Structural Foundations for Differentiable Programming

Mathieu Huot

St Cross College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

October 2022

Thesis

This dissertation supports the following thesis:

> Differentiable programming introduces new foundational challenges. First, it is subject to a classic expressivity versus efficiency tradeoff. Second, there is a gap between the mathematical theories underpinning learning and optimization on the one hand, and their actual implementations on the other hand. These challenges suggest developing mathematical tools better tailored for what programs actually compute and which properties they satisfy. We argue that category theory as a backbone for denotational semantics, and other standard techniques from programming language theory, can give a solid structural foundation to differentiable programming, as needed to support the rapid growth of AI and related fields over the past few years.

Declaration

This dissertation and research are a product of my original work, except where indicated otherwise by explicit statement or reference. All ideas, quotations, and data originating from the work of others, published or otherwise, are fully acknowledged according to standard practices of the academic discipline.

Copyright

Lay Abstract

Increasingly, computer software is used to manipulate continuous values, in domains such as machine learning, probabilistic programming, and scientific computing. At the same time, it is more and more common for such programs to be integrated into safety-critical systems. Usually, these algorithms are analysed within a known mathematical framework, e.g. convex differentiable functions, and call subroutines computing programs satisfying strong mathematical properties, e.g. computing exact derivatives. On top of the errors introduced by finite-precision real-number arithmetic, such as floating point arithmetic, programming with continuous values extends far beyond real arithmetic, involving objects such as probability distributions, programs with unbounded loops, approximate solvers, functions taking functions as arguments or returning functions, and more. What's more, these expressive programs may be implementations of differentiable functions for which we wish to compute a derivative, expressed as a program, and obtained automatically. This paradigm of automatic computation of differentiable functions expressed via highly expressive programs has been called *differentiable programming*.

My thesis lays semantic foundations for differentiable programming. Programs are interpreted as mathematical functions that go beyond the simple setting of real-valued differentiable functions. This interpretation is compositional, which means that the sequencing of programs is interpreted by function composition, and this enables reducing the complexity of studying all programs to finitely many basic building blocks. The key operation of interest here, differentiation, can then be studied on programs by the way it changes the basic blocks. In this dissertation, I develop new expressive mathematical models for differentiable programming, using tools from category theory, differential geometry, and programming language theory. They enable the study of correctness and main properties of the fundamental algorithm underpinning differentiable programming, so-called *automatic differentiation* (AD).

Abstract

This dissertation supports the broader thesis that categorical semantics is a powerful tool to study and design programming languages. It focuses on the foundational aspects of differentiable programming in a simply typed functional setting. Although most of the category theory used can be boiled down to a more elementary presentation, its influence was certainly key in obtaining the results presented in this dissertation. The conciseness of certain proofs and the compactness of certain definitions and insights were made easier thanks to my background in category theory.

Backpropagation is the key algorithm that allows fast learning on neural networks. It enabled some of the impressive recent advancements in machine learning. With models of increasing complexity, data structures equally complex are required, which calls for the ability to go beyond standard differentiability. This emerging generalization was coined as differentiable programming. The idea is to allow users to write expressive programs representing (a generalization of) differentiable functions, whose gradient computation can be automated using automatic differentiation. In this dissertation, I lay some foundations for differentiable programming. This is done in three ways.

Firstly, I present a simple higher-order functional language and define automatic differentiation as a structure-preserving program transformation. The language is given a denotational semantics using diffeological spaces, and it is shown that the transformation is correct, i.e. that AD produces programs that do compute gradients of the original programs, using a logical relations argument.

Secondly, I extend the language from the previously described chapter to introduce new expressive program constructs such as conditionals and recursion. In such a setting, even first-order programs may represent functions that need not be differentiable. I introduce better-fitted denotational semantics for such a language and show how to extend AD to such a setting and what guarantees about AD now hold. This extended language models the more realistic needs in expressiveness that can be found in the literature, e.g. in modern probabilistic programming languages.

Thirdly, I present detailed applications of the developed theory. I first show a general recipe for extending AD to non-trivial new types and new primitives. I then show how the guarantees about AD are sufficient for usage in certain applications,

such as the change of variable formula of stochastic gradient descent, but how it may not be sufficient, for instance, in simple gradient descent. Finally, more applications in the specific context of probabilistic programming are explored. First, a denotational proof that the trace semantics of a probabilistic program is almost everywhere differentiable is given. Second, a characterization of posterior distributions of probabilistic programs valued in Euclidean spaces is obtained: they have densities with respect to (w.r.t.) some sum-of-Hausdorff measure on a countable union of smooth manifolds.

Overall, these contributions give us better insights into differentiable programming. They form a foundational setting to study the differentiability-like properties of realistic complex programs, beyond usual settings such as differentiability or convexity. They give general recipes to prove some properties of such programs and modularly extend automatic differentiation to richer contexts with new types and primitives.

Acknowledgements

I have been lucky enough to be surrounded by great people at all times throughout my education, more than I would usually acknowledge, and this section is my attempt to partially amend this.

Firstly, I am grateful to my supervisor Sam Staton, for giving me countless pieces of advice and always staying positive and supportive. He is a great supervisor that pushed me intellectually and taught me a good deal of category theory while leaving me enough freedom to pursue the research I wanted to, and he let me develop my own take on research. Thank you, Sam, for having high research standards and always being curious and open-minded about the different research topics I came up with.

Secondly, I want to thank Andrzej Murawski and Neel Krishnaswami for examining this thesis and providing valuable feedback that greatly improved it.

Thirdly, I want to express my gratitude towards Ohad Kammar and Marcelo Fiore for supervising me during a research internship in Cambridge. This led me to do another research internship with Sam, which then led me to have Sam as a PhD supervisor, which eventually led to this dissertation. You taught me a lot about mathematics when I barely knew what a category was, I learned a lot about research under your guidance. I am also particularly thankful to Ohad for being a great listener, and helping me the best he could when I needed it.

I am grateful to some memorable teachers I had in high school, who taught me something important one way or another. They left a mark and became memorable to me, helping me climb part of the infinitely high mountain that is science. So thank you, Jean-Paul Heim, Olivier Bourre, Pascal Gelfi, Jean Goubault-Larrecq, Xavier Leroy, and Emmanuel Haucourt.

I am grateful to my scientific collaborators. You gave me the confidence that I can help do great research but also entrenched my belief that great research is a collaborative effort. Communication at the research level (and in general) between people is hard but worth it. I particularly want to thank Alex Lew, Amir Shaikhha, and Matthijs Vákár. Matthijs, you made me discover differentiable programming and helped me progress a lot during our collaboration. Alex, I am especially thankful to you for showing me some of the beauty of your ideas that I think are really worth pushing, and I hope we will continue our collaboration. Amir, thank you for listening to me bringing sometimes weird mathematical ideas into our collaborations

and keeping an open mind, and always being nice to be around in general.

I am thankful to our research group at Oxford. I think this is a rare opportunity to be around so many great people and great researchers. In particular thank you, Swaraj Dash, Carol Mak, Cristina Matache, Sean Moss, Hugo Paquet, Paolo Perrone, Philip Saville, Jesse Sigal, Sam Speight, Dario Stein, Ned Summers, Dominik Wagner, Fabian Zaiser.

I am grateful to RelationalAI for allowing me to do an internship with them. I met great and passionate people, and I learned a lot in the query optimizer team. Thank you in particular Hung Ngo, Mahmoud Abo-Khamis, David Sanders, Huda Nassar, and Ryan Curtin for the great work and discussions. I also want to thank Nikolas Göbel, Nathan Dely, Marton Bur, and Remy Wang for the fun discussions. Also thank you Molham Aref for giving me this opportunity.

I am thankful for the École Normale Supérieure Paris-Saclay, and in particular to the department of Computer Science: I really had a great time when I was there.

I am in debt to my amazing friends for supporting my bad jokes and many other flaws, you are the best! Thank you, Emmanuel Arrighi, William Babonnaud, Clément Beauseigneur, Louis Cohen, Leo Colisson, Robin Dall Armellina, Thomas Dupriez, Alexis Ghyselen, Luca Grillotti, Guillaume Hocquet, Tristan Jadoul, Younesse Kaddar, Alexis Laouar, Laurent Noël, Antonin Penon, Clément Ringeade, Thibault Vernier.

I also want to thank my CS class at ENS Paris-Saclay for being the nicest people. I want to thank Maria Gorinova and Petar Veličković for the great discussions. Thank you, Falk Schneider, for being a fantastic roommate during my time in Oxford. Furthermore, I want to acknowledge the great people I met on the way since I started my undergraduate studies, but whom I, unfortunately, have not been as much in touch with as I would have liked. These include Théo Auclair, Emile Auria, Pierre Becker, Alexandre Boyer, Thomas Blandin, Charles Bravetti, Lorenzo Carré, Alexandre Comyn, Clement Denis, Vianney Devavelaere, Joris Duguépéroux, Quentin Gazda, Thomas Guennoc, Raji Haddad, Lucas Jager, Aliaume Lopez, Théodore Lopez, Alexandre Montoriol, Hoël Plantec, Nicolas Shaeffer, Joan Thibault, Cheikh Touré, Etienne Toussaint, Julien Trevisan.

