mathbb{R} \to [\![\tau_1 \to \tau_2]\!], g : \mathbb{R} \to [\![\mathcal{D}\{\tau_1 \to \tau_2\}]\!]) \mid \forall (j, k) \in \mathscr{R}_{\tau_1}. (\lambda r. f(r)(j(r)), \lambda r. g(r)(k(r))) \in \mathscr{R}_{\tau_2}\right\}$$

Fig. 6. Definition of the dual-number logical relation at each type. For each smooth base type κ, we write $\iota_\kappa : [\![\kappa]\!] \to \mathbb{R}$ for the canonical injection.

Definition 2.3 (dual-number relation). *For a type τ, a dual-number relation for τ is a relation $\mathscr{R}_\tau$ over the sets $\mathbb{R} \to [\![\tau]\!]$ and $\mathbb{R} \to [\![\mathcal{D}\{\tau\}]\!]$, that is, a subset $\mathscr{R}_\tau \subseteq (\mathbb{R} \to [\![\tau]\!]) \times (\mathbb{R} \to [\![\mathcal{D}\{\tau\}]\!])$.*

For each τ, we choose $\mathscr{R}_\tau$ so that it relates continuously-parameterized $[\![\tau]\!]$ values (i.e., curves $\mathbb{R} \to [\![\tau]\!]$) with continuously-parameterized dual-number values (curves $\mathbb{R} \to [\![\mathcal{D}\{\tau\}]\!]$) that use their dual-number storage to correctly track a local linear approximation to how the $[\![\tau]\!]$ value is changing when the real-valued parameter changes. This allows us to define correct dual-number derivatives at any type automatically:

Definition 2.4 (correct dual-number derivative (general)). *Suppose that dual-number relations $\mathscr{R}_\tau$ have been chosen for every τ, and let $f : [\![\tau_1]\!] \to [\![\tau_2]\!]$. We say $f_D : [\![\mathcal{D}\{\tau_1\}]\!] \to [\![\mathcal{D}\{\tau_2\}]\!]$ is a correct dual-number derivative of f if, for all $(g, g') \in \mathscr{R}_{\tau_1}$, the functions $(f \circ g, f_D \circ g') \in \mathscr{R}_{\tau_2}$.*

This last definition is the one we will use as our inductive hypothesis in proving the AD macro correct overall. Although the proof by induction ultimately needs to cover all terms, the power of the technique is that the inductive steps are largely covered by existing, well-studied machinery, and so most of the AD-specific action happens at base types and primitives.

2.4 Proof of Correctness for Forward-mode AD

We now apply the logical relations technique from the last section to prove our AD macro correct. To do so, we define dual-number relations $\mathscr{R}_\tau$ which encode a notion of *derivative* at each type (Figure 6). For the reals (and continuous subsets of the reals), this notion coincides with the usual derivative: the relation $\mathscr{R}_{\mathbb{R}}$, for example, relates a differentiable function f with the function $g(\theta) = (f(\theta), f'(\theta))$. For discrete types, such as $\mathbb{N}$, because the derivative will necessarily be zero, we can avoid storing a dual number.

We then need to prove what is often called the *fundamental lemma* of a logical relations argument:

Lemma 2.1 (Fundamental lemma). *For every term-in-context $\Gamma \vdash t : \tau$, $[\![\mathcal{D}\{t\}]\!]$ is a correct dual-number derivative of $[\![t]\!]$, with respect to the relations $\mathscr{R}_\tau$ given in Figure 6.*

This is proved by induction on the typing derivation of t, and as our definitions of $\mathscr{R}_\tau$ for product types and function types are completely standard, the inductive cases can be handled by standard logical relations machinery [Huot et al. 2020]. The only interesting cases are the base cases, where we must show that the interpretation of every primitive function c has a correct dual-number derivative given by the interpretation of its translation c_D (Definition 2.1).

The last step is to use our proof of the fundamental lemma to establish the more basic correctness criterion we outlined in Definition 2.2:

Theorem 2.2 (correctness of forward-mode AD [Huot et al. 2020]). *For all closed terms* $\vdash t : \mathbb{R} \to \mathbb{R}$, $[\![\lambda\theta : \mathbb{R}.\mathbf{snd}(\mathcal{D}\{t\}\ (\theta, 1))]\!]$ *is the derivative of* $[\![t]\!]$.

Proof. By the fundamental lemma (2.1), $[\![\mathcal{D}\{t\}]\!] \in \mathscr{R}_{\mathbb{R}\to\mathbb{R}}$, so for functions $(f, g) \in \mathscr{R}_{\mathbb{R}}$, we have $([\![t]\!] \circ f, [\![\mathcal{D}\{t\}]\!] \circ g) \in \mathscr{R}_{\mathbb{R}}$. Take $f = id$ and $g = \lambda\theta.(\theta, 1)$, and note that $(f, g) \in \mathscr{R}_{\mathbb{R}}$, because $g(\theta) = (f(\theta), f'(\theta))$. Then $([\![t]\!] \circ f, [\![\mathcal{D}\{t\}]\!] \circ g) = ([\![t]\!], \lambda\theta.[\![\mathcal{D}\{t\}]\!](\theta, 1)) \in \mathscr{R}_{\mathbb{R}}$, and so by the definition of $\mathscr{R}_{\mathbb{R}}$, for all $\theta \in \mathbb{R}$, $[\![\mathcal{D}\{t\}]\!](\theta, 1) = ([\![t]\!](\theta), [\![t]\!]'(\theta))$. Applying π_2 to extract just the second component, we have $\pi_2[\![\mathcal{D}\{t\}]\!](\theta, 1) = [\![t]\!]'(\theta)$. As a function of θ, the left-hand side is precisely $[\![\lambda\theta : \mathbb{R}.\mathbf{snd}(\mathcal{D}\{t\}(\theta, 1))]\!]$, and the right-hand side is the derivative of $[\![t]\!]$. □

3 WARM-UP: DIFFERENTIATING A PROBABILISTIC COMBINATOR DSL

Now that we have set the stage, we can begin to introduce our main characters: new types and terms for probabilistic programming. In this section, we tackle only a small warm-up extension: we study the most basic setting where probability arises, a simple and restrictive DSL for composing probability distributions with combinators. Unlike general probabilistic programming languages, which we study in Sections 4-6, the DSL in this section does not allow for arbitrary sequencing of probabilistic computations. Despite the simplicity of this setting, our development here provides an important foundation for the fancier extensions we will add next.

3.1 The Type $\widetilde{\mathbb{R}}$ of Random Real Numbers

Typically, in differentiable programming languages, users aim to construct a closed expression $\vdash t : \mathbb{R} \to \mathbb{R}$, implementing a differentiable function whose derivative they wish to compute. But for the rest of this paper, we consider a different workflow: instead of constructing a program of type $\mathbb{R} \to \mathbb{R}$, the user constructs a program of type $\mathbb{R} \to \widetilde{\mathbb{R}}$, where $\widetilde{\mathbb{R}}$ is a new type of *random* real numbers, whose *expected values* are the quantities of interest. We call the values of $\widetilde{\mathbb{R}}$ *unbiased real-valued estimators*: sampling them yields unbiased estimates of the true values we care about.

Definition 3.1 (real-valued estimator). *We denote by* $\widetilde{\mathbb{R}}$ *the set of unbiased real-valued estimators: probability measures* μ *on the measurable space* $(\mathbb{R}, \mathcal{B}(\mathbb{R}))$. *If* $\mathbb{E}_{x\sim\mu}[x] = \int x\mu(dx)$ *exists, i.e. is finite and equal to some number* $r \in \mathbb{R}$, *we say* μ *estimates (or is an unbiased estimator of)* r.

Remark 3.1. *A distribution* $\mu \in \widetilde{\mathbb{R}}$ *need not have a density function, and may be supported on a finite, countable, or uncountable set of reals. For example, the Dirac distribution,* δ_r, *which assigns all its mass to the number* r, *is an unbiased estimator of* r, *as is the Gaussian distribution* $\mathcal{N}(r, 1)$.

Although the user's program $\vdash \widetilde{t} : \mathbb{R} \to \widetilde{\mathbb{R}}$ denotes a map into the space of probability distributions, the function they wish to differentiate is the loss function $\mathcal{L} : \mathbb{R} \to \mathbb{R} = \lambda\theta.\mathbb{E}_{x\sim[\![\widetilde{t}]\!](\theta)}[x]$, if $\mathcal{L}$ is well-defined (i.e., if the expectation always exists). As illustrated in Figure 1, applying ADEV to the program $\widetilde{t}$, we get a new program $\vdash \widetilde{s} : \mathbb{R} \to \widetilde{\mathbb{R}}$ that estimates $\mathcal{L}'$, the derivative of the loss. It will be useful to have a word for the relationship between $[\![\widetilde{t}]\!]$ and $[\![\widetilde{s}]\!]$; we coin *unbiased derivative*:

Definition 3.2 (unbiased derivative). *Given a function* $\widetilde{f} : U \to \widetilde{\mathbb{R}}$, *for* $U \subseteq \mathbb{R}$, *suppose* $\mathcal{L} : U \to \mathbb{R} = \lambda\theta.\mathbb{E}_{x\sim\widetilde{f}(\theta)}[x]$ *is well-defined and differentiable. We say that a function* $\widetilde{g} : U \to \widetilde{\mathbb{R}}$ *is an unbiased derivative of* $\widetilde{f}$ *if for all* $\theta \in U$, $\widetilde{g}(\theta)$ *estimates* $\mathcal{L}'(\theta)$, *that is,*

$$\mathbb{E}_{x\sim\widetilde{g}(\theta)}[x] = \mathcal{L}'(\theta) = \frac{d}{d\theta}\mathbb{E}_{x\sim\widetilde{f}(\theta)}[x].$$

Note that estimator-valued functions may have many unbiased derivatives.

$$\text{Types } \tau ::= \ldots \mid \widetilde{\mathbb{R}} \mid \widetilde{\mathbb{R}}_{\mathcal{D}}$$

$$\text{Primitives } c ::= \ldots \mid \textbf{minibatch} : \mathbb{N} \to \mathbb{N} \to (\mathbb{N} \to \mathbb{R}) \to \widetilde{\mathbb{R}} \mid \textbf{fst}_*, \textbf{snd}_* : \widetilde{\mathbb{R}}_{\mathcal{D}} \to \widetilde{\mathbb{R}}$$
$$\mid +^{\widetilde{\mathbb{R}}}, \times^{\widetilde{\mathbb{R}}} : \widetilde{\mathbb{R}} \times \widetilde{\mathbb{R}} \to \widetilde{\mathbb{R}} \mid exp^{\widetilde{\mathbb{R}}} : \widetilde{\mathbb{R}} \to \widetilde{\mathbb{R}} \mid \textbf{exact} : \mathbb{R} \to \widetilde{\mathbb{R}}$$

Fig. 7. Syntax for the Probabilistic Combinator DSL (§3), as an extension to Fig. 3. Gray highlights indicate syntax only present in the target language of the AD macro.

Semantics of types:

$$[\![\widetilde{\mathbb{R}}]\!] = \{\mu \mid \mu \text{ a probability measure on } (\mathbb{R}, \mathcal{B}(\mathbb{R}))\}$$
$$[\![\widetilde{\mathbb{R}}_{\mathcal{D}}]\!] = \{\mu \mid \mu \text{ a probability measure on } (\mathbb{R} \times \mathbb{R}, \mathcal{B}(\mathbb{R} \times \mathbb{R}))\}$$

Example primitive and its built-in derivative:

$\times^{\widetilde{\mathbb{R}}}(\widetilde{x} : \widetilde{\mathbb{R}}, \widetilde{y} : \widetilde{\mathbb{R}})$:
  $r \sim \widetilde{x}$
  $s \sim \widetilde{y}$,
  return $r \times s$
end

$\times^{\widetilde{\mathbb{R}}}_{\mathcal{D}}(\widetilde{dx} : \widetilde{\mathbb{R}}_{\mathcal{D}}, \widetilde{dy} : \widetilde{\mathbb{R}}_{\mathcal{D}})$:
  $dr \sim \widetilde{dx}$
  $ds \sim \widetilde{dy}$,
  return $dr \times_{\mathcal{D}} ds$
end

Fig. 8. Semantics of the new types for the Combinator DSL and an example of a new primitive.

$$\mathcal{D}\{\widetilde{\mathbb{R}}\} = \widetilde{\mathbb{R}}_{\mathcal{D}} \qquad \mathcal{D}\{exp^{\widetilde{\mathbb{R}}}\} = exp^{\widetilde{\mathbb{R}}}_{\mathcal{D}}$$
$$\mathcal{D}\{\textbf{minibatch}\} = \textbf{minibatch}_{\mathcal{D}} \qquad \mathcal{D}\{\textbf{exact}\} = \textbf{exact}_{\mathcal{D}}$$
$$\mathcal{D}\{+^{\widetilde{\mathbb{R}}}\} = +^{\widetilde{\mathbb{R}}}_{\mathcal{D}} \qquad \mathcal{D}\{\times^{\widetilde{\mathbb{R}}}\} = \times^{\widetilde{\mathbb{R}}}_{\mathcal{D}}$$

Fig. 9. ADEV macro for the Combinator DSL (§3), extending Fig. 4

3.2 Syntax and Semantics of the Combinator DSL

We now present our combinator language, a toy DSL for constructing values of type $\widetilde{\mathbb{R}}$. The syntax is given in Fig. 7, and the semantics in Fig. 8 (some primitives deferred to Fig. 24).

Beyond the new base type $\widetilde{\mathbb{R}}$, our extended source language exposes a small collection of combinators for implementing stochastic loss functions. The **minibatch** primitive constructs a probabilistic estimator of a large sum $\sum_{i=1}^{M} f(i)$, that works by subsampling $m \ll M$ indices $(i_1, \ldots, i_m)$ uniformly at random, and evaluating f only at those indices, returning $\frac{M}{m}\sum_{j=1}^{m} f(i_j)$. The **exact** primitive constructs the trivial deterministic estimator of a real value that returns the value with probability 1. Our other new primitives *transform* existing estimators, creating a new estimator with expected value equal to some function (e.g., a sum, product, or exponentiation) of the inputs' expected values. We give one example ($\times^{\widetilde{\mathbb{R}}}$) in Figure 8; the full semantics can be found in Appendix C, Figure 24.

3.3 ADEV for the Combinator DSL: Differentiating through $\widetilde{\mathbb{R}}$

Suppose a user has written a program $\vdash \widetilde{t} : \mathbb{R} \to \widetilde{\mathbb{R}}$, representing a stochastic estimator $[\![\widetilde{t}]\!]$ of a loss function $\mathcal{L}(\theta) = \mathbb{E}_{x \sim [\![\widetilde{t}]\!](\theta)}[x]$. Our algorithm, ADEV, differentiates $\mathcal{L}$ by constructing a program $\vdash \widetilde{s} : \mathbb{R} \to \widetilde{\mathbb{R}}$ that implements an *unbiased derivative* of $[\![\widetilde{t}]\!]$ (Definition 3.2), in two steps:

- First, we will apply an extended version of the AD macro $\mathcal{D}\{\cdot\}$ to the user's program, yielding a new program $\mathcal{D}\{\widetilde{t}\} : \mathbb{R} \times \mathbb{R} \to \widetilde{\mathbb{R}}_{\mathcal{D}}$. It accepts a *dual number* as input, and instead of estimating a single real value, estimates a dual number value: the type $\widetilde{\mathbb{R}}_{\mathcal{D}}$ denotes the set of probability distributions over *pairs* of reals. The key correctness property in this extended setting is that if we are given as an input dual number $(h(\theta), h'(\theta))$ for some differentiable function h, then $[\![\mathcal{D}\{\widetilde{t}\}]\!]$ should send it to an estimator of the dual number $((\mathcal{L} \circ h)(\theta), (\mathcal{L} \circ h)'(\theta))$.

- Second, we output the term $\widetilde{s} = \lambda\theta : \mathbb{R}.\mathbf{snd}_*(\mathcal{D}\{\widetilde{t}\}(\theta, 1))$. As before, since $(\theta, 1) = (id(\theta), id'(\theta))$, we know $\pi_{2*} \circ [\![\mathcal{D}\{\widetilde{t}\}]\!]$ returns a new estimator that estimates $(\mathcal{L} \circ id)'(\theta)$, as desired.

As in the standard AD algorithm from Section 2, the correctness of the second step follows directly if we can prove the first step works correctly, so we now turn to extending $\mathcal{D}\{\cdot\}$.

