λ_S: Computable Semantics for Differentiable Programming with Higher-Order Functions and Datatypes

BENJAMIN SHERMAN, MIT, USA
JESSE MICHEL, MIT, USA
MICHAEL CARBIN, MIT, USA

Deep learning is moving towards increasingly sophisticated optimization objectives that employ higher-order functions, such as integration, continuous optimization, and root-finding. Since differentiable programming frameworks such as PyTorch and TensorFlow do not have first-class representations of these functions, developers must reason about the semantics of such objectives and manually translate them to differentiable code.

We present a differentiable programming language, λ_S, that is the first to deliver a semantics for higher-order functions, higher-order derivatives, and Lipschitz but nondifferentiable functions. Together, these features enable λ_S to expose differentiable, higher-order functions for integration, optimization, and root-finding as first-class functions with automatically computed derivatives. λ_S's semantics is computable, meaning that values can be computed to arbitrary precision, and we implement λ_S as an embedded language in Haskell.

We use λ_S to construct novel differentiable libraries for representing probability distributions, implicit surfaces, and generalized parametric surfaces – all as instances of higher-order datatypes – and present case studies that rely on computing the derivatives of these higher-order functions and datatypes. In addition to modeling existing differentiable algorithms, such as a differentiable ray tracer for implicit surfaces, without requiring any user-level differentiation code, we demonstrate new differentiable algorithms, such as the Hausdorff distance of generalized parametric surfaces.

CCS Concepts: • **Mathematics of computing** → *Arbitrary-precision arithmetic*; **Continuous functions**; *Point-set topology*; • **Theory of computation** → *Categorical semantics*.

Additional Key Words and Phrases: Constructive Analysis, Diffeological Spaces, Automatic Differentiation

ACM Reference Format:
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, Article 3 (January 2021), 31 pages. https://doi.org/10.1145/3434284

1 INTRODUCTION

Deep learning is centered on optimizing objectives $\ell : \Theta \to \mathbb{R}$ over some parameter space Θ by *gradient descent*, following the derivative of ℓ at some particular value $\theta \in \Theta$ to move in a direction that decreases $\ell(\theta)$. Before deep-learning practitioners adopted frameworks such as TensorFlow and PyTorch, creating a new model (i.e., parameter space and objective) was a laborious and error-prone endeavor, since it involved manually determining and computing the derivative of the objective. The advent of deep-learning frameworks that provide *automatic differentiation* (*AD*)—the automated computation of derivatives of a function given just the definition of the function itself—has made creating and modifying models much easier: a user simply writes the objective

Authors' addresses: Benjamin Sherman, MIT, USA, sherman@csail.mit.edu; Jesse Michel, MIT, USA, jmmichel@mit.edu; Michael Carbin, MIT, USA, mcarbin@csail.mit.edu.

2475-1421/2021/1-ART3
https://doi.org/10.1145/3434284

and its derivative is computed automatically. As a result, progress in deep learning has rapidly accelerated – a testament to the value of programming-language abstractions.

However, the creativity of deep-learning practitioners has exceeded the capabilities of current AD frameworks: practitioners have devised objectives that current AD frameworks cannot handle directly. A simple example is an objective including an expectation over a probability distribution whose parameters may vary, like this:

$$\ell(\theta) = \mathbb{E}_{x \sim \mathcal{N}(\mu(\theta), \sigma^2(\theta))}[f(x)].$$

If this ℓ is translated naïvely to PyTorch, by approximating the expectation with Monte Carlo sampling, the automatically generated derivative will be incorrect. Numerous algorithms have been proposed to compute the derivatives of objectives that average over parameterized probability distributions [Figurnov et al. 2018; Jang et al. 2017; Jankowiak and Obermeyer 2018; Naesseth et al. 2017]. How does one compute derivatives of objectives like these in general? No existing differentiable-programming semantics has tackled the problem of differentiating through expectations such as these.

Other objectives are sufficiently complex that they do not even beg an incorrect naïve implementation. Objectives ℓ that optimize over compact sets Δ

$$\ell(\theta) = \max_{\delta \in \Delta} f(\theta, \delta)$$

arise in adversarial contexts, including adversarial training and generative adversarial networks (GANs). Conceptually, optimizing this objective with gradient-based techniques requires a semantics for a differentiable max operation over a compact set, which, to date, has not been covered in the literature on the semantics of differentiable programs. Devising the appropriate derivative for these kinds of objectives is an object of current study [Lorraine et al. 2019; Wang et al. 2020].

Sometimes, an objective involves *root-finding*,

$$\ell(\theta) = \text{let } x \text{ be such that } g(\theta, x) = 0 \text{ in } f(\theta, x).$$

This arises in learning implicit surfaces, with applications both to learning the decision boundaries of classifiers as well as to reconstructing surfaces from point-cloud data or other visual data. How to compute the derivative of objectives like this is a key contribution of several papers [Atzmon et al. 2019; Bai et al. 2019; Niemeyer et al. 2020].

What do these objectives all have in common? They all involve *higher-order functions*: their definitions introduce variables that are subject to integration, optimization, or root-finding. Not only are these three operations troublesome in practice, but no semantics of differentiable programming has yet addressed them.

Approach. We present λ_S, a differentiable programming language that includes higher-order functions for integration, optimization, and root-finding. A key technical challenge is that these functions are higher-order and our semantic approach must wed higher-order functions with higher-order derivatives and nonsmooth functions to encompass these and other modern deep learning objectives. As a toy example, consider computing the derivative $f'(0.6)$ of the function

$$f(c) \triangleq \int_0^1 \text{ReLU}(x - c)\, dx,$$

where $\text{ReLU}(x) = \max(0, x)$. We can compute $f'(0.6) = -0.4$ with the λ_S expression

```
eps=1e-2> deriv (λ c ⇒ integral01 (λ x ⇒ relu (x - c))) 0.6
[-0.407, -0.398]
```

where there is a type $\mathfrak{R}$ for real numbers, a function `relu : ℜ → ℜ` for ReLU, a higher-order function `integral01 : (ℜ → ℜ) → ℜ` for integration over the unit interval [0,1], and a higher-order function for differentiation of real-valued functions `deriv : (ℜ → ℜ) → ℜ → ℜ`. The result can be queried to any precision, returning an interval guaranteed to include the true answer. Here, the precision is specified in the prompt as `eps=1e-2`.

λ_S is the first language that gives semantics to such an operation and moreover is the first to support its computation to arbitrary precision. Note that, in order to determine this derivative, we must evaluate the derivative of the ReLU function everywhere from -0.6 to 0.4, which includes 0, where ReLU is not (classically) differentiable.

Our work is unique compared to related work in supporting higher-order functions, higher-order derivatives, and nondifferentiable functions. We combine the use of Clarke derivatives in Di Gianantonio and Edalat [2013] to support nondifferentiable functions, the diffeological approach of Vákár et al. [2018] to support higher-order functions, and the derivative towers of Elliott [2008] to support higher-order derivatives. §9 covers related work in more detail. Merging these techniques gives us a platform to accomplish the contributions described below.

Contributions. We present λ_S, a differentiable programming language whose types are (generalized) smooth spaces and whose functions are (generalized) smooth maps. Our contributions are:

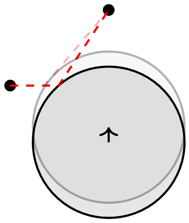
(1) The first semantics for a differentiable programming language that admits all of the following: 1) higher-order functions (§5), 2) higher-order derivatives (§4), and 3) Lipschitz but nonsmooth functions, such as `min`, `max`, and ReLU (§4).
(2) The first semantics for differentiable integration, optimization, and root-finding (§5), enabled by the features above.
(3) An implementation of this semantics, including implementations for higher-order functions such as integration (§6). Our implementation is based directly on a constructive categorical semantics that demonstrates how these constructs can be computed to arbitrary precision.
(4) New smooth libraries for constructing and computing on three higher-order datatypes: probability distributions, implicit surfaces, and generalized parametric surfaces (§7).

λ_S's semantics allows computation with and reasoning about the derivatives of higher-order functions, such as integration, optimization, and root-finding. λ_S elucidates foundational principles for how to program with smooth values in a sound, arbitrarily precise manner, including which operations are possible to compute soundly and which are not. While in many cases λ_S is not practically efficient, in some cases, programs can serve as executable specifications to guide programming in other frameworks, to validate separately developed systems, and to suggest new functionality that could be added to other differentiable programming frameworks.

2 AN INTRODUCTION TO λ_S

We demonstrate λ_S's core functionality by implementing a simple differentiable *ray tracer*, an algorithm that generates an image of a scene as viewed by a camera by tracing how rays of light emanate from a light source, bounce off the scene, and then enter the camera's aperture. *Differentiable ray tracing* is a new deep-learning technique that propagates derivatives through image rendering algorithms, permitting the use of inverse graphics to solve computer-vision tasks [Li et al. 2018; Niemeyer et al. 2020]. These techniques optimize the parameters of a scene representation to make the image generated by the ray tracer more closely match a target image.

As a simple example, consider computing the brightness of a particular scene at a particular direction, using the λ_S library for representing scenes and a function for performing ray tracing, both of which we present in Fig. 1:



(a) A ray of light from a source above bounces off a circle before hitting a camera. How does the brightness change when the circle is moved up?

```
type Surface A = A → ℜ

firstRoot : (ℜ → ℜ) → ℜ ! language primitive

let dot (x y : ℜ^2) : ℜ = x[0] * y[0] + x[1] * y[1]
let scale (c : ℜ) (x : ℜ^2) : ℜ^2 = (c * x[0], c * x[1])
let norm2 (x : ℜ^2) : ℜ = x[0]^2 + x[1]^2
let normalize (x : ℜ^2) : ℜ^2 = scale (1 / sqrt (norm2 x)) x

deriv : (ℜ → ℜ) → (ℜ → ℜ) ! library function
let gradient (f : ℜ^2 → ℜ) (x : ℜ^2) : ℜ^2 =
  (deriv (λ z : ℜ ⇒ f (z, x[1])) x[0],
   deriv (λ z : ℜ ⇒ f (x[0], z)) x[1])
```

(b) Basic definitions used in `raytrace` below.

```
! camera assumed to be at the origin
let raytrace (s : Surface (ℜ^2)) (lightPos : ℜ^2) (rayDirection : ℜ^2) : ℜ =
  let tStar = firstRoot (λ t : ℜ ⇒ s (scale t rayDirection)) in
  let y = scale tStar rayDirection in let normal : ℜ^2 = - gradient s y in
  let lightToSurf = y - lightPos in
  max 0 (dot (normalize normal) (normalize lightToSurf))
  / (norm2 y * norm2 lightToSurf)
```

(c) A λ_S function for differentiable ray tracing of implicit surfaces.

Fig. 1. A library for differentiable ray tracing and scene representation.

```
eps=1e-5> raytrace (circle (1, -3/4) 1) (1, 1) (1, 0)
[2.587289, 2.587299]
```

Fig. 1a depicts the computation at hand. The camera is located at the origin `(0, 0)`, the circle is centered at `(1, -3/4)` and has radius `1`, the light source is at `(1, 1)`, and we consider a ray pointing horizontally to the right from the camera, in the direction `(1, 0)`. The computation returns an interval and the `eps=1e-5` specifies the precision tolerance, such that the interval-valued result, `[2.587289, 2.587299]`, has a width at most 10^{-5}. Our implementation guarantees that whenever it returns a finite-width interval, the true, real-valued result is contained within that interval.

λ_S permits differentiation of any functions in the language, so we can compute how the brightness would change if the circle were moved up by an infinitesimal amount:

```
eps=1e-3> deriv (λ y : ℜ ⇒ raytrace (circle (0, y) 1) (1, 1) (1, 0)) (-3/4)
[1.3477, 1.3484]
```

The λ_S function `deriv : (ℜ → ℜ) → (ℜ → ℜ)` computes the derivative of a scalar-valued real function. The result indicates that when the circle is moved up infinitesimally from its current location, the brightness increases infinitesimally at a rate of ~1.35 units brightness per unit distance the circle is moved up.

Several changes occur when the circle is moved up that affect the image brightness. The point at which the light ray bounces off the circle moves closer to the camera, decreasing the distance from the camera to the circle (increasing brightness) but increasing the distance from the light to the camera (decreasing brightness). Both the direction of the surface normal of the circle at the point where the light deflects and the direction from the light source to that point change, increasing the angle between the surface normal of the circle and the light ray (decreasing brightness). Automatic differentiation automatically takes all of these effects into account.

Figure 1c shows the implementation of the differentiable ray tracing in λ_S. The function `firstRoot : (ℜ → ℜ) → ℜ` in the definition of `raytrace` computes the distance that the light travels from the scene to the camera. Given a function `f : ℜ → ℜ`, `firstRoot f` performs *root finding*, computing $\min\{x \in [0,1] \mid \mathtt{f}(x) = 0\}$. λ_S's higher-order functions for root finding are novel, and accordingly, λ_S's ability to express differentiable ray tracing of implicit surfaces (embodied in `raytrace`) without needing any custom code for specifying derivatives.

The differentiable ray tracer `raytrace` critically depends on λ_S's unique support for higher-order functions, higher-order derivatives, and Lipschitz but nondifferentiable functions such as min, max, and ReLU. We now provide a brief introduction to these three features.

2.1 Higher-Order Functions

The `raytrace` function must compute the distance the ray of light travels from the scene to the camera, represented by the let-definition `tStar` in `raytrace`. When applied to the scene `circle (1, y) 1`, the definition reduces to

```
let tStar y = firstRoot (λ t : ℜ ⇒ 1 - y^2 - (t - 1)^2)
```

The function `firstRoot : (ℜ → ℜ) → ℜ` is a higher-order function since it takes a function as input. In order to admit a function like this in a differentiable programming language, the language must be able to compute how the result of `firstRoot` changes when there is an infinitesimal perturbation to its input function. In this example, we want to know how `tStar` changes when `y` changes. To answer this, define `f t y = 1 - y^2 - (t - 1)^2`. Then `tStar` finds a solution for the variable `t` to the equation `f t y = 0`. So whatever change is induced by changing `y` must be counterbalanced by changing `tStar`. λ_S's semantics validate the equation (for values of `y` giving well-defined roots)

```
deriv tStar y = - deriv (λ y0 : ℜ ⇒ f (tStar y) y0) y /
                  deriv (λ t : ℜ ⇒ f t y) (tStar y)
```

This equation for the derivative of root finding is known as the *implicit function theorem*. By the rules of calculus, we can further simplify this to

```
deriv tStar y = - y / (tStar y - 1).
```

Note that the semantics of λ_S ensures that these equations are indeed program equivalences: one can substitute one expression for the other within the context of a larger expression without affecting its meaning. Indeed, taking `y = -3/4`, and evaluating both sides of the expression above in λ_S produces compatible answers, roughly -1.1, which indicates that moving the circle up decreases the distance that the light travels from the circle to the camera.