I am forever in debt to my girlfriend, Tiffany Luong. I also want to thank her family, who have always been charming and very welcoming to me.

Likewise, I am deeply grateful to my family, my parents and my sister, for always believing in me, helping me in all the ways you could for so many years without ever complaining, and always being present during tough times. You taught me great values that I will keep for the rest of my life, and I will always try to give back to the world some of what you gave and taught me.

I finally want to thank the people I might have missed and the reader.

Foreword

Journey

I first learned about category theory during the first year of my master's. I quickly became interested in the depth of the theory, and its deep connection to mathematics and computer science. Soon, this led me to try to do research on this topic during internships. I had the pleasure to be supervised by Ohad Kammar and Marcelo Fiore in Cambridge. They introduced me to Sam Staton, and I learned about probabilistic programming and the exciting recent developments in denotational semantics for higher-order probabilistic programs. I first joined Sam for a research internship during my last year at ENS Paris-Saclay, where I worked on quantum theory. This was a blast, but I really wanted to catch up on these fancy, novel new developments in the theory of probabilistic programming. Around the same time, I learned about differentiable programming from Matthijs Vákár. I immediately thought there seemed to be interesting research to be done for the programming language community on this topic. This seemed to be a perfect middle ground to later bridge the gap between the very exciting explosion of research in machine learning (AI more generally) and semantics for probabilistic programming. Indeed, probability and differentiability have always been intertwined, and both play a key role in some parts of machine learning and Bayesian inference. I think this is how the idea that eventually led to this thesis was born: conduct, with differentiable programming, a similar development to the one I saw with the denotational semantics of probabilistic programming, eventually trying to catch up with it and connect the two. As I said above, several applications fundamentally need to reason about both differentiability and probabilities. The outline of my dissertation, therefore, somewhat closely follows this idea. First, develop some framework for differentiable programming for programs including higher-order functions, which is the content of Chapter 2. It turns out that practice requires going beyond usual differentiability: functions can be partial, conditionals can break continuity (and therefore differentiability), and other programming features such as recursion are non-trivial to reason about. This is the exploration that now forms Chapter 3. Finally, I wanted to look at applications of the previously developed theory. The first question was whether the theoretical guarantees were enough to

ensure the sound usage of AD in other algorithms. A second question was the investigation of some links between differentiable and probabilistic programming. This now forms Chapter 4 of this dissertation.

Publications

Chapter 2 is mainly taken from the published work [Huot et al., 2020], and its extended journal version [Vákár, Staton, and Huot, 2022]. Chapter 3 and 4 are based on Huot et al. [2023b] which was recently accepted at LICS 2023, extending the workshop paper Lew, Huot, and Mansinghka [2021] (Chapter 3), and Lew, Huot, and Mansinghka [2021] and Huot et al. [2023a] (Chapter 4). During my DPhil in Oxford, I was also an author of the following papers that will not be the main focus of this dissertation: Huot and Staton [2019a], Huot and Staton [2019b], Huot and Shaikhha [2022] as well as Ghorbani, Huot, Hashemian, and Shaikhha [2022], Shaikhha, Huot, Ghasemirad, Fitzgibbon, Jones, and Vytiniotis [2022a], Lew, Huot, Staton, and Mansinghka [2023], Shaikhha, Huot, Smith, and Olteanu [2022b].

Prerequisites

Throughout the dissertation, familiarity with simply typed lambda calculus will be assumed, as well as standard extensions such as term-recursion and algebraic data types. Likewise, familiarity with programming language theory will be assumed, which includes proofs by logical relations, rewriting rules, operational and denotational semantics, soundness and adequacy of a semantics w.r.t. another, call-by-value evaluation. Basic knowledge of calculus and topology will also be assumed throughout the dissertation. Several proofs and sections utilize category theory and will assume familiarity with the basic theory, which includes functors and natural transformations, universal properties, presheaves, the Yoneda embedding, adjunctions, monads, limits and colimits, initial algebras for an endofunctor, categorical semantics of the simply-typed lambda-calculus in Cartesian closed categories. These parts will be indicated with a  symbol. Particular sections or points that will require or introduce more advanced category theory will be indicated with a  symbol. In particular, some familiarity with fibrations and sheaves will be helpful when reading Part II of Chapter 3. Chapter 3 will assume basic knowledge and familiarity with real analytic functions. In the same chapter, familiarity with the standard semantics of PCF in the category of complete partial orders and continuous functions will help. Finally, Chapter 4 will assume some knowledge of topology, measure theory and probability theory. In addition, albeit not strictly necessary, basic knowledge and a certain familiarity with differential geometry will be useful.

Contents

List of Figures

List of Abbreviations

AD . . . . . Automatic Differentiation

a.e . . . . . Almost everywhere

AST . . . . . Abstract Syntax Tree

cdf . . . . . Cumulative distribution function

det . . . . . Determinant

DSL . . . . . Domain Specific Language

epi . . . . . Epimorphism

FP . . . . . Functional Programming

GD . . . . . Gradient Descent

iff . . . . . If and only if

i.i.d . . . . . Independent identically distributed

IR . . . . . Intermediate Representation

iso . . . . . Isomorphism

LHS . . . . . Left-hand side

lubs . . . . . Least upper bounds

MCMC . . . . Markov Chain Monte Carlo

mono . . . . . Monomorphism

ODE . . . . . Ordinary differential equation

PAP . . . . . Piece-wise analytic under analytic partition

PDE . . . . . Partial differential equation

pdf . . . . . Probability density function

PL . . . . . Programming Language

PPL . . . . . Probabilistic Programming Language

RHS . . . . . Right-hand side

RNN . . . . . Recurrent neural network

SGD . . . . . Stochastic Gradient Descent

sqr . . . . . Square

sqrt . . . . . Square root

Notations

27, 103

Category Theory

$\cong$	Isomorphism
$\mathcal{Y}$	Yoneda Embedding
$\mathcal{J}$	Coverage
$\rightarrowtail$	Monomorphism
$\mathcal{M}$	Class of monomorphisms
$\widehat{\mathbf{C}}$	Category of presehaves on $\mathbf{C}$
1	Terminal object
0	Initial object
$Kl_T\mathbf{C}$	Kleisli category for the monad T on $\mathbf{C}$
$\mathbf{C}(A, B)$	Morphisms in $\mathbf{C}$ from A to B
$\mathbf{list}(A)$	List object on A
Diff	Diffeological spaces and smooth maps
CartSp	Cartesian spaces and smooth maps
OpenCont	Open spaces of $\mathbb{R}^n$ and continuous maps
Sbs	Standard Borel spaces and measurable maps
Set	Sets and functions
Pred	Subsets and subset-preserving functions
ω**CPO**	ω-complete partial orders and continous maps
$\omega Conc(\mathbf{C}, \mathcal{J})$	ω-concrete sheaves and natural transformations
CCC	Cartesian-closed category
bi-CCC	Cocartesian and Cartesian-closed category
Syn	Syntactic category for a language
ωPAP	ω-PAP spaces and smooth maps
$A \times B$	Cartesian product of A and B
$A + B$	Coproduct of A and B
A^B	Exponential object for morphisms from A to B
id_A	Identity morphism on A
$f; g$	Sequential composition

Differential Geometry

$\mathcal{C}^k$	Functions with continuous derivatives of order up to k
$\mathcal{C}^\infty$	Functions which are $\mathcal{C}^k$ for all k
H^k	k-Hausdorff measure
$J_x f$	Jacobian matrix of f at x
$\frac{\partial f}{\partial x}$	Partial derivative w.r.t. variable x

$\mathcal{T}$	Tangent bundle functor on manifolds

General Math

ι	injection
poset	Partially ordered set
$g \circ f$	Function composition
Im	Image
$\mathbb{N}$	Natural numbers
$\mathbb{B}$	Booleans
$\mathbb{R}$	Real numbers
Dom	Domain
$\lvert X \rvert$	Underlying set
$A \setminus B$	Set difference
$\bar{A}$	Smallest closed set containing A
$\emptyset$	Empty set

I97

Measure Theory

$\ll$	Absolute continuity
$f_*\mu$	Pushforward measure of μ along f
λ	Lebesgue measure
μ, ν	Distribution
ρ	Density

Programming languages

$\Gamma \vdash t : \tau$	Typing judgement for t in context Γ
$\lambda x.e$	Lambda Abstraction
$\gg\!=$	Monadic bind
$\mu f.e$	Recursive term
fold	Fold right operator
$\langle t_1, t_2 \rangle$	Pairing
$[\![t]\!]$	Denotational semantics of t

1 Introduction

1.1 The need for Differentiable Programming

The use of derivatives is ubiquitous in science. With the growing importance of computer science in the past sixty years, increasingly complex programs have been written as modelling and computational tools. From the early days of computer science, there has been a need to compute derivatives of functions expressed as programs. More recently, derivatives expressed as programs have been vastly popularized by their extensive application for optimization in the field of machine learning and AI, perhaps most spectacularly in the subfield of deep learning via the backpropagation algorithm.