Defining the ADEV Macro at the Type Level. Our AD macro from Section 2 had one key job: replacing every real number flowing through the program with a dual number, and all real number operations with dual number operations. In doing so, it translated terms of type τ to terms of type $\mathcal{D}\{\tau\}$—the dual-number version of the type τ. We now have a type of *estimated* real numbers, $\widetilde{\mathbb{R}}$, and the dual-number version of an estimator should be an *estimator* of a dual number:

Definition 3.3 (unbiased dual-number estimator). *We denote by $\widetilde{\mathbb{R}}_{\mathcal{D}}$ the set of unbiased dual-number estimators: probability measures μ on $(\mathbb{R} \times \mathbb{R}, \mathcal{B}(\mathbb{R} \times \mathbb{R}))$. If $\mathbb{E}_{(x,\delta x)\sim\mu}[x] = r$ and $\mathbb{E}_{(x,\delta x)\sim\mu}[\delta x] = \delta r$ for finite real numbers r and δr, we say that μ estimates the dual number $(r, \delta r)$.*

This type, which appears in the target language in Figure 7 but not in our source language, represents random processes for estimating a dual number. Note that it allows for the two components of the estimate to depend on the same random choices: it is a distribution over pairs, not a pair of distributions. We set $\mathcal{D}\{\widetilde{\mathbb{R}}\} := \widetilde{\mathbb{R}}_{\mathcal{D}}$.

Defining the ADEV Macro at the Term Level. To extend the macro $\mathcal{D}\{\cdot\}$ from Section 2 to handle our extended language, we need to say what it does on each new term. But the only new terms we have added to our source language are the new primitives, like **minibatch** and $exp^{\widetilde{\mathbb{R}}}$. Thus, the only new behavior we need to specify is how to translate each primitive—in other words, we need to attach to each new primitive $c : \tau \to \sigma$ a custom built-in derivative $c_{\mathcal{D}}$. We give one example in Figure 8, with the full list in Appendix C, Figure 24.

If a primitive $c : \tau \to \widetilde{\mathbb{R}}$ builds an estimator of some loss, the goal of $c_{\mathcal{D}} : \mathcal{D}\{\tau\} \to \widetilde{\mathbb{R}}_{\mathcal{D}}$ is to estimate both the loss and the derivative of the loss. In many cases, this is quite straightforward. For example, the primitive $\times^{\widetilde{\mathbb{R}}}$ in Fig. 8 estimates the product of the two numbers x and y that its input arguments $\widetilde{x}$ and $\widetilde{y}$ estimate. Its built-in derivative $\times^{\widetilde{\mathbb{R}}}_{\mathcal{D}}$ does the same but with dual numbers: it independently generates estimates $dr = (r, \delta r)$ of $(x, \delta x)$ and $ds = (s, \delta s)$ of $(y, \delta y)$, then returns $(r, \delta r) \times_{\mathcal{D}} (s, \delta s) = (rs, s\delta r + r\delta s)$. Because r and s are independent random variables, their product's expectation is the product of their expectations, $\mathbb{E}[rs] = xy$. And by linearity of expectation, $\mathbb{E}[s\delta r + r\delta s] = \mathbb{E}[s\delta r] + \mathbb{E}[r\delta s]$; exploiting again the fact that s and δr are independent (and likewise for r and δs), we obtain the desired result $y\delta x + x\delta y$. With such built-in derivatives for all the primitives (Figure 24), the ADEV macro now covers the new source language.

3.4 Correctness Criterion for ADEV

Before trying to prove ADEV correct, let's formulate a definition of correctness for the programs it produces (an updated version of Definition 2.1):

Definition 3.4 (correct dual-number derivative at $\widetilde{\mathbb{R}}$). *Let $\widetilde{f} : \mathbb{R} \to \widetilde{\mathbb{R}}$ be an estimator-valued function, and suppose that the map $\mathcal{L} : \mathbb{R} \to \mathbb{R}$ that sends θ to $\mathbb{E}_{x\sim\widetilde{f}(\theta)}[x]$ is well-defined (i.e., the expectation exists for all θ) and differentiable. Then $\widetilde{f_D} : \mathbb{R} \times \mathbb{R} \to \widetilde{\mathbb{R}}_{\mathcal{D}}$ is a correct dual-number derivative of $\widetilde{f}$ if for all differentiable $h : \mathbb{R} \to \mathbb{R}$, the dual number estimator $\widetilde{f_D}(h(\theta), h'(\theta))$ estimates the dual number $((\mathcal{L} \circ h)(\theta), (\mathcal{L} \circ h)'(\theta))$.*

Then our macro should compute these correct dual-number derivatives:

$$\mathscr{R}_{\widetilde{\mathbb{R}}} = \{(\widetilde{f} : \mathbb{R} \to \widetilde{\mathbb{R}}, \widetilde{g} : \mathbb{R} \to \widetilde{\mathbb{R}}_{\mathcal{D}}) \mid \mathcal{L} := \theta \mapsto \mathbb{E}_{x \sim \widetilde{f}(\theta)}[x] \text{ is well-defined and differentiable} \\ \wedge \text{ for all } \theta \in \mathbb{R},\ \widetilde{g}(\theta) \text{ estimates } (\mathcal{L}(\theta), \mathcal{L}'(\theta)) \text{ (Def. 3.3)}\}$$

Fig. 10. Logical relation for the Probabilistic Combinator DSL

Definition 3.5 (correctness of $\mathcal{D}\{\cdot\}$ (ADEV)). *The ADEV macro $\mathcal{D}\{\cdot\}$ is correct if for all closed terms $\vdash \widetilde{t} : \mathbb{R} \to \widetilde{\mathbb{R}}$, $[\![\vdash \mathcal{D}\{\widetilde{t}\} : \mathbb{R} \times \mathbb{R} \to \widetilde{\mathbb{R}}_{\mathcal{D}}]\!]$ is a correct dual-number derivative of $[\![\vdash \widetilde{t} : \mathbb{R} \to \widetilde{\mathbb{R}}]\!]$.*

This notion of correctness is *intensional* [Lee et al. 2020a], in that there is more than one correct dual-number derivative of a function $\widetilde{f} : \mathbb{R} \to \widetilde{\mathbb{R}}$. In practice, which derivative a user gets will depend on which primitives a user's program invokes, intuitively because "all the action" in forward-mode AD happens at the primitives (each primitive is equipped with a built-in derivative, and these are composed to implement a program's derivative). By providing users with a library of primitives, some of which have the same meaning but different built-in derivatives, we give users a compositional way to explore the space of gradient estimation strategies (see Section 4.1).

3.5 Proving the ADEV Algorithm Correct

To extend Section 2's proof to cover the ADEV algorithm, we need to:

(1) Define a dual-number relation $\mathscr{R}_{\widetilde{\mathbb{R}}} \subseteq (\mathbb{R} \to \widetilde{\mathbb{R}}) \times (\mathbb{R} \to \widetilde{\mathbb{R}}_{\mathcal{D}})$, characterizing when a function that estimates dual numbers is a correct derivative of a function that estimates reals. ***Intuition:*** this step is about defining a notion of derivative for $\mathbb{R} \to \widetilde{\mathbb{R}}$ functions; we will base our choice on our earlier Definition 3.4.
(2) Prove an updated version of the fundamental lemma (Lemma 2.1) for the extended language, with respect to all the old relations $\mathscr{R}_\tau$, but also the new relation $\mathscr{R}_{\widetilde{\mathbb{R}}}$. ***Intuition:*** this step updates an inductive proof now that we have more base cases (new primitives). Luckily, the inductive steps don't change at all, and it suffices to check the base cases. Concretely, this means showing that each of our new primitives has a correct built-in derivative, using the definition of 'correctness' arising from our choice in step (1) together with Definition 2.4.
(3) Prove an updated version of Theorem 2.2, to show how correctness of ADEV follows from the updated fundamental lemma. ***Intuition:*** This step shows that if our program estimates dual numbers correctly (implied by step 2), then our "wrapper" that extracts the second component of the dual number to return as the derivative is correct. As in Section 2, this step is straightforward.

For the first step, we extend our definitions of dual-number relations $\mathscr{R}_\tau$ from Section 2 to cover our new type, $\widetilde{\mathbb{R}}$. The new relation $\mathscr{R}_{\widetilde{\mathbb{R}}}$ is presented in Figure 10, and captures what it means to be a correct dual-number derivative estimator. Since our logical relations have changed, we need to reprove the fundamental lemma:

Lemma 3.1 (fundamental lemma (revised with $\widetilde{\mathbb{R}}$)). *For every term $\Gamma \vdash t : \tau$, $[\![\mathcal{D}\{t\}]\!]$ is a correct dual-number derivative of $[\![t]\!]$, with respect to the relations $\mathscr{R}_\tau$ defined at each type (incl. $\widetilde{\mathbb{R}}$).*

Proof. The proof is the same inductive proof we used for Lemma 2.1, except that there are now new base cases: we must show that the interpretation of every new primitive function c has a correct dual-number derivative given by the interpretation of its translation c_D.

- For **exact** $: \mathbb{R} \to \widetilde{\mathbb{R}}$, we must check that $\textbf{exact}_{\mathcal{D}}(h(\theta), h'(\theta))$ estimates $(h(\theta), h'(\theta))$ for differentiable h (which it clearly does: it returns its input dual number exactly).

$$\text{Types } \tau ::= \ldots \mid \mathbb{B} \mid P\,\tau \mid \boxed{P_{\mathcal{D}}\,\tau}$$

$$\text{Terms } t ::= \ldots \mid \textbf{True} \mid \textbf{False} \mid \textbf{if } t \textbf{ then } t_1 \textbf{ else } t_2 \mid \textbf{do}\,\{\,m\,\} \mid \boxed{\textbf{do}_{\mathcal{D}}\,\{\,m\,\}}$$
$$\mid \textbf{return}\,t \mid \boxed{\textbf{return}_{\mathcal{D}}\,t}$$

$$\text{Do notation } m ::= t \mid x \leftarrow t;\, m$$

$$\text{Primitives } c ::= \ldots \mid \textbf{flip}_{\texttt{REINFORCE}}, \textbf{flip}_{\texttt{ENUM}} : \mathbb{I} \to P\,\mathbb{B} \mid \mathbb{E} : P\,\mathbb{R} \to \widetilde{\mathbb{R}}$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \textbf{return}\,t : P\,\tau} \qquad \frac{\Gamma \vdash t : P\,\tau}{\Gamma \vdash \textbf{do}\{t\} : P\,\tau} \qquad \frac{\Gamma \vdash t : P\,\tau_1 \quad \Gamma, x : \tau_1 \vdash \textbf{do}\{m\} : P\,\tau}{\Gamma \vdash \textbf{do}\{x \leftarrow t; m\} : P\,\tau}$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \textbf{return}_{\mathcal{D}}\,t : P_{\mathcal{D}}\,\tau} \qquad \frac{\Gamma \vdash t : P_{\mathcal{D}}\,\tau}{\Gamma \vdash \textbf{do}_{\mathcal{D}}\{t\} : P_{\mathcal{D}}\,\tau} \qquad \frac{\Gamma \vdash t : P_{\mathcal{D}}\,\tau_1 \quad \Gamma, x : \tau_1 \vdash \textbf{do}_{\mathcal{D}}\{m\} : P_{\mathcal{D}}\,\tau}{\Gamma \vdash \textbf{do}_{\mathcal{D}}\{x \leftarrow t; m\} : P_{\mathcal{D}}\,\tau}$$

$$\frac{\Gamma \vdash t : \mathbb{B} \quad \Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash \textbf{if } t \textbf{ then } t_1 \textbf{ else } t_2 : \tau}$$

$\textbf{let}\ x = t; m$ is sugar for $x \leftarrow \textbf{return}\ t; m$ and $t; m$ for $\_ \leftarrow t; m$

Fig. 11. Syntax of the discrete probabilistic language, as an extension to Figs. 3 and 7. Gray highlights indicate syntax only present in the target language of the AD macro.

- For $exp^{\widetilde{\mathbb{R}}}$, $\times^{\widetilde{\mathbb{R}}}$ and $+^{\widetilde{\mathbb{R}}}$, implementing n-ary operations op on estimators, we must check that for all n-tuples differentiable functions $(r_1, \ldots, r_n) : \mathbb{R} \to \mathbb{R}$, if $\widetilde{dr}_i$ estimates $(r_i(\theta), r_i'(\theta))$, then $op(\widetilde{dr}_1, \ldots, \widetilde{dr}_n)$ estimates $(op(r_1(\theta), \ldots, r_n(\theta)), \frac{d}{d\theta} op(r_1(\theta), \ldots, r_n(\theta)))$.
- For the primitive **minibatch**, we must check that if $df : \mathbb{N} \to \mathbb{R} \times \mathbb{R}$ maps each natural number i to the dual number $(f_i(\theta), f_i'(\theta))$ for some differentiable function f_i, then $\textbf{minibatch}_{\mathcal{D}}\ M\ m\ df$ estimates the dual number $(\sum_{i=1}^{M} f_i(\theta), \sum_{i=1}^{M} f_i'(\theta))$.

Once we check all these primitives (given in Figure 24), the proof is done. □

Finally, we can conclude correctness of the ADEV algorithm on the extended language:

Theorem 3.2 (correctness of ADEV on the combinator DSL). *For all closed terms* $\vdash \widetilde{t} : \mathbb{R} \to \widetilde{\mathbb{R}}$, $[\![\lambda\theta : \mathbb{R}.\textbf{snd}_*(\mathcal{D}\{\widetilde{t}\}\ (\theta, 1))]\!]$ *is an unbiased derivative of* $[\![\widetilde{t}]\!]$.

4 DIFFERENTIATING EXPECTED VALUES OF DISCRETE PROBABILISTIC PROGRAMS

In this section, we develop one of the most important ideas in the paper: how to differentiate *expressive* probabilistic programs, with sequencing and branching, compositionally. For now we make the simplifying assumption that primitive probability distributions have finite support (e.g., a coin flip, which can take only two possible values). But this is only to simplify the proofs; when we add continuous distributions in Section 5, the ADEV algorithm itself won't change, only the theory.

4.1 Syntax and Semantics of the Discrete Probabilistic Programming Language

Figure 11 gives the syntax of this section's language. It is an extension of the language from Section 3 with two new features: *sequencing* of probabilistic computations, and *branching*. This greatly increases the expressiveness of the language; e.g., even without the advances of Sections 5 and 6, we can already express and differentiate the motivating example program in Figure 2.