We implement the `firstRoot` function as a language primitive by specifying not only how `firstRoot` acts on values but also how derivatives propagate through it, via the implicit function theorem (see §5.2 for more detail).

2.2 Higher-Order Derivatives

The brightness of the image computed by the `raytrace` function depends on the angle at which the ray of light deflects as it bounces off the circle, so we need to know which direction the circle faces where the light hits it, which is known as the *surface normal.* In the code for `raytrace`, the surface normal is computed as

```
let normal : ℜ² = - gradient s y
```

Consider, for instance, the unit circle centered at $(0, 0)$, i.e., `circle (0, 0) 1`, given by the function $f(x, y) = 1 - x^2 - y^2$. The surface normal is given by the negative gradient,

$$-\nabla f(x,y) = -\left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right) = (2x, 2y)$$

So, for instance, the point $(1/\sqrt{2}, 1/\sqrt{2})$ on the upper-right of the circle has a surface normal that points up and to the right, in the direction $(2/\sqrt{2}, 2/\sqrt{2})$.

Note that, in the `raytrace` code itself, this gradient computation requires the computation of derivatives of the implicitly defined surface in order to compute the image brightness. Accordingly, computing the derivative of the image brightness with respect to an infinitesimal perturbation in the scene requires computing the second derivatives of the implicitly defined surface with respect to its arguments. Thus, higher-order differentiation is a valuable language feature.

In λ_S, differentiation is a first-class programming construct, so higher-order differentiation is naturally supported, as we can compute higher-order derivatives by applying the `deriv : (ℜ → ℜ) → ℜ → ℜ` function multiple times. Note that some approaches to differentiable programming do not support higher-order differentiation (see Table 1) and thus do not have differentiation as a first-class construct. Higher-order derivatives are also used for numerical integration, in optimization algorithms, and in other contexts.

The requirement to support higher-order derivatives means that language primitives, such as `firstRoot`, must specify not only how they act on values but also how derivatives *of all orders* propagate through them.

2.3 Nondifferentiability

Note that the `raytrace` code uses the built-in function `max : ℜ → ℜ → ℜ` in computing the image brightness. If the light source is behind the scene, the dot product of the surface normal and the vector from the light to the surface will be negative, but the brightness should be 0, rather than this negative value. Hence, we clamp the value to be at least zero by applying `max 0`. Note that this function is exactly the rectified linear unit (ReLU) that is common in deep learning:

```
let relu (x : ℜ) : ℜ = max 0 x
```

ReLU is not differentiable at 0. When we compute its derivative at 0 in λ_S, we get a *nonmaximal* result. That means that, for sufficiently fine (≤ 1) precision tolerances, we get nontermination:

```
eps=1e-1> deriv relu 0
(nontermination)
```

```
eps=2> deriv relu 0
[0.0, 1.0]
```

The interval approximations never converge to intervals smaller than $[0, 1]$. The type $\mathfrak{R}$ contains, in addition to the real numbers, nonmaximal elements such as this one, which we name $[0, 1]$, e.g., we find that $\mathrm{ReLU}'(0) = [0, 1]$.

Differentiable programming frameworks such as PyTorch admit min and max operations, but they are unsound, in the sense that one can define $f(x) = \max(x, 0) + \min(0, x)$, which is the identity function, but compute in PyTorch that $f'(0) = 2$, whereas it should be $f'(0) = 1$. Because of this issue, most differentiable programming semantics leave the derivative of max undefined at 0.

However, λ_S's interval-valued semantics for functions like max enables productive computational functionality that the partiality approach would not permit. For instance, suppose rather than having a point light source for ray-tracing, we instead have a line light source, so we integrate over the entire line, using the primitive higher-order function `integral01 : (ℜ → ℜ) → ℜ`, where for `f : ℜ → ℜ`, `integral01 f` computes the integral of `f` over the unit interval, $\int_0^1 \mathtt{f}(x)\,dx$. For simplicity, consider a camera located at $(0, 1)$ pointing downwards at a flat surface that stretches from $(-1, y)$ to $(1, y)$, with a light source stretching from $(1, 0)$ to $(1, 1)$. Furthermore, let us disregard the effect of brightness decreasing when the light travels longer distances, such that the brightness is

```
let brightness (y : ℜ) : ℜ =
  integral01 (λ y0 : ℜ ⇒ max 0 ((y0 - y) / sqrt (1 + (y0 - y)²)))
```

When $0 \le$ `y` ≤ 1, the integrand will be nondifferentiable with respect to `y` at the point where `y0 = y`. For instance, taking `y0 = y = 1/2`, we find that the derivative of the integrand is

```
deriv (λ y : ℜ ⇒ max 0 ((1/2 - y) / sqrt (1 + (1/2 - y)²))) (1/2) = [-1, 0].
```

When `y0` is just greater than `y`, the derivative will be near -1, but when `y0` is just less than `y`, the derivative will be near 0. Because the derivative at this point is a bounded interval, rather than a completely undefined result, it ends up being soundly neglected when it is integrated over:

```
eps=1e-3> deriv brightness (1/2)

[-0.4476, -0.4469]
```

The expression `deriv brightness (1/2)` is indeed maximal, meaning that it can be evaluated to arbitrary precision. Were the derivative of the integrand to be undefined rather than interval-valued, `deriv brightness (1/2)` would necessarily need to be undefined as well, but with these semantics, we can soundly compute the correct derivative.

This generalized notion of derivative that works for ReLU is based on *Clarke's generalized derivative* [Clarke 1990]. The basic idea can be motivated by the desire for continuity and robustness in the numerical computation. The derivative of ReLU is 1 for numbers imperceptibly greater than 0, and the derivative is 0 for numbers imperceptibly smaller than 0, so the derivative of ReLU at 0 should be consistent with those nearby answers. The *specialization relation* $\sqsubseteq$ on ℜ formalizes this notion of compatible behavior, where we have $[0, 1] \sqsubseteq 0$ and $[0, 1] \sqsubseteq 1$. We will prove a consistency theorem for our language (Proposition 5.4) that says that derivatives are always compatible, i.e., related by $\sqsubseteq$, with the infinitesimal rates of change indicated by its value-level operation.

3 SYNTAX AND SEMANTICS OF λ_S

```
0, 1, 2, ... : ℜ                       tangent A B : (A → B) → Tan A → Tan B
(+), (-), (*), (/) : ℜ → ℜ → ℜ         tangentValue A : Tan A → A
max : ℜ → ℜ → ℜ
sin, exp : ℜ → ℜ                       record (≅) A B = { to : A → B,
                                                          from : B → A }
integral01 : (ℜ → ℜ) → ℜ               tangent_R : Tan ℜ ≅ ℜ * ℜ
cutRoot : (ℜ → ℜ) → ℜ                  tangentProd A B : Tan (A * B) ≅ Tan A * Tan B
firstRoot : (ℜ → ℜ) → ℜ                tangentTo_R A : Tan (A → ℜ) ≅ (A → ℜ) * (A → ℜ)
max01 : (ℜ → ℜ) → ℜ
argmax01 : (ℜ → ℜ) → ℜ
```

Fig. 2. λ_S constants and their types.

λ_S is the simply-typed lambda calculus with the constants shown in Fig. 2. These include basic operators, such as arithmetic and trigonometric operators, higher-order operators, and primitives to compute derivatives. The syntax permits polymorphic type signatures, but semantically we treat polymorphism at the metatheoretic level.

Higher-Order Operators. The function `integral01` gives the Riemannian integral of a function on the interval $[0, 1]$. `max01` maximizes a function over the interval $[0, 1]$, and `argmax01` finds its maximizing argument. `cutRoot` finds the root of a function `f : ℜ → ℜ`, assuming that it has a single root and is negative for smaller values and positive for larger values. `firstRoot`, on input `f : ℜ → ℜ`, finds the first root of `f` on a region starting at 0.

Derivatives. `tangent` is a first-class function that computes derivatives, where the type function `Tan` gives the space of tangent bundles over a space; conceptually, a space of pairs of values and derivatives. The function `tangentValue` projects the value part of this tangent bundle. The isomorphisms of tangent bundles – i.e., `tangent_R`, `tangentProd`, and `tangentTo_R` – assist with manipulating the information that corresponds to the derivative part of the tangent bundle when it is possible for certain spaces. To concretize the concept behind these isomorphisms, we now present the implementation of `deriv` from Fig. 1, which uses `tangent` and these isomorphisms:

```
let deriv (f : ℜ→ℜ) (x : ℜ) : ℜ =
  snd (tangent_R.to (tangent f (tangent_R.from (x, 1))))
```

This implementation calls `tangent` with f and a query for the derivative of f at x in the direction 1. The query is a tangent bundle constructed with the isomorphism `tangent_R` from the pair $(x, 1)$. `deriv` then projects out the derivative part of `tangent`'s result, using `tangent_R` in the opposite direction and the standard second projection on binary products.

Semantics. Over the next sections, we develop the full syntax and semantics of λ_S. In §4, we describe a first-order (i.e., no higher-order functions) differentiable language that supports Clarke semantics and higher-order derivatives, by defining a Cartesian monoidal category **AD**. In §5, we will define semantics for the higher-order language λ_S by taking a category of presheaves, **HAD**, over **AD**. We defer computability concerns to §6.

4 SEMANTICS OF A FIRST-ORDER DIFFERENTIABLE LANGUAGE (AD)

In this section, we describe a first-order (i.e., no higher-order functions) differentiable language that supports Clarke semantics and higher-order derivatives, by defining a Cartesian monoidal category **AD**. Fig. 3 presents the syntax and typing rules for the language for **AD**. The $*$ type represents the unit type, having a single value ! in it. Given any object $K \in \mathbf{AD}$, the type expression $\lfloor K \rfloor$ represents the type whose semantics is K. Given any arrow $f : [\![\tau_1]\!] \rightsquigarrow [\![\tau_2]\!]$ of **AD** and given some expression $\Gamma \vdash e : \tau_1$ the syntax $\lfloor f \rfloor (e)$ applies the map f to the result of e. When the constants are binary operators like $+$ and $\times$, we permit syntactic sugar to write them infix, such that, e.g., $e_1 + e_2$ is shorthand for $+(e_1, e_2)$. The syntax $\frac{\partial e_y}{\partial x} |_{x=e_x} \cdot e_{dx}$ computes the directional derivative of e_y with respect to x at $x = e_x$ in the direction of infinitesimal perturbation e_{dx}.

Fig. 4 presents the semantics for the language for **AD**, which we explain in this section. Our semantics of derivatives is phrased in terms of *Clarke's generalized derivative* [Clarke 1990], which enables capturing differentiable properties of locally Lipschitz but nonsmooth functions such as max, min, and ReLU. We will now present the background material we use to define the semantics of the language for **AD**.

Syntax

$$
\begin{aligned}
&\text{variables } x\\
&\text{types } \tau ::= * \mid \tau_1 \times \tau_2 \mid \lfloor K \rfloor\\
&\text{contexts } \Gamma ::= \cdot \mid \Gamma, x : \tau\\
&\text{functions } f \in \mathrm{Arr}(\mathbf{AD})\\
&\text{expressions } e ::= x \mid \lfloor f \rfloor (e)\\
&\qquad \mid\ ! \mid (e, e)\\
&\qquad \mid \frac{\partial e}{\partial x}\Big|_{x=e} \cdot e\\
&\qquad \mid \text{let } x \triangleq e \text{ in } e
\end{aligned}
$$

Typing rules

$$
\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}
\qquad
\frac{\Gamma \vdash e : \tau_1 \qquad f : [\![\tau_1]\!] \rightsquigarrow [\![\tau_2]\!]}{\Gamma \vdash \lfloor f \rfloor (e) : \tau_2}
$$

$$
\frac{}{\Gamma \vdash ! : *}
\qquad
\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}
$$

$$
\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x \triangleq e_1 \text{ in } e_2 : \tau_2}
$$

$$
\frac{\Gamma, x : \tau_1 \vdash e_y : \tau_2 \qquad \Gamma \vdash e_x : \tau_1 \qquad \Gamma \vdash e_{dx} : \tau_1}{\Gamma \vdash \dfrac{\partial e_y}{\partial x}\Big|_{x=e_x} \cdot e_{dx} : \tau_2}
$$

Fig. 3. Syntax and typing rules for the language for **AD**.

Types

$$
\frac{\tau \text{ type}}{[\![\tau]\!] \in \mathrm{Ob}(\mathbf{AD})}
$$

$$
\begin{aligned}
[\![*]\!] &\triangleq 1_{\mathbf{AD}}\\
[\![\tau_1 \times \tau_2]\!] &\triangleq [\![\tau_1]\!] \times [\![\tau_2]\!]\\
[\![\lfloor K \rfloor]\!] &\triangleq K
\end{aligned}
$$

Contexts

$$
\frac{\Gamma \text{ context}}{[\![\Gamma]\!] \in \mathrm{Ob}(\mathbf{AD})}
$$

$$
\begin{aligned}
[\![\cdot]\!] &\triangleq 1_{\mathbf{AD}}\\
[\![\Gamma, x : \tau]\!] &\triangleq [\![\Gamma]\!] \times [\![\tau]\!]
\end{aligned}
$$

Terms

$$
\frac{\Gamma \vdash e : \tau}{[\![e]\!] : [\![\Gamma]\!] \rightsquigarrow [\![\tau]\!]}
$$

$$
\begin{aligned}
[\![\lfloor f \rfloor (e)]\!] &\triangleq f \circ [\![e]\!]\\
[\![!]\!] &\triangleq !\\
[\![(e_1, e_2)]\!] &\triangleq \langle [\![e_1]\!], [\![e_2]\!] \rangle\\
[\![\text{let } x \triangleq e_1 \text{ in } e_2]\!] &\triangleq [\![e_2]\!] \circ \langle \mathrm{id}, [\![e_1]\!] \rangle\\
[\![\frac{\partial e_y}{\partial x}\Big|_{x=e_x} \cdot e_{dx}]\!] &\triangleq [\![e_y]\!]' \circ \langle \langle \mathrm{id}, e_x \rangle, \langle 0, e_{dx} \rangle \rangle
\end{aligned}
$$

Fig. 4. Semantics of the language for **AD**.

4.1 Preliminaries

A *domain* D is a set with a partial-order structure $\sqsubseteq$ that supports *directed joins* $\bigsqcup_{d \in S} d$, which are just joins of *directed* subsets $S \subseteq D$, which are those subsets such that if $x, y \in D$, then there is some $z \in D$ such that $x \sqsubseteq z$ and $y \sqsubseteq z$. We call the partial-order relation $\sqsubseteq$ *specialization.* The relation $x \sqsubseteq y$ intuitively means that x behaves in a way that is compatible with how y behaves. An element $x \in D$ is *maximal* if for any $y \in D$, if $x \sqsubseteq y$, then $y \sqsubseteq x$.