As the problem of computing derivatives of functions expressed as programs is already quite old, a natural question is whether there is still a need to study it further. The first way to compute approximate derivatives of functions is known as the finite difference method. The idea is that the derivative $f'(x)$ of a differentiable function $f : \mathbb{R} \to \mathbb{R}$ at x can be approximated as

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

for a sufficiently small epsilon. While very simple to implement and fast to run, it has one major drawback: it is usually not very accurate. In addition, it can be quite tricky to make work in practical, complex cases. In brief, we usually use floating point arithmetic for doing fast computations with reals on computers. Two of the main most basic pieces of advice from the numerical computing community are the following:

- Do not add a tiny number to a number
- Do not subtract two similar numbers

We could add a third: beware of overflow, so in particular beware when dividing by a tiny number. Now let us look back at what the finite difference method does: it computes $x + \epsilon$, so it adds a tiny number to a number. Then, it subtracts $f(x)$ to $f(x + \epsilon)$, two similar numbers. Finally, it divides by the tiny number ϵ.

The idea of automatic differentiation is instead to compute exact derivatives (up to machine precision). It will produce a whole new program that computes the derivative f'. To do so, it will differentiate elementary blocks that compose a program, and compose these derivatives via the chain rule. Basic blocks are typically elementary functions such as $\exp, +, \cos$, whose derivatives are definable using a few elementary functions, which are therefore exact. The chain rule only involves products, which are numerically stable. The main limitation is that it requires access to an analytic expression for the function we are implementing as such a program.

Recently, popular automatic differentiation tools such as TensorFlow [Abadi et al., 2016] or PyTorch [Paszke et al., 2017] play a central aspect in machine learning, as computing gradients is key to many optimization methods. In the early days of automatic differentiation, one would only differentiate simple straight-line programs, whose building blocks are elementary differentiable functions. In this situation, semantic models and the correctness of the AD algorithm is pretty straightforward. Nowadays, programs of increasing complexity are being written, and AD tools need to adapt. These richer features include data structures such as arrays, lists, trees, or dictionaries. They can also include conditionals, recursive programs, or higher-order functions. Suddenly, the simple mental model of differentiating first-order programs representing simple differentiable functions no longer applies. This need for differentiation as a core operation in expressive languages is what we call differentiable programming. It is the extension of automatic differentiation that goes beyond differentiability and poses several interesting challenges in terms of implementations, efficiency, accuracy, and correctness. In this dissertation, we will lay some theoretical foundations for studying such languages, through the lens of the study of programming languages, and programming language semantics.

With this view in mind, the study of differentiable programming raises several basic semantics questions, such as

- What's the meaning of differentiating a function w.r.t. a function? Do we even need to answer this question to reason about higher-order programs?

- When a program represents a differentiable function, but its internals may use non-smooth primitives such as conditionals, does AD compute the correct derivative? More generally, what is AD computing exactly?

- Differentiable functions are usually defined on an open domain, so for programs denoting generalized differentiable functions, does the domain need to be an open set?

- Are the guarantees we can obtain on AD sufficient for usage in other algorithms? For instance, is almost everywhere differentiability a desirable property?

1.2 Contribution and Outline

The objective of this thesis is to develop structural foundations for differentiable programming, based on categorical models of higher-order functional languages, to answer questions such as the ones above. More specifically, it aims at doing the following:

(i) Argue that categorical semantics in specific sheaf models constitutes a powerful mathematical language to reason about differentiable programming in expressive higher-order functional languages.

(ii) Expand our understanding of mathematical differentiation, automatic differentiation of programs, their links, and differences.

(iii) Provide a unified and modular approach to extending differentiation and its variants, such as computing higher-order derivatives.

(iv) Showcase different applications of the framework by proving non-trivial properties on differentiable programs and their usage in other algorithms.

In order to do so, this dissertation follows a linear structure, summarized as follows:

- ✚ Chapter 2 presents a simple higher-order language with smooth primitives, defines AD as a source-to-source program transform, introduces a denotational semantics for the language using a generalization of Euclidean spaces and smooth functions, and shows that the transformed program does compute correct derivatives.

- ✚ Chapter 3 shows how to go beyond standard differentiability, as is necessary for a richer language including conditionals and recursion. First, it introduces a new categorical model of generalized smooth spaces. Second, it proves correctness of the AD transform in such a setting. Both are enabled by using a general categorical methodology.

- ✚ Chapter 4 studies several applications of the theory of Chapter 3. It gives a general recipe for extending the higher-order language with new types and new primitives. It looks at gradient descent in the case where gradients are computed using AD. Furthermore, it considers the differentiability of probabilistic programs and the use of AD for the change of variable formula. Finally, it gives a novel characterization of the posterior distributions of probabilistic programs.

2

Differentiable Programming in a smooth setting

Summary. This chapter is about semantic correctness proofs of automatic differentiation. We consider a forward-mode automatic differentiation method on a higher-order language (Section 2.2.2) with algebraic data types (Section 2.4), and we characterise it as the unique structure-preserving macro given a choice of derivatives for basic operations (Section 2.5.1). We describe a rich semantics for differentiable programming (Sections 2.3.2, 2.4.2), based on diffeological spaces (Section 2.3.1). We show that it interprets our language, and we phrase what it means for the automatic differentiation method to be correct with respect to this semantics (Theorems 2.3.2, 2.5.6). We show that our characterisation of automatic differentiation gives rise to an elegant semantic proof of its correctness based on a gluing construction on diffeological spaces (Section 2.5.2). We explain how this is, in essence, a logical relations argument. We show how the analysis extends to AD methods for computing higher-order derivatives using a Taylor approximation (Section 2.6).

Publication. The content of this chapter is taken from Huot et al. [2020], Vákár et al. [2022], which were jointly written with Matthijs Vákár and Sam Staton. Section 2.1 is taken from the same section in Huot et al. [2020], Vákár et al. [2022]. Sections 2.2-2.5 correspond to the same sections in Huot et al. [2020]. Section 2.6 is extracted from Sections 2.2-6 in Vákár et al. [2022], avoiding redundancies between the two papers. Section 2.7 is extracted from Sections 7-8 in Vákár et al. [2022]. Some differences include the following: the higher-order derivatives are delayed until Section 2.6; the discussion and related work is only at the end; numerous figures have been reworked, several proofs (e.g. Prop. 2.5.2) have been expanded to give more details; the exposition of the Faà di Bruno formula has been expanded.

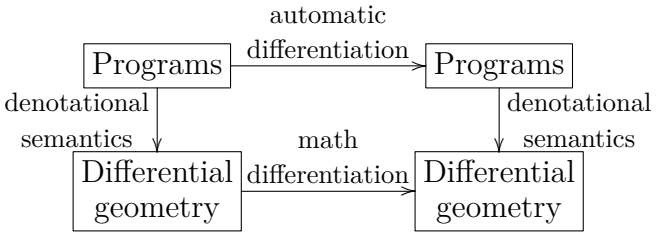


Figure 2.1: Overview of semantics/correctness of AD.

2.1 Introduction

Automatic differentiation, loosely speaking, is the process of taking a program describing a function, and building the derivative of that function by applying the chain rule across the program code. As gradients play a central role in many aspects of machine learning, so too do automatic differentiation systems such as TensorFlow [Abadi et al., 2016] or Stan [Carpenter et al., 2015].

Differentiation has a well-developed mathematical theory in terms of differential geometry. The goal of this chapter is to formalize this connection between differential geometry and the syntactic operations of AD. In this way we achieve two main things:

1. A compositional, denotational understanding of differentiable programming and AD

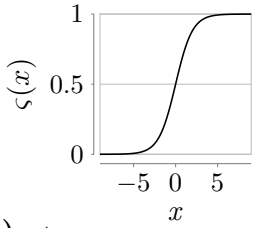
2. An explanation of the correctness of AD

This intuitive correspondence (summarized in Fig. 2.1) is in fact rather complicated. In this chapter, we focus on resolving the following problem: higher-order functions play a key role in programming, and yet they have no counterpart in traditional differential geometry. Moreover, we resolve this problem while retaining the compositionality of denotational semantics.

2.1.1 higher-order functions and differentiation.