Types: We introduce a type $\mathbb{B}$ of Booleans, and for each type τ, a new monadic type $P\,\tau$, of (finitely supported) probability distributions over $[\![\tau]\!]$. In our semantics, we need to fix a way of representing these distributions, and we choose $[\![P\,\tau]\!]$ to be the set of probability mass functions

Semantics of types:

$$\llbracket P\,\tau \rrbracket = \{\mu : \llbracket \tau \rrbracket \to [0,\infty) \mid \mu \text{ a probability distribution on } \llbracket \tau \rrbracket \text{ with finite support } \}$$
$$\llbracket P_{\mathcal{D}}\,\tau \rrbracket = (\llbracket \mathcal{D}\{\tau\} \rrbracket \to \widetilde{\mathbb{R}}_{\mathcal{D}}) \to \widetilde{\mathbb{R}}_{\mathcal{D}} \qquad \llbracket \mathbb{B} \rrbracket = \{\mathbf{True}, \mathbf{False}\}$$

Semantics of terms:

$$\llbracket \mathbf{return} \rrbracket(x)(y) = [x == y] \qquad \llbracket \mathbf{return}_{\mathcal{D}} \rrbracket(dr) = \lambda dl.dl(dr) \qquad \llbracket \mathbb{E} \rrbracket(\mu) = \mu$$
$$\llbracket \mathbf{flip}_{\texttt{ENUM}} \rrbracket(\theta)(b) = b\,?\,\theta\ :\ (1-\theta) \qquad \llbracket \mathbb{E}_{\mathcal{D}} \rrbracket(f) = f(\llbracket \mathbf{exact}_{\mathcal{D}} \rrbracket) \qquad \llbracket \mathbf{flip}_{\texttt{REINFORCE}} \rrbracket = \llbracket \mathbf{flip}_{\texttt{ENUM}} \rrbracket$$
$$\llbracket \mathbf{if}\ t\ \mathbf{then}\ t_1 \mathbf{else}\ t_2 \rrbracket(\rho) = \llbracket t \rrbracket(\rho)\,?\,\llbracket t_1 \rrbracket(\rho)\ :\ \llbracket t_2 \rrbracket(\rho)$$
$$\llbracket \mathbf{do}\ \{x \leftarrow t; m\} \rrbracket(\rho)(y) = \textstyle\sum_{z \in \mathrm{supp}(\llbracket t \rrbracket(\rho))} (\llbracket t \rrbracket(\rho)(z) \times \llbracket m \rrbracket(\rho, z)(y))$$
$$\llbracket \mathbf{do}_{\mathcal{D}}\ \{x \leftarrow t; m\} \rrbracket(\rho)(dl) = \llbracket t \rrbracket(\rho)(\lambda v. \llbracket m \rrbracket(\rho[x := v])(dl))$$

Fig. 12. Semantics of the discrete probabilistic language

$\llbracket \tau \rrbracket \to [0,\infty)$ with finitely many non-zero values, which form a monad over **Set**. For $\mu \in \llbracket P\,\tau \rrbracket$, we write $\mathrm{supp}(\mu) \subseteq \llbracket \tau \rrbracket$ for the finite subset of inputs at which it is non-zero.[3]

Terms: Since we now have Booleans, we can introduce an **if** statement. We will not introduce discontinuous comparators like $\leq$ until Section 6, so for now the **if** statement is primarily useful for branching on the outcomes of random coin flips. The new term $\texttt{flip} : \mathbb{I} \to P\,\mathbb{B}$ (which comes in two flavors, $\mathbf{flip}_{\texttt{ENUM}}$ and $\mathbf{flip}_{\texttt{REINFORCE}}$, for reasons we defer to Section 4.3) is the key primitive probability distribution. It is parameterized by a number $\theta \in (0,1)$, and returns **True** with probability θ and **False** with probability $1-\theta$. More complex probability distributions can be constructed using the Haskell-inspired $\mathbf{do}\ \{x \leftarrow t; m\}$ syntax: it builds a new probabilistic program that first samples x from $\llbracket t \rrbracket$, then runs $\llbracket m \rrbracket$ in an environment extended with the sampled x. This allows us to, for example, sequence two coin flips, where the second flip's probability depends on the outcome of the first: $\mathbf{do}\ \{b_1 \leftarrow \mathbf{flip}_{\texttt{REINFORCE}}\ 0.5;\ b_2 \leftarrow \mathbf{flip}_{\texttt{REINFORCE}}(\mathbf{if}\ b_1\ \mathbf{then}\ 0.2\ \mathbf{else}\ 0.4);\ \mathbf{return}\ (b_1 \wedge b_2)\} : P\,\mathbb{B}$.

The Expectation Operator. We now have two types denoting probability distributions over reals:

- $\widetilde{\mathbb{R}}$, the type of *estimators*—arbitrary probability distributions over $\mathbb{R}$, that can be composed using the combinator DSL from Section 3.
- $P\,\mathbb{R}$, the type of *monadic probabilistic programs returning reals*, which may be composed arbitrarily with downstream probabilistic computation.

The *expectation operator* $\mathbb{E} : P\,\mathbb{R} \to \widetilde{\mathbb{R}}$ casts a probabilistic program returning random real numbers into an *unbiased estimator* of the original program's *expectation*. ADEV can then be applied to the resulting esitmator to construct an estimator of its expectation's derivative. Returning to the example from Fig. 2, the term $\mathcal{L}$ has type $\mathbb{R} \to \widetilde{\mathbb{R}}$, and is thus a suitable 'main function' for ADEV to differentiate—but it is *constructed* by applying $\mathbb{E}$ to a term of type $P\,\mathbb{R}$. Note that because our language has primitives that transform and combine $\widetilde{\mathbb{R}}$ terms, the user's main function need not be a simple expectation of a probabilistic program—it can also be the $exp^{\widetilde{\mathbb{R}}}$ of an expectation, for example, or the $+^{\widetilde{\mathbb{R}}}$ of two expectations.

4.2 Differentiating the Probabilistic Language: Three False Starts

We now face the challenge of extending our ADEV macro $\mathcal{D}\{\cdot\}$ to handle this much more expressive probabilistic language. The first step is to define the macro's action on each new *type* in our language. The Booleans are simple enough ($\mathcal{D}\{\mathbb{B}\} = \mathbb{B}$, since they do not track derivative

[3]We emphasize that the choice to represent probability distributions as mass functions in our semantics does *not* mean that the *operational* meaning of a $P\,\tau$ term is a mass function evaluator: we think of $P\,\tau$ terms as probabilistic programs, which are tractable to *run* (i.e., to draw samples from), but for which it may be extremely expensive to evaluate probabilities.

information), but the monadic types $P\,\tau$ pose a real hurdle. If a source language term has type $P\,\tau$, what type should its translation have? In this section, we first explore three superficially appealing but ultimately problematic answers, before introducing our solution in Section 4.3.

False Start 1: Probabilistic Dual-Number Programs. In Section 3, we saw how for simple probabilistic computations over $\mathbb{R}$, it sufficed to translate them to simple probabilistic computations over $\mathbb{R} \times \mathbb{R}$. Naively, we might wonder whether this approach works at all types: can we define $\mathcal{D}\{P\,\tau\} = P\,(\mathcal{D}\{\tau\})$? Unfortunately, this simple, structure-preserving choice doesn't work. Values of type $\mathcal{D}\{\tau\}$ must track both a primal value of type τ, and the way that value depends continuously on an external parameter. At $\mathbb{R}$, for instance, this is done explicitly using a pair of reals. But now consider a program of type $P\,\mathbb{B}$, for example $\texttt{flip}\,\theta$. Even though $\mathbb{B}$ is discrete, probability distributions over Booleans *may* depend continuously on parameters, and so $\mathcal{D}\{P\,\mathbb{B}\}$ values must somehow track both the primal value (a distribution over $\mathbb{B}$) and a dual value (how that distribution changes when θ changes). But if we choose $\mathcal{D}\{P\,\tau\} := P\,\mathcal{D}\{\tau\}$, then we get that $\mathcal{D}\{P\,\mathbb{B}\} = P\,\mathbb{B}$, which can only track the primal value. This loss of information is one of the key reasons why Standard AD can fail when naively applied to probabilistic programs, as depicted in Figure 2.

False Start 2: Differentiating the Mass Function Semantics. Our semantics interprets a term of type $P\,\tau$ as a mass function, mapping values of $[\![\tau]\!]$ to non-negative real probability values. Viewed in this light, a primitive like `flip` is actually a real-valued function, in this case from $\mathbb{I} \to \mathbb{B} \to [0, \infty)$. We already know how to make AD work compositionally with functions of this type; would it work to set $\mathcal{D}\{P\,\tau\} := \mathcal{D}\{\tau \to \mathbb{R}\} = \mathcal{D}\{\tau\} \to \mathbb{R} \times \mathbb{R}$? The idea would be that applying AD to a probabilistic program $\vdash p : \mathbb{R} \to P\,\mathbb{R}$ would give us the derivative of its mass function, $\lambda(\theta, x).\frac{d}{d\theta}[\![p]\!](\theta)(x)$. To get derivatives of an expectation, $\frac{d}{d\theta}\sum_{x \in \text{supp}([\![p]\!](\theta))}[\![p]\!](x, \theta) \cdot x$, we would then differentiate term-by-term, using the automatically computed derivative. Unfortunately, it is not clear how to handle the fact that $[\![p]\!]$'s *support* can depend on θ, and relatedly, that the mass functions of probabilistic programs are not always differentiable. Consider, for example, the program $t = \lambda\theta : \mathbb{R}.\textbf{do}\,\{b \leftarrow \texttt{flip}\,0.3; \textbf{if}\ b\ \textbf{then return}\ \theta\ \textbf{else return}\ (2\theta)\}$, whose support $\{\theta, 2\theta\}$ depends on θ and whose mass function, $[\![t]\!](\theta)(r) = 0.3[\theta = r] + 0.7[2\theta = r]$, is not differentiable with respect to θ.

False Start 3: Differentiating the Expectation Directly. Ultimately, we only need to differentiate terms of type $P\,\tau$ because we care about how they affect the *expectation* of the program they are used within. This suggests that when we translate a term p of type $P\,\tau$, we might wish to produce a term that tells us not how $[\![p]\!]$ itself depends on a parameter θ, but how *expectations* with respect to the distribution $[\![p]\!]$ depend on the parameter θ. That is, can we differentiate the expectation $\mathbb{E}_{x \sim [\![p]\!]}[f(x)] = \sum_{x \in \text{supp}([\![p]\!])}[\![p]\!](x) \cdot f(x)$, for a formal expectand $f : \tau \to \mathbb{R}$?

One way to make good on this intuition is to set $\mathcal{D}\{P\,\tau\} := \mathcal{D}\{(\tau \to \mathbb{R}) \to \mathbb{R}\}$. Here, we understand a probability distribution μ to be a *higher-order function*, taking in an expectand $f : \tau \to \mathbb{R}$, and outputting the expectation $\sum_{x \in \text{supp}(\mu)} \mu(x) \cdot f(x)$. If we know how to differentiate μ *as an expectation operator*, then we will know how to differentiate expectations with respect to μ.

What would this look like in practice? For the primitive `flip`, we would need to implement a built-in derivative $\texttt{flip}_{\mathcal{D}}$, of type $\mathbb{I} \times \mathbb{R} \to (\mathbb{B} \to \mathbb{R} \times \mathbb{R}) \to \mathbb{R} \times \mathbb{R}$. Intuitively, it takes in a dual number $(\theta, \delta\theta) : \mathbb{I} \times \mathbb{R}$ representing the probability of heads, and dual-number *expectand* $df : \mathbb{B} \to \mathbb{R} \times \mathbb{R}$, and returns a dual number with the value and derivative of

$$\mathbb{E}_{x \sim \texttt{flip}(\theta)}[\pi_1(df(x))] = \theta \cdot \pi_1(df(\textbf{True})) + (1 - \theta) \cdot \pi_1(df(\textbf{False})).$$

$$
\begin{array}{l|l}
\mathcal{D}\{\mathbb{B}\} = \mathbb{B} & \mathcal{D}\{\textbf{if } t \textbf{ then } t_1 \textbf{ else } t_2\} = \textbf{if } \mathcal{D}\{t\} \textbf{ then } \mathcal{D}\{t_1\} \textbf{ else } \mathcal{D}\{t_2\} \\
\mathcal{D}\{P\,\tau\} = P_{\mathcal{D}}\,\mathcal{D}\{\tau\} & \mathcal{D}\{\textbf{return } t\} = \textbf{return}_{\mathcal{D}}\,\mathcal{D}\{t\} \\
\quad = (\mathcal{D}\{\tau\} \to \widetilde{\mathbb{R}}_{\mathcal{D}}) \to \widetilde{\mathbb{R}}_{\mathcal{D}} & \mathcal{D}\{\textbf{do}\{m\}\} = \textbf{do}_{\mathcal{D}}\,\{\mathcal{D}\{m\}\} \\
 & \mathcal{D}\{x \leftarrow t; m\} = x \leftarrow \mathcal{D}\{t\}; \mathcal{D}\{m\}
\end{array}
$$

+ new primitives for the built-in derivatives of $\textbf{flip}_{\texttt{ENUM}}$, $\textbf{flip}_{\texttt{REINFORCE}}$, and $\mathbb{E}$.

Fig. 13. Extended ADEV macro for the Discrete Probabilistic Programming Language (§4)

$\textbf{flip}_{\texttt{ENUM}\,\mathcal{D}}(dp : \mathbb{I} \times \mathbb{R}, \widetilde{dl} : \mathbb{B} \to \widetilde{\mathbb{R}}_{\mathcal{D}})$:
  $dl_1 \sim \widetilde{dl}\ \textbf{True}$
  $dl_2 \sim \widetilde{dl}\ \textbf{False}$
  $dr1 \leftarrow (dp \times_{\mathcal{D}} dl_1)$
  $dr2 \leftarrow ((1,0) -_{\mathcal{D}} dp) \times_{\mathcal{D}} dl_2)$
  return $dr1 +_{\mathcal{D}} dr2$
end

$\mathbb{E}_{\mathcal{D}}(\widetilde{dl} : P_{\mathcal{D}}\,(\mathbb{R} \times \mathbb{R}))$:
  $\widetilde{dr} \sim \widetilde{dl}(\llbracket \textbf{exact}_{\mathcal{D}} \rrbracket)$
  return $\widetilde{dr}$
end

$\textbf{flip}_{\texttt{REINFORCE}\,\mathcal{D}}(dp : \mathbb{I} \times \mathbb{R}, \widetilde{dl} : \mathbb{B} \to \widetilde{\mathbb{R}}_{\mathcal{D}})$:
  $b \sim \text{Bernouilli}(\textbf{fst}\ dp)$
  $(l_1, l_2) \sim \widetilde{dl}\ b$
  $dlp \leftarrow \textbf{if } b \textbf{ then } \log_{\mathcal{D}}\ dp \textbf{ else } \log_{\mathcal{D}}\ ((1,0) -_{\mathcal{D}} dp)$
  $\delta logpdf \leftarrow \textbf{snd}\ dlp$
  return $(l_1, l_2 + l_1 \times \delta logpdf)$
end

Fig. 14. Built-in derivatives for our new probabilistic primitives.

This looks reasonable, and is not hard to implement in practice, using the dual number operators $\times_{\mathcal{D}}$ and $+_{\mathcal{D}}$. Indeed, this choice turns out to be quite nice. Expectation operators of probability distributions form a submonad of the continuation monad [Vákár et al. 2019], so we would be translating one term of monadic type $(t : P\,\tau)$ to a new term of monadic type $(\mathcal{D}\{t\} : (\mathcal{D}\{\tau\} \to \mathcal{D}\{\mathbb{R}\}) \to \mathcal{D}\{\mathbb{R}\}$, whose type is equivalent to $\textbf{Cont}_{\mathcal{D}\{\mathbb{R}\}}\,(\mathcal{D}\{\tau\}))$. Furthermore, if we translate $\textbf{do}\,\{x \leftarrow t; m\}$ into $\textbf{do}_{\textbf{Cont}}\,\{x \leftarrow \mathcal{D}\{t\}; \mathcal{D}\{m\}\}$ and $\textbf{return}\ t$ to $\textbf{return}_{\textbf{Cont}}\,\mathcal{D}\{t\}$, we obtain correct (exact) derivatives of compound probabilistic programs' expectations. This is nice in that $\mathcal{D}\{\cdot\}$ still preserves even a monadic program's structure, with "all the action" happening at the primitives.

But there is one fatal flaw with this otherwise appealing approach: it computes *exact* derivatives of expectations, by summing over all possible random paths through a program, and in practice this will generally be completely intractable.

4.3 ADEV for the Probabilistic Language, Correctly

We present our approach to extending the ADEV macro in Figures 13-14. Our strategy reaps all the benefits of False Start 3 from the previous section, but avoids the fatal flaw: everywhere that False Start 3 must compute exact expectations of type $\mathbb{R}$, we permit estimated expectations of type $\widetilde{\mathbb{R}}$. For example, instead of requiring each primitive to compute intractable exact derivatives of expectations of arbitrary expectands, we allow primitives p to expose procedures for *estimating* the derivatives of expectations $\mathbb{E}_{x \sim p}[f(x)]$, given as input a procedure $\widetilde{df} : \llbracket \mathcal{D}\{\tau\} \rrbracket \to \widetilde{\mathbb{R}}_{\mathcal{D}}$ for *estimating* the value and derivative of the expectand f. We describe the intuition behind the translation:

- **Understanding the macro at the type level:** Our macro translates terms of probabilistic program type $P\ \tau$ into terms of *dual-number expectation estimator* type

$$P_{\mathcal{D}}\ \mathcal{D}\{\tau\} := \textbf{Cont}_{\widetilde{\mathbb{R}}_{\mathcal{D}}}\ \mathcal{D}\{\tau\} = (\mathcal{D}\{\tau\} \to \widetilde{\mathbb{R}}_{\mathcal{D}}) \to \widetilde{\mathbb{R}}_{\mathcal{D}}.$$

  Given a probabilistic program $\vdash p : P\,\tau$, the translation produces an *algorithm* $\vdash \mathcal{D}\{p\} : (\mathcal{D}\{\tau\} \to \widetilde{\mathbb{R}}_{\mathcal{D}}) \to \widetilde{\mathbb{R}}_{\mathcal{D}}$ for estimating the value and derivative of a $\llbracket p \rrbracket$*-expectation.* The generated procedure $\mathcal{D}\{p\}$ takes as input a function $\widetilde{df} : \llbracket \mathcal{D}\{\tau\} \rrbracket \to \widetilde{\mathbb{R}}_{\mathcal{D}}$, which, on input $(x, \delta x)$, estimates some true (dual-number) expectand $(f(x), \delta f(x, \delta x)) : \mathbb{R} \times \mathbb{R}$. The goal of the procedure $\llbracket \mathcal{D}\{p\} \rrbracket$ is then to estimate $\mathbb{E}_{x \sim \llbracket p \rrbracket}[f(x)]$ and its tangent value.