Define

$$
\mathcal{R} \triangleq \{[a, b] \mid a, b \in \mathbb{R}, a \leq b\} \cup \{\mathbb{R}\}
$$

as the domain of *interval reals*, partially ordered ($\sqsubseteq$) by reverse set inclusion. Its maximal elements are the intervals of the form $[a, a]$, which we often just write as a. Arithmetic operations can be extended from $\mathbb{R}$ to $\mathcal{R}$ (see, e.g., Edalat and Lieutier [2004]). Note that $\mathbb{R}$ serves as a bottom element, and we refer to it with the symbol $\bot$. For any vector space V (over $\mathbb{R}$), let $\mathbf{C}(V)$ be the set of nonempty convex sets in V, with an order relation $\sqsubseteq$ also corresponding to reverse inclusion. Note that V serves as a bottom element, and we refer to it with the symbol $\bot$. Note that we have the sequence of embeddings $\mathbb{R}^n \hookrightarrow \mathcal{R}^n \hookrightarrow \mathbf{C}(\mathbb{R}^n)$: every vector $v \in \mathbb{R}^n$ can be treated as a tuple of singleton intervals $\mathcal{R}^n$, and every element $x \in \mathcal{R}^n$ can be treated as a (convex) hyperrectangle, where some dimensions of the hyperrectangle may be infinite. We use the notation $\iota_{\mathbb{R}^n \hookrightarrow \mathcal{R}^n}$ and $\iota_{\mathcal{R}^n \hookrightarrow \mathbf{C}(\mathbb{R}^n)}$ to denote these embeddings, respectively.

The Clarke Derivative. Let $f : \mathbb{R}^n \to \mathbb{R}^m$. If f is locally Lipschitz on $X \subseteq U$, let $Z_f \subseteq X$ be the points of nondifferentiability of f. The *Bouligand subdifferential* of f at $x \in X$ is the *set* of matrices

$$\partial_B f(x) \triangleq \left\{ H : \mathbb{R}^{m\times n} \mid \begin{array}{c} H = \lim_{j\to\infty} Jf(x_j) \text{ for some sequence } (x_j)_{j\in\mathbb{N}} \\ \text{where } x_j \in X \setminus Z_f \text{ for all } j \in \mathbb{N} \text{ and } \lim_{j\to\infty} x_j = x \end{array} \right\},$$

where J is the Jacobian operator defining the derivative of a function at a point where it is differentiable. The *Clarke Jacobian* of f at x is given by the convex hull $\partial f(x) \triangleq \mathrm{hull}(\partial_B f(x))$. The Clarke Jacobian $\partial f(x) \in \mathbf{C}(\mathbb{R}^{m\times n})$ is always compact (since f is locally Lipschitz).

Given $f : \mathbb{R}^n \to \mathbb{R}^m{}_\perp$, let U be the largest open set on which f is both defined and locally Lipschitz. We can define the *partial Clarke Jacobian* of f to be

$$\partial_\perp f(x) = \begin{cases} \partial f(x) & x \in U \\ \perp & x \notin U \end{cases}$$

such that $\partial_\perp : (\mathbb{R}^n \to \mathbb{R}^m{}_\perp) \to \mathbb{R}^n \to \mathbf{C}(\mathbb{R}^{m\times n})$. We can map values of $\mathbf{C}(A)$ to $A_\perp$ (for any A) by mapping maximal elements $\{x\} \in \mathbf{C}(A)$ to $x \in A_\perp$ and everything else to $\perp$. Using this conversion, we can also give the partial Clarke Jacobian the type $\partial_\perp : (\mathbb{R}^n \to \mathbf{C}(\mathbb{R}^m)) \to \mathbb{R}^n \to \mathbf{C}(\mathbb{R}^{m\times n})$, and thus we can also iterate the partial Clarke Jacobian construction to get higher-order derivatives $\partial_\perp^k : (\mathbb{R}^n \to \mathbb{R}^m{}_\perp) \to \mathbb{R}^n \to \mathbf{C}(\mathbb{R}^{m\times n^k})$. Note that the $k+1$th-order Clarke Jacobian is $\perp$ unless the kth-order Clarke Jacobian is maximal; thus, when defined, higher-order Clarke Jacobians are just Clarke Jacobians of conventional higher-order derivatives.

When a function is differentiable, its partial Clarke Jacobian is a maximal element. When it is locally Lipschitz but not differentiable, the partial Clarke Jacobian is a compact convex set. When it is not locally Lipschitz, the partial Clarke Jacobian is the entire space, corresponding to $\perp$.

4.2 Smoothish Maps

We will now define **AD**. The objects of **AD** are the natural numbers, where $n \in \mathbb{N}$ corresponds to n-dimensional Euclidean space. To emphasize that we are thinking of Euclidean space, we write the object $n \in \mathbb{N}$ as $\mathbb{R}^n$. A morphism of **AD** is a *smoothish map*: a *derivative tower* that is *successively consistent*. A *derivative tower* f between spaces $\mathbb{R}^n$ and $\mathbb{R}^m$, $f : \mathbb{R}^n \rightsquigarrow \mathbb{R}^m$, is a collection of continuous maps (taking the Scott topology for $\mathcal{R}$)

$$f^{(k)} : \mathbb{R}^n \times (\mathbb{R}^n)^k \to \mathcal{R}^m$$

for each $k \in \mathbb{N}$, where $f^{(k)}$ represents the kth-order derivative. This defines a smoothish map as a power series, where the first $\mathbb{R}^n$ argument is the point where the map is evaluated, and the remaining k arguments represent the inputs to a multilinear map representing the derivative.[1] Given vectors $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^k$, let $x \otimes y \in \mathbb{R}^{n\times k}$ denote the tensor product. Define $\mathrm{Mat}_k : (\mathbb{R}^n \times (\mathbb{R}^n)^k \to \mathcal{R}^m) \to \mathbb{R}^n \to \mathbb{R}^{m\times n^k}{}_\perp$ at a point $x \in \mathbb{R}^n$ such that $\mathrm{Mat}_k(f)(x) = M$ if there is a matrix $M \in \mathbb{R}^{m\times n^k}$ such that for all $dx_1, \ldots, dx_k \in \mathbb{R}^n$, we have

$$f(x; dx_1, \ldots, dx_k) = \iota_{\mathbb{R}^m \hookrightarrow \mathcal{R}^m}\left(M \cdot (dx_1 \otimes \ldots \otimes dx_k)\right), \tag{1}$$

and $\mathrm{Mat}_k(f)(x) = \perp$ if there is no such matrix (M). Equation 1 requires that f is multilinear in its $dx_1, \ldots, dx_k$ arguments, which means that f has a form that permits differentiation.

[1] This representation as *derivative towers* is largely drawn from [Elliott 2008].

Definition 4.1. We define a consistency relation $\mathrm{Cons}_k(g, f)$ for a function $g : \mathbb{R}^n \times (\mathbb{R}^n)^k \to \mathcal{R}^m$ and a function $f : \mathbb{R}^n \to \mathbf{C}(\mathbb{R}^{m \times n^k})$ to hold if for all $x \in \mathbb{R}^n$ and for all $dx_1, \ldots, dx_k \in \mathbb{R}^n$,

$$\iota_{\mathcal{R}^m \hookrightarrow \mathbf{C}(\mathbb{R}^m)}\,(g(x; dx_1, \ldots, dx_k)) \sqsubseteq f(x) \cdot (dx_1 \otimes \ldots \otimes dx_k).$$

A derivative tower f is *successively consistent* if for all $k \in \mathbb{N}$, we have

$$\mathrm{Cons}_{k+1}(f^{(k+1)}, \partial_{\perp}\mathrm{Mat}_k(f^{(k)})),$$

meaning that each successive derivative $f^{(k+1)}$ is consistent with the value-level behavior of $f^{(k)}$.

A *smoothish map f* is a successively consistent derivative tower. We call a smoothish map *smooth* if $f^{(k)}$ is maximal for all k (which agrees with the standard definition of a smooth map). We will later show (in §4.4) that smoothish maps form a category **AD**, and then by categorical semantics, that all expressions in the first-order language map to that category.

4.3 Primitives

Any first-order primitive may be implemented by giving its power-series representation. We use the notation $f^{(k)}(x; \vec{v})$ to denote the kth derivative of f at x in directions $\vec{v}$; a smoothish map f is defined by the collection of these functions for all $k \in \mathbb{N}$. These data provide power-series expansions around any input point. There is a map $0 : \Gamma \rightsquigarrow A$ (for any $\Gamma, A \in \mathbf{AD}$) that always returns zero regardless of its input. A linear map $f : A \to B$ determines a smooth map $\mathsf{linear}(f) : A \rightsquigarrow B$ by

$$\begin{aligned}\mathsf{linear}(f)^{(0)}(x) &\triangleq f(x)\\ \mathsf{linear}(f)^{(1)}(x; v) &\triangleq f(v)\\ \mathsf{linear}(f)^{(k+2)}(x; \vec{v}) &\triangleq 0\end{aligned}$$

Derivative-Tower Construction. A derivative tower can be viewed as a stream of a function and all of its derivatives. Streams are characterized by the isomorphism

$$\mathsf{Stream}(A) \cong A \times \mathsf{Stream}(A)$$

that says that a stream $s : \mathsf{Stream}(A)$ is exactly composed of its head, $\mathsf{head}(s) : A$, and its tail, $\mathsf{tail}(s) : \mathsf{Stream}(A)$. To construct a derivative tower, we define the map foldDer as an analogue to the *cons* operation on streams. For instance, given value-level definitions of sine and cosine, sin and cos, it is well-founded to define their derivative towers as

$$\begin{aligned}[\![\mathtt{sin}]\!]_{\mathbf{AD}} &\triangleq \mathsf{foldDer}(\sin, [\![x, dx \vdash \mathtt{cos}(x) * dx]\!]_{\mathbf{AD}})\\ [\![\mathtt{cos}]\!]_{\mathbf{AD}} &\triangleq \mathsf{foldDer}(\cos, [\![x, dx \vdash -\mathtt{sin}(x) * dx]\!]_{\mathbf{AD}}),\end{aligned}$$

just as it would be to define two mutually recursive streams $\mathsf{evens} = \mathsf{cons}(0, \mathsf{map}(\lambda x.\ x+1, \mathsf{odds}))$ and $\mathsf{odds} = \mathsf{cons}(1, \mathsf{map}(\lambda x.\ x+3, \mathsf{evens})))$.

We define foldDer as follows, where $f : A \to B$ and $g : A \times A \rightsquigarrow B$, such that $\mathsf{foldDer}(f, g) : A \rightsquigarrow B$.

$$\begin{aligned}\mathsf{foldDer}(f, g)^{(0)}(x) &\triangleq f(x)\\ \mathsf{foldDer}(f, g)^{(k+1)}(x; v_1, \ldots, v_{k+1}) &\triangleq g^{(k)}((x, v_1); (v_2, 0), \ldots, (v_{k+1}, 0)) \qquad (k \in \mathbb{N})\end{aligned}$$

One of the perturbations v_1 is passed in as the value to g, and then that perturbation is not considered to have any derivatives itself, hence the 0s in the second components of the perturbation passed to g. Setting the first components of the derivatives to $v_2, \ldots, v_{k+1}$ establishes these as independent infinitesimal perturbations of the first value component, x.

4.3.1 Arithmetic Operations. The binary arithmetic operations are first-order functions and so can be represented in **AD** as functions with the type $\mathbb{R} \times \mathbb{R} \rightsquigarrow \mathbb{R}$. Addition and subtraction are linear, so their semantics is simply $[\![+]\!]_{\mathbf{AD}} \triangleq \mathsf{linear}(+)$ and $[\![-]\!]_{\mathbf{AD}} \triangleq \mathsf{linear}(-)$. We define the smooth multiplication operator by

$$[\![*]\!]_{\mathbf{AD}} \triangleq \mathsf{foldDer}(\lambda(x, y).\ x \times y, \left[\!\left[(x, y), (dx, dy) \vdash x * dy + y * dx\right]\!\right]_{\mathbf{AD}}),$$

whose derivative is the familiar product rule. Note that our definition of $[\![*]\!]_{\mathbf{AD}}$ has two recursive references to multiplication's own *smooth* map. This recursive reference is well-founded because the result is used in a way that does not demand any further differentiation. This recursive pattern is similar to defining the stream of natural numbers $\mathsf{nats} : \mathsf{Stream}(\mathbb{N})$ by

$$\mathsf{nats} \triangleq \mathsf{cons}(0, \mathsf{map}\ (\lambda x.\ x + 1)\ \mathsf{nats}),$$

where mapping a function over nats does not demand any further calls to tail. Reciprocals (used for division) can be defined using foldDer as well, where all kth-order derivatives will return $\bot$ when the input is 0.

4.3.2 Lipschitz but Nonsmooth Functions. Many functions, such as max, min, and ReLU, are locally Lipschitz but not smooth. These functions are used pervasively in contexts that require differentiation, so their admissibility in a differential-programming semantics is paramount. Whereas most differential-programming semantics say that derivative of max is undefined when its arguments are equal, our use of Clarke derivatives permits a non-$\bot$ result.

We define `max` as follows, where hull computes the interval corresponding to the convex hull of the union of a set of points.

$$[\![\mathsf{max}]\!]^{(0)}_{\mathbf{AD}}\,(x, y) \triangleq \max(x, y)$$

$$[\![\mathsf{max}]\!]^{(1)}_{\mathbf{AD}}\,((x, y); (dx, dy)) \triangleq \begin{cases} dx & x > y \\ dy & y < x \\ \mathrm{hull}(\{dx, dy\}) & x = y \end{cases}$$

$$[\![\mathsf{max}]\!]^{(k+2)}_{\mathbf{AD}}\,((x, y); \vec{v}) \triangleq \begin{cases} 0 & x \neq y \\ \bot & x = y \end{cases}$$

4.3.3 Differentiation Operator. To give a semantics to the syntax $\frac{\partial e_y}{\partial x}\ |_{x=e_x}\ \cdot e_{dx}$ for differentiation, we first define a differentiation operator, postfix $'$, on smoothish maps, where $f : A \rightsquigarrow B$ maps to $f' : A \times A \rightsquigarrow B$. Defining this operator is nontrivial, because all the derivatives of f' must consider not only perturbations to the function value but also perturbations to the derivative argument, which are not accounted for in the original derivative tower: note that the kth derivative of f is a multilinear map from A^k, whereas the kth derivative of f' is a multilinear map from A^{2k}. We show the value and first few derivatives; because x will always be applied as the value argument to derivatives of f, we elide those arguments:

$$\begin{aligned} f'^{(0)}(x, v) &= f^{(1)}(v) \\ f'^{(1)}((x, v); (dx_a, dv_a)) &= f^{(2)}(v, dx_a) + f^{(1)}(dv_a) \\ f'^{(2)}((x, v); (dx_a, dv_a), (dx_b, dv_b)) &= f^{(3)}(v, dx_a, dx_b) + f^{(2)}(dv_a, dx_b) + f^{(2)}(dx_a, dv_b) \end{aligned}$$

The general formula is:

$$\begin{aligned} &f'^{(k)}((x,v);(dx_1,dv_1),\dots,(dx_k,dv_k)) \triangleq \\ &\qquad f^{(k+1)}(x;v,dx_1,\dots,dx_k) + \sum_{j=1}^{k} f^{(k)}(x;dx_1,\dots,dx_{j-1},dv_j,dx_{j+1},\dots dx_k). \end{aligned}$$

4.3.4 Revisiting Derivative Tower Construction. The $'$ operator is analogous to the tail operator of a stream, in that derivative towers have the section-retraction pair

$$\begin{tikzcd} A \rightsquigarrow B \arrow[r, shift left, tail, "\lambda f.(f^{(0)},f')"] & (A \to B) \times (A \times A \rightsquigarrow B) \arrow[l, shift left, two heads, "\mathsf{foldDer}"] \end{tikzcd}$$

that characterizes a derivative tower $f : A \rightsquigarrow B$ as a function $f^{(0)} : A \to B$ for the evaluation map of f together with a derivative tower $f' : A \times A \rightsquigarrow B$ where $f'(x,v)$ represents the directional derivative of f at x in the direction v.