A major application of higher-order functions is to support disciplined code reuse. Code reuse is particularly acute in machine learning. For example, a multi-layer neural network might be built of millions of near-identical neurons, as follows.

$$\mathrm{neuron}_n : (\mathbf{real}^n \times (\mathbf{real}^n \times \mathbf{real})) \to \mathbf{real}$$
$$\mathrm{neuron}_n \stackrel{\text{def}}{=} \lambda\langle x, \langle w, b\rangle\rangle.\, \varsigma(w \cdot x + b)$$
$$\mathrm{layer}_n : ((\tau_1 \times P) \to \tau_2) \to (\tau_1 \times P^n) \to \tau_2^n$$
$$\mathrm{layer}_n \stackrel{\text{def}}{=} \lambda f.\, \lambda\langle x, \langle p_1, \ldots, p_n\rangle\rangle.\, \langle f\langle x, p_1\rangle, \ldots, f\langle x, p_n\rangle\rangle$$
$$\mathrm{comp} : (((\tau_1 \times P) \to \tau_2) \times ((\tau_2 \times Q) \to \tau_3)) \to (\tau_1 \times (P \times Q)) \to \tau_3$$
$$\mathrm{comp} \stackrel{\text{def}}{=} \lambda\langle f, g\rangle.\, \lambda\langle x, (p, q)\rangle.\, g\langle f\langle x, p\rangle, q\rangle$$



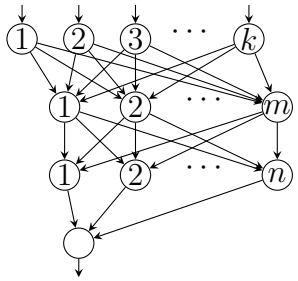


Figure 2.2: The network in (2.1) with k inputs and two hidden layers.

(Here $\varsigma(x) \stackrel{\text{def}}{=} \frac{1}{1+e^{-x}}$ is the sigmoid function, as illustrated.) We can use these functions to build a network as follows (see also Fig. 2.2):

$$\mathrm{comp}\langle \mathrm{layer}_m(\mathrm{neuron}_k), \mathrm{comp}\langle \mathrm{layer}_n(\mathrm{neuron}_m), \mathrm{neuron}_n\rangle\rangle : (\mathbf{real}^k \times P) \to \mathbf{real} \tag{2.1}$$

Here $P \cong \mathbf{real}^p$ with $p = (m(k{+}1){+}n(m{+}1){+}n{+}1)$. This program (2.1) describes a smooth (infinitely differentiable) function. The goal of automatic differentiation is to find its derivative.

If we β-reduce all the λ's, we end up with a very long function expression just built from the sigmoid function and linear algebra. We can then find a program for calculating its derivative by applying the chain rule. However, automatic differentiation can also be expressed without first β-reducing, in a compositional way, by explaining how higher-order functions like (layer) and (comp) propagate derivatives. We provide a semantic analysis of this compositional approach.

The general idea of denotational semantics is to interpret types as spaces and programs as functions between the spaces. We propose to use diffeological spaces and smooth functions Souriau [1980], Iglesias-Zemmour [2013] to this end. These satisfy the following three desiderata:

- $\mathbb{R}$ is a space, and the smooth functions $\mathbb{R} \to \mathbb{R}$ are exactly the functions that are infinitely differentiable;
- The set of smooth functions $X \to Y$ between spaces again forms a space, so we can interpret function types.
- The disjoint union of a sequence of spaces again forms a space, so we can interpret variant types and inductive types.

We emphasise that the most standard formulation of differential geometry, using manifolds, does not support spaces of functions. Diffeological spaces seem to us the simplest notion of space that satisfies these conditions, but there are other candidates Baez and Hoffnung [2011], Stacey [2011]. A diffeological space is in particular a set X equipped with a chosen set of curves $C_X \subseteq X^{\mathbb{R}}$ and a smooth map $f : X \to Y$ must be such that if $\gamma \in C_X$ then $\gamma; f \in C_Y$. This is reminiscent of the method of logical relations.

2.1.2 From smoothness to automatic derivatives at higher types.

Our denotational semantics in diffeological spaces guarantees that all definable functions are smooth. But we need more than just to know that a definable function happens to have a mathematical derivative: we need to be able to find that derivative.

We focus on a simple, forward-mode automatic differentiation method, which is a macro translation on syntax (called $\overrightarrow{\mathcal{D}}$ in Section 2.2). We are able to show that it is correct, using our denotational semantics.

Here there is one subtle point that is central to our development. Although differential geometry provides established derivatives for first-order functions (such as neuron above), there is no canonical notion of derivative for higher-order functions (such as layer and comp) in the theory of diffeological spaces (e.g. Christensen and Wu [2014]). We propose a new way to resolve this, by interpreting types as triples (X, X', S) where, intuitively, X is a space of inhabitants of the type, X' is a space serving as a chosen bundle of tangents over X, and $S \subseteq X^{\mathbb{R}} \times X'^{\mathbb{R}}$ is a binary relation between curves, informally relating curves in X with their tangent curves in X'. This new model gives a denotational semantics for automatic differentiation.

In Section 2.3 we boil this new approach down to a straightforward and elementary logical relations argument for the correctness of automatic differentiation. The approach is explained in detail in Section 2.5.

2.1.3 Summary of contributions.

We have provided a semantic analysis of automatic differentiation. Our syntactic starting point is a well-known forward-mode AD macro on a typed higher-order language (e.g. Shaikhha et al. [2019], Wang et al. [2019]). We recall this in Section 2.2 for function types, and in Section 2.4 we extend it to inductive types and variants. The main contributions of this chapter are as follows.

- We give a denotational semantics for the language in diffeological spaces, showing that every definable expression is smooth (Section 2.3).
- We show correctness of the AD macro by a logical relations argument (Theorem 2.3.2).
- We give a categorical analysis of this correctness argument with two parts: canonicity of the macro in terms of syntactic categories, and a new notion of glued space that abstracts the logical relation (Section 2.5).
- We then use this analysis to state and prove a correctness argument at all first-order types (Theorem 2.5.6).
- We show that our method is not specific to one particular AD macro, by also considering an AD macro for computing higher-order derivatives (Section 2.6).

2.2 A simple forward-mode AD translation

2.2.1 first-order differentiation: the chain rule and dual numbers.

We will now recall the definition of gradient of a differentiable function, the goal of AD and, and what it means for AD to be correct. Recall that the derivative of a function $f : \mathbb{R} \to \mathbb{R}$, if it exists, is a function $\nabla f : \mathbb{R} \to \mathbb{R}$ such that for all a, $\nabla f(a)$ is the gradient of f at a in the sense that the function $x \mapsto f(a)+\nabla f(a)\cdot(x-a)$ gives the best linear approximation of f at a. (The gradient $\nabla f(a)$ is often written $\frac{\mathrm{d}f(x)}{\mathrm{d}x}(a)$.)

The chain rule for differentiation tells us that we can calculate $\nabla(f;g)(a) = \nabla f(a) \cdot \nabla g(f(a))$. In that sense, the chain rule tells us how linear approximations to a function transform under post-composition with another function.

To find ∇f in a compositional way, using the chain rule, two generalizations are reasonable:

- We need both f and ∇f when calculating $\nabla(f;g)$ of a composition $f;g$, using the chain rule, so we are really interested in the pair $(f, \nabla f) : \mathbb{R} \to \mathbb{R} \times \mathbb{R}$;
- In building f we will need to consider functions of multiple arguments, such as $+ : \mathbb{R}^2 \to \mathbb{R}$, and these functions should propagate derivatives.

Thus we are more generally interested in transforming a function $g : \mathbb{R}^n \to \mathbb{R}$ into a function $h : (\mathbb{R} \times \mathbb{R})^n \to \mathbb{R} \times \mathbb{R}$ in such a way that for any $f_1 \dots f_n : \mathbb{R} \to \mathbb{R}$,

$$(f_1, \nabla f_1, \dots, f_n, \nabla f_n); h = ((f_1, \dots, f_n); g, \nabla((f_1, \dots, f_n); g)). \tag{2.2}$$

Computing automatically the program representing h, given a program representing g, is the goal of automatic differentiation. An intuition for h is often given in terms of dual numbers. The transformed function operates on pairs of numbers, (x, x'), and it is common to think of such a pair as $x + x'\epsilon$ for an 'infinitesimal' ϵ. But while this is a helpful intuition, the formalization of infinitesimals can be intricate, and the development in this chapter is focussed on the elementary formulation in (2.2).

A function h satisfying (2.2) encodes all the partial derivatives of g. For example, if $g\colon \mathbb{R}^2 \to \mathbb{R}$, then with $f_1(x) \stackrel{\text{def}}{=} x$ and $f_2(x) \stackrel{\text{def}}{=} x_2$, by applying (2.2) to x_1 we obtain $h(x_1, 1, x_2, 0) = (g(x_1, x_2), \frac{\partial g(x,x_2)}{\partial x}(x_1))$ and similarly $h(x_1, 0, x_2, 1) = (g(x_1, x_2), \frac{\partial g(x_1,x)}{\partial x}(x_2))$. And conversely, if g is differentiable in each argument, then a unique h satisfying (2.2) can be found by taking linear combinations of partial derivatives, for example:

$$h(x_1, x_1', x_2, x_2') = (g(x_1, x_2), x_1' \cdot \tfrac{\partial g(x,x_2)}{\partial x}(x_1) + x_2' \cdot \tfrac{\partial g(x_1,x)}{\partial x}(x_2)).$$

(Here, recall that the partial derivative $\frac{\partial g(x,x_2)}{\partial x}(x_1)$ is a particular notation for the gradient $\nabla(g(-, x_2))(x_1)$, i.e. with x_2 fixed.)