- **Understanding the macro on return:** One of the simplest probabilistic programs is $\mathbf{return}\, x : P\,\tau$, which implements the Dirac delta distribution that returns x with probability 1. The expectation of a function f with respect to this distribution is just $f(x)$. Indeed, our macro translates this term to $\mathbf{return}_{\mathcal{D}}\, dx : P_{\mathcal{D}}\, \mathcal{D}\{\tau\}$, which, unfolding the definitions, is equivalent to $\lambda \widetilde{df} : \mathcal{D}\{\tau\} \to \widetilde{\mathbb{R}}_{\mathcal{D}}.\widetilde{df}(dx) : (\mathcal{D}\{\tau\} \to \widetilde{\mathbb{R}}_{\mathcal{D}}) \to \widetilde{\mathbb{R}}_{\mathcal{D}}$. Intuitively, if we know how to estimate the dual number $df(dx)$ for any dx, we can also estimate its expectation under the Dirac delta—just plug in dx.
- **Understanding the macro on `flip`:** For the primitive distribution `flip`, we must attach a built-in derivative $\texttt{flip}_{\mathcal{D}}$, capable of estimating expectations with respect to the Bernoulli distribution, as well as their derivatives. It turns out there are multiple sensible choices, which strike different trade-offs between computational cost and variance. To afford the user maximum flexibility in navigating these trade-offs, we expose two *versions* of `flip`, $\mathbf{flip}_{\texttt{ENUM}}$ and $\mathbf{flip}_{\texttt{REINFORCE}}$, which have the same semantics, but different built-in derivatives. Our implementations of these built-in derivatives are given in Figure 14. The estimator $\mathbf{flip}_{\texttt{ENUM}\,\mathcal{D}}$ is the costlier but lower-variance option: to estimate an expected loss, it estimates the expectand on both possible sample values, **True** and **False**, and computes a (dual-number) weighted average. By contrast, $\mathbf{flip}_{\texttt{REINFORCE}\,\mathcal{D}}$ *samples* a value b, and only estimates the expectand for that sample value. It then uses the REINFORCE or score-function estimator to estimate the derivative of the expectation. In both cases, we emphasize how the logic of a particular derivative estimation strategy is encapsulated inside a procedure attached to the $\mathbf{flip}_{\texttt{ENUM}}$ or $\mathbf{flip}_{\texttt{REINFORCE}}$ primitive. This modular design supports future extensions with new gradient estimation strategies, or with new primitive distributions.
- **Understanding the macro on do:** To translate a term that *sequences* probabilistic computations, $\mathbf{do}\,\{x \leftarrow t; m\}$, the macro outputs $\mathbf{do}_{\mathcal{D}}\,\{dx \leftarrow \mathcal{D}\{t\}; \mathcal{D}\{m\}\}$. This is sugar for the continuation monad; desugaring, if $t : P\,\sigma$ and $x : \sigma \vdash \mathbf{do}\{m\} : \tau$, we get that the produced term is equivalent to $\lambda \widetilde{df} : (\mathcal{D}\{\tau\} \to \widetilde{\mathbb{R}}_{\mathcal{D}}).\, \mathcal{D}\{t\}(\lambda dx : \mathcal{D}\{\sigma\}.\mathbf{do}_{\mathcal{D}}\{\mathcal{D}\{m\}\}(\widetilde{df}))$. How should we understand this term? In order to estimate an expectation with respect to the *sequence* of computations, we apply the law of iterated expectation ($\mathbb{E}_{(x,y)\sim p}[f(y)] = \mathbb{E}_{x\sim p}[\mathbb{E}_{y\sim p(\cdot|x)}[f(y)]]$): we estimate an *expected* [expectation with respect to $[\![\mathbf{do}\{m\}]\!]$] with respect to $[\![t]\!]$. The inner expectation is estimated using the translation of m, and the outer one is estimated using the translation of t.
- **Understanding the macro on $\mathbb{E}$:** Once the user has constructed a term $t : P\,\mathbb{R}$, they can construct a term $\mathbb{E}\, t : \widetilde{\mathbb{R}}$, of estimator type. We think of $\mathbb{E}\, t$ as an estimator of $[\![t]\!]$'s expectation, i.e., the expectation of the identity function under the distribution $[\![t]\!]$. When our macro is applied to $\mathbb{E}\, t$, we get the term $\mathbb{E}_{\mathcal{D}}\mathcal{D}\{t\} : \widetilde{\mathbb{R}}_{\mathcal{D}}$, which, as can be seen from Fig. 14, is equivalent to $\mathcal{D}\{t\}(\lambda dr : \mathbb{R} \times \mathbb{R}.\mathbf{exact}_{\mathcal{D}} dr) : \widetilde{\mathbb{R}}_{\mathcal{D}}$. The idea is that $\mathcal{D}\{t\}$ is a procedure for estimating expectations (and their derivatives) of *any* dual-number function with respect to $[\![t]\!]$; we want the expectation of the identity, so we pass in $\lambda dr.\mathbf{exact}_{\mathcal{D}}\, dr$, (a zero-variance estimator of) the dual-number *id* function.

4.4 Correctness of ADEV on the Probabilistic Language

The overall ADEV workflow, and the statement of the overall correctness theorem for ADEV, will not change from Section 3: the user still ultimately constructs a program of type $\mathbb{R} \to \widetilde{\mathbb{R}}$, and it is still our job to differentiate the expectation of that program. The novelty in this section is that now, the user has more tools for *constructing* the final $\mathbb{R} \to \widetilde{\mathbb{R}}$ function, most notably the ability to construct probabilistic programs and pass them to $\mathbb{E}$. To establish correctness, we need to define new dual-number logical relations, defining appropriate notions of correct derivative for each new type. Then, we will need to reprove the fundamental lemma, by adding cases to our inductive proof for every new term constructor we added in this section.

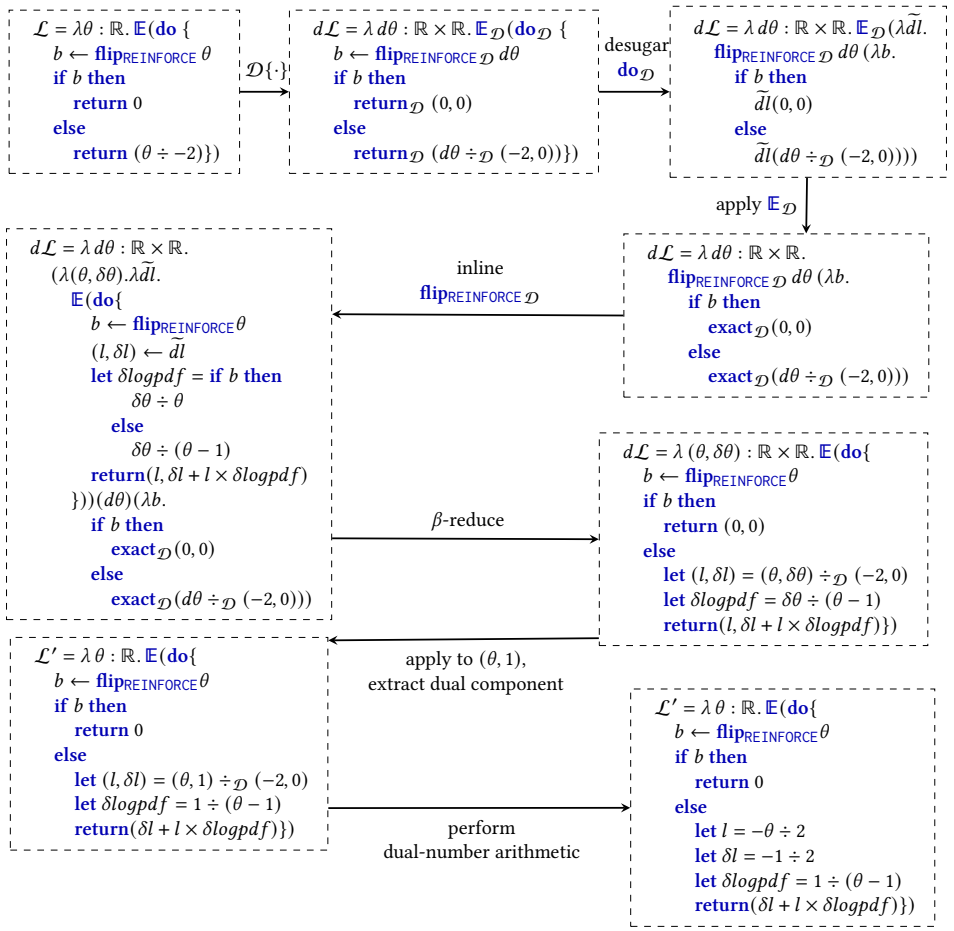

Fig. 15. How ADEV, applied to the example program from Fig. 2, derives the term on the bottom right. The ADEV macro $\mathcal{D}\{\cdot\}$ is itself very simple, changing only constants and primitives, just as in forward-mode AD. After applying it, we partially evaluate the resulting term for clarity, but these are *not* new transformations. (NB: We overload $\mathbb{E} : P\,\mathbb{R} \to \widetilde{\mathbb{R}}$ to also work on inputs of $P\,(\mathbb{R} \times \mathbb{R})$ type, yielding output of type $\widetilde{\mathbb{R}}_{\mathcal{D}}$.)

Defining the New Logical Relations. The new dual number logical relations for $\mathbb{B}$ and $P\,\tau$ are given in Fig. 16. The relation for $\mathbb{B}$ is essentially the same one we had for $\mathbb{N}$, another discrete type, reflecting that the only differentiable functions from $\mathbb{R}$ into the Booleans are the constant functions.

The relation at the probabilistic program type $P\,\tau$ is more interesting. Its goal is to relate a parameterized probabilistic program $f : \mathbb{R} \to [\![P\,\tau]\!]$ to the algorithm $\widetilde{g} : \mathbb{R} \to ([\![\mathcal{D}\{\tau\}]\!] \to \widetilde{\mathbb{R}}_{\mathcal{D}}) \to \widetilde{\mathbb{R}}_{\mathcal{D}}$ for estimating derivatives of expectations with respect to it. As an intermediate step for understanding the correctness relationship that must hold between f and $\widetilde{g}$, let's first consider a simpler algorithm than $\widetilde{g}$, that simply estimates expectations of f, not their derivatives. Such an algorithm $\widetilde{a}$ would have a similar type to $\widetilde{g}$, but would have no need for dual numbers: the program $\widetilde{a} : \mathbb{R} \to ([\![\tau]\!] \to \widetilde{\mathbb{R}}) \to \widetilde{\mathbb{R}}$ would take as input a parameter θ and an estimated expectand $\widetilde{l} : [\![\tau]\!] \to \widetilde{\mathbb{R}}$, and return an estimator of $\mathbb{E}_{x \sim f(\theta)}[l(x)]$, where $l(x) = \mathbb{E}_{y \sim \widetilde{l}(x)}[y]$. One way of implementing such an $\widetilde{a}$ would be to have it sample $x \sim f(\theta)$, then sample $y \sim \widetilde{l}(x)$, then return y. That is, $\widetilde{a}$ is just the *monadic bind*, in the underlying semantic space of probability measures, of

$$\mathscr{R}_{\mathbb{B}} = \{(f : \mathbb{R} \to \mathbb{B}, g : \mathbb{R} \to \mathbb{B}) \mid f \text{ is constant } \wedge f = g\}$$
$$\mathscr{R}_{P\,\tau} = \{(f : \mathbb{R} \to P\,\tau, \widetilde{g} : \mathbb{R} \to (\mathcal{D}\{\tau\} \to \widetilde{\mathbb{R}}_{\mathcal{D}}) \to \widetilde{\mathbb{R}}_{\mathcal{D}}) \mid$$
$$(\lambda\theta.\lambda\widetilde{l} : \tau \to \widetilde{\mathbb{R}}.\ \widetilde{\oint} \widetilde{l}(x) f(\theta)(dx), \widetilde{g}) \in \mathscr{R}_{(\tau \to \widetilde{\mathbb{R}}) \to \widetilde{\mathbb{R}}}\}$$

Fig. 16. Definition of the dual-number logical relation for our Discrete Probabilistic language

$f(\theta)$ (a probability distribution over $[\![\tau]\!]$) with the continuation $\widetilde{l}$ (a probability kernel from $[\![\tau]\!]$ to $\mathbb{R}$). Mathematically, we can write

$$\widetilde{a} = \lambda\theta.\lambda\widetilde{l}.\ \widetilde{\oint} \widetilde{l}(x) f(\theta, \mathrm{d}x),$$

where we have borrowed *Kock integral* notation $\widetilde{\oint} \nu(y)\mu(x, dy)$ for binding a kernel $\mu : X \to P\,Y$ to a continuation kernel $\nu : Y \to P\,Z$ from synthetic measure theory [Kock 2011; Ścibior et al. 2018].

Now, what we want from the algorithm $\widetilde{g}$, which estimates *dual-number derivatives* of expectations, is that it be a correct dual-number derivative of *the expectation estimation algorithm* $\widetilde{a}$. This is exactly what our logical relation $\mathscr{R}_{P\,\tau}$ says (Figure 16, inlining the definition of $\widetilde{a}$ we gave above).

Correctness of Primitives. Using this definition, we can work out a precise statement of the specification that a custom built-in derivative for a primitive $\mathbb{R} \to P\,\tau$ (such as $\mathbf{flip}_{\mathsf{ENUM}}$ and $\mathbf{flip}_{\mathsf{REINFORCE}}$) must meet. It arises as a special case of Definition 2.4, for the type $\mathbb{R} \to P\,\tau$:

Definition 4.1 (correct dual-number expectation estimator). *Let $p \in \mathbb{R} \to [\![P\,\tau]\!]$ be a probability kernel from $\mathbb{R}$ to $[\![\tau]\!]$. Then $p_D : [\![\mathcal{D}\{\mathbb{R}\}]\!] \to [\![P_{\mathcal{D}}\ \mathcal{D}\{\tau\}]\!]$ is a correct dual-number expectation estimator for p if for all $(\widetilde{f} : \mathbb{R} \to [\![\tau]\!] \to \widetilde{\mathbb{R}}, \widetilde{g} : \mathbb{R} \to [\![\mathcal{D}\{\tau\}]\!] \to \widetilde{\mathbb{R}}_{\mathcal{D}}) \in \mathscr{R}_{\tau \to \widetilde{\mathbb{R}}}$, and all differentiable functions $h : \mathbb{R} \to \mathbb{R}$, $p_D(h(\theta), h'(\theta))(\widetilde{g}(\theta))$ estimates the dual number $(\mathbb{E}_{x \sim p(h(\theta))}[\mathbb{E}_{y \sim \widetilde{f}(\theta)(x)}[y]], \frac{d}{d\theta}\mathbb{E}_{x \sim p(h(\theta))}[\mathbb{E}_{y \sim \widetilde{f}(\theta)(x)}[y]])$.*

The idea is that a built-in derivative for a probabilistic primitive (e.g. $\mathbf{flip}_{\mathsf{REINFORCE}\,\mathcal{D}}$ for the primitive $\mathbf{flip}_{\mathsf{REINFORCE}}$) receives two inputs: (1) a dual-number parameter, $(h(\theta), h'(\theta))$, that is already tracking its own derivative with respect to some underlying parameter θ, and (2) the expectand-estimator $\widetilde{g}_\theta : [\![\mathcal{D}\{\tau\}]\!] \to \widetilde{\mathbb{R}}_{\mathcal{D}}$, which is in general a *closure* that may have captured the underlying parameter θ. When returning an estimated expectation and derivative of the expectation, $\mathbf{flip}_{\mathsf{REINFORCE}\,\mathcal{D}}$ must account for both the way that θ influences the sampling distribution of $x \sim \mathbf{flip}_{\mathsf{REINFORCE}}(\theta)$, and also how it influences the closure whose expectation is being estimated. It is instructive to go through the exercise of showing *why* (for example) $\mathbf{flip}_{\mathsf{REINFORCE}}$'s built-in derivative satisfies this specification: the argument combines the standard REINFORCE estimator with the use of dual numbers to propagate derivatives.