Given this observation, we may for convenience in the rest of the paper define a smoothish map f by its value-level function $f^{(0)}$ and its smoothish derivative f', denoting an implicit use of foldDer. For example, we can equivalently define the smooth multiplication operator (§4.3.1) by

$$\begin{aligned} [\![*]\!]^{(0)}_{\mathbf{AD}} &\triangleq \lambda(x,y).\ x \times y \\ [\![*]\!]'_{\mathbf{AD}} &\triangleq \big[\!\big[(x,y),(dx,dy) \vdash x * dy + y * dx\big]\!\big]_{\mathbf{AD}}. \end{aligned}$$

4.4 Categorical Operations

AD forms a Cartesian monoidal category. We describe the categorical operations here, and prove that they satisfy the expected properties in Proposition 4.5. The maps $\mathrm{id} : A \to A$ (for all A), $! : \Gamma \to *$ (for all Γ), $\mathrm{fst} : A \times B \to A$ and $\mathrm{snd} : A \times B \to B$ (for all A, B) are all in fact linear maps and so can be made into smooth maps with the linear operator described above. Given $f : \Gamma \rightsquigarrow A$ and $g : \Gamma \rightsquigarrow B$, we define their product $\langle f, g \rangle : \Gamma \rightsquigarrow A \times B$ by

$$\langle f,g\rangle^{(k)}(x;\vec{v}) \triangleq (f^{(k)}(x;\vec{v}), g^{(k)}(x;\vec{v})),$$

It only remains to define composition. Composition of smooth maps is given by Faà di Bruno's formula. The definition is perhaps easier to understand by example for small k. The following shows derivatives of $g \circ f$ at x; since g is always differentiated at $f(x)$ and f is always differentiated at x, we elide those arguments:

$$\begin{aligned} (g \circ f)^{(0)}() &= g^{(0)}() \\ (g \circ f)^{(1)}(v_a) &= g^{(1)}(f^{(1)}(v_a)) \\ (g \circ f)^{(2)}(v_a,v_b) &= g^{(2)}(f^{(1)}(v_a), f^{(1)}(v_b)) + g^{(1)}(f^{(2)}(v_a,v_b)) \\ (g \circ f)^{(3)}(v_a,v_b,v_c) &= g^{(3)}(f^{(1)}(v_a), f^{(1)}(v_b), f^{(1)}(v_c)) \\ &\quad + g^{(2)}(f^{(2)}(v_a,v_b), f^{(1)}(v_c)) + g^{(2)}(f^{(2)}(v_a,v_c), f^{(1)}(v_b)) \\ &\quad + g^{(2)}(f^{(2)}(v_b,v_c), f^{(1)}(v_a)) + g^{(1)}(f^{(3)}(v_a,v_b,v_c)) \end{aligned}$$

The general formula is

$$(g \circ f)^{(k)}(x;\vec{v}) \triangleq \sum_{\pi \in \mathcal{H}(\{1,\dots,k\})} \text{let } n \triangleq |\pi| \text{ in } g^{(n)}\left(f(x); \begin{array}{c} f^{(|\pi_1|)}(x; v_{\pi_{1\,1}}, \dots, v_{\pi_{1\,|\pi_1|}}), \\ \vdots, \\ f^{(|\pi_n|)}(x; v_{\pi_{n\,1}}, \dots, v_{\pi_{n\,|\pi_n|}}) \end{array} \right),$$

where $\mathcal{H}(S)$ is the set of partitions of a set S, and $|S|$ is the cardinality of a set. Note that in the general case, the inputs to $g^{(n)}$ may be elements of $\mathcal{R}^b$ rather than $\mathbb{R}^b$ (for some $b \in \mathbb{N}$). Given any nth derivative $g^{(n)} : \mathbb{R}^b \times (\mathbb{R}^b)^k \to \mathcal{R}^m$, we extend it to apply to inputs $x \in \mathcal{R}^b$ and $dx_1, \ldots, dx_k \in \mathcal{R}^b$ by

$$g^{(n)}(x; dx_1, \ldots, dx_k) \triangleq \text{hull}\left\{g^{(n)}(y; dy_1, \ldots, dy_k) \mid y \in x, dy_1 \in dx_1, \ldots, dy_k \in dx_k\right\}.$$

Faà di Bruno's formula simplifies drastically in the case that either function is linear:

Proposition 4.2. *For any $g : B \to C$ and any derivative tower $f : A \rightsquigarrow B$, for any $k \in \mathbb{N}$ and any $x \in A$ and $v_1, \ldots, v_k \in A$,*

$$(\mathsf{linear}(g) \circ f)^{(k)}(x; v_1, \ldots, v_k) = g(f^{(k)}(v_1, \ldots, v_k))$$

Proof sketch. Because $\mathsf{linear}(g)^{(j)}(v_1, \ldots, v_j) = 0$ whenever $j > 1$ by definition of linear, all terms in the sum given by the Faà di Bruno formula where $|\pi| > 1$ will be 0. We can thus remove those terms, and the only term in the sum that will remain is the one where $|\pi| = 1$. □

Proposition 4.3. *For any consistent derivative tower $g : B \rightsquigarrow C$ and any $f : A \to B$ that maps maximal elements to maximal elements, for any $k \in \mathbb{N}$ and any maximal $x \in A$ and maximal $v_1, \ldots, v_k \in A$,*

$$(g \circ \mathsf{linear}(f))^{(k)}(x; v_1, \ldots, v_k) = g^{(k)}(f(v_1), \ldots, f(v_k)).$$

Proof sketch. Note that the term in the sum given by the Faà di Bruno formula where $|\pi| = k$ gives the right-hand side $g^{(k)}(f(v_1), \ldots, f(v_k))$. For all other terms in the sum, where $|\pi| < k$, we have that one of the inputs to $g^{(|\pi|)}$ will be 0, because we have $\mathsf{linear}(f)^{(j)}(v_1, \ldots, v_j) = 0$ whenever $j > 1$ by definition of linear.

We need to know that adding all these terms to the term $|\pi| = k$ makes no difference to the sum, which can happen either if all of the terms are 0, or if already $g^{(k)}(f(v_1), \ldots, f(v_k)) = \bot$, in which case the addition of any elements will not change the result. Thus, it suffices to prove that if $g^{(k)}(f(v_1), \ldots, f(v_k)) \neq \bot$, then all of those other terms in the sum are 0. A detailed technical argument can show that this is the case. □

The chain rule for Clarke derivatives is a specialization relation rather than an equality:

Proposition 4.4 (Chain rule for $\partial_\bot$). *Given $f : \mathbb{R}^n \to \mathbb{R}^m{}_\bot$, and $g : \mathbb{R}^m \to \mathbb{R}^k{}_\bot$ for all $x \in \mathbb{R}^n$ and all $dx \in \mathbb{R}^n$,*

$$\text{hull}\left(\left\{G \cdot F \cdot dx \mid G \in (\partial_\bot g)\,(f(x)), F \in \partial_\bot f(x)\right\}\right) \sqsubseteq \partial_\bot(g \circ f)(x) \cdot dx.$$

Proof sketch. A minor variation of [Clarke 1990, Corollary on page 75]. □

For example, at the value level, `max x 0 + min 0 x = x`, but the derivative of the left-hand side at 0 is $[0, 2]$ while the derivative at the right-hand side is 1, noting $[0, 2] \sqsubseteq 1$. This has important ramifications for λ_S, where we construct functions as compositions of others and need composition to be computable. Because of the specialization relation, we know that any behavior of a function in λ_S (e.g., $[0, 2]$) will be *compatible* with the ideal derivative of its value-level function (e.g., 1), but it may not return the maximal such value.

Proposition 4.5. *These operations (identity, composition, pairing, projections) give* **AD** *the structure of a Cartesian monoidal category. Therefore,* **AD** *admits the internal language described in Fig. 3.*

Proof sketch. There are two main classes of properties we must confirm about these categorical operations. First, we must verify that all of the operations preserve successive consistency, taking consistent derivative towers to consistent derivative towers. Second, we must confirm that the algebraic laws of a Cartesian monoidal category.

1. Operations preserve consistency. Because several of the categorical operations are of the form $\mathsf{linear}(f)$ we first prove a lemma that these maps are consistent:

   Lemma 4.6. *Call a map $f : \mathbb{R}^n \to \mathcal{R}^k$* linear *if it always outputs values in $\mathbb{R}^k$ and if it is linear in the traditional sense, i.e., $f(u) + f(v) = f(u + v)$ for all $u, v \in \mathbb{R}^n$ and $c \cdot f(v) = f(c \cdot v)$ for all $c \in \mathbb{R}$ and all $v \in \mathbb{R}^n$. Whenever $f : \mathbb{R}^n \to \mathcal{R}^k$ is linear,* $\mathsf{linear}(f)$ *is consistent.*

   Proof. Since f is linear in the above-defined sense, it is smooth, and so its derivatives will always be maximal, and will coincide with the traditional derivatives, which is exactly what $\mathsf{linear}(f)$ computes. □

   - *Identity maps are consistent.* Follows from Lemma 4.6.
   - *Product projections are consistent.* Also follows from Lemma 4.6.
   - *Pairing preserves consistency.* Essentially reduces to the following lemma:

     Proposition 4.7. *Given two maps $f : \mathbb{R}^n \to \mathbb{R}^m{}_\perp$ and $g : \mathbb{R}^n \to \mathbb{R}^k{}_\perp$, for any $x \in \mathbb{R}^n$,*

     $$\{\begin{bmatrix} u & v \end{bmatrix} \mid u \in \partial_\perp f(x), v \in \partial_\perp g(x)\} \sqsubseteq \partial_\perp(\lambda z.(f(z), g(z)))(x),$$

     where the pairing operation $(\cdot,\cdot) : \mathbb{R}^m{}_\perp \times \mathbb{R}^k{}_\perp \to \mathbb{R}^{m+k}{}_\perp$ returns $\perp$ if either of its arguments it $\perp$, or the pair of values if both inputs are not $\perp$.

     Proof sketch. Note that the set defined by the set comprehension on the left-hand side of the relation is convex, since both $\partial_\perp f(x)$ and $\partial_\perp g(x)$ are. Suppose $\begin{bmatrix} H & L \end{bmatrix}$ is in the Bouligand subdifferential of $\lambda z.(f(z), g(z))$ at x. Then it must be the case that H is in the Bouligand subdifferential for f and that L is in the Bouligand subdifferential for g. □

   - *Composition preserves consistency.* The full proof is quite detailed and technical. At its core, the proof proceeds much like the proof of the conventional Faà di Bruno formula, which can proceed by induction on the order of differentiation. However, whereas conventionally there is an equality between the Faá di Bruno formula and the derivative, in our case, their is an order relation that the Faá di Bruno formula is at most the derivative. The base case is the general chain rule of calculus, which in our case corresponds to the chain rule for Clarke derivatives, Proposition 4.4. The key step in the inductive case is the tensor product rule:

     Proposition 4.8 (Tensor product rule for $\partial_\perp$). *Given $g : D \to \mathbb{R}^{m \times n_j \times \ldots \times n_1}{}_\perp$ and for all $i \in \{1, \ldots, j\}$, $f_i : D \to \mathbb{R}^{n_i}{}_\perp$, for all $x \in D$,*

     $$\partial_\perp g(x) \cdot \left(\bigotimes_{k=1}^{j} f_k(x)\right) + g(x) \cdot \sum_{i=1}^{j} \left(\bigotimes_{k=1}^{i-1} f_k(x)\right) \otimes \partial_\perp f_i(x) \otimes \left(\bigotimes_{k=i+1}^{j} f_k(x)\right)$$
     $$\sqsubseteq$$
     $$\partial_\perp(g \cdot (f_1 \otimes \ldots \otimes f_j))(x).$$

     Proof. By repeated application of the product rule for Clarke derivatives. □

2. Algebraic laws hold.
   - *Composition is associative.* Follows from associativity and commutativity of $+$ and the associativity of taking partitions of partitions (in the appropriate sense).
   - $f \circ \mathrm{id} = f = \mathrm{id} \circ f$. Follows from Proposition 4.3 and Proposition 4.2, since id is linear.
   - *β and η laws for product projections.* Follows from the fact that linear commutes with pairing, i.e., $\mathsf{linear}(\langle f, g\rangle) = \langle \mathsf{linear}(f), \mathsf{linear}(g)\rangle$, and from Proposition 4.3 and Proposition 4.2. □

Syntax

$$
\begin{aligned}
&\text{variables } x\\
&\text{constant types } K \in \mathrm{Ob}(\mathbf{HAD})\\
&\text{types } \tau ::= * \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2\\
&\qquad \mid \lfloor K \rfloor\\
&\text{contexts } \Gamma ::= \cdot \mid \Gamma, x : \tau\\
&\text{constants } k \in \mathrm{Arr}(\mathbf{HAD})\\
&\text{expressions } e ::= x \mid \lfloor k \rfloor \mid e\ e \mid \lambda x.\, e\\
&\qquad \mid\ ! \mid (e, e)\\
&\qquad \mid \text{let } x \triangleq e \text{ in } e
\end{aligned}
$$

Typing rules

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda\, x : \tau_1.\, e : \tau_1 \to \tau_2}$$

$$\frac{k \in [\![\Gamma]\!] \to_{\mathbf{HAD}} [\![\tau]\!]}{\Gamma \vdash \lfloor k \rfloor : \tau} \qquad \frac{}{\Gamma \vdash\ ! : *}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x \triangleq e_1 \text{ in } e_2 : \tau_2}$$

Fig. 5. Syntax and typing rules for λ_S. The constants are those listed in Fig. 2.