In summary, the idea of differentiation with dual numbers is to transform a differentiable function $g : \mathbb{R}^n \to \mathbb{R}$ to a function $h : \mathbb{R}^{2n} \to \mathbb{R}^2$ which captures g

and all its partial derivatives. We packaged this up in (2.2) as an invariant which is useful for building derivatives of compound functions $\mathbb{R} \to \mathbb{R}$ in a compositional way. The idea of (first-order) forward-mode automatic differentiation is to perform this transformation at the source code level.

We say that a macro for AD is correct if, given a semantic model $[\![-]\!]$, the program P representing $g = [\![P]\!]$ is transformed by the macro to a program P' representing $h = [\![P']\!]$. This means in particular that P' computes correct partial derivatives of (the function represented by) P.

Smooth functions.

In what follows we will often speak of *smooth* functions $\mathbb{R}^k \to \mathbb{R}$, which are functions that are continuous and differentiable, such that their derivatives are also continuous and differentiable, and so on.

2.2.2 A simple language of smooth functions.

We consider a standard higher-order typed language with a first-order type **real** of real numbers. The types (τ, σ) and terms (t, s) are as follows.

τ, σ, ρ	::=	types	
	$\mid$ **real**	real numbers	
	$\mid$ $(\tau_1 \times \ldots \times \tau_n)$	finite product	
	$\mid$ $\tau \to \sigma$	function	
t, s, r	::=	terms	
	x	variable	
	$\mid$ $\underline{c} \mid t + s \mid t * s \mid \varsigma(t)$	operations/constants	
	$\mid$ $\langle t_1, \ldots, t_n \rangle \mid \mathbf{case}\, t\, \mathbf{of}\, \langle x_1, \ldots, x_n \rangle \to s$	tuples/pattern matching	
	$\mid$ $\lambda x.t \mid t\, s$	function abstraction/app.	

Figure 2.3: Syntax of our language

The typing rules are in Figure 2.4. We have included a minimal set of operations for the sake of illustration, but it is not difficult to add further operations. We add some simple syntactic sugar $t - u \stackrel{\text{def}}{=} t + \underline{(-1)} \times u$. We intend ς to stand for the sigmoid function, $\varsigma(x) \stackrel{\text{def}}{=} \frac{1}{1+e^{-x}}$. We further include syntactic sugar $\mathbf{let}\, x = t\, \mathbf{in}\, s$ for $(\lambda x.s)\, t$ and $\lambda\langle x_1, \ldots, x_n\rangle.t$ for $\lambda x.\mathbf{case}\, x\, \mathbf{of}\, \langle x_1, \ldots, x_n\rangle \to t$. A ground type is a type that does not contain a function type. A ground context is a context where all the variables have ground types. A first-order term is a term of ground type in a ground context.

$$\frac{}{\Gamma \vdash \underline{c} : \mathbf{real}}(c \in \mathbb{R}) \quad \frac{\Gamma \vdash t : \mathbf{real} \quad \Gamma \vdash t : \mathbf{real}}{\Gamma \vdash t + t : \mathbf{real}} \quad \frac{\Gamma \vdash t : \mathbf{real} \quad \Gamma \vdash t : \mathbf{real}}{\Gamma \vdash t * t : \mathbf{real}}$$

$$\frac{\Gamma \vdash t : \mathbf{real}}{\Gamma \vdash \varsigma(t) : \mathbf{real}} \quad \frac{\Gamma \vdash t_1 : \tau_1 \quad \ldots \quad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash \langle t_1, \ldots, t_n \rangle : (\tau_1 \times \ldots \times \tau_n)}$$

$$\frac{\Gamma \vdash t : (\tau_1 \times \ldots \times \tau_n) \quad \Gamma, x_1 : \tau_1, ..., x_n : \tau_n \vdash t : \tau}{\Gamma \vdash \mathbf{case}\, t\, \mathbf{of}\, \langle x_1, \ldots, x_n \rangle \to t : \tau}$$

$$\frac{}{\Gamma \vdash x : \tau}((x : \tau) \in \Gamma) \quad \frac{\Gamma, x : \tau \vdash t : \sigma}{\Gamma \vdash \lambda x : \tau . t : \tau \to \sigma} \quad \frac{\Gamma \vdash t : \tau \to \sigma \qquad \Gamma \vdash s : \tau}{\Gamma \vdash t\, s : \sigma}$$

Figure 2.4: Typing rules for the simple language.

2.2.3 Syntactic automatic differentiation: a functorial macro.

The aim of forward-mode AD is to find the dual numbers representation of a function by syntactic manipulations. For our simple language, we implement this as the following inductively defined macro $\overrightarrow{\mathcal{D}}$ on both types and terms (see also Wang et al. [2019], Shaikhha et al. [2019]):

$$\begin{array}{lcl}
\overrightarrow{\mathcal{D}}(\mathbf{real}) & \stackrel{\text{def}}{=} & \mathbf{real} \times \mathbf{real} \\
\overrightarrow{\mathcal{D}}(\tau \to \tau_2) & \stackrel{\text{def}}{=} & \overrightarrow{\mathcal{D}}(\tau) \to \overrightarrow{\mathcal{D}}(\tau_2) \\
\overrightarrow{\mathcal{D}}(\tau_1 \times \ldots \times \tau_n) & \stackrel{\text{def}}{=} & \overrightarrow{\mathcal{D}}(\tau_1) \times \ldots \times \overrightarrow{\mathcal{D}}(\tau_n)
\end{array}$$

$$\begin{array}{lcl}
\overrightarrow{\mathcal{D}}(x) & \stackrel{\text{def}}{=} & x \\
\overrightarrow{\mathcal{D}}(\underline{c}) & \stackrel{\text{def}}{=} & \langle \underline{c}, 0 \rangle \\
\overrightarrow{\mathcal{D}}(t + s) & \stackrel{\text{def}}{=} & \mathbf{case}\, \overrightarrow{\mathcal{D}}(t)\, \mathbf{of}\, \langle x, x' \rangle \to \mathbf{case}\, \overrightarrow{\mathcal{D}}(s) \\
 & & \quad \mathbf{of}\, \langle y, y' \rangle \to \langle x + y, x' + y' \rangle \\
\overrightarrow{\mathcal{D}}(t * s) & \stackrel{\text{def}}{=} & \mathbf{case}\, \overrightarrow{\mathcal{D}}(t)\, \mathbf{of}\, \langle x, x' \rangle \to \mathbf{case}\, \overrightarrow{\mathcal{D}}(s) \\
 & & \quad \mathbf{of}\, \langle y, y' \rangle \to \langle x * y, x * y' + x' * y \rangle \\
\overrightarrow{\mathcal{D}}(\varsigma(t)) & \stackrel{\text{def}}{=} & \mathbf{case}\, \overrightarrow{\mathcal{D}}(t)\, \mathbf{of}\, \langle x, x' \rangle \to \mathbf{let}\, y = \varsigma(x) \\
 & & \quad \mathbf{in}\, \langle y, x' * y * (1 - y) \rangle \\
\overrightarrow{\mathcal{D}}(\lambda x.t) & \stackrel{\text{def}}{=} & \lambda x. \overrightarrow{\mathcal{D}}(t) \\
\overrightarrow{\mathcal{D}}(t\, s) & \stackrel{\text{def}}{=} & \overrightarrow{\mathcal{D}}(t)\, \overrightarrow{\mathcal{D}}(s) \\
\overrightarrow{\mathcal{D}}(\langle t_1, \ldots, t_n \rangle) & \stackrel{\text{def}}{=} & \langle \overrightarrow{\mathcal{D}}(t_1), \ldots, \overrightarrow{\mathcal{D}}(t_n) \rangle \\
\overrightarrow{\mathcal{D}}(\mathbf{case}\, t\, \mathbf{of}\, \langle x_1, \ldots, x_n \rangle \to s) & \stackrel{\text{def}}{=} & \mathbf{case}\, \overrightarrow{\mathcal{D}}(t)\, \mathbf{of}\, \langle x_1, \ldots, x_n \rangle \to \overrightarrow{\mathcal{D}}(s)
\end{array}$$

Figure 2.5: AD translation.

We extend $\overrightarrow{\mathcal{D}}$ to contexts: $\overrightarrow{\mathcal{D}}(\{x_1{:}\tau_1, ..., x_n{:}\tau_n\}) \stackrel{\text{def}}{=} \{x_1{:}\overrightarrow{\mathcal{D}}(\tau_1), ..., x_n{:}\overrightarrow{\mathcal{D}}(\tau_n)\}$. This turns $\overrightarrow{\mathcal{D}}$ into a well-typed, functorial macro in the following sense.