Lemma 4.1. *$[\![\mathbf{flip}_{\mathsf{REINFORCE}\,\mathcal{D}}]\!]$ is a correct dual-number expectation estimator for $[\![\mathbf{flip}_{\mathsf{REINFORCE}}]\!]$.*

Proof. Let $h : \mathbb{R} \to \mathbb{R}$ differentiable, and $(\widetilde{f} : \mathbb{R} \to \mathbb{B} \to \widetilde{\mathbb{R}}, \widetilde{g} : \mathbb{R} \to \mathbb{B} \to \widetilde{\mathbb{R}}_{\mathcal{D}}) \in \mathscr{R}_{\mathbb{B} \to \widetilde{\mathbb{R}}}$. Then by the definition of correct dual-number derivative, $[\![\mathbf{flip}_{\mathsf{REINFORCE}\,\mathcal{D}}]\!]$ should estimate

$$\frac{d}{d\theta}\mathbb{E}_{x \sim Bern(h(\theta))}[\mathbb{E}_{y \sim \widetilde{f}(\theta)(x)}[y]] \tag{1}$$

(standard REINFORCE estimator, based on log derivative trick)

$$= \mathbb{E}_{x \sim Bern(h(\theta))}\left[\left(\frac{d}{d\theta}\log Bern(x; h(\theta))\right)\mathbb{E}_{y \sim \widetilde{f}(\theta)(x)}[y] + \frac{d}{d\theta}\mathbb{E}_{y \sim \widetilde{f}(\theta)(x)}[y]\right] \tag{2}$$

(use the fact that $(\widetilde{f}, \widetilde{g}) \in \mathscr{R}_{\tau \to \widetilde{\mathbb{R}}}$ to rewrite both terms)

$$= \mathbb{E}_{x \sim Bern(h(\theta))} \left[\left(\frac{d}{d\theta} \log Bern(x; h(\theta)) \right) \mathbb{E}_{(y,\delta y) \sim \widetilde{g}(\theta)(x)} [y] + \mathbb{E}_{(y,\delta y) \sim \widetilde{g}(\theta)(x)} [\delta y] \right] \tag{3}$$

(push log density term inside expectation, then combine expectations)

$$= \mathbb{E}_{x \sim Bern(h(\theta))} \left[\mathbb{E}_{(y,\delta y) \sim \widetilde{g}(\theta)(x)} \left[y \cdot \left(\frac{d}{d\theta} \log Bern(x; h(\theta)) \right) + \delta y \right] \right] \tag{4}$$

(evaluating the derivative)

$$= \mathbb{E}_{x \sim Bern(h(\theta))} \left[\mathbb{E}_{(y,\delta y) \sim \widetilde{g}(\theta)(x)} \left[y \cdot \left(\frac{-1^{1-x} \cdot h'(\theta)}{Bern(x; h(\theta))} \right) + \delta y \right] \right]. \tag{5}$$

This final expression can be estimated using just the dual number $(h(\theta), h'(\theta))$, and the function $\widetilde{g}$, by generating $x \sim Bern(x; h(\theta))$, then $(y, \delta y) \sim \widetilde{g}(\theta)(x)$, and then returning the value from line (5) above. This is exactly what $[\![\mathbf{flip}_{\texttt{REINFORCE}\,\mathcal{D}}]\!]$ does, to compute the tangent value it returns. □

Proving every primitive correct in a similar manner, following Section 4.3's logic, we can derive:

Lemma 4.2 (Fundamental lemma (revised with $\mathbb{B}$ and $P\,\tau$)). *For every term $\Gamma \vdash t : \tau$, $[\![\mathcal{D}\{t\}]\!]$ is a correct dual-number derivative of $[\![t]\!]$, w.r.t. the relations $\mathscr{R}_\tau$ defined at each type (incl. $\mathbb{B}$ and $P\,\tau$).*

Having reproved the fundamental lemma, the proof of Theorem 3.2 goes through unchanged:

Theorem 4.3 (correctness of ADEV for the discrete probabilistic language). *For all closed terms $\vdash t : \mathbb{R} \to \widetilde{\mathbb{R}}$, $[\![\lambda\theta : \mathbb{R}.\mathbf{snd}_*(\mathcal{D}\{t\}\ (\theta, 1))]\!]$ is an unbiased derivative of $[\![t]\!]$.*

Corollary 4.4. *For all continuous numeric types $\kappa \in \{\mathbb{R}, \mathbb{R}_{>0}, \mathbb{I}\}$, and all closed terms $\vdash t : \kappa \to \widetilde{\mathbb{R}}$, $[\![\lambda\theta : \kappa.snd_* (\mathcal{D}\{t\}(\theta, 1))]\!]$ is an unbiased derivative of $[\![t]\!]$.*

Proof. We have the result for $\kappa = \mathbb{R}$ from Thm 4.3, so first consider $\kappa = \mathbb{I}$. Let $\vdash t : \mathbb{I} \to \widetilde{\mathbb{R}}$. By the fundamental lemma, $[\![\mathcal{D}\{t\}]\!]$ is a correct dual-number derivative of $[\![t]\!]$, so for any $(h, h_{\mathcal{D}}) \in \mathscr{R}_{\mathbb{I}}$, we have $([\![t]\!] \circ h, [\![\mathcal{D}\{t\}]\!] \circ h_{\mathcal{D}}) \in \mathscr{R}_{\widetilde{\mathbb{R}}}$. In particular, this means that for any $r \in \mathbb{R}$, $\mathbb{E}_{(x,\delta x) \sim [\![\mathcal{D}\{t\}]\!](h_{\mathcal{D}}(d))} [\delta x] = (\lambda r.\mathbb{E}_{x \sim [\![t]\!](h(r))} [x])'(r)$. Now let $\theta \in \mathbb{I}$ and consider $h := \lambda r. \frac{\theta}{\theta + (1-\theta)e^{-r/(\theta - \theta^2)}}$, $h_{\mathcal{D}} := \lambda r.(h(r), h'(r))$. Because $h_{\mathcal{D}}$ computes h's derivative, $(h, h_{\mathcal{D}}) \in \mathscr{R}_{\mathbb{I}}$. The important property of this function h is that $h(0) = \theta$ and $h'(0) = 1$. Plugging this h into the equation from above, and setting r to 0, we get that $\mathbb{E}_{(x,\delta x) \sim [\![\mathcal{D}\{t\}(\theta,1)]\!]} [\delta x] = (\lambda r.\mathbb{E}_{x \sim [\![t]\!](h(r))} [x])'(0)$. The left-hand side is the expected value of $[\![\lambda\theta : \mathbb{I}.\mathbf{snd}_* \mathcal{D}\{t\}(\theta, 1)]\!](\theta)$. The right-hand side can be rewritten, using the chain rule, to yield $h'(0) \cdot (\lambda z.\mathbb{E}_{x \sim [\![t]\!](z)} [x])'(h(0)) = 1 \cdot (\lambda z.\mathbb{E}_{x \sim [\![t]\!](z)} [x])'(\theta)$. The fact that the LHS and RHS are equal implies that $[\![\lambda\theta : \mathbb{I}.\mathbf{snd}_* \mathcal{D}\{t\}(\theta, 1)]\!]$ is an unbiased derivative of $[\![t]\!]$.For the type $\mathbb{R}_{>0}$, the argument is the same, expect that we define $h := \lambda r.\theta.e^{r/\theta}$, which also has the property that $h(0) = \theta$ and $h'(0) = 1$ but has codomain $\mathbb{R}_{>0}$ instead of $\mathbb{I}$. □

5 EXTENDING ADEV TO CONTINUOUS PROBABILISTIC PROGRAMS

We now lift the key restriction from Section 4: we add to our language new primitives for sampling from continuous distributions (Fig. 17). Perhaps surprisingly, nothing about the ADEV macro or the user's workflow changes with this extension. Adding continuous primitives is no different from adding discrete primitives: just as in Section 4, the key task is to design built-in derivative-of-expectation estimators of type $P_{\mathcal{D}}\,\tau$ for every $P\,\tau$ primitive we add. What *does* change is the correctness proof: as we will see in Section 5.2, the introduction of continuous probability adds several wrinkles to our semantics and logical relations.

$$\text{Primitives } c ::= \ldots \mid \mathbf{uniform} : P\,\mathbb{I} \mid \mathbf{normal}_{\texttt{REPARAM}} : \mathbb{R} \times \mathbb{R}_{>0} \to P\,\mathbb{R} \mid \mathbf{normal}_{\texttt{REINFORCE}} : \mathbb{R} \times \mathbb{R}_{>0} \to P\,\mathbb{R} \mid \mathbf{geometric}_{\texttt{REINFORCE}} : \mathbb{I} \to P\,\mathbb{N}$$

Fig. 17. Extended syntax for Continuous Probabilistic Programming

5.1 Syntax and Algorithm

Our extended language (Fig. 17) has four new primitives: $\mathbf{uniform} : P\,\mathbb{I}$ (which samples on the unit interval), $\mathbf{normal}_{\texttt{REPARAM}}$, $\mathbf{normal}_{\texttt{REINFORCE}} : \mathbb{R} \times \mathbb{R}_{>0} \to P\,\mathbb{R}$ (which sample a normal distribution with user-specified parameters), and $\mathbf{geometric}_{\texttt{REINFORCE}} : \mathbb{I} \to P\,\mathbb{N}$ (which samples a geometric).[4]

Following our development in Section 4, we equip each primitive with a built-in derivative estimation procedure, based on the REINFORCE and reparameterization-trick gradient estimators, well-studied in the machine learning literature [Kingma and Welling 2014]. (See Fig. 25 in Appendix C.) As in Section 4, the novelty here is not in the estimators themselves but in the modularity, with gradient estimators exposed to the user in the form of composable primitives. Beyond the translation of these new primitives, the ADEV macro requires no further extensions.

5.2 Correctness of ADEV in the Continuous Language

New Challenges. All we have done is add a few new primitives, but formally justifying the extension raises two significant technical difficulties:

- **Measurability issues.** In Section 4 we chose $[\![P\,\tau]\!]$ to be the monad of finitely supported mass functions. This choice was nice, because (1) the set of finitely supported mass functions on $[\![\tau]\!]$ is well-defined for *any* set $[\![\tau]\!]$, and (2) the expectation of *any* function $f : [\![\tau]\!] \to \mathbb{R}$ with respect to a finitely-supported distribution is well-defined (it is just a finite sum). We exploited property (2) in defining our logical relations $\mathscr{R}_{P\,\tau}$, which talk about expectations of arbitrary functions. In our newly extended language, we must revise our choice of $[\![P\,\tau]\!]$, setting it to (something like) the set of probability measures on $[\![\tau]\!]$. But this breaks both of the nice properties above: (1) there is no nice way to define the set of probability measures over $[\![\tau]\!]$ when τ is higher-order (e.g. $\tau = \mathbb{R} \to \mathbb{R}$) [Heunen et al. 2017], and (2) in general expectations can only be taken of measurable functions. The challenge, then, is to find a way of defining semantics for the extended language, and updating our logical relations, that doesn't break anything we've done so far.
- **Edge cases where primitive gradient estimators are incorrect.** The standard proofs that the REINFORCE and reparameterization trick estimators are correct come with regularity conditions on the function f whose expectation's derivative is being estimated. As such, our new primitives and their ADEV translations $(c, c_{\mathcal{D}})$ do not technically satisfy the correctness criterion implied by our logical relations (Definition 4.1), which quantifies over all possible expectands. An updated correctness theorem will need to somehow account for these regularity conditions.

Resolving the First Challenge: Quasi-Borel Semantics. A long line of research has recently culminated in a new setting for measure theory where function spaces are well-behaved: the quasi-Borel spaces [Heunen et al. 2017]. Like a measurable space, a *quasi-Borel space* X pairs an underlying set $|X|$ with additional structure for reasoning precisely about probability; see Ścibior et al. [2018] for an overview. Here we just summarize our application of the theory:

- For every quasi-Borel space X, there is a quasi-Borel space $P\,X$ of probability measures on X, and these form a strong commutative monad. Using it, we were able to reformulate our language's semantics in terms of quasi-Borel spaces: every type τ is interpreted by a space $[\![\tau]\!]$, and terms t are interpreted as *quasi-Borel morphisms* $[\![t]\!]$, which are just functions satisfying a generalized measurability property ensuring they work nicely with quasi-Borel measures. Our interpretations

[4]The geometric distribution is not continuous, but violates a different restriction from Section 4—finite support.

$$
\begin{aligned}
\mathscr{R}_{\widetilde{\mathbb{R}}} = \Big\{ & (f : \mathbb{R} \to \widetilde{\mathbb{R}}, g : \mathbb{R} \to (\widetilde{\mathbb{R}}_{\mathcal{D}} \times (S \to \mathbb{R} \times \mathbb{R}))) \mid h_i := \lambda\theta.\lambda s.\pi_i((\pi_2 \circ g)(\theta)(s)) \\
& \wedge \forall\theta. \int_{\mathbb{R}} h_1(\theta)(s)ds = \mathbb{E}_{x \sim f(\theta)}[x] = \mathbb{E}_{x \sim \pi_{1*}(\pi_1 \circ g)(\theta)}[x] \\
& \wedge \forall\theta. \int_{\mathbb{R}} h_2(\theta)(s)ds = \mathbb{E}_{x \sim \pi_{2*}(\pi_1 \circ g)(\theta)}[x] \\
& \wedge (\lambda\theta.\lambda s.h_1(\theta)(s), \lambda\theta.\lambda s.(h_1, h_2)(\theta)(s)) \in \mathscr{R}_{S \to \mathbb{R}} \Big\} \\
\mathscr{R}_S = \big\{ & (f : \mathbb{R} \to S, g : \mathbb{R} \to S) \mid f \text{ is constant } \wedge f = g \big\}
\end{aligned}
$$

Fig. 18. Revised logical relation for our type $\widetilde{\mathbb{R}}$.

are standard, matching those of Ścibior et al. [2018]. We note that, unlike in Section 4, we now interpret $\widetilde{\mathbb{R}}$ and $P\,\mathbb{R}$ the same way: as the quasi-Borel probability measures on $\mathbb{R}$.

- Now that we have changed our semantics, how should we think about the definitions, appearing throughout our paper, of logical relations $\mathscr{R}_\tau$? We can read the definitions exactly as they are written, but interpreting them as relations over sets of quasi-Borel morphisms $\mathbb{R} \to [\![\tau]\!]$ (or $\mathbb{R} \to [\![\mathcal{D}\{\tau\}]\!]$), rather than over sets of arbitrary functions. Any expectations and integrals appearing in our definitions should now be understood as expectations and integrals of quasi-Borel morphisms with respect to quasi-Borel measures.[5]
- Having re-interpreted our language and our definitions of logical relations, we should do a sanity check that our proofs from Section 4 still go through (using the new semantics, but not yet adding our new primitives for continuous sampling). It turns out they do:

  Lemma 5.1 (fundamental lemma for the old language, new semantics). *For every term $\Gamma \vdash t : \tau$ in the discrete probabilistic language of Section 4, $[\![\mathcal{D}\{t\}]\!]_{Qbs}$ is a correct dual-number derivative of $[\![t]\!]_{Qbs}$, with respect to the $\mathscr{R}_\tau$ obtained by interpreting our previous definitions as relations of quasi-Borel morphisms.*[6]

Resolving the Second Challenge: Surfacing Regularity Conditions with a Lightweight Static Analysis. Now that we have a clear semantics, we can move onto the second problem: our logical relations $\mathscr{R}_\tau$ are too strict. In particular, they require that a primitive distribution must be able to estimate the derivative of the expectation of *any* (smooth, quasi-Borel) expectand, when in practice, nearly every gradient estimator needs additional regularity conditions to ensure unbiasedness. We address this issue in three stages:

(1) First, we develop a *weaker definition* of *unbiased derivative* (Definition 3.2):

  Definition 5.1 (weak unbiased derivative). *Let $\widetilde{f} : \mathbb{R} \to \widetilde{\mathbb{R}}$, and suppose that the map $\mathcal{L} : \mathbb{R} \to \mathbb{R}$ sending θ to $\mathbb{E}_{x \sim \widetilde{f}(\theta)}[x]$ is well-defined. Then $\widetilde{g} : \mathbb{R} \to \widetilde{\mathbb{R}}$ is a weak unbiased derivative of $\widetilde{f}$ if there exists a measurable function $h : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, continuously differentiable in its first argument, such that (1) $\mathcal{L}(\theta) = \int_{\mathbb{R}} h(\theta, s)ds$, and (2) $g(\theta)$ estimates $\int_{\mathbb{R}} \frac{\partial}{\partial\theta} h(\theta, s)ds$.*

[5] Existing work on quasi-Borel spaces usually defines integration of $X \to [0, \infty]$ functions, and not of $X \to \mathbb{R}$ functions. But just as in standard measure theory, we can extend the definition for non-negative functions to one for arbitrary real functions: we split the integrand into a positive part and negative part, separately integrate each, and then subtract the results. Because each result can be either finite or infinite, their difference can be either finite, infinite, *or undefined* (if both the positive part and negative part are infinite). In this paper, when we say that an expectation exists or is well-defined, we mean that the result of the integral is *finite*.