4.5 Consistency

The derivatives that our semantics defines are *consistent*: the behaviors of kth derivative that is computed, $[\![e]\!]^{(k)}_{\mathbf{AD}}$, are compatible with the derivatives that would be abstractly defined by looking at its value-level behavior, $\partial^k_\perp \mathrm{Mat}_0([\![e]\!]^{(0)}_{\mathbf{AD}})$. This proposition follows by first demonstrating that derivative towers are *successively consistent.*

Proposition 4.9. *Given any term $\Gamma \vdash e : \tau$, the derivative tower $[\![e]\!]_{\mathbf{AD}}$ is successively consistent.*

Proof sketch. By induction on the typing derivation of e. We then see that, to know the proposition is true, we must know that the derivative towers for all primitives are consistent (including product projections) and that pairing and composition preserve successive consistency (proof sketch in Proposition 4.5). □

Proposition 4.10 (Consistency of differentiation in the first-order language). *Given any term $\Gamma \vdash e : \tau$, for all $k \in \mathbb{N}$,* $\mathrm{Cons}_{k+1}\left([\![e]\!]^{(k+1)}_{\mathbf{AD}}, \partial^{k+1}_\perp \mathrm{Mat}_0([\![e]\!]^{(0)}_{\mathbf{AD}})\right)$.

Proof sketch. By Proposition 4.9, $[\![e]\!]_{\mathbf{AD}}$ is successively consistent. Then the proof proceeds by a simple induction on k. □

5 HIGHER-ORDER SEMANTICS (HAD)

The category **AD** does not admit exponentiation (function spaces), since its objects are limited to $\mathbb{R}^n$. However, higher-order functions yield novel expressive power that is critical for §7. To admit higher-order functions, λ_S uses a category **HAD** of *presheaves* over **AD**, i.e., $\mathbf{HAD} = [\mathbf{AD}^{\mathrm{op}}, \mathbf{Set}]$.

Syntax and Semantics. The basic syntax of **HAD** is that of the simply-typed lambda calculus, shown in Fig. 5, where the constants are those listed in Fig. 2. Fig. 6 presents the semantics of λ_S (generic to any Cartesian closed category of presheaves). However, the categorical semantics in **HAD** means that λ_S is inherently extensible and not limited to just those constants in Fig. 2; any object or morphism in **HAD** could be added to the language.

We now proceed to describe the semantics of the higher-order constants in Fig. 2.

Types

$$\begin{aligned}
[\![*]\!](\Gamma) &\triangleq 1_{\mathbf{Set}} \\
[\![\tau_1 \times \tau_2]\!](\Gamma) &\triangleq [\![\tau_1]\!](\Gamma) \times [\![\tau_2]\!](\Gamma) \\
[\![\lfloor K \rfloor]\!](\Gamma) &\triangleq K(\Gamma) \\
[\![\tau_1 \to \tau_2]\!](\Gamma) &\triangleq \int_{\Delta \in \mathbf{AD}} (\Delta \rightsquigarrow \Gamma) \times [\![\tau_1]\!](\Delta) \to [\![\tau_2]\!](\Delta)
\end{aligned}$$

Terms

$$\begin{aligned}
[\![e_1\, e_2]\!](\gamma) &\triangleq [\![e_1]\!](\gamma)(\mathrm{id}, [\![e_2]\!](\gamma)) \\
[\![\lambda\, x : \tau_1.\, e]\!] &\triangleq \mathsf{abstract}([\![e]\!]) \\
[\![\lfloor k \rfloor]\!](\gamma) &\triangleq k(\gamma) \\
[\![!]\!] &\triangleq ! \\
[\![(e_1, e_2)]\!](\gamma) &\triangleq ([\![e_1]\!](\gamma), [\![e_2]\!](\gamma)) \\
[\![\text{let } x \triangleq e_1 \text{ in } e_2]\!] &\triangleq [\![(\lambda x.\, e_2)\, e_1]\!]
\end{aligned}$$

Fig. 6. The semantics of λ_S.

5.1 Ground Types and First-Order Primitives

Any space $X \in \mathbf{AD}$ can be lifted into a presheaf $\mathbf{HAD}$ by the *Yoneda embedding*, $\overline{X} \in \mathbf{HAD}$, which acts as $\overline{X}(\Gamma) \triangleq \Gamma \rightsquigarrow X$. Because the Yoneda embedding is full and faithful and preserves products, ground types (and their products) in $\mathbf{HAD}$ represent Cartesian spaces and first-order functions represent smoothish maps. In particular, $[\![\mathfrak{R}]\!] \triangleq \overline{\mathcal{R}}$. Note that all first-order functions from $\mathbf{AD}$ can be lifted into $\mathbf{HAD}$ by the Yoneda embedding.

5.2 Smoothish Higher-Order Primitive Functions

Each of the smoothish higher-order primitive functions has a type $(\mathfrak{R} \to \mathfrak{R}) \to \mathfrak{R}$ in λ_S. To construct primitives of this type, we can equivalently construct maps $(\Gamma \times \mathcal{R} \rightsquigarrow \mathcal{R}) \to (\Gamma \rightsquigarrow \mathcal{R})$ for all $\Gamma \in \mathbf{AD}$ in λ_C.[2] Such a map takes as input $\mathcal{R}$-valued expression in a context $\Gamma \times \mathcal{R}$ and produce an $\mathcal{R}$-valued expression in the context Γ (for any Γ).

Accordingly, we defined these second-order primitives with parametrically polymorphic mappings of derivative towers. We must confirm that these definitions preserve successive consistency, i.e., they must map successively consistent derivative towers to successively consistent derivative towers. In general, this boils down to confirming that taking the derivative of the value-level definitions of each of these primitives (when applied to any possible function $f : \Gamma \times \mathcal{R} \rightsquigarrow \mathcal{R}$) yields the definitions for the derivatives of these primitives. It is possible to confirm for each definition that this is the case.

5.2.1 *Smooth integral.*

The integral `integral01` is defined as follows for any $f : \Gamma \times \mathbb{R} \rightsquigarrow \mathbb{R}$:

$$[\![\texttt{integral01}]\!]_{\mathbf{HAD}}\,(f)^{(k)}(\gamma; d\gamma_1, \ldots, d\gamma_k) \triangleq \int_0^1 f^{(k)}(\gamma, x; (d\gamma_1, 0), \ldots, (d\gamma_k, 0))\, dx.$$

Since integration is a linear operator, we essentially just integrate the first-order infinitesimal perturbations arising from f at every order of derivative. Integration is *smooth* in the sense that if its input is smooth, its output will be smooth as well. Note the similarity between the above AD tower and the result of postcomposing a linear function ℓ after a function f arising from Faà di Bruno's formula described previously. The reader may wonder how a semantics invoking integration might be computable; we discuss this in §6.

[2] In any category of presheaves $[C^{\mathrm{op}}, \mathbf{Set}]$, letting $\overline{\cdot}$ denote the Yoneda embedding and letting $\Rightarrow$ denote the internal hom, then there is an equivalence between constants with the second-order type $(\overline{A} \Rightarrow \overline{B}) \Rightarrow \overline{C}$ and the end $\int_\Gamma (\Gamma \times A \to_C B) \to (\Gamma \to_C C)$:

$$1 \to_{[C^{\mathrm{op}}, \mathbf{Set}]} (\overline{A} \Rightarrow \overline{B}) \Rightarrow \overline{C} \cong (\overline{A} \Rightarrow \overline{B}) \to_{[C^{\mathrm{op}}, \mathbf{Set}]} \overline{C} = \int_\Gamma (\overline{A} \Rightarrow \overline{B})(\Gamma) \to \overline{C}(\Gamma) \cong \int_\Gamma (\Gamma \times A \to_C B) \to (\Gamma \to_C C)$$

5.2.2 Smoothish Root Finding. The primitive `cutRoot` $: (\mathfrak{R} \to \mathfrak{R}) \to \mathfrak{R}$ smoothly finds the root of any function with a single isolated root that is positive to its left and negative to its right.

Equivalently, `cutRoot` is a map $(\Gamma \times \mathbb{R} \rightsquigarrow \mathbb{R}) \longrightarrow (\Gamma \rightsquigarrow \mathbb{R})$. We will define `cutRoot` by using the stream characterization of smooth maps, defining it with a function for its evaluation map and a smooth map for its derivative:

$$\llbracket \texttt{cutRoot} \rrbracket_{\mathbf{HAD}}(f)^{(0)} \triangleq \lambda\gamma.\ [\sup\{x : \mathbb{R} \mid f^{(0)}(\gamma, x) > 0\}, \inf\{x : \mathbb{R} \mid f^{(0)}(\gamma, x) < 0\}]$$

$$\llbracket \texttt{cutRoot} \rrbracket_{\mathbf{HAD}}(f)' \triangleq \left\llbracket \gamma, d\gamma \vdash \text{let } y \triangleq \left\lfloor \llbracket \texttt{cutRoot} \rrbracket_{\mathbf{HAD}}(f) \right\rfloor(\gamma) \text{ in } -\frac{\lfloor f' \rfloor((\gamma, y), (d\gamma, 0))}{\lfloor f' \rfloor((\gamma, y), (0, 1))} \right\rrbracket_{\mathbf{AD}}$$

The formula for the derivative is a simple application of the *implicit function theorem.* Note that we have a well-founded recursive reference following the same pattern as with multiplication.

`cutRoot` enables root-finding only for functions that have only one root. In graphics, for ray tracing of implicit surfaces, it is useful to be able to find for a function `f` $: \mathfrak{R} \to \mathfrak{R}$ the least root $x \in [0, 1]$ such that f switches from positive for values just less than x to negative for values just greater than x. `firstRoot` $: (\mathfrak{R} \to \mathfrak{R}) \to \mathfrak{R}$ accomplishes this:

$$\llbracket \texttt{firstRoot} \rrbracket_{\mathbf{HAD}}(f)^{(0)} \triangleq \lambda\gamma.\ \begin{matrix}[\sup\{x \in [0,1] \mid \forall q \in [0,x].\ f^{(0)}(\gamma, q) > 0\} \\ , \inf\{x \in [0,1] \mid \exists q \in [0,x].\ f^{(0)}(\gamma, x) < 0\}]\end{matrix}$$

$$\llbracket \texttt{firstRoot} \rrbracket_{\mathbf{HAD}}(f)' \triangleq \left\llbracket \gamma, d\gamma \vdash \begin{matrix}\text{let } y \triangleq \left\lfloor \llbracket \texttt{firstRoot} \rrbracket_{\mathbf{HAD}}(f) \right\rfloor(\gamma) \text{ in} \\ -\frac{\lfloor f' \rfloor((\gamma, y), (d\gamma, 0))}{\lfloor f' \rfloor((\gamma, y), (0, 1))}\end{matrix} \right\rrbracket_{\mathbf{AD}}$$

Like with `cutRoot`, its derivatives are determined by the implicit function theorem; the only difference is in the definition of the value of the root.

5.2.3 Smoothish Optimization. λ_S admits primitives `argmax01, max01` $: (\mathfrak{R} \to \mathfrak{R}) \to \mathfrak{R}$ that find the maximizing argument and the maximum, respectively, of a function `f` $: \mathfrak{R} \to \mathfrak{R}$ over the unit interval. Equivalently, each of `argmax01` and `max01` are maps $(\Gamma \times \mathbb{R} \rightsquigarrow \mathbb{R}) \longrightarrow (\Gamma \rightsquigarrow \mathbb{R})$.

We first describe `argmax01`, which is defined as follows:

$$\llbracket \texttt{argmax01} \rrbracket_{\mathbf{HAD}}(f)^{(0)} \triangleq \lambda\gamma.\ \text{hull}\left(\{x \in [0,1] \mid f(\gamma, x) = \max_{z \in [0,1]} f(\gamma, z)\}\right)$$

$$\llbracket \texttt{argmax01} \rrbracket_{\mathbf{HAD}}(f)' \triangleq \left\llbracket \gamma, d\gamma \vdash \begin{matrix}\text{let } y \triangleq \left\lfloor \llbracket \texttt{argmax01} \rrbracket_{\mathbf{HAD}}(f) \right\rfloor(\gamma) \text{ in} \\ \text{let } f'_y \triangleq \lfloor f' \rfloor((\gamma, y), (0, 1)) \text{ in} \\ \begin{cases} -\frac{\lfloor f'' \rfloor(((\gamma, y), (0,1)), ((d\gamma, 0), (0,0)))}{\lfloor f'' \rfloor(((\gamma, y), (0,1)), ((0,1), (0,0)))} & 0 < y < 1 \\ 0 & y = 0 \land f'_y < 0 \\ 0 & y = 1 \land f'_y > 0 \\ \bot & \text{otherwise} \end{cases}\end{matrix} \right\rrbracket_{\mathbf{AD}}$$

The input is a smooth map $f : \Gamma \times \mathbb{R} \rightsquigarrow \mathbb{R}$. In general, for a $\gamma \in \Gamma$, there may be many values of x achieving the same maximum $f(\gamma, x)$, so the value-level definition takes the convex hull of the set of those maximizing arguments. The derivative of `argmax01` is not $\bot$ only when its value is maximal, i.e., there is only one maximizing argument, which we will call y. There are three possibilities for y: either $0 < y < 1$, or $y = 0$, or $y = 1$. In the case that $0 < y < 1$, then if f'' is defined at (γ, y), then we know that $f'_y(\gamma, y) = 0$ and that this argmax is an isolated root of f'_y, where f'_y is the derivative of f with respect to its latter argument. Any infinitesimal perturbation $d\gamma$ to γ results in an infinitesimal perturbation to the root of f'_y, so the implicit function theorem defines how the root changes. If the maximizing argument y is on the boundary, i.e., $y = 0$ or $y = 1$, then if we additionally know that either $f'_y(\gamma, y) < 0$ or $f'_y(\gamma, y) > 0$, respectively, then it must be the case that the derivative of the argmax is 0, because the argmax will be stuck at the boundary no matter how γ might be infinitesimally perturbed.

We can now proceed to describe `max01`, whose derivative is defined in terms of `argmax01`:

$$[\![\texttt{max01}]\!]_{\mathbf{HAD}}\,(f)^{(0)} \triangleq \lambda\gamma.\ \max_{x\in[0,1]} f(\gamma, x)$$

$$[\![\texttt{max01}]\!]_{\mathbf{HAD}}\,(f)' \triangleq (f \circ [\![\texttt{argmax01}]\!]_{\mathbf{HAD}}\,(f))'$$

Just as the derivative of `max` depends on which argument results in the max, similarly the derivative of `max01` is a function of the maximizing argument. If we can isolate a single argmax, then `max01 f = f (argmax01 f)`, and thus all the derivatives of `max01 f` follow from the chain rule and the smooth derivatives of `f` and `argmax01 f`.