Lemma 2.2.1 (Functorial macro)**.** *If* $\Gamma \vdash t : \tau$ *then* $\overrightarrow{\mathcal{D}}(\Gamma) \vdash \overrightarrow{\mathcal{D}}(t) : \overrightarrow{\mathcal{D}}(\tau)$*.*
If $\Gamma, x : \tau_2 \vdash t : \tau$ *and* $\Gamma \vdash s : \tau_2$ *then* $\overrightarrow{\mathcal{D}}(\Gamma) \vdash \overrightarrow{\mathcal{D}}(t[{}^{s}/_{x}]) = \overrightarrow{\mathcal{D}}(t)[{}^{\overrightarrow{\mathcal{D}}(s)}/_{x}]$*.*

The lemma can be proved by a simple induction on t, but we will provide a different argument in Section 2.5.

Example 2.2.1 (Inner products)**.** *Let us write* τ^n *for the n-fold product* $(\tau \times \ldots \times \tau)$*. Then, given* $\Gamma \vdash t, s : \mathbf{real}^n$ *we can define their inner product*

$$\begin{aligned}\Gamma \vdash t \cdot_n s \stackrel{\text{def}}{=}\ & \mathbf{case}\, t\, \mathbf{of}\, \langle z_1, \ldots, z_n\rangle \to \\ & \mathbf{case}\, s\, \mathbf{of}\, \langle y_1, \ldots, y_n\rangle \to z_1 * y_1 + \cdots + z_n * y_n : \mathbf{real}\end{aligned}$$

To illustrate the calculation of $\overrightarrow{\mathcal{D}}$*, let us expand (and* β*-reduce)* $\overrightarrow{\mathcal{D}}(t \cdot_2 s)$*:*

$\mathbf{case}\, \overrightarrow{\mathcal{D}}(t)\, \mathbf{of}\, \langle z_1, z_2\rangle \to \mathbf{case}\, \overrightarrow{\mathcal{D}}(s)\, \mathbf{of}\, \langle y_1, y_2\rangle \to \mathbf{case}\, z_1\, \mathbf{of}\, \langle z_{1,1}, z_{1,2}\rangle \to$
$\mathbf{case}\, y_1\, \mathbf{of}\, \langle y_{1,1}, y_{1,2}\rangle \to \mathbf{case}\, z_2\, \mathbf{of}\, \langle z_{2,1}, z_{2,2}\rangle \to \mathbf{case}\, y_2\, \mathbf{of}\, \langle y_{2,1}, y_{2,2}\rangle \to$
$\langle z_{1,1} * y_{1,1} + z_{2,1} * y_{2,1}\ ,\ z_{1,1} * y_{1,2} + z_{1,2} * y_{1,1} + z_{2,1} * y_{2,2} + z_{2,2} * y_{2,1}\rangle$

Example 2.2.2 (Neural networks)**.** *In our introduction* (2.1)*, we provided a program in our language to build a neural network out of expressions* neuron, layer, comp*; this program makes use of the inner product of Example 2.2.1. We can similarly calculate* $\overrightarrow{\mathcal{D}}$ *of such neural nets by mechanically applying the macro.*

2.3 Semantics of differentiation

Consider for a moment the first-order fragment of the language in Section 2.2, with only one type, **real**, and no λ's or pairs. This has a simple semantics in the category of cartesian spaces and smooth maps. Indeed, a term $x_1 \ldots x_n : \mathbf{real} \vdash t : \mathbf{real}$ has a natural reading as a function $[\![t]\!] : \mathbb{R}^n \to \mathbb{R}$ by interpreting our operation symbols by the well-known operations on $\mathbb{R}^n \to \mathbb{R}$ with the corresponding name. In fact, the functions that are definable in this first-order fragment are smooth, which means that they are continuous, differentiable, and their derivatives are continuous, differentiable, and so on. Let us write **CartSp** for this category of cartesian spaces ($\mathbb{R}^n$ for some n) and smooth functions.

The category **CartSp** has cartesian products, and so we can also interpret product types, tupling and pattern matching, giving us a useful syntax for constructing functions into and out of products of $\mathbb{R}$. For example, the interpretation of (neuron_n) in (2.1) becomes

$$\mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R} \xrightarrow{[\![\cdot_n]\!] \times \mathrm{id}_{\mathbb{R}}} \mathbb{R} \times \mathbb{R} \xrightarrow{[\![+]\!]} \mathbb{R} \xrightarrow{[\![\varsigma]\!]} \mathbb{R}.$$

where $[\![\cdot_n]\!]$, $[\![+]\!]$ and $[\![\varsigma]\!]$ are the usual inner product, addition and the sigmoid function on $\mathbb{R}$, respectively.

Inside this category, we can straightforwardly study the first-order language without λ's, and automatic differentiation. In fact, we can prove the following by plain induction on the syntax:

The interpretation of the (syntactic) forward AD $\overrightarrow{\mathcal{D}}(t)$ *of a first-order term t equals the usual (semantic) derivative of the interpretation of t as a smooth function.*

However, as is well known, the category **CartSp** does not support function spaces. We quickly review the argument here. For each natural number d, we have polynomial terms

$$x_1, \ldots, x_d : \mathbf{real} \vdash \lambda y.\ \textstyle\sum_{n=1}^{d} x_n y^n : \mathbf{real} \to \mathbf{real}$$

If we could interpret (**real** $\to$ **real**) as a Euclidean space $\mathbb{R}^N$ for some N then, by interpreting these polynomial expressions, we would be able to find continuous injections $\mathbb{R}^d \to \mathbb{R}^N$ for every d, which is topologically impossible for any $d > N$, for example as a consequence of the Borsuk-Ulam theorem.

This means that we cannot interpret the functions (layer) and (comp) from (2.1) in **CartSp**, as they are higher-order functions, even though they are very useful and innocent building blocks for differential programming. We call it innocent, as any first-order function built using $\mathrm{stack}_{n,m}^{k,l}$ and other smooth first-order functions is again an ordinary smooth function. The neural network $\mathrm{nn}_{n,m}$ is a prime example. Clearly, we could define neural nets such as (2.1) directly as smooth functions without any higher-order subcomponents, though that would quickly become cumbersome for deep networks. A problematic consequence of the lack of a semantics for higher-order differential programs is that we have no obvious way of establishing compositional semantic correctness of $\overrightarrow{\mathcal{D}}$ for the given implementation of (2.1).

2.3.1 Diffeological spaces.

This motivates us to turn to a more general notion of differential geometry for our semantics, based on *diffeological spaces* [Iglesias-Zemmour, 2013]. The key idea will be that a higher-order function is called smooth if it sends smooth functions to smooth functions, meaning that we can never use it to build first-order functions that are not smooth. For example, (comp) in (2.1) has this property.

Definition 2.3.1. *A* diffeological space $(X, \mathcal{P}_X)$ *consists of a set X together with, for each n and each open subset U of $\mathbb{R}^n$, a set $\mathcal{P}_X^U \subseteq [U \to X]$ of functions, called* plots*, such that*

- *all constant functions are plots;*
- *if $f : V \to U$ is a smooth function and $p \in \mathcal{P}_X^U$, then $f; p \in \mathcal{P}_X^V$;*
- *if $\left(p_i \in \mathcal{P}_X^{U_i}\right)_{i \in I}$ is a compatible family of plots ($x \in U_i \cap U_j \Rightarrow p_i(x) = p_j(x)$) and $(U_i)_{i \in I}$ covers U, then the gluing $p : U \to X : x \in U_i \mapsto p_i(x)$ is a plot.*

Definition 2.3.2. *We call a function $f : X \to Y$ between diffeological spaces* smooth *if, for all plots $p \in \mathcal{P}_X^U$, we have that $p; f \in \mathcal{P}_Y^U$. We write* $\mathbf{Diff}(X, Y)$ *for the set of smooth maps from X to Y.*

Smooth functions compose, and so we have a category **Diff** of diffeological spaces and smooth functions.

A diffeological space is therefore a set equipped with structure. Many constructions of sets carry over straightforwardly to diffeological spaces. In particular,

the category is well-pointed, meaning that for $f, g : X \to Y$, we have that $f = g$ iff $f(x) = g(x)$ for all $x : 1 \to X$. In particular, morphisms are monos if and only if the underlying functions are injective.

Example 2.3.1. *We can interpret a subset $Y \subseteq X$ of a diffeological space X as a* subspace *by taking the plots of X which factor over Y as its plots.*

Example 2.3.2 (Cartesian diffeologies)**.** *Each open subset U of $\mathbb{R}^n$ can be given the structure of a diffeological space by taking all the smooth functions $V \to U$ as $\mathcal{P}_U^V$. It is easily seen that smooth functions from $V \to U$ in the traditional sense coincide with smooth functions in the sense of diffeological spaces. Thus, diffeological spaces have a profound relationship with ordinary calculus.*

Remark 2.3.1. *In categorical terms, this gives a full embedding of* **CartSp** *in* **Diff***.*

Example 2.3.3 (Product diffeologies)**.** *Given a family $(X_i)_{i\in I}$ of diffeological spaces, we can equip the product $\prod_{i\in I} X_i$ of sets with the* product diffeology *in which U-plots are precisely the functions of the form $(p_i)_{i\in I}$ for $p_i \in \mathcal{P}_{X_i}^U$.*

This gives us the categorical product in **Diff**.