[6] For the categorically-minded reader, we provide another presentation of this logical relations argument in Appendix D.

This definition captures "unbiased, up to interchange of an integral with a derivative": instead of requiring that g unbiasedly estimate $\mathcal{L}'(\theta)$, we require that there is some way to write $\mathcal{L}$ as an integral such that, if you could swap the derivative and the integral, g would estimate $\mathcal{L}'(\theta)$.

(2) Second, we develop a lightweight static analysis that, given a term $\vdash \widetilde{t} : \mathbb{R} \to \widetilde{\mathbb{R}}$, *finds* the measurable function $h : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ (from Definition 5.1) that justifies ADEV's output $\widetilde{s} : \mathbb{R} \to \widetilde{\mathbb{R}}$ as a *weak* unbiased derivative estimator. To do so, we create a modified version of $\mathcal{D}\{\cdot\}$, where $\mathcal{D}\{\widetilde{\mathbb{R}}\} = \widetilde{\mathbb{R}}_{\mathcal{D}} \times (S \to \mathbb{R} \times \mathbb{R})$. Here, S is a new type of *random seeds*: $[\![S]\!] = \mathbb{R}$, but $\mathcal{D}\{S\} = S$ (no dual numbers), and derivatives of S computations are not tracked. Intuitively, this new $\mathcal{D}\{\cdot\}$ translates a term of type $\widetilde{\mathbb{R}}$ to a *pair*, where the first component is the same dual number estimator (of type $\widetilde{\mathbb{R}}_{\mathcal{D}}$) that we produced in Sections 3 and 4, and the second is the *justification of the estimator as a weak unbiased derivative*. A crucial part of this translation is what should happen at the primitives: because $\mathcal{D}\{\cdot\}$'s behavior on $\widetilde{\mathbb{R}}$ has changed, to include an extra component, all our primitives involving $\widetilde{\mathbb{R}}$ need to be updated, to produce or handle this extra component (see Figure 26 in Appendix). A more formal understanding can be gained by examining the new logical relation we define for $\mathscr{R}_{\widetilde{\mathbb{R}}}$, given in Figure 18. (What it calls h_1 is the witness h from Definition 5.1, and what it calls h_2 is its derivative.) By proving the fundamental lemma using this new relation, we obtain a weak correctness result for the language:

Lemma 5.2 (Weak correctness of ADEV). *Let* $\vdash \widetilde{t} : \mathbb{R} \to \widetilde{\mathbb{R}}$. *Letting*

$$h_i = \lambda(\theta, x).\pi_i(\pi_2([\![\mathcal{D}\{\widetilde{t}\}]\!](\theta, 1))(x)) : \mathbb{R} \times \mathbb{R} \to \mathbb{R},$$

assume that $\forall \theta \in \mathbb{R}$, $\int_{\mathbb{R}} h_1(\theta, x)dx$ *and* $\int_{\mathbb{R}} h_2(\theta, x)dx$ *are well-defined. Then:*

- *For all* $(\theta, x) \in \mathbb{R} \times \mathbb{R}$, $h_2(\theta, x) = \frac{\partial}{\partial \theta} h_1(\theta, x)$
- *For all* $\theta \in \mathbb{R}$, $[\![\widetilde{t}]\!](\theta)$ *is an unbiased estimator of* $\int_{\mathbb{R}} h_1(\theta, x)dx$.
- *For all* $\theta \in \mathbb{R}$, $\mathbf{snd}_*(\pi_1([\![\mathcal{D}\{\widetilde{t}\}]\!](\theta, 1)))$ *is an unbiased estimator of* $\int_{\mathbb{R}} h_2(\theta, x)dx$.

Therefore, $\lambda\theta : \mathbb{R}.\mathbf{snd}_*(\pi_1([\![\mathcal{D}\{\widetilde{t}\}]\!](\theta, 1)))$ *is a weak unbiased derivative of* $[\![\widetilde{t}]\!]$.

(3) Finally, we state a sufficient condition for a weak unbiased derivative to be fully unbiased:

Definition 5.2 (Locally Dominated). *We say that a function* $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ *is locally dominated if, for every* $\theta \in \mathbb{R}$, *there is a neighborhood* $U(\theta) \subseteq \mathbb{R}$ *of* θ *and an integrable function* $m_{U(\theta)} : \mathbb{R} \to [0, +\infty)$ *such that* $\forall \theta' \in U(\theta), \forall x \in \mathbb{R}, |f(\theta', x)| \leq m_{U(\theta)}(x)$.

Combining it with our static analysis that finds h and h', we get our final correctness theorem:

Theorem 5.3 (Correctness of ADEV (continuous language)). *Let* $\vdash \widetilde{t} : \mathbb{R} \to \widetilde{\mathbb{R}}$ *be a closed term, and suppose that* $[\![\widetilde{t}]\!](\theta)$ *has a well-defined expectation for every* $\theta \in \mathbb{R}$. *If* $[\![\lambda(\theta, x) : \mathbb{R} \times \mathbb{R}.\mathbf{snd}(\mathbf{snd}(\mathcal{D}\{\widetilde{t}\}(\theta, 1))(x))]\!]$ *is locally dominated, then* $[\![\lambda\theta : \mathbb{R}.\mathbf{snd}_*(\mathbf{fst}(\mathcal{D}\{\widetilde{t}\}(\theta, 1)))]\!]$ *is a correct unbiased derivative of* $[\![\widetilde{t}]\!]$.

Note that the final conclusion is the same *full* correctness property we proved in earlier sections; there is now just a single local domination condition for the user to verify, before the guarantee kicks in. This local domination condition is only *one* of the preconditions for swapping derivatives and integrals to be valid; crucially, the other hypotheses of the Dominated Convergence Theorem are automatically discharged by our proofs. Furthermore, even if the user's program composes different primitives, each using different gradient estimation strategies and making different assumptions, the static analysis performed by our modified $\mathcal{D}\{\cdot\}$ macro automatically generates a *single* term $\lambda(\theta, x).\mathbf{snd}(\mathbf{snd}(\mathcal{D}\{\widetilde{t}\}(\theta, 1))(x))$ (the h_2 from Lemma 5.2) whose local domination should be checked. Because this is an explicit term in our language, we are optimistic that future, more sophisticated static analyses could be developed to automatically discharge this local domination condition in many cases.

$$\begin{aligned}\text{Types } \tau &::= \ldots \mid \kappa^* \quad \text{(for every smooth base type } \kappa\text{)}\\ \text{Primitives } c &::= \ldots \mid \textbf{uniform} : P\,\mathbb{I}^* \mid \textbf{normal}_{\texttt{REINFORCE}} : \mathbb{R}\times\mathbb{R}_{>0} \to P\,\mathbb{R}^* \mid \lfloor\cdot\rfloor_\kappa : \kappa^* \to \kappa\\ &\quad \mid \leq_\kappa : \kappa^*\times\kappa^* \to \mathbb{B} \mid =_\kappa : \kappa^*\times\kappa^* \to \mathbb{B}\end{aligned}$$

Fig. 19. Revised syntax for smooth-tracking types

$$\mathcal{D}\{\kappa^*\} = \kappa^* \qquad \mathscr{R}_{\kappa^*} = \{(f : \mathbb{R} \to [\![\kappa]\!], g : \mathbb{R} \to [\![\kappa]\!]) \mid f \text{ is constant } \wedge\, f = g\}$$

Fig. 20. Definition of the dual-number type and the dual-number logical relation for smooth-tracking types

Accepted by the Type-checker		**Rejected**
$\mathcal{L}_1 = \lambda\theta : \mathbb{R}.\,\mathbb{E}(\textbf{do}\,\{$ $x^* \leftarrow \textbf{normal}_{\texttt{REINFORCE}}\ \theta\ 1$ $y \leftarrow \textbf{normal}_{\texttt{REPARAM}}\ \lfloor x^*\rfloor\ 1$ $\textbf{if}\ x^* \leq 3\ \textbf{then}$ $\quad\textbf{return}\ 0$ $\textbf{else}$ $\quad\textbf{return}\ -(\theta \div 2)\})$	$\mathcal{L}_2 = \lambda\theta : \mathbb{R}.\,\mathbb{E}(\textbf{do}\,\{$ $x \leftarrow \textbf{normal}_{\texttt{REPARAM}}\ \theta\ 1$ $y^* \leftarrow \textbf{normal}_{\texttt{REINFORCE}}\ x\ 1$ $\textbf{if}\ y^* \leq 3\ \textbf{then}$ $\quad\textbf{return}\ 0$ $\textbf{else}$ $\quad\textbf{return}\ -(\theta \div 2)\})$	$\mathcal{L}_3 = \lambda\theta : \mathbb{R}.\,\mathbb{E}(\textbf{do}\,\{$ $x^* \leftarrow \textbf{normal}_{\texttt{REINFORCE}}\ \theta\ 1$ $y \leftarrow \textbf{normal}_{\texttt{REPARAM}}\ \lfloor x^*\rfloor\ 1$ $\textbf{if}\ y \leq 3\ \textbf{then}$ $\quad\textbf{return}\ 0$ $\textbf{else}$ $\quad\textbf{return}\ -(\theta \div 2)\})$

Fig. 21. Smoothness-tracking types allow us to enforce preconditions for ADEV's correctness. In these programs, variables of type $\mathbb{R}^*$ – those that can be used non-smoothly – are indicated with a star. Type checking will reject the unsound program on the right and accept the two programs on the left. The error comes from the fact that $y : \mathbb{R}$ cannot be cast to a variable of type $\mathbb{R}^*$, for use with $\leq$: y has to be used smoothly for $\textbf{normal}_{\texttt{REPARAM}}$'s built-in derivative to be correct.

6 STRONGER GUARANTEES WITH SMOOTHNESS-TRACKING TYPES

The correctness guarantee of Theorem 5.3 covers an expressive language with discrete and continuous sampling, higher-order functions, monadic probabilistic programming, and conditional branching. But ADEV sometimes produces correct derivatives in cases our theory does *not* yet cover, namely when the user's program uses discontinuous primitives like $\leq$. Consider, for example,

$$\lambda\theta : \mathbb{R}.\textbf{do}\,\{x \leftarrow \textbf{normal}_{\texttt{REINFORCE}}(\theta, 1); \textbf{if}\ x \leq 3\ \textbf{then return}\ 1\ \textbf{else return}\ 0\}, \tag{6}$$

which *uses* $\leq$ but has expectation $\mathbb{P}_{x\sim\mathcal{N}(\theta,1)}[x \leq 3]$, which is itself differentiable with respect to θ. If we equip $\leq$ with a built-in derivative that ignores the tangent part of any dual-number inputs, we can apply ADEV to this program, and in this case, we *do* get out a correct derivative. Why is this, and can we state a more general theorem about when ADEV is correct?

It turns out that primitives like $\leq$ can be safely added to our language, but only if their use is carefully restricted. This is because the proof that establishes Theorem 5.3 from Lemma 5.2 relies on the measure-theoretic formulation of the Leibniz integral rule, which requires us to ensure that $h_1(\theta, x)$ is differentiable with respect to θ for almost all x. Importantly, we do *not* need h_1 to be differentiable with respect to x. Intuitively, we can allow $\leq$ in cases where it introduces discontinuities with respect to the *random seed* x, but not to the input parameter θ.

Types for Smoothness Tracking. We can make these intuitions precise by carefully extending our language of study to allow restricted uses of discontinuous primitives, then re-proving our correctness theorem for the extended language. We do this by adding, for each *smooth type* κ ($\mathbb{R}$, $\mathbb{I}$, and $\mathbb{R}_{>0}$), a *non-smooth type* κ^*. The semantics of a smooth type and its corresponding non-smooth type are the same, but our macro $\mathcal{D}\{\cdot\}$ does not attach dual numbers to non-smooth values ($\mathcal{D}\{\kappa^*\} = \kappa^*$), and our logical relations $\mathscr{R}_{\kappa^*}$ (Fig. 20) treat them as if they were discrete.

From a user's perspective, values of non-smooth type are *allowed* to be used non-smoothly, whereas values of smooth type *must* be used smoothly. This is reflected in the type of the primitive $\leq : \kappa^* \times \kappa^* \to \mathbb{B}$. In addition to $\leq$, we introduce a coercion $\lfloor \cdot \rfloor : \kappa^* \to \kappa$ from non-smooth to smooth types, but not in the other direction: you are always allowed to promise (unnecessarily) to use a value smoothly, but not to go back on your promise. In fact, as can be seen from the definition of $\mathcal{R}_{\mathbb{R}^*}$, *any* $\mathbb{R} \to \mathbb{R}^*$ function expressible in our language must necessarily be constant—no information can 'leak' from the smooth world to the non-smooth world.

Smooth and non-smooth types can be mixed to create functions whose types perform fine-grained tracking of *which* arguments they are differentiable with respect to. For example, a term $\vdash t : \mathbb{R} \times \mathbb{R}^* \to \mathbb{R}$ is guaranteed to have a denotation differentiable with respect to its first argument, but not its second. The key feature unlocked by this fine-grained tracking is that we can now assign more permissive types to some of our primitives from Section 5: namely, **uniform** and $\mathbf{normal}_{\texttt{REINFORCE}}$ now generate samples of non-smooth type, indicating that their correctness proofs do *not* require sampled values to be used smoothly in the rest of the program. $\mathbf{normal}_{\texttt{REPARAM}}$, by contrast, still generates samples of type $\mathbb{R}$. With these new types, we can accept program (6) from above, while rejecting programs on which ADEV would fail (Figure 21).

With these typing rules, we can import Section 5's results with no major hurdles:

Theorem 6.1 (Correctness of ADEV (full)). *Let $\vdash \widetilde{t} : \mathbb{R} \to \widetilde{\mathbb{R}}$ be a closed term in the extended language of Section 6, such that $[\![\widetilde{t}]\!](\theta)$ has a well-defined expectation for every θ. If $[\![\lambda(\theta, x) : \mathbb{R} \times \mathbb{R}.\mathbf{snd}(\mathbf{snd}(\mathcal{D}\{\widetilde{t}\}(\theta, 1))(x))]\!]$ is locally dominated, then $[\![\lambda\theta : \mathbb{R}.\mathbf{snd}_*(\mathbf{fst}(\mathcal{D}\{\widetilde{t}\}(\theta, 1)))]\!]$ is a correct unbiased derivative of $[\![\widetilde{t}]\!]$.*

7 SUMMARY: FULL LANGUAGE AND ADEV MACRO

Our full language is summarized in Figure 22, and our full AD macro in Figure 23. By combining Thm. 6.1 with Cor. 4.4, we arrive at the following general correctness result:

Corollary 7.1. *Let $\vdash \widetilde{t} : \kappa \to \widetilde{\mathbb{R}}$ be a closed term in the full language, where $\kappa \in \{\mathbb{R}, \mathbb{R}_{\geq 0}, \mathbb{I}\}$. If $[\![\widetilde{t}]\!](\theta)$ has a well-defined expectation for every $\theta \in \kappa$, and $[\![\lambda(\theta, x) : \kappa \times \mathbb{R}.\mathbf{snd}(\mathbf{snd}(\mathcal{D}\{\widetilde{t}\}(\theta, 1))(x))]\!]$ is locally dominated, then $[\![\lambda\theta : \kappa.\mathbf{snd}_*(\mathbf{fst}(\mathcal{D}\{\widetilde{t}\}(\theta, 1)))]\!]$ is a correct unbiased derivative of $[\![\widetilde{t}]\!]$. When $\widetilde{t}$ samples only from finite discrete distributions, the domination condition is always satisfied.*

8 RELATED WORK

Gradient Estimation in Machine Learning. ADEV's primitives compositionally package many gradient estimation strategies developed in the machine learning community [Kingma and Welling 2014; Lee et al. 2018; Mohamed et al. 2020; Ranganath et al. 2014]. It also extends a growing literature on *stochastic computation graphs* (SCGs) [Foerster et al. 2018; Schulman 2016; Schulman et al. 2015; Weber et al. 2019], the goal of which is to help practitioners derive unbiased gradient estimators for expectations of probabilistic processes represented as graphs. Recently, van Krieken et al. [2021] presented Storchastic, a practical system for AD of stochastic computation graphs. Storchastic provides reverse-mode AD (often more efficient than the forward-mode AD in our paper); and is implemented for PyTorch [Paszke et al. 2019], a widely used, practical deep learning framework. Our work on ADEV is complementary. We precisely formalize the general problem of automatic differentiation of expected values of probabilistic processes, in a way that applies to broad classes of probabilistic programs (including higher-order) that cannot easily be represented as computation graphs. Furthermore, our logical relations allow us to precisely formulate general conditions that new primitives' gradient estimators must satisfy to be compositionally added to the language. See Appendix B for further discussion on the consequences of these differences, including: (1) how ADEV can exploit dependency structure that is more explicit in SCGs, (2) how Storchastic gradient