5.3 Internal Derivatives of Functions at All Types

The primitive `tangent A B : (A → B) → Tan A → Tan B` permits the expression of the derivative of any function in λ_S, with any input type `A` and output type `B`. These types are much more general than those on which differentiation in classically defined in mathematics. In this section, we will explain the semantics of `tangent` and `Tan`, which generalize the notion of differentiation from **AD** to apply to all objects in **HAD** (i.e., all types in λ_S).

We need to systematically generalize the derivative of **AD**, as expressed with the postfix $'$ operator, to apply to **HAD**. Following Vákár et al. [2018], we can apply the categorical technique of left Kan extensions, which extend a functor on a base category to one that acts on presheaves over that category. Our definition of generalized tangent spaces and its properties will also be similar to the dvs diffeology on internal tangent bundles as described by Christensen and Wu [2017]. Accordingly, we can lift the operation of forward-mode differentiation from the first-order language **AD** to the higher-order language **HAD**. Defining

$$\mathsf{valueWithDer} : (A \rightsquigarrow B) \longrightarrow (A \times A \rightsquigarrow B \times B)$$

$$\mathsf{valueWithDer}(f) \triangleq \left[\!\!\left[x, dx \vdash (\lfloor f \rfloor\,(x), \lfloor f' \rfloor\,(x, dx)) \right]\!\!\right]_{\mathbf{AD}},$$

we find that valueWithDer defines a functor on **AD** acting on objects by $X \mapsto X \times X$ from a space X to its tangent bundle $X \times X$, where the tangent bundle $X \times X$ represents a point of X together with an infinitesimal perturbation of that point. The functoriality of valueWithDer follows from the chain rule of differentiation (and that $\lfloor \mathrm{id}' \rfloor\,(x, dx) = dx$).

This functor can be extended to **HAD** via a left Kan extension to produce a functor `Tan` and its functorial map `tangent A B : (A → B) → Tan A → Tan B`, which runs generalized forward-mode derivatives, interpreted geometrically as a pushforward of the tangent bundles. Concretely, we define the *tangent bundle* functor `Tan`, as a left Kan extension, corresponds to a *coend*:

$$\texttt{Tan}(F)(\Gamma) \cong \int^{\Delta} (\Gamma \rightsquigarrow \Delta^2) \times F(\Delta).$$

Informally, the tangent bundle over the λ_S type F in a context Γ is represented by a pair of a value and infinitesimal perturbation $\Gamma \rightsquigarrow \Delta^2$ for some Cartesian space Δ (i.e., $\Delta = \mathbb{R}^n$ for some $n \in \mathbb{N}$), together with a map from the space Δ into the type F. Thus, if we wish to define an infinitesimal perturbation into a complicated type F, we are able to do it by choosing a Cartesian space Δ to express that infinitesimal perturbation, and then we construct a map from Δ to F. All elements of the tangent bundle of F arise in that way.

We now explain how these tangent bundles work with an example. Suppose $F = \mathbb{R}^2$ and we want to represent the tangent bundle $((0, 1), (1, 0)) \in \mathbb{R}^2 \times \mathbb{R}^2$, i.e., the vector $(0, 1)$ moving infinitesimally in the $(1, 0)$ direction. Since there are no variables in the context, we can define the tangent bundle at once for all Γ. The type of generalized tangent bundles is

$$\mathtt{Tan}(\overline{\mathbb{R}^2})(\Gamma) \cong \int^{\Delta} (* \rightsquigarrow \Delta^2) \times (\Delta \rightsquigarrow \mathbb{R}^2).$$

We can represent the tangent bundle $((0,1),(1,0)) \in \mathbb{R}^2 \times \mathbb{R}^2$ in two equivalent ways. The straightforward way is to take $\Delta = \mathbb{R}^2$ and put the point and its perturbation in the first component and the identity map in the second,

$$(\mathbb{R}^2, (\langle (0,1),(1,0)\rangle, \mathrm{id})).$$

Alternatively, we can represent it with a parametric function $f : \mathbb{R} \to \mathbb{R}^2$ defined by $f(t) = (0,1) + t \cdot (1,0)$, describing a point that moves from $(0,1)$ at $t = 0$ in the direction of $(1,0)$ as t increases:

$$(\mathbb{R}, (\langle 0,1\rangle, \lambda t.\ (0,1) + t \cdot (1,0))).$$

Two members $(\tau_1, (f_1, g_1))$ and $(\tau_2, (f_2, g_2))$ of $\mathtt{Tan}(\overline{\mathbb{R}^2})(\Gamma)$ are equivalent if

$$\mathsf{valueWithDer}(g_1) \circ f_1 = \mathsf{valueWithDer}(g_2) \circ f_2.$$

Indeed, this is the case for the two examples above, as both compositions yield $\langle (0,1),(1,0)\rangle$. This criterion for equivalence is for representable types such as $\overline{\mathbb{R}^2}$ but generalizes for tangent bundles over types that are not representable. It intuitively captures the notion that the first component of the tuple represents a tangent bundle of a representable space, whereas the second is a map that applies to that output but is yet to be differentiated. This is the justification for applying valueWithDer above.

The Kan extension is genuinely an extension of the underlying functor valueWithDer. That is, we have the equivalence $\mathtt{Tan}(\overline{A}) \cong \overline{A^2}$, where $\overline{\cdot}$ is the Yoneda embedding (Proposition 5.2). This means that the generalized tangent bundle for Cartesian spaces $\mathbb{R}^n$ is indeed $\mathbb{R}^n \times \mathbb{R}^n$: one $\mathbb{R}^n$ for the point and one $\mathbb{R}^n$ for the infinitesimal perturbation.

The generalized tangent bundle functor supports other operations as well. A polymorphic function `tangentValue A : Tan A → A` projects out the base point. The primitive `tangentProd A B : Tan (A * B) ≅ Tan A * Tan B` implements the following isomorphism:

Proposition 5.1. *Tangent bundles commute with products, i.e.,* $\mathtt{Tan}(F \times G) \cong \mathtt{Tan}(F) \times \mathtt{Tan}(G)$.

Proof sketch. First, we construct mappings in both directions: We easily have the product projections $\mathtt{Tan}(F \times G) \to \mathtt{Tan}(F)$ and $\mathtt{Tan}(F \times G) \to \mathtt{Tan}(G)$. Conversely, given $(\mathtt{Tan}(F) \times \mathtt{Tan}(G))(\Gamma)$, we get Δ_1 and Δ_2 with $f : \Gamma \rightsquigarrow \Delta_1^2$ and $g : \Gamma \rightsquigarrow \Delta_2^2$ and $F(\Delta_1)$ and $G(\Delta_2)$. Taking $\Delta \triangleq \Delta_1 \times \Delta_2$, we can define $h(\gamma) \triangleq ((x,y),(dx,dy))$ where $(x, dx) = f(\gamma)$ and $(y, dy) = g(\gamma)$. Using the pullbacks $\pi_1^* : F(\Delta_1) \to F(\Delta_1 \times \Delta_2)$ and $\pi_2^* : G(\Delta_2) \to G(\Delta_1 \times \Delta_2)$, we can produce $\mathtt{Tan}(F \times G)(\Gamma)$.

Next, it is possible to confirm that these mappings are mutually inverse, using the fact that

$$\mathsf{valueWithDer}(\lambda(x,y).(f(x), g(y)))((x,y),(dx,dy)) = ((f(x), g(x)), (f'(x,dx), g'(y,dy))),$$

together with general properties of limits and functoriality of $\mathtt{Tan}$, F, and G. □

The primitive `tangent_R : Tan` $\mathfrak{R}$ `≅` $\mathfrak{R}$ `*` $\mathfrak{R}$ that implements the following isomorphism (for the special case of $\mathfrak{R}$):

Proposition 5.2. *We have the equivalence* $\mathtt{Tan}(\overline{A}) \cong \overline{A^2}$, *where* $\overline{\cdot}$ *is the Yoneda embedding.*

Proof.

$$\mathtt{Tan}(\overline{A})(\Gamma) \cong \int^{\Delta} (\Gamma \rightsquigarrow \Delta^2) \times (\Delta \rightsquigarrow A).$$

Given $f : \overline{A^2}(\Gamma) = \Gamma \rightsquigarrow A^2$, we can take $\Delta = A$ and use (f, id). Given an element of $\mathtt{Tan}(\overline{A})(\Gamma)$, i.e., some Δ and $f : \Gamma \rightsquigarrow \Delta^2$ and $g : \Delta \rightsquigarrow A$, then $\mathsf{valueWithDer}(g) \circ f : \overline{A^2}(\Gamma)$. □

Note that we have $\mathtt{tangentValue} \circ \mathtt{tangent\_R} = \mathtt{fst}$, i.e., the first component is the base point and the second is the infinitesimal perturbation.

Note that the types to represent isomorphisms of tangent bundles are not necessarily isomorphisms in λ_S: the type ≅ just corresponds to pairs of maps back and forth. The primitive `tangentTo_R A : Tan (A → 𝕽) ≅ (A → 𝕽) * (A → 𝕽)`, in which tangent bundles distribute over functions into $\mathfrak{R}$, implements mappings that are an isomorphism only when we restrict $\mathfrak{R}$ to $\mathbb{R}$ (rather than all of $\mathcal{R}$):

Proposition 5.3. *There is an isomorphism* $\mathtt{Tan}(A \Rightarrow \overline{\mathbb{R}}) \cong A \Rightarrow \mathtt{Tan}(\overline{\mathbb{R}}) \cong A \Rightarrow \overline{\mathbb{R}^2}$.

Proof. First, we construct the mappings in each direction. Note that these types are:

$$\mathtt{Tan}(A \Rightarrow \overline{\mathbb{R}})(\Gamma) \cong \int^{\Delta} (\Gamma \rightsquigarrow \Delta^2) \times \int_X (X \rightsquigarrow \Delta) \to A(X) \to (X \rightsquigarrow \mathbb{R})$$

$$(A \Rightarrow \overline{\mathbb{R}^2})(\Gamma) \cong \int_X (X \rightsquigarrow \Gamma) \to A(X) \to (X \rightsquigarrow \mathbb{R}^2)$$

Given $f : (A \Rightarrow \overline{\mathbb{R}^2})$, we take $\Delta = \Gamma \times \mathbb{R}$, and use

$$\begin{gathered}\exists \Gamma \times \mathbb{R}.\ (\lambda \gamma.\ ((\gamma, 0), (0, 1)), \Lambda X.\ \lambda(e : X \rightsquigarrow \Gamma \times \mathbb{R}).\ \lambda(a : A(X)).\\ \text{let } (g, dg) = f(X, \pi_1 \circ e, a) \text{ in } \lambda x : X.\ g(x) + \pi_2(e(x)) \cdot dg(x))\end{gathered}$$

Conversely, given a member of $\mathtt{Tan}(A \Rightarrow \overline{\mathbb{R}})(\Gamma)$, i.e., a Δ with $d : \Gamma \rightsquigarrow \Delta^2$ and $f : \int_X (X \rightsquigarrow \Delta) \to A(X) \to (X \rightsquigarrow \mathbb{R})$, we can provide

$$\Lambda X.\ \lambda(e : X \rightsquigarrow \Gamma).\ \lambda(a : A(X)).\ \mathsf{valueWithDer}(f(X, \pi_1 \langle f(X, \pi_1 \circ d \circ e, a), f(X, \pi_2 \circ d \circ e, a) \rangle)).$$

Next, we must confirm that these mappings are mutually inverse. This boils down to the basic identity $f'(x; v) = \frac{\partial(f(x + t \cdot v))}{\partial t} \,|_{t=0}$. □

Note that it is not an isomorphism for all of $\mathfrak{R}$, because we rely on the algebraic law $x + 0 \cdot y = x$ for all y, but if we allow $y \in \mathcal{R} \setminus \mathbb{R}$, there is the counterexample $x + 0 \cdot \bot = \bot$.

5.4 Consistency

Proposition 5.4 (Consistency of differentiation in the higher-order language). *Given any term* $\Gamma \vdash e : \tau$ *in* λ_S *where* Γ *is a context of all ground types and* τ *is a ground type, then* $[\![e]\!]_{\mathbf{HAD}}$ *is equivalent to some first-order smoothish map* f*, i.e., successively consistent derivative tower.*

Proof. Since the Yoneda embedding is full and faithful, first-order terms in **HAD** correspond to morphisms in **AD**, so this statement reduces to Proposition 4.10. □

6 COMPUTABILITY AND NUMERICALLY-SOUND IMPLEMENTATION

It is not obvious that the categorical semantics of λ_S we present in §4-5 is actually implementable (in a sound manner). The semantics critically uses reals and real arithmetic, rather than some approximation like floating point (which would fail to give even the most basic equalities such as $1/5 + 2/5 = 3/5$). And value-level definitions of higher-order primitives in λ_S are expressed in terms of mathematical operations for integration, optimization, and root finding applied to arbitrary continuous maps. In fact, our semantic development is computable, and we have implemented it in a numerically sound manner as an embedded DSL in Haskell.

Our semantics can be developed constructively and interpreted within the *internal language* of another topos, which we call λ_C, in order to provide a computable interpretation. We base λ_C on MarshallB [Sherman et al. 2019]. Our implementation of λ_S more-or-less directly follows interpreting the semantics of λ_S within λ_C and in turn implementing λ_C in Haskell.

λ_C is a topos of sheaves over a Cartesian monoidal category that we call **CTop**. **CTop** is a category of computably presented topological spaces and computable continuous maps. λ_C is the topos of sheaves over **CTop** with the open cover topology (along the lines of [Fourman 1984]).

What results is a stack of languages: λ_S reducing to **AD**, implemented in λ_C, which reduces to **CTop**, which carries the final executable content of ground terms. We can view it like a stack of metaprogramming languages on top of **CTop**: ultimately, when a closed term of λ_S (or any other language in the stack) of ground type is evaluated and displayed as a sequence of improving approximations, it is in fact a closed term of **CTop**, i.e., a computable point of a topological space.