Example 2.3.4 (Functional diffeology)**.** *We can equip the set $\mathbf{Diff}(X, Y)$ of smooth functions between diffeological spaces with the* functional diffeology *in which U-plots consist of functions $f : U \to \mathbf{Diff}(X, Y)$ such that $(u, x) \mapsto f(u)(x)$ is an element of $\mathbf{Diff}(U \times X, Y)$.*

This specifies the categorical function object in **Diff**.

Remark 2.3.2. *This construction satisfies the usual universal property of function spaces, making* **Diff** *a Cartesian-closed category.*

2.3.2 Semantics and correctness of AD.

We can now give a denotational semantics to our language from Section 2.2. We interpret each type τ as a set $[\![\tau]\!]$ equipped with the relevant diffeology, by induction on the structure of types:

$$\begin{array}{lcl} [\![\mathbf{real}]\!] & \stackrel{\text{def}}{=} & \mathbb{R} \\ [\![(\tau_1 \times \ldots \times \tau_n)]\!] & \stackrel{\text{def}}{=} & \prod_{i=1}^n [\![\tau_i]\!] \\ [\![\tau \to \sigma]\!] & \stackrel{\text{def}}{=} & \mathbf{Diff}([\![\tau]\!], [\![\sigma]\!]) \end{array}$$

A context $\Gamma = (x_1 \colon \tau_1 \ldots x_n \colon \tau_n)$ is interpreted as a diffeological space $[\![\Gamma]\!] \stackrel{\text{def}}{=} \prod_{i=1}^n [\![\tau_i]\!]$.

Now, well typed terms $\Gamma \vdash t : \tau$ are interpreted as smooth functions $[\![t]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$, giving a meaning for t for every valuation of the context. This is routinely defined by induction on the structure of typing derivations. Constants $\underline{c}$: **real** are interpreted as constant functions; and the first-order operations $(+, *, \varsigma)$ are

interpreted by composing with the corresponding functions, which are smooth. For example, $[\![\varsigma(t)]\!](\rho) \stackrel{\text{def}}{=} \varsigma([\![t]\!](\rho))$, where $\rho \in [\![\Gamma]\!]$. Variables are interpreted as $[\![x_i]\!](\rho) \stackrel{\text{def}}{=} \rho_i$. The remaining constructs are interpreted as follows, and it is straightforward to show that smoothness is preserved.

$$\begin{array}{lcl}
[\![\langle t_1, \ldots, t_n\rangle]\!](\rho) & \stackrel{\text{def}}{=} & ([\![t_1]\!](\rho), \ldots, [\![t_n]\!](\rho)) \\
[\![\lambda x{:}\tau.t]\!](\rho)(a) & \stackrel{\text{def}}{=} & [\![t]\!](\rho, a)\ (a \in [\![\tau]\!]) \\
[\![\mathbf{case}\, t\, \mathbf{of}\, \langle x_1, x_2\rangle \to s]\!](\rho) & \stackrel{\text{def}}{=} & [\![s]\!](\rho, [\![t]\!](\rho)) \\
[\![t\, s]\!](\rho) & \stackrel{\text{def}}{=} & [\![t]\!](\rho)([\![s]\!](\rho))
\end{array}$$

As is standard in a Cartesian-closed category, the interpretation $[\![-]\!]$ respects the $\beta\eta$-equational theory:

Proposition 2.3.1 (Soundness)**.** *Let* $\Gamma \vdash t, s : \tau$*. If* $t \stackrel{\beta\eta}{=} s$*, then* $[\![t]\!] = [\![s]\!]$*.*

Notice that a term $x_1\colon \mathbf{real}, \ldots, x_n\colon \mathbf{real} \vdash t : \mathbf{real}$ is interpreted as a smooth function $[\![t]\!] : \mathbb{R}^n \to \mathbb{R}$, even if t involves higher-order functions (like (2.1)). Moreover the macro differentiation $\overrightarrow{\mathcal{D}}(t)$ is a function $[\![\overrightarrow{\mathcal{D}}(t)]\!] : (\mathbb{R} \times \mathbb{R})^n \to (\mathbb{R} \times \mathbb{R})$. This enables us to state a limited version of our main correctness theorem:

Theorem 2.3.2 (Semantic correctness of $\overrightarrow{\mathcal{D}}$ (limited))**.** *For any term* $x_1\colon \mathbf{real}, \ldots, x_n\colon \mathbf{real} \vdash t : \mathbf{real}$*, the function* $[\![\overrightarrow{\mathcal{D}}(t)]\!]$ *is the dual numbers representation* (2.2) *of* $[\![t]\!]$*. In detail: for any smooth functions* $f_1 \ldots f_n : \mathbb{R} \to \mathbb{R}$*,*

$$(f_1, \nabla f_1, \ldots, f_n, \nabla f_n); [\![\overrightarrow{\mathcal{D}}(t)]\!] = \Big((f_1 \ldots f_n); [\![t]\!], \nabla((f_1 \ldots f_n); [\![t]\!])\Big).$$

For instance, if $n = 2$, then $[\![\overrightarrow{\mathcal{D}}(t)]\!](x_1, 1, x_2, 0) = ([\![t]\!](x_1, x_2), \frac{\partial [\![t]\!](x, x_2)}{\partial x}(x_1))$. More generally, this is sufficient to show that $\overrightarrow{\mathcal{D}}(t)$ computes the correct partial derivatives of t, and therefore correct gradients.

Proof. We prove this by logical relations. Although the following proof is elementary, we found it by using the categorical methods in Section 2.5. We sketch the key arguments here, the fuller proof will again be given in Section 2.5.

For each type τ, we define a binary relation S_τ between curves in $[\![\tau]\!]$ and curves in $[\![\overrightarrow{\mathcal{D}}(\tau)]\!]$, i.e. $S_\tau \subseteq \mathcal{P}^{\mathbb{R}}_{[\![\tau]\!]} \times \mathcal{P}^{\mathbb{R}}_{[\![\overrightarrow{\mathcal{D}}(\tau)]\!]}$, by induction on τ:

$$\begin{array}{rcl}
S_{\mathbf{real}} & \stackrel{\text{def}}{=} & \{(f, (f, \nabla f)) \mid f : \mathbb{R} \to \mathbb{R} \text{ smooth}\} \\
S_{(\tau_1 \times \ldots \times \tau_n)} & \stackrel{\text{def}}{=} & \{(\langle f_1, \ldots, f_n\rangle, \langle g_1, \ldots, g_n\rangle) \mid (f_1, g_1) \in S_{\tau_1}, \ldots, (f_n, g_n) \in S_{\tau_n}\} \\
S_{\tau \to \sigma} & \stackrel{\text{def}}{=} & \{(f_1, f_2) \mid \forall (g_1, g_2) \in S_\tau.(x \mapsto f_1(x)(g_1(x)), x \mapsto f_2(x)(g_2(x))) \in S_\sigma\}
\end{array}$$

Then, we establish the following 'fundamental lemma':

> If $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t : \tau_2$ and, for all $1 \le i \le n$, $f_i : \mathbb{R} \to [\![\tau_i]\!]$ and $g_i : \mathbb{R} \to [\![\overrightarrow{\mathcal{D}}(\tau_i)]\!]$ are such that (f_i, g_i) is in S_{τ_i}, then we have that $\Big((f_1, \ldots, f_n); [\![t]\!], (g_1, \ldots, g_n); [\![\overrightarrow{\mathcal{D}}(t)]\!]\Big)$ is in S_{τ_2}.

This is proved routinely by induction on the typing derivation of t. The case for $*$ relies on the precise definition of $\overrightarrow{\mathcal{D}}(t * s)$, and similarly for $+, \varsigma$.

We conclude the theorem from the fundamental lemma by considering the case where $\tau_i = \sigma = \mathbf{real}$, $m = n$ and $s_i = y_i$. □

2.4 Extending the language: variant and inductive types

In this section, we show that the definition of forward AD and the semantics generalize if we extend the language of Section 2.2 with variants and inductive types. As an example of inductive types, we consider lists. This specific choice is only for expository purposes and the whole development works at the level of generality of arbitrary algebraic data types generated as initial algebras of (polynomial) type constructors formed by finite products and variants.