Smooth base types $\kappa ::= \mathbb{R} \mid \mathbb{R}_{>0} \mid \mathbb{I}$

Types $\tau ::= \mathbf{1} \mid \mathbb{N} \mid \kappa \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \widetilde{\mathbb{R}} \mid \mathbb{B} \mid P\,\tau \mid \kappa^*$ (for every smooth base type κ)
$\mid \widetilde{\mathbb{R}}_{\mathcal{D}} \mid P_{\mathcal{D}}\,\tau \mid \mathrm{S}$

Terms $t ::= () \mid r\ (\in \kappa) \mid c \mid x \mid (t_1, t_2) \mid \lambda x.t \mid \mathbf{let}\ x = t_1\ \mathtt{in}\ t_2 \mid \mathbf{fst}\ t \mid \mathbf{snd}\ t \mid t_1\ t_2$
$\mid \mathbf{True} \mid \mathbf{False} \mid \mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \mid \mathbf{do}\ \{\, m \,\} \mid \mathbf{return}\ t$
$\mid c_{\mathcal{D}} \mid d \mid \mathbf{do}_{\mathcal{D}}\ \{\, m \,\} \mid \mathbf{return}_{\mathcal{D}}\ t$

Do notation $m ::= t \mid x \leftarrow t;\ m$

Source primitives $c ::= + \mid - \mid \times \mid \div \mid \mathtt{exp} \mid \mathtt{log} \mid \mathtt{sin} \mid \mathtt{cos} \mid \mathtt{pow} \mid \mathbb{E} : P\,\mathbb{R} \to \widetilde{\mathbb{R}}$
$\mid \mathbf{minibatch} : \mathbb{N} \to \mathbb{N} \to (\mathbb{N} \to \mathbb{R}) \to \widetilde{\mathbb{R}} \mid +^{\widetilde{\mathbb{R}}}, \times^{\widetilde{\mathbb{R}}} : \widetilde{\mathbb{R}} \times \widetilde{\mathbb{R}} \to \widetilde{\mathbb{R}}$
$\mid exp^{\widetilde{\mathbb{R}}} : \widetilde{\mathbb{R}} \to \widetilde{\mathbb{R}} \mid \mathbf{exact} : \mathbb{R} \to \widetilde{\mathbb{R}} \mid \lfloor - \rfloor_\kappa : \kappa^* \to \kappa$ (for each κ)
$\mid \mathbf{flip}_{\mathtt{REINFORCE}}, \mathbf{flip}_{\mathtt{ENUM}} : \mathbb{I} \to P\,\mathbb{B} \mid \leq_\kappa : \kappa^* \times \kappa^* \to \mathbb{B} \mid =_\kappa : \kappa^* \times \kappa^* \to \mathbb{B}$
$\mid \mathbf{uniform} : P\,\mathbb{I}^* \mid \mathbf{normal}_{\mathtt{REPARAM}} : \mathbb{R} \times \mathbb{R}_{>0} \to P\,\mathbb{R}$
$\mid \mathbf{normal}_{\mathtt{REINFORCE}} : \mathbb{R} \times \mathbb{R}_{>0} \to P\,\mathbb{R}^* \mid \mathbf{geometric}_{\mathtt{REINFORCE}} : \mathbb{I} \to P\,\mathbb{N}$

Target primitives $d ::= \mid \mathbf{fst}_*, \mathbf{snd}_* : \widetilde{\mathbb{R}}_{\mathcal{D}} \to \widetilde{\mathbb{R}}$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathbf{return}\ t : P\,\tau} \qquad \frac{\Gamma \vdash t : P\,\tau}{\Gamma \vdash \mathbf{do}\{t\} : P\,\tau} \qquad \frac{\Gamma \vdash t : P\,\tau_1 \quad \Gamma, x : \tau_1 \vdash \mathbf{do}\{m\} : P\,\tau}{\Gamma \vdash \mathbf{do}\{x \leftarrow t; m\} : P\,\tau}$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathbf{return}_{\mathcal{D}}\ t : P_{\mathcal{D}}\,\tau} \qquad \frac{\Gamma \vdash t : P_{\mathcal{D}}\,\tau}{\Gamma \vdash \mathbf{do}_{\mathcal{D}}\{t\} : P_{\mathcal{D}}\,\tau}$$

$$\frac{\Gamma \vdash t : P_{\mathcal{D}}\,\tau_1 \quad \Gamma, x : \tau_1 \vdash \mathbf{do}_{\mathcal{D}}\{m\} : P_{\mathcal{D}}\,\tau}{\Gamma \vdash \mathbf{do}_{\mathcal{D}}\{x \leftarrow t; m\} : P_{\mathcal{D}}\,\tau} \qquad \frac{\Gamma \vdash t : \mathbb{B} \quad \Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash \mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 : \tau}$$

$\mathbf{let}\ x = t; m$ is sugar for $x \leftarrow \mathbf{return}\ t; m$ and $t; m$ for $t \leftarrow \_; m$

When clear from context, we omit the brackets $\lfloor \cdot \rfloor_\kappa$.

We also assume that each source-language primitive c has a corresponding built-in derivative $c_{\mathcal{D}}$ in the target language.

Fig. 22. Full grammar and selected typing rules of the language we study. Gray highlights indicate syntax only present in the target language of the AD macro.

estimation methods can be exposed compositionally in ADEV, (3) how ADEV's continuations let it work robustly with multi-sample gradient estimators, whereas Storchastic's broadcasting approach can cause it to fail e.g. in programs with Python `if` statements, (4) how ADEV can achieve polynomial time complexity for estimators that would be exponential-time in Storchastic, and (5) how higher-order ADEV primitives can encapsulate sophisticated gradient estimation strategies that *don't* decompose into sample-by-sample estimators (as Storchastic's design would require).

Concurrently with our work, Arya et al. [2022] developed an intriguing new approach to AD of probabilistic programs, which like ADEV, arises by extending forward-mode AD, but which unlike ADEV, is not based on composing existing, well-understood estimation strategies. It is unclear what source-language features are covered by their algorithm (the authors caution, e.g., that general `if` statements are unsupported), but the low variance their estimators appear to achieve may open the door to stable optimization of objectives that have been out of reach using existing estimators. We have worked with the authors to incorporate their approach into ADEV as an additional estimator; our preliminary implementation (available on Github) adds little new code, inherits ADEV's support for `if` statements and other control flow, and interoperates with all of ADEV's other estimators.

To our knowledge, among frameworks for deriving unbiased gradient estimators (based on SCGs or Arya et al. [2022]'s stochastic triples), ADEV is the only one that handles objectives defined as *functions of* one or more expected values (e.g., $\mathrm{exp}_{\widetilde{\mathbb{R}}}\ (\mathbb{E}\ p) +_{\widetilde{\mathbb{R}}} \mathrm{exp}_{\widetilde{\mathbb{R}}}\ (\mathbb{E}\ q)$).

$\mathcal{D}\{-\}$ on contexts

$$\mathcal{D}\{\bullet\} = \bullet \qquad \mathcal{D}\{\Gamma, x : \tau\} = \mathcal{D}\{\Gamma\}, x : \mathcal{D}\{\tau\}$$

$\mathcal{D}\{-\}$ on types

$$\begin{array}{llll} \mathcal{D}\{\widetilde{\mathbb{R}}\} & = \widetilde{\mathbb{R}}_{\mathcal{D}} \times (S \to \mathbb{R} \times \mathbb{R}) & \mathcal{D}\{\tau_1 \times \tau_2\} & = \mathcal{D}\{\tau_1\} \times \mathcal{D}\{\tau_2\} \\ \mathcal{D}\{P\,\tau\} & = P_{\mathcal{V}}\,\mathcal{D}\{\tau\} & \mathcal{D}\{\tau_1 \to \tau_2\} & = \mathcal{D}\{\tau_1\} \to \mathcal{D}\{\tau_2\} \\ \mathcal{D}\{\kappa\} & = \kappa \times \mathbb{R} & \mathcal{D}\{\mathbb{B}\} & = \mathbb{B} \\ \mathcal{D}\{\mathbb{N}\} & = \mathbb{N} & \mathcal{D}\{\kappa^*\} & = \kappa^* \end{array}$$

$\mathcal{D}\{-\}$ on expressions

$$\begin{array}{llll} \mathcal{D}\{x\} & = x & \mathcal{D}\{r : \kappa\} & = (r, 0) \\ \mathcal{D}\{\lambda x.t\} & = \lambda x.\mathcal{D}\{t\} & \mathcal{D}\{r : \kappa^*\} & = r \\ \mathcal{D}\{t_1 t_2\} & = \mathcal{D}\{t_1\}\mathcal{D}\{t_2\} & \mathcal{D}\{r : \mathbb{N}\} & = r \\ \mathcal{D}\{\textbf{let } x = t_1 \textbf{ in } t_2\} & = \textbf{let } x = \mathcal{D}\{t_1\} \textbf{ in } \mathcal{D}\{t_2\} & \mathcal{D}\{()\} & = () \\ \mathcal{D}\{(t_1, t_2)\} & = (\mathcal{D}\{t_1\}, \mathcal{D}\{t_2\}) & \mathcal{D}\{\textbf{return } t\} & = \textbf{return}_{\mathcal{V}}\ \mathcal{D}\{t\} \\ \mathcal{D}\{\textbf{fst } t\} & = \textbf{fst } \mathcal{D}\{t\} & \mathcal{D}\{\textbf{do}\{m\}\} & = \textbf{do}_{\mathcal{V}}\ \{\mathcal{D}\{m\}\} \\ \mathcal{D}\{\textbf{snd } t\} & = \textbf{snd } \mathcal{D}\{t\} & \mathcal{D}\{x \leftarrow t; m\} & = x \leftarrow \mathcal{D}\{t\}; \mathcal{D}\{m\} \end{array}$$

We assume built-in primitives $c_{\mathcal{D}}$ for the derivatives of source primitives c, including $\textbf{flip}_{\text{ENUM}}$, $\textbf{flip}_{\text{REINFORCE}}$, $\textbf{normal}_{\text{REINFORCE}}$, $\textbf{normal}_{\text{REPARAM}}$, $\textbf{geometric}_{\text{REINFORCE}}$, **uniform**, **minibatch**, **exact**, and $\mathbb{E}$. For those we have

$$\mathcal{D}\{c\} = c_{\mathcal{D}}$$

$P_{\mathcal{V}}\,\tau$, $\textbf{return}_{\mathcal{V}}$, $\textbf{do}_{\mathcal{V}}$ are syntactic sugar for the continuation monad given by $P_{\mathcal{V}}\,\tau := (\tau \to \mathcal{D}\{\widetilde{\mathbb{R}}\}) \to \mathcal{D}\{\widetilde{\mathbb{R}}\}$.

Fig. 23. Full AD translation $\mathcal{D}\{-\}$. We have the following invariant: if $\Gamma \vdash t : \tau$, then $\mathcal{D}\{\Gamma\} \vdash \mathcal{D}\{t\} : \mathcal{D}\{\tau\}$. On terms $\Gamma \vdash t : \widetilde{\mathbb{R}}$, the first projection of $\mathcal{D}\{t\}$ is the dual-number derivative, and the second is the witness program for the function whose weak domination has to be checked (see Section 5).

Correctness and Semantics for Probabilistic and Differentiable Programming. Partly enabled by new semantic foundations for probabilistic [Ehrhard et al. 2018; Heunen et al. 2017; Zhang and Amin 2022] and differentiable [Huot et al. 2020; Sherman et al. 2021; Vákár 2020] programming, researchers have recently established a variety of correctness results for both automatic differentiation [Abadi and Plotkin 2020; Krawiec et al. 2022; Lee et al. 2020a; Mazza and Pagani 2021] and probabilistic program transformations [Lee et al. 2020b; Lew et al. 2020; Ścibior et al. 2018] for increasingly expressive languages. We build most closely on *logical relations* approaches [Ahmed 2006; Appel et al. 2007; Katsumata 2013; Pientka et al. 2019] for proving properties of AD algorithms [Barthe et al. 2020; Brunel et al. 2020; Huot et al. 2020; Mazza and Pagani 2021], and on works that use quasi-Borel spaces as a model of synthetic measure theory [Kock 2011; Ścibior et al. 2018; Vákár et al. 2019]. Recent work has begun to formally investigate interactions of differentiability and probabilistic programming [Lee et al. 2020b; Lew et al. 2021; Mak et al. 2021; Sherman et al. 2021], but not yet the properties of *AD* in the general probabilistic programming setting.

AD of Languages with Integration. Researchers have recently proposed languages with support both for integration and AD, including Teg [Bangaru et al. 2021], a differentiable first-order expression language with compact-domain integrals and arithmetic, and λ_S [Sherman et al. 2021], a higher-order language with computable integration on $[0, 1]$ as a primitive. Using compact-domain integration, it is possible to express some probabilistic program expectations, but not all (e.g., λ_S cannot express probabilistic programs that use Gaussian distributions). Furthermore, unlike in Teg and λ_S, the output of ADEV is a new probabilistic program, that can be directly run to produce gradient estimates for optimization. A unique aspect of Teg is its support for *parametric discontinuities*, which can sometimes be mimicked in ADEV programs using discrete random choices like `flip`, but are in general prohibited by Section 6's type system.

AD in PPLs. Many practical probabilistic programming languages [Bingham et al. 2019; Cusumano-Towner et al. 2019; Narayanaswamy et al. 2017] support the automated estimation of gradients

of a *particular* expectation with respect to probabilistic programs q: the gradient of the ELBO, $\nabla_\theta \mathbb{E}_{x\sim q_\theta}[\log p_\theta(x) - \log q_\theta(x)]$. ADEV formalizes and proves correct a more general algorithm for arbitrary expected values, giving theory that could help to understand when these algorithms are correct (as studied in a first-order language for independent Gaussians by Lee et al. [2020b]), and how they can be modularly extended to support new gradient estimation strategies, or the estimation of other expectations. Many PPLs also rely on AD for reasons *other than* differentiating expectations. Typically, these languages differentiate *deterministic* programs that are derived from or related to probabilistic ones. For example, Wingate et al. [2011] differentiate log densities of probabilistic programs, as does the widely-used and highly-optimized Stan [Carpenter et al. 2017] probabilistic programming system, for use within Hamiltonian Monte Carlo. Venture [Mansinghka et al. 2014, 2018] and Gen [Cusumano-Towner et al. 2019] also differentiate log densities, for HMC, gradient-based MAP optimization, and Metropolis-Adjusted Langevin Ascent. Gen also computes derivatives of user-defined *involutions* to automatically compute Jacobian corrections in reversible-jump MCMC [Cusumano-Towner et al. 2020]. It would be interesting to investigate whether our semantic setting—where we can reason about smoothness via logical relations, and measurability via quasi-Borel semantics—could be used to establish the soundness of these PPL applications.

9 DISCUSSION

Multivariate Functions. To simplify the presentation, we have presented everything in terms of $\mathbb{R} \to \widetilde{\mathbb{R}}$ functions with scalar, not vector, inputs and outputs. But the same general strategies used to extend deterministic forward-mode to multivariate functions apply in our case:

Given a term $\vdash t : \mathbb{R}^n \to P\,\mathbb{R}^m$, and an input vector $x \in \mathbb{R}^n$, we can consider the terms $t_{ij} := \lambda\theta : \mathbb{R}.\mathbb{E}(\mathbf{do}\{y \leftarrow t(x_1, \ldots, x_{i-1}, \theta, x_{i+1}, \ldots, x_n); \mathbf{return}\ (\pi_j\ y)\}$. The translation $\mathcal{D}\{t_{ij}\}$ of such a term yields an unbiased estimator of the partial derivative $\frac{\partial y_j}{\partial x_i}$. One (costly) option for estimating the entire Jacobian matrix would be to separately estimate each partial derivative. To reduce the variance of this estimate, the same random seed can be used when generating each term's estimate, without compromising unbiasedness of the overall estimate. For a fixed i, the computation of t_{ij}'s derivative estimate proceeds identically for all j; it is only at the end that we extract the j^{th} component of a result vector. There is therefore no need to run the computation m times: we must only run the calculation once for each $i \in \{1, \ldots, n\}$, to generate an entire vector of m different $\frac{\partial y_j}{\partial x_i}$ values. This is a well-understood feature of forward-mode AD: it is especially efficient when there are many outputs but few inputs. As in ordinary forward-mode AD, then, we can compute a Jacobian via n runs of the translated program. Also as in standard forward-mode AD, it is possible to trade memory for time: if instead of dual numbers $\mathbb{R} \times \mathbb{R}$ we use dual vectors $\mathbb{R} \times \mathbb{R}^n$, we can run the n copies of the computation 'in parallel.'[7] However, for memory- and time-efficient gradients of functions with high-dimensional inputs, reverse-mode is usually preferred.