Semantics of λ_C and Implications for λ_S. λ_C is a language whose types are (generalized) topological spaces with computable structure and whose functions are (generalized) computable continuous maps. λ_C permits all the higher-order functions and higher-order types that we will seek to define in λ_S and enables their computation to arbitrary precision. This section describes λ_C by example. In λ_C, the type $\mathfrak{R}$ in λ_C represents the interval reals $\mathcal{R}$. One closed term, or value, of type $\mathfrak{R}$ is `sqrt 2`. A value of $\mathfrak{R}$ represents a point of the space $\mathcal{R}$ and is computationally represented by streams of increasingly precise approximations (i.e., monotone with respect to $\sqsubseteq$):

```
> sqrt 2 : ℜ

[1.4142135619, 1.4142135624]
[1.414213562370, 1.414213562384]
[1.4142135623729, 1.4142135623733]
...
```

Note that these streams of increasingly precise approximations can be used to provide the arbitrary-precision interface where one asks for a precision tolerance and gets a result. Each interval $[\underline{x}, \overline{x}]$, where $\underline{x} \in \{-\infty\} \cup \mathbb{D}, \overline{x} \in \mathbb{D} \cup \{\infty\}$, has either infinite or dyadic-rational ($\mathbb{D} = \{k/2^n \mid k \in \mathbb{Z}, n \in \mathbb{N}\}$) endpoints and represents partial information about `sqrt 2`: the first component represents a rational lower bound (with $-\infty$ being a vacuous bound) and the second an upper bound (with ∞ vacuous). λ_C is *sound* in the sense that these bounds are guaranteed to hold of the true value. Two closed terms of $\mathfrak{R}$ in λ_C are considered equivalent if their streams always overlap, even if the streams are not identical. For instance, $(\texttt{sqrt 2})^2 = 2$:

```
> (sqrt 2)²

[1.9999999986, 2.0000000009]
[1.999999999985, 2.000000000058]
[1.9999999999991, 2.0000000000009]
...
```

```
> 2

[2.0000000000, 2.0000000000]
[2.000000000000, 2.000000000000]
[2.0000000000000, 2.0000000000000]
...
```

The equivalence means that one can substitute $(\texttt{sqrt 2})^2$ for 2 within any program without affecting its meaning. In contrast, the floating-point computation for many languages and CPUs returns 2.0000000000000004, which is not 2 and does not itself indicate a larger range of possible values that includes 2, and would not validate the equation `(sqrt 2)^2` $= 2$.

First-order functions in λ_C are stream transformers of their approximations. For instance, applying the squaring function $(\texttt{-})^2 : \mathfrak{R} \rightarrow \mathfrak{R}$ to `sqrt 2` yields the following result:

```
> sqrt 2 : ℜ

[1.4142135619, 1.4142135624]
[1.414213562370, 1.414213562384]
[1.4142135623729, 1.4142135623733]
...
```

```
> (sqrt 2)^2 : ℜ

[1.9999999986, 2.0000000009]
[1.999999999985, 2.000000000058]
[1.9999999999991, 2.0000000000009]
...
```

In this case, the squaring function squares each input interval to produce output intervals. The computation is continuous in the sense that the computation of each interval result of `(sqrt 2)`^{2} needs only an interval approximation of `sqrt 2`. First-order functions such as `(-)`^{2} are *continuous maps*, meaning that in order to approximate the output to any finite level of precision, it suffices to inspect the input to only a finite level of precision.

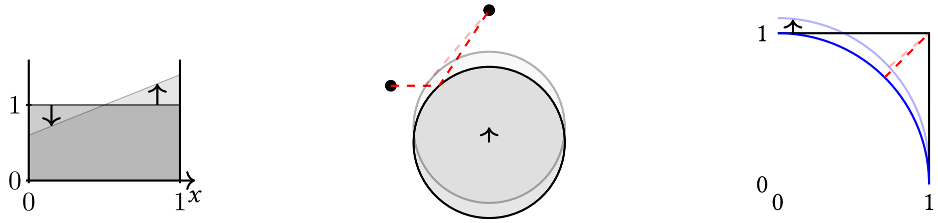
Implementing Higher-Order Primitives. The value-level definitions of higher-order primitives in λ_S are expressed in terms of mathematical operations for integration, optimization, and root finding. It's not obvious that these are computable. However, MarshallB [Sherman et al. 2019] demonstrates how to endow a language with computable implementations of Riemannian integration, maximization over compact sets, as well as a *Dedekind cut* primitive that is essentially equivalent to the root finding of `cutRoot` and can be used to implement the root finding of `firstRoot`. We were able to implement these MarshallB primitives in λ_C and use them to implement the higher-order primitives in λ_S.

Haskell Implementation. We implemented λ_S as an embedded language within Haskell. Because $\mathbb{R}^n$ and $\mathcal{R}^n$ are representable within **CTop**, we actually implement **AD** directly using **CTop** within Haskell, rather than working internally to λ_C. We implement **CTop** using an interval-arithmetic library that in turn uses MPFR [Fousse et al. 2007], a library for multi-precision floating-point arithmetic. We include this implementation and all the code examples as supplementary material, and will make it publicly available. See the readme file for more information about the code.

Computability and Numerical Soundness. The semantics for λ_S supports a realistic machine model for computing real-valued results to arbitrary precision. This is in contrast to semantics that permit Boolean-valued comparison of real numbers, and computational models like Real RAM, in which a machine can compare real numbers in constant time. When algorithms are designed based on such models but implemented with floating-point arithmetic, those implementations may fail to be robust to floating-point error (e.g., [Kettner et al. 2008]). In contrast, the continuity inherent in λ_S's semantics provides a robustness guarantee: arbitrary-precision approximations of the output can be produced by inspecting only finite-precision approximations of the input.

7 HIGHER-ORDER DATATYPES AND LIBRARIES

This section demonstrates the unique expressivity and computability of λ_S. We use the novel higher-order primitives available in λ_S – including integration, optimization, and root-finding – to build libraries for constructing and computing with three different higher-order datatypes: probability distributions (and measures), implicit surfaces, and generalized parametric surfaces. Since these libraries are implemented in λ_S, they are differentiable (arbitrarily many times). For each library, we compute an example differentiation task. Fig. 7 shows a high-level overview of each example. We now detail the implementation of each of the libraries and provide the implementations for each of the corresponding examples.



(a) *Probability distributions*: How does the mean and variance of the uniform distribution change as you weight its mass to tilt more towards higher values and away from lower values?

(b) *Implicit surfaces*: A ray of light from a source above bounces off a circle before hitting a camera. How does the brightness change when the circle is moved up?

(c) *Generalized parametric surfaces*: How does the Hausdorff distance between the quarter circle and the "L" shape change as the quarter circle is moved up?

Fig. 7. Three example differentiation problems we will express and compute with libraries in λ_S.

7.1 Probability Distributions (and Measures)

Probability is central to many machine-learning applications. Loss functions for Bayesian neural networks, GANs, etc. involve expectations over probability distributions. However, no previous work on the semantics of AD supports probability distributions[3]. The interaction between probabilistic choice and differentiation is nontrivial, and the lack of a semantic treatment of their interaction has real consequences for machine-learning practitioners using AD libraries who seek to combine them. Practitioners often use Monte Carlo sampling to approximate expectations, but because derivatives cannot be propagated through the samplers in common frameworks such as PyTorch and TensorFlow, code that *looks* correct and produces appropriate approximations of its value-level output can end up producing incorrect derivatives when AD is applied (as mentioned in the introduction). This common pitfall, which can be difficult to detect, necessitates *the reparameterization trick*, where code is rewritten such that samplers do not depend on any parameters that are to be differentiated.

λ_S can represent a monad of probability distributions $\mathcal{P}$, making it the first language semantics to support differentiation through probabilistic choice, including through distributions such as the uniform distribution on the unit interval. Supporting probability distributions is hard because they must involve higher-order functions: expectations are higher-order functions $\mathcal{P}(A) \times (A \to \mathbb{R}) \to \mathbb{R}$, as is the monadic bind operator $\mathcal{P}(A) \times (A \to \mathcal{P}(B)) \to \mathcal{P}(B)$ that supports compositional construction of complex probability distributions from simple ones.

A λ_S Library for Probability Distributions and Measures. Probability distributions, measures, and distributions (in the sense of generalized functions) can all be described as integrals,

```
type Integral A = (A → ℜ) → ℜ,
```

detailed in Fig. 8. Integrals are functions $i : (A \to \mathbb{R}) \to \mathbb{R}$ which are linear in their arguments. Measures are those integrals i satisfying $i(f) \geq 0$ whenever $f(x) \geq 0$ for all $x \in \mathbb{R}$. Probability distributions are those measures i satisfying $i(\lambda x.\ 1) = 1$; the integral for a probability distribution computes the expectation of a real-valued function under that distribution.

[3] While other works can represent expectations over distributions with finite support as sums, this would not work for distributions with infinite support. Loss functions frequently involve expectations over distributions with infinite support.

```
type Integral A = (A → ℜ) → ℜ

let dirac A (x : A) : Integral A = λ f : A → ℜ ⇒ f x
let bind A B (x : Integral A) (f : A → Integral B) : Integral B
  = λ k : B → ℜ ⇒ x (λ a : A ⇒ f a k)
let zero A : Integral A = λ f : A → Real ⇒ 0
let add A (x y : Integral A) : Integral A = λ f : A → ℜ ⇒ x f + y f
let map A B (f : A → B) (e : Integral A) : Integral B =
  λ k : B → ℜ ⇒ e (λ x : A ⇒ k (f x))
let factor (x : ℜ) : Integral unit = λ f : unit → ℜ ⇒ f () * x
let measToProb A (e : Integral A) : Integral A = λ f : A → ℜ ⇒ e f / e (λ x : A ⇒ 1)
let bernoulli (p : ℜ) : Integral 𝔅 = λ f : 𝔅 → ℜ ⇒ p * f tt + (1 - p) * f ff
let uniform : Integral ℜ = integral01

let total_mass A (mu : Integral A) = mu (λ x : A ⇒ 1)
let mean (mu : Integral ℜ) = mu (λ x : ℜ ⇒ x)
let variance (mu : Integral ℜ) = mu (λ x : ℜ ⇒ (x - mean mu)²)
```

Fig. 8. Integrals and λ_S programs that manipulate them.

Example. What happens if we make an infinitesimal perturbation to the uniform distribution as in Fig. 7a? How will its mean and variance change? Differentiation answers these questions.

The uniform distribution over the interval $[0, 1]$ is equivalent to the integral of $[0, 1]$, namely `uniform : Integral ℜ = integral01`. It satisfies $\int_0^1 1dx = 1$ (as any probability distribution must), and has mean $\int_0^1 xdx = 1/2$ and variance $\int_0^1 (x - 1/2)^2 dx = 1/12$.

Next, we must craft a perturbation to consider. There is an isomorphism `Tan (Integral A)` ≅ `Integral A * Integral A`, which says that a perturbation to an integral itself has the form of an integral as well. Hence, our perturbation must also be an integral. In addition, because we are perturbing a probability distribution, whose total mass must sum to 1, the total mass of our perturbation must be 0: if we are to increase mass somewhere, we must decrease it elsewhere. Given these design considerations, consider the following perturbation to the uniform distribution that makes 1 more likely, 0 less likely, 1/2 equally likely as before, and interpolates between these:[4]

```
let change : Integral ℜ = λ f : ℜ → ℜ ⇒ integral01 (λ x : ℜ ⇒ (x - 1/2) * f x)
```

The perturbation is an integral with total mass 0: $\int_0^1 (x - 1/2)dx = 0$.

Returning to our question of how this perturbation changes the mean and variance of `uniform`, for convenience let `der : (Integral A → ℜ) → Integral A → Integral A → ℜ` compute the derivative of its argument at a point and infinitesimal perturbation, using the appropriate coercions and projections to and from tangent spaces.[5] Since `mean` is linear, its derivative is independent of the current value and is just the original `mean` function applied to the infinitesimal perturbation:

```
der mean uniform change
  = mean change
  = integral01 (λ x : ℜ ⇒ (x - 1/2) * x)
  = 1/12
```

And indeed, that's what we compute:

[4] Fig. 7a shows a schematic of this perturbation.

[5] `let der f x dx = snd (tangetTo_R.to (tangent f (tangetTo_R.from (x, dx))))`

```
type Surface A = A → ℜ

let circle (c : ℜ²) (r : ℜ) : Surface (ℜ²) =
  λ x : ℜ² ⇒ r² - (x[0] - c[0])² - (x[1] - c[1])²
let halfplane A (normal : ℜ²) : Surface (ℜ²) = λ x : ℜ² ⇒ dot normal x
let union A (s s' : Surface A) : Surface A = λ x : A ⇒ max (s x) (s' x)
let intersection A (s s' : Surface A) : Surface A =  λ x : A ⇒ min (s x) (s' x)
let complement A (s : Surface A) : Surface A = λ x : A ⇒ - (s x)
```

Fig. 9. A λ_S library for implicit surfaces.

```
eps=1e-3> der mean uniform change
[0.0829, 0.0837]
```

However, `variance` is nonlinear, so its derivative does depend on the current point. Let's compute it and then reason about the answer:

```
eps=1e-2> der variance uniform change
[-0.005, 0.004]
```

We can reason about the change in the variance with the laws about derivatives, just as we would in first-order cases:

```
der variance uniform change
  = der (λ mu : Integral ℜ ⇒ mu (λ x : ℜ ⇒ x²) - (mean mu)²) uniform change
  = change (λ x : ℜ ⇒ x²) - 2 * mean uniform * mean change
  = integral01 (λ x : ℜ ⇒ (x-1/2)*x²) - 2 * 1/2 * 1/12
  = 1/12 - 1/12
  = 0
```

So it turns out that this infinitesimal perturbation will actually not change the variance.

7.2 Implicit Surfaces and Root-Finding

§2 and Fig. 1 presented a library for implicit surfaces and a function for performing ray tracing on scenes represented by implicit surfaces.

Fig. 9 presents a library for constructing implicit surfaces. An *implicit surface* is a representation of a surface (such as a sphere or plane) with the zero-set of a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ (where usually we consider $n = 3$ for 3-dimensional space). Whether $f(x, y)$ is positive, negative, or zero indicates whether (x, y) is inside, outside, or on the border of the surface, respectively. The angle at which a ray deflects is determined by the *surface normal* at the location where the ray hits the surface, which is the vector that is orthogonal to the plane that is tangent to the surface.

In λ_S, we can represent implicit surfaces as `type Surface A = A → ℜ`. Fig. 9 presents a small library for constructing implicit surfaces. The Boolean operations of Constructive Solid Geometry (CSG) – union, intersection, and complement – are available for these implicit surfaces. Because λ_S permits nonsmooth functions, it is able to represent implicit surfaces that don't necessarily correspond to manifolds, such as the union of two spheres that are offset and equally sized. Where they touch, there is a corner, and thus there is no (unique) surface normal.

Our smooth ray tracer, shown in Fig. 1c, renders the image of an implicit surface with a single light source and a Lambertian reflectance model, computing the angle at which light reflects off of the surface using automatic differentiation. The code in Fig. 1c reflects the contributions of Niemeyer et al. [2020], who use a differentiable ray-tracing renderer to learn implicit 3D representations of surfaces, noting their "key insight is that depth gradients can be derived analytically using the concept of implicit differentiation."

We can implement a smooth (and thus differentiable) ray tracer for implicit surfaces in λ_S in just a few lines of code, and the use of implicit differentiation automatically falls out.

7.3 Generalized Parametric Surfaces and Optimization

We now build a library within λ_S for constructing shapes and computing operations on them. For instance, we can represent the quarter disk and unit square in Fig. 7c as shapes and compute the Hausdorff distance between them, which equals $\sqrt{2} - 1$, as:

```
eps=1e-3> hausdorffDist R2Dist lShape (quarterCircle 0)
[0.4138, 0.4145]
```

We can also compute derivatives, such as the infinitesimal perturbation in the Hausdorff distance that would result if the quarter circle were to infinitesimally move up by a unit magnitude:

```
eps=1e-1> deriv (λ y : ℜ ⇒ hausdorffDist R2Dist lShape (quarterCircle y))
0
[-0.752, -0.664]
```

This application is admittedly more speculative in its practical applications, but it demonstrates a novel domain in which we can define and compute derivatives. We will now explain how this library for shapes works.

We represent these generalized parametric surfaces as *maximizers*, represented in λ_S as

```
type Maximizer A = (A → ℜ) → ℜ.
```

Maximizers are functions $F : (A \to \mathbb{R}) \to \mathbb{R}$ that satisfy the algebraic laws $F(\lambda x : A.\ k) = k$ for all $k \in \mathbb{R}$ and $F(\lambda x : A.\ \max(f(x), g(x))) = \max(F(f), F(g))$ (analogously to how integrals are functions that satisfy the algebraic laws of linearity). A generalized parametric surface `k : Maximizer A`, when applied to a function `f : A → ℜ`, returns the maximum value that `f` attains on the region represented by `k`.

Fig. 10 shows an excerpt of the library for generalized parametric surfaces. Note that generalized parametric surfaces shapes form a monad (representing nondeterminism), with `point` and `indexedUnion` as *return* and *bind*, yielding a programming model for constructing shapes.

Returning to the earlier Hausdorff-distance example, note that the maximal distance on the "L" shape occurs at the corner point, which is represented twice, as the endpoint of each line; thus, a maximum is taken over two equal distances. In [Abadi and Plotkin 2020], because the maximum operator is defined with a partial conditional statement, the result — not to mention the derivative — would be undefined. Because both the values and derivatives are the same for the two representations of this corner point, the derivative is a maximal element. Also note that we need second derivatives to compute the derivative of the Hausdorff distance, due to the use of `max01`.

8 DISCUSSION

In this section, we discuss the capability of λ_S to represent control flow as well as the opportunity to soundly speed up execution of higher-order primitives using derivative information.

```
type Maximizer A = (A → ℜ) → ℜ
let point A (x : A) : Maximizer A = λ f : A → ℜ ⇒ f x
let indexedUnion A B (ka : Maximizer A) (kb : A → Maximizer B) : Maximizer B =
  λ f : B → ℜ ⇒ ka (λ a : A ⇒ kb a f)
let union A (k1 k2 : Maximizer A) : Maximizer A =
  λ f : A → ℜ ⇒ max (k1 f) (k2 f)
let map A B (g : A → B) (k : Maximizer A) : Maximizer B =
  λ f : B → ℜ ⇒ k (λ a : ℜ ⇒ f (g a))
let sup A (k : Maximizer A) (f : A → ℜ) : ℜ = k f
let inf A (k : Maximizer A) (f : A → ℜ) : ℜ = - k (λ x : A ⇒ - (f x))
let hausdorffDist A (d : A → A → ℜ) (k1 k2 : Maximizer A) : ℜ =
  max (sup k1 (λ x1 : A ⇒ inf k2 (λ x2 : A ⇒ d x1 x2)))
      (sup k2 (λ x2 : A ⇒ inf k1 (λ x1 : A ⇒ d x1 x2)))

let unitInterval : Maximizer ℜ = max01
let quarterCircle (y : ℜ) : Maximizer (ℜ^2) = map
  (λ theta : ℜ ⇒ (cos (pi / 2 * theta), sin (pi / 2 * theta) + y))
  unitInterval
let lShape : Maximizer (ℜ^2) =
  union (map (λ x : ℜ → (x, 1)) unitInterval)
        (map (λ y : ℜ ⇒ (1, y)) unitInterval)
let R2Dist (a b : ℜ^2) : ℜ = sqrt ((a[0] - b[0])^2 + (a[1] - b[1])^2)
```

Fig. 10. Generalized parametric surfaces and λ_S programs that manipulate them.

8.1 Control Flow: Conditionals and Recursion

λ_S supports discrete spaces, including in particular the Booleans $\mathbb{B}$ and any well-founded set (such as the natural numbers). The recursion principles for these yield, respectively, if-then-else expressions and well-founded recursion. These control-flow expressions must be independent of "continuous data": all maps from *connected* spaces to discrete spaces are constant. This property defines connected spaces. Connected spaces include all vector spaces, such as $\mathbb{R}^n$. Di Gianantonio and Edalat [2013] explain some particular issues that demonstrate why implementing piecewise-differentiable functions with branching is problematic.

8.2 Optimizing Higher-Order Primitives with Derivative Information

We can also use the fact that functions in λ_S come equipped with all their derivatives to opportunistically speed up some operations. For instance, consider applying `cut_root` to some function f. Its value-level definition naturally maps to a bisection-like algorithm on the values of f. However, since we have access to $f^{(1)}$, we can use a variation of Newton's method generalized to interval arithmetic to speed up the convergence drastically, and indeed we do this in our implementation. Note that we are guaranteed that this optimization is sound, because consistency of differentiation ensures that $f^{(1)}$ appropriately reflects $f^{(0)}$. It may be the case that $f^{(1)}$ returns $\perp$ at some points, or even everywhere, in which case the algorithm falls back on bisection to ensure progress.

Similarly, the literal interpretation of the value-level definition of Riemannian integration in §5 maps to a quadrature method that uses only the values $f^{(0)}$ of f. However, the availability of higher derivatives of f makes it possible to use interval-based versions of higher-order integration methods, which can also drastically speed up the convergence. We do not use these higher-order methods by default in our actual implementation.

Table 1. Summary of other approaches to semantics of differentiable programming and their properties. *Higher-order derivatives*: The differentiation operator can be iterated arbitrarily many times (when applied to smooth functions). *Higher-order functions*: A concrete test: is twice$(f : \mathbb{R} \to \mathbb{R})(x : \mathbb{R}) : \mathbb{R} \triangleq f(f(x))$ admitted? *Non-differentiable functions*: Some nondifferentiable functions are admitted. A concrete test: is max : $\mathbb{R}^2 \to \mathbb{R}$ admitted? "*Clarke derivative*" indicates that locally Lipschitz functions support derivatives in the sense of Clarke derivatives or L-derivatives [Edalat and Lieutier 2004], whereas "*partiality*" indicates that nondifferentiable maps are supported by considering them to be partial at their discontinuities.

	higher-order functions	higher-order derivatives	nondifferentiable functions
Vákár et al. [2018]	✓	✓	✗
Di Gianantonio and Edalat [2013]	✓	✗	✓ (Clarke derivative)
Elliott [2018]	✗	✗	✗
Abadi and Plotkin [2020]	✗	✓	✓ (partiality)
Sigal [2018]	✗	✓	✓ (partiality)
Vytiniotis et al. [2019]	✓	✗	✗
Huot et al. [2020]	✓	✓	✗
Ehrhard and Regnier [2003]	✓	✓	✗
λ_S (this work)	✓	✓	✓ (Clarke derivative)

9 RELATED WORK

Table 1 illustrates the unique set of features that λ_S provides and their relationship to other approaches to AD semantics. We note that these other approaches also have features that λ_S lacks.

Di Gianantonio and Edalat [2013] describe a programming language for nonexpansive (i.e., Lipschitz constant 1) functions on the interval $[-1, 1]$ with a differentiation operator that applies to functions from $[-1, 1]$ to $[-1, 1]$. The semantics of this differentiation operator are that of the L-derivative [Edalat 2008; Edalat and Lieutier 2004], which is closely related to the Clarke Jacobian definition we use. Their domain-theoretic account ensures computability: in theory, results can be computed to arbitrary precision. Their semantics is fundamentally limited to first-order derivatives: their interval type denotes $[-1, 1] \times [-1, 1]$, corresponding to a *dual-number* representation, baking in that limited capability. It is unclear how that representation could be generalized directly to permit higher-order differentiation and appropriately handle nested differentiation (without the *perturbation confusion* [Siskind and Pearlmutter 2005] that may arise with nested differentiation).

Elliott [2008] presents a data type for representing smooth maps, where a smooth map f is represented by the collection of its kth derivatives for all k. Elliott [2008] defines the derivatives of some arithmetic functions as well as some categorical operations, though the definition of composition of smooth maps is incorrect. We support higher-order derivatives by adapting this representation for the Clarke derivative.

Vákár et al. [2018] presents the semantics of a differentiable programming language that supports higher-order functions and higher-order derivatives using the quasitopos of diffeological spaces. As a quasitopos, the semantics supports higher-order functions and quotient types. Vákár et al. [2018] show an internal derivative operator that can be applied to any function of any type, and thus can be applied repeatedly for higher-order derivatives. We based our internal derivative operator on theirs. Functions such as `max` that are not smooth are not admissible. It is not made clear how one could implement a differentiable programming language supporting the expressive possibilities suggested by the semantics.

None of the works in Table 1 describe higher-order functions for root-finding, optimization, or integration, nor do they describe datatypes for implicit surfaces, compact shapes, or probability distributions. Edalat and Lieutier [2004] describe an integration operator in a domain-theoretic framework for differential calculus, but it does not handle higher-order derivatives. Sherman et al. [2019] describe computable higher-order functions and libraries for root-finding, optimization, and integration, but does not admit differentiation of any sort.

We follow Sherman et al. [2019] in our approach to computability. We are unaware of any system that computes arbitrary-precision derivatives (given the definition of the function) in any capacity.

10 CONCLUSION

This paper demonstrates how to compute and make sense of derivatives of higher-order functions, such as integration, optimization, and root-finding and at higher-order types, such as probability distributions, implicit surfaces, and generalized parametric surfaces. Our libraries and case studies model existing differentiable algorithms, for instance, a differentiable ray tracer for implicit surfaces, without requiring any user-level differentiation code, in addition to demonstrating new differentiable algorithms, such as computing derivatives of the Hausdorff distance of generalized parametric surfaces. Ideally, the ideas λ_S demonstrates may enable differentiable programming frameworks to support the new abstractions and expressivity suggested by this paper.

ACKNOWLEDGMENTS

We thank Eric Atkinson, Tej Chajed, Alexander Lew, Alex Renda, and David Spivak, as well as the anonymous reviewers, for their helpful feedback and discussions. This work was supported in part by the Office of Naval Research (ONR-N00014-17-1-2699). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the Office of Naval Research.

REFERENCES

Martín Abadi and Gordon D. Plotkin. A simple differentiable programming language. In *Principles of Programming Languages*, 2020.

Matan Atzmon, Niv Haim, Lior Yariv, Ofer Israelov, Haggai Maron, and Yaron Lipman. Controlling neural level sets. In *Advances in Neural Information Processing Systems*. 2019.

Shaojie Bai, J Zico Kolter, and Vladlen Koltun. Deep equilibrium models. In *Advances in Neural Information Processing Systems*, 2019.

J Daniel Christensen and Enxin Wu. Tangent spaces and tangent bundles for diffeological spaces. *American Mathematical Society*, 2017.

Frank H Clarke. *Optimization and nonsmooth analysis*. 1990.

Pietro Di Gianantonio and Abbas Edalat. A language for differentiable functions. In *Foundations of Software Science and Computational Structures*, 2013.

Abbas Edalat. A continuous derivative for real-valued functions. In *New Computational Paradigms*. 2008.

Abbas Edalat and André Lieutier. Domain theory and differential calculus (functions of one variable). *Mathematical Structures in Computer Science*, 14(6), 2004.

Thomas Ehrhard and Laurent Regnier. The differential lambda-calculus. *Theoretical Computer Science*, 309(1-3), 2003.

Conal Elliott. Higher-dimensional, higher-order derivatives, functionally. 2008. URL http://conal.net/blog/posts/higher-dimensional-higher-order-derivatives-functionally.

Conal Elliott. The simple essence of automatic differentiation. In *International Conference on Functional Programming*, 2018.

Mikhail Figurnov, Shakir Mohamed, and Andriy Mnih. Implicit reparameterization gradients. In *Advances in Neural Information Processing Systems*. 2018.

Michael P Fourman. Continuous truth I: Non-constructive objects. *Studies in Logic and the Foundations of Mathematics*, 112, 1984.

Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2), 2007.

Mathieu Huot, Sam Staton, and Matthijs Vákár. Correctness of automatic differentiation via diffeologies and categorical gluing. In *Foundations of Software Science and Computation Structures*, 2020.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *International Conference on Learning Representations*, 2017.

Martin Jankowiak and Fritz Obermeyer. Pathwise derivatives beyond the reparameterization trick. In *International Conference on Machine Learning*, 2018.

Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee Yap. Classroom examples of robustness problems in geometric computations. *Computational Geometry*, 40(1), 2008.

Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable Monte Carlo ray tracing through edge sampling. In *Special Interest Group on Computer Graphics and Interactive Techniques*, 2018.

Jonathan Lorraine, Paul Vicol, and David Duvenaud. Optimizing millions of hyperparameters by implicit differentiation. *arXiv preprint arXiv:1911.02590*, 2019.

Christian A. Naesseth, Francisco J. R. Ruiz, Scott W. Linderman, and David M. Blei. Reparameterization gradients through acceptance-rejection sampling algorithms. In *Artificial Intelligence and Statistics*, 2017.

Michael Niemeyer, Lars Mescheder, Michael Oechsle, and Andreas Geiger. Differentiable volumetric rendering: Learning implicit 3D representations without 3D supervision. In *Computer Vision and Pattern Recognition*, 2020.

Benjamin Sherman, Jesse Michel, and Michael Carbin. Sound and robust solid modeling via exact real arithmetic and continuity. In *International Conference on Functional Programming*, 2019.

Jesse Sigal. Denotational semantics for differentiable programming with manifolds. In *Student Research Competition at the Internation Conference on Functional Programming*, 2018.

Jeffrey Mark Siskind and Barak A Pearlmutter. Perturbation confusion and referential transparency: Correct functional implementation of forward-mode ad. *Workshop on Implementation and Application of Functional Languages*, 2005.

Dimitrios Vytiniotis, Dan Belov, Richard Wei, Gordon Plotkin, and Martin Abadi. The differentiable curry. In *Neural Information Processing Systems Workshop Program Transformations*, 2019.

Matthijs Vákár, Ohad Kammar, and Sam Staton. Diffeological spaces and semantics for differential programming. In *Domains*, 2018. URL https://andrejbauer.github.io/domains-floc-2018/slides/Matthijs-Kammar-Staton.pdf.

Yuanhao Wang, Guodong Zhang, and Jimmy Ba. On solving minimax optimization locally: A follow-the-ridge approach. In *International Conference on Learning Representations*, 2020.