Limitations of Differentiability Analysis. Our type system enforces that the user's main program is smooth with respect to the input parameter θ. This limitation has several consequences:

(1) ADEV's type system rejects many programs do not *have* differentiable expectations. Estimating their derivatives would be an ill-defined task, so we consider this 'a feature, not a bug.'
(2) ADEV's type system also prevents users from expressing some programs that do have differentiable expectations, but for which efficient gradient estimators are not known or cannot be derived using standard strategies. We would love to differentiate such programs, but we suspect

[7] Built-in derivatives $c_{\mathcal{D}}$ must be updated to work on these n-ary dual numbers. For deterministic primitives, this is straightforward, but for probabilistic primitives that employ certain gradient estimation strategies, e.g. measure-valued differentiation, it is less clear whether there are efficient ways to simultaneously estimate derivatives in multiple directions.

that for expert users hoping to apply ADEV, this limitation would seem natural. (Several recent works [Bangaru et al. 2021; Lee et al. 2018] present gradient estimation strategies for restricted classes of discontinuities; these estimators are not yet widely used by practitioners, but we are interested in exploring how they might be incorporated into future versions of ADEV.)

(3) Finally, ADEV rejects some programs too eagerly. For example, if a parameter θ is used non-smoothly but only in a probability-zero set of random executions (i.e., for almost all executions, the function is differentiable for all θ), our type system will reject it, even though our existing gradient estimators would have been correct for the program. More subtly, certain Lipschitz-continuous but non-differentiable uses of a parameter θ may be permissible, if for any θ the non-differentiability itself is encountered with probability 0 (e.g., $ReLU(x-\theta)$ for x sampled from a Gaussian). A less conservative static analysis could help make ADEV applicable to such programs, which do arise in practice. But we expect this to be a tricky problem. For example, concurrently with our work, Lee et al. [2022] present a static analysis based on abstract interpretation for careful reasoning about smoothness, including local Lipschitz continuity. Their analysis accepts $ReLU(x-\theta)$, but it also seems to accept, for example, $ReLU(ReLU(x)-\theta)$, a term we would want to reject in ADEV (at $\theta = 0$, the program is not differentiable for a positive-measure set of x values). Finding more sophisticated static analyses that admit a larger set of programs while still ensuring soundness is an interesting direction for future work, which could broaden the range of applications that AD of probabilistic programs might have.

Haskell Prototype. Our Haskell prototype (Appx. A) illustrates how ADEV integrates with existing libraries for probabilistic and differentiable programming. On Github, we also implement 11 extensions from Appx. B, and we intend the library to be practical both as a platform for future research and as a tool for small- or medium-scale applications.[8] Interestingly, although our analysis does not cover general recursion, our prototype successfully differentiates many recursive programs. The only failure cases we know of are programs whose translations do not halt almost surely, e.g. $\text{geom} = \lambda\theta : \mathbb{1}.\mathbf{do}\{b \leftarrow \mathbf{flip}_{\text{ENUM}}\,\theta; \mathbf{if}\ b\ \mathbf{then}\ 0\ \mathbf{else\ do}\{n \leftarrow \text{geom}\,\theta; \mathbf{return}\ (n+1)\}\}$. The use of $\mathbf{flip}_{\text{ENUM}}$ causes ADEV's gradient estimator to attempt an enumeration of program paths, of which there are infinitely many. In this example, the problem could be avoided by using (e.g.) $\mathbf{flip}_{\text{REINFORCE}}$, but we leave to future work a full account of ADEV's correctness that covers recursive programs.

ACKNOWLEDGMENTS

We have benefited from discussing this work with many friends and colleagues, including Martin Rinard, Tan Zhi-Xuan, Wonyeol Lee, Faustyna Krawiec, Gaurav Arya, Ohad Kammar, Feras Saad, Cathy Wong, McCoy Becker, Cameron Freer, Michele Pagani, Jesse Michel, Ben Sherman, Kevin Mu, Jesse Sigal, Paolo Perrone, Sean Moss, Younesse Kaddar and the Oxford group. We are also grateful to anonymous referees for very helpful feedback. This material is based on work supported by the NSF Graduate Research Fellowship under Grant No. 1745302. Our work is also supported by a Royal Society University Research Fellowship, the ERC BLAST grant, the Air Force Office of Scientific Research (Award No. FA9550–21–1–0038), and the DARPA Machine Common Sense and SAIL-ON projects.

[8] Like our theoretical presentation, our implementation extends forward-mode AD, whose cost scales linearly with the number of input parameters. For models with low- to medium-dimensional parameter spaces, forward-mode can be more efficient than reverse-mode, but models containing large neural networks with many parameters, for example, cannot be efficiently differentiated with our current prototype. Improvements to and analyses of reverse-mode algorithms have often built directly on earlier work studying the simpler forward-mode case; our hope is that by showing how standard forward-mode AD algorithms and their proofs can be extended cleanly to handle probabilistic programs, ADEV may lay the groundwork for future research investigating more efficient reverse-mode AD algorithms for probabilistic programs.

REFERENCES

Martín Abadi and Gordon D. Plotkin. 2020. A simple differentiable programming language. *Proc. ACM Program. Lang.* 4, 38:1–38:28. https://doi.org/10.1145/3371106

Amal J. Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3924)*, Peter Sestoft (Ed.). Springer, 69–83. https://doi.org/10.1007/11693024_6

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 109–122. https://doi.org/10.1145/1190216.1190235

Gaurav Arya, Moritz Schauer, Frank Schäfer, and Chris Rackauckas. 2022. Automatic Differentiation of Programs with Discrete Randomness. *CoRR* abs/2210.08572 (2022). https://doi.org/10.48550/arXiv.2210.08572 arXiv:2210.08572

Sai Praveen Bangaru, Jesse Michel, Kevin Mu, Gilbert Bernstein, Tzu-Mao Li, and Jonathan Ragan-Kelley. 2021. Systematically differentiating parametric discontinuities. *ACM Trans. Graph.* 40, 4 (2021), 107:1–107:18. https://doi.org/10.1145/3450626.3459775

Gilles Barthe, Raphaëlle Crubillé, Ugo Dal Lago, and Francesco Gavazzo. 2020. On the Versatility of Open Logical Relations - Continuity, Automatic Differentiation, and a Containment Theorem. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 56–83. https://doi.org/10.1007/978-3-030-44914-8_3

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *J. Mach. Learn. Res.* 20 (2019), 28:1–28:6. http://jmlr.org/papers/v20/18-403.html

Aloïs Brunel, Damiano Mazza, and Michele Pagani. 2020. Backpropagation in the simply typed lambda-calculus with linear negation. *Proc. ACM Program. Lang.* 4, POPL (2020), 64:1–64:27. https://doi.org/10.1145/3371132

Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *Journal of Statistical Software* 76, 1 (2017), 1–32. https://doi.org/10.18637/jss.v076.i01

Marco Cusumano-Towner, Alexander K Lew, and Vikash K Mansinghka. 2020. Automating involutive MCMC using probabilistic and differentiable programming. *arXiv preprint arXiv:2007.09871* (2020).

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

SW Director and R Rohrer. 1969. Automated network design-the frequency-domain case. *IEEE Transactions on Circuit Theory* 16, 3 (1969), 330–337.

Thomas Ehrhard, Michele Pagani, and Christine Tasson. 2018. Measurable cones and stable, measurable functions: a model for probabilistic higher-order programming. *Proc. ACM Program. Lang.* 2, POPL (2018), 59:1–59:28. https://doi.org/10.1145/3158147

Mikhail Figurnov, Shakir Mohamed, and Andriy Mnih. 2018. Implicit Reparameterization Gradients. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 439–450. https://proceedings.neurips.cc/paper/2018/hash/92c8c96e4c37100777c7190b76d28233-Abstract.html

Jakob N. Foerster, Gregory Farquhar, Maruan Al-Shedivat, Tim Rocktäschel, Eric P. Xing, and Shimon Whiteson. 2018. DiCE: The Infinitely Differentiable Monte Carlo Estimator. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer G. Dy and Andreas Krause (Eds.). PMLR, 1524–1533. http://proceedings.mlr.press/v80/foerster18a.html

Andreas Griewank and Andrea Walther. 2008. *Evaluating derivatives - principles and techniques of algorithmic differentiation, Second Edition.* SIAM. https://doi.org/10.1137/1.9780898717761

Bernd Heidergott and Felisa J Vázquez-Abad. 2000. *Measure valued differentiation for stochastic processes: The finite horizon case.* Eurandom.

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, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 1–12. https://doi.org/10.1109/LICS.2017.8005137

Mathieu Huot, Sam Staton, and Matthijs Vákár. 2020. Correctness of Automatic Differentiation via Diffeologies and Categorical Gluing. In *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12077)*, Jean Goubault-Larrecq and Barbara König (Eds.). Springer, 319–338. https://doi.org/10.1007/978-3-030-45231-5_17

Shin-ya Katsumata. 2013. Relating computational effects by TT-lifting. *Information and Computation* 222 (2013), 228–246.

Diederik P. Kingma and Max Welling. 2014. Auto-Encoding Variational Bayes. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1312.6114

Nathan L. Kleinman, James C. Spall, and Daniel Q. Naiman. 1999. Simulation-Based Optimization with Stochastic Approximation Using Common Random Numbers. *Management Science* 45 (1999), 1570–1578.

Anders Kock. 2011. Commutative monads as a theory of distributions. *arXiv preprint arXiv:1108.5952* (2011).

Faustyna Krawiec, Simon Peyton Jones, Neel Krishnaswami, Tom Ellis, Richard A. Eisenberg, and Andrew W. Fitzgibbon. 2022. Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–30. https://doi.org/10.1145/3498710

Wonyeol Lee, Xavier Rival, and Hongseok Yang. 2022. Smoothness Analysis for Probabilistic Programs with Application to Optimised Variational Inference. *CoRR* abs/2208.10530 (2022). https://doi.org/10.48550/arXiv.2208.10530 arXiv:2208.10530

Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. 2020a. On Correctness of Automatic Differentiation for Non-Differentiable Functions. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/4aaa76178f8567e05c8e8295c96171d8-Abstract.html

Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. 2020b. 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

Wonyeol Lee, Hangyeol Yu, and Hongseok Yang. 2018. Reparameterization Gradient for Non-differentiable Models. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 5558–5568. https://proceedings.neurips.cc/paper/2018/hash/b096577e264d1ebd6b41041f392eec23-Abstract.html

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

Alexander K. Lew, Mathieu Huot, and Vikash K. Mansinghka. 2021. Towards Denotational Semantics of AD for Higher-Order, Recursive, Probabilistic Languages. *CoRR* abs/2111.15456 (2021). arXiv:2111.15456 https://arxiv.org/abs/2111.15456

Carol Mak, C.-H. Luke Ong, Hugo Paquet, and Dominik Wagner. 2021. Densities of Almost Surely Terminating Probabilistic Programs are Differentiable Almost Everywhere. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 432–461. https://doi.org/10.1007/978-3-030-72019-3_16

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

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

Damiano Mazza and Michele Pagani. 2021. Automatic differentiation in PCF. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–27. https://doi.org/10.1145/3434309

Andriy Mnih and Karol Gregor. 2014. Neural Variational Inference and Learning in Belief Networks. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014 (JMLR Workshop and Conference Proceedings, Vol. 32)*. JMLR.org, 1791–1799. http://proceedings.mlr.press/v32/mnih14.html

Shakir Mohamed, Mihaela Rosca, Michael Figurnov, and Andriy Mnih. 2020. Monte Carlo Gradient Estimation in Machine Learning. *J. Mach. Learn. Res.* 21 (2020), 132:1–132:62. http://jmlr.org/papers/v21/19-346.html

Christian A. Naesseth, Francisco J. R. Ruiz, Scott W. Linderman, and David M. Blei. 2017. Reparameterization Gradients through Acceptance-Rejection Sampling Algorithms. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA (Proceedings of Machine Learning Research, Vol. 54)*, Aarti Singh and Xiaojin (Jerry) Zhu (Eds.). PMLR, 489–498. http://proceedings.mlr.press/v54/naesseth17a.html

Siddharth Narayanaswamy, Brooks Paige, Jan-Willem van de Meent, Alban Desmaison, Noah D. Goodman, Pushmeet Kohli, Frank D. Wood, and Philip H. S. Torr. 2017. Learning Disentangled Representations with Semi-Supervised Deep Generative Models. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5925–5935. https://proceedings.neurips.cc/paper/2017/hash/9cb9ed4f35cf7c2f295cc2bc6f732a84-Abstract.html

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. (2019), 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

Brigitte Pientka, David Thibodeau, Andreas Abel, Francisco Ferreira, and Rébecca Zucchini. 2019. A Type Theory for Defining Logics and Proofs. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*. IEEE, 1–13. https://doi.org/10.1109/LICS.2019.8785683

Louis B. Rall. 1981. *Automatic Differentiation: Techniques and Applications*. Lecture Notes in Computer Science, Vol. 120. Springer. https://doi.org/10.1007/3-540-10861-0

Rajesh Ranganath, Sean Gerrish, and David M. Blei. 2014. Black Box Variational Inference. 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, Vol. 33)*. JMLR.org, 814–822. http://proceedings.mlr.press/v33/ranganath14.html

John Schulman. 2016. *Optimizing Expectations: From Deep Reinforcement Learning to Stochastic Computation Graphs*. Ph.D. Dissertation. University of California, Berkeley, USA. https://www.escholarship.org/uc/item/9z908523

John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. 2015. Gradient Estimation Using Stochastic Computation Graphs. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett (Eds.). 3528–3536. https://proceedings.neurips.cc/paper/2015/hash/de03beffeed9da5f3639a621bcab5dd4-Abstract.html

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

Adam Ścibior, Vaden Masrani, and Frank Wood. 2021. Differentiable Particle Filtering without Modifying the Forward Pass. *CoRR* abs/2106.10314 (2021). arXiv:2106.10314 https://arxiv.org/abs/2106.10314

Benjamin Sherman, Jesse Michel, and Michael Carbin. 2021. λ_S: computable semantics for differentiable programming with higher-order functions and datatypes. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–31. https://doi.org/10.1145/3434284

Matthijs Vákár. 2020. Denotational Correctness of Forward-Mode Automatic Differentiation for Iteration and Recursion. *arXiv preprint arXiv:2007.05282* (2020).

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

Emile van Krieken, Jakub M. Tomczak, and Annette ten Teije. 2021. Storchastic: A Framework for General Stochastic Automatic Differentiation. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (Eds.). 7574–7587. https://proceedings.neurips.cc/paper/2021/hash/3dfe2f633108d604df160cd1b01710db-Abstract.html

Théophane Weber, Nicolas Heess, Lars Buesing, and David Silver. 2019. Credit Assignment Techniques in Stochastic Computation Graphs. In *The 22nd International Conference on Artificial Intelligence and Statistics, AISTATS 2019, 16-18 April 2019, Naha, Okinawa, Japan (Proceedings of Machine Learning Research, Vol. 89)*, Kamalika Chaudhuri and Masashi Sugiyama (Eds.). PMLR, 2650–2660. http://proceedings.mlr.press/v89/weber19a.html

David Wingate, Noah D. Goodman, Andreas Stuhlmüller, and Jeffrey Mark Siskind. 2011. Nonstandard Interpretations of Probabilistic Programs for Efficient Inference. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain*, John Shawe-Taylor, Richard S. Zemel, Peter L. Bartlett, Fernando C. N. Pereira, and Kilian Q. Weinberger (Eds.). 1152–1160. https://proceedings.neurips.cc/paper/2011/hash/0d7de1aca9299fe63f3e0041f02638a3-Abstract.html

Yizhou Zhang and Nada Amin. 2022. Reasoning about "reasoning about reasoning": semantics and contextual equivalence for probabilistic programs with nested queries and recursion. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–28. https://doi.org/10.1145/3498677

Received 2022-07-07; accepted 2022-11-07