Similarly, our choice of operations is for expository purposes. More generally, assume given a family of operations $(\mathsf{Op}_n)_{n\in\mathbb{N}}$ indexed by their arity n. Further assume that each $op \in \mathsf{Op}_n$ has type $\mathbf{real}^n \to \mathbf{real}$. We then ask for a certain closure of these operations under differentiation, that is we define

$$\begin{aligned}\overrightarrow{\mathcal{D}}(op(t_1,\ldots,t_n)) \stackrel{\text{def}}{=}\ & \mathbf{case}\,\overrightarrow{\mathcal{D}}(t_1)\,\mathbf{of}\,\langle x_1, x_1'\rangle \to \ldots \to \mathbf{case}\,\overrightarrow{\mathcal{D}}(t_n)\,\mathbf{of}\,\langle x_n, x_n'\rangle \to \\ & \langle op(x_1,\ldots,x_n), \textstyle\sum_{i=1}^n x_i' * \partial_i op(x_1,\ldots,x_n)\rangle\end{aligned}$$

where $\partial_i op(x_1,\ldots,x_n)$ is some chosen term in the language, involving free variables from $x_1,\ldots,x_n$, which we think of as implementing the partial derivative of op with respect to its i-th argument. For constructing the semantics, every op must be interpreted by some smooth function, and, to establish correctness, the semantics of $\partial_i op(x_1,\ldots,x_n)$ must be the semantic i-th partial derivative of the semantics of $op(x_1,\ldots,x_n)$.

2.4.1 Language.

We assume given a countably infinite set of labels L. We additionally consider the following types and terms, where $\ell_i \in L$.

τ, σ, ρ	$::=$		types
	$\mid$	$\mathbf{list}(\tau)$	list
	$\mid$	$\{\ell_1\,\tau_1 \mid \ldots \mid \ell_\mathsf{n}\,\tau_n\}$	variant
t, s, r	$::=$		terms
	$\mid$	$\tau.\ell\,t$	variant constructor
	$\mid$	$[] \mid t :: s$	empty list and cons
	$\mid$	$\mathbf{case}\,t\,\mathbf{of}\,\{\ell_1\,x_1 \to s_1 \mid \cdots \mid \ell_\mathsf{n}\,x_n \to s_n\}$	pattern matching: variants
	$\mid$	$\mathbf{fold}\,(x_1, x_2).t\,\mathbf{over}\,s\,\mathbf{from}\,r$	list fold

Figure 2.6: Syntax for the extended language.

We extend the type system according to:

$$\frac{\Gamma \vdash t : \tau_i}{\Gamma \vdash \tau.\ell_i\, t : \tau}((\ell_i\, \tau_i) \in \tau) \qquad \frac{}{\Gamma \vdash [\,] : \mathbf{list}(\tau)} \qquad \frac{\Gamma \vdash t : \tau \qquad \Gamma \vdash s : \mathbf{list}(\tau)}{\Gamma \vdash t :: s : \mathbf{list}(\tau)}$$

$$\frac{\Gamma \vdash t : \{\ell_1\, \tau_1 \;\big|\; \ldots \;\big|\; \ell_{\mathsf{n}}\, \tau_n\} \qquad \text{for each } 1 \le i \le n\text{: } \Gamma, x_i : \tau_i \vdash s_i : \tau}{\Gamma \vdash \mathbf{case}\, t\, \mathbf{of}\, \{\ell_1\, x_1 \to s_1 \;\big|\; \cdots \;\big|\; \ell_{\mathsf{n}}\, x_n \to s_n\} : \tau}$$

$$\frac{\Gamma \vdash s : \mathbf{list}(\tau) \qquad \Gamma \vdash r : \sigma \qquad \Gamma, x_1 : \tau, x_2 : \sigma \vdash t : \sigma}{\Gamma \vdash \mathbf{fold}\, (x_1, x_2).t\, \mathbf{over}\, s\, \mathbf{from}\, r : \sigma}$$

Figure 2.7: Typing rules for the extended language.

We can then extend $\overrightarrow{\mathcal{D}}$ to our new types and terms by

$$\begin{aligned}
\overrightarrow{\mathcal{D}}(\{\ell_1\, \tau_1 \;\big|\; \ldots \;\big|\; \ell_{\mathsf{n}}\, \tau_n\}) &\stackrel{\text{def}}{=} \{\ell_1\, \overrightarrow{\mathcal{D}}(\tau_1) \;\big|\; \ldots \;\big|\; \ell_{\mathsf{n}}\, \overrightarrow{\mathcal{D}}(\tau_n)\} \\
\overrightarrow{\mathcal{D}}(\mathbf{list}(\tau)) &\stackrel{\text{def}}{=} \mathbf{list}(\overrightarrow{\mathcal{D}}(\tau)) \\
\\
\overrightarrow{\mathcal{D}}(\tau.\ell\, t) &\stackrel{\text{def}}{=} \overrightarrow{\mathcal{D}}(\tau).\ell\, \overrightarrow{\mathcal{D}}(t) \\
\overrightarrow{\mathcal{D}}([\,]) &\stackrel{\text{def}}{=} [\,] \\
\overrightarrow{\mathcal{D}}(t :: s) &\stackrel{\text{def}}{=} \overrightarrow{\mathcal{D}}(t) :: \overrightarrow{\mathcal{D}}(s) \\
\overrightarrow{\mathcal{D}}(\mathbf{fold}\, (x_1, x_2).t\, \mathbf{over}\, s\, \mathbf{from}\, r) &\stackrel{\text{def}}{=} \mathbf{fold}\, (x_1, x_2).\overrightarrow{\mathcal{D}}(t)\, \mathbf{over}\, \overrightarrow{\mathcal{D}}(s)\, \mathbf{from}\, \overrightarrow{\mathcal{D}}(r) \\
\overrightarrow{\mathcal{D}}(\mathbf{case}\, t\, \mathbf{of}\, \{\ell_1\, x_1 \to s_1 \;\big|\; \cdots \;\big|\; \ell_{\mathsf{n}}\, x_n \to s_n\}) &\stackrel{\text{def}}{=} \\
\mathbf{case}\, \overrightarrow{\mathcal{D}}(t)\, \mathbf{of}\, \{\ell_1\, x_1 \to \overrightarrow{\mathcal{D}}(s_1) \;\big|\; \cdots \;\big|\; \ell_{\mathsf{n}}\, x_n \to \overrightarrow{\mathcal{D}}(s_n)\}
\end{aligned}$$

Figure 2.8: AD translation for the extended language.

To demonstrate the practical use of expressive type systems for differential programming, we consider the following two examples.

Example 2.4.1 (Lists of inputs for neural nets)**.** *Usually, we run a neural network on a large data set, the size of which might be determined at runtime. To evaluate a neural network on multiple inputs, in practice, one often sums the outcomes. This can be coded in our extended language as follows. Suppose that we have a network* $f : (\mathbf{real}^n \times P) \to \mathbf{real}$ *that operates on single input vectors. We can construct one that operates on lists of inputs as follows:*

$$g \stackrel{\text{def}}{=} \lambda\langle l, w\rangle.\mathbf{fold}\, (x_1, x_2).f\langle x_1, w\rangle + x_2\, \mathbf{over}\, l\, \mathbf{from}\, \underline{0} : (\mathbf{list}(\mathbf{real}^n) \times P) \to \mathbf{real}$$

Example 2.4.2 (Missing data)**.** *In practically every application of statistics and machine learning, we face the problem of* missing data*: for some observations, only partial information is available. In an expressive typed programming language like we consider, we can model missing data conveniently using the data type*

$\mathbf{maybe}(\tau) = \{\mathsf{Nothing} \mid \mathsf{Just}\,\tau\}$. *In the context of a neural network, one might use it as follows. First, define some helper functions*

$$\mathrm{fromMaybe} \stackrel{\text{def}}{=} \lambda x.\lambda m.\mathbf{case}\, m\, \mathbf{of}\, \{\mathsf{Nothing}\_ \to x \mid \mathsf{Just}\, x' \to x'\}$$

$$\mathrm{fromMaybe}^n \stackrel{\text{def}}{=} \lambda\langle x_1, ..., x_n\rangle.\lambda\langle m_1, ..., m_n\rangle.\langle \mathrm{fromMaybe}\, x_1\, m_1, ..., \mathrm{fromMaybe}\, x_n\, m_n\rangle : (\mathbf{maybe}(\mathbf{real}))^n \to \mathbf{real}^n \to \mathbf{real}^n$$

$$\mathrm{map} \stackrel{\text{def}}{=} \lambda f.\lambda l.\mathbf{fold}\,(x_1, x_2).f\, x_1 :: x_2\, \mathbf{over}\, l\, \mathbf{from}\, [\,] : (\tau \to \sigma) \to \mathbf{list}(\tau) \to \mathbf{list}(\sigma)$$

Given a neural network $f : (\mathbf{list}(\mathbf{real}^k) \times P) \to \mathbf{real}$*, we can build a new one that operates on on a data set for which some covariates (features) are missing, by passing in default values to replace the missing covariates:*

$$\lambda\langle l, \langle m, w\rangle\rangle.f\langle \mathrm{map}\,(\mathrm{fromMaybe}^k\, m)\, l, w\rangle : (\mathbf{list}((\mathbf{maybe}(\mathbf{real}))^k) \times (\mathbf{real}^k \times P)) \to \mathbf{real}$$

Then, given a data set l *with missing covariates, we can perform automatic differentiation on this network to optimize, simultaneously, the ordinary network parameters* w and *the default values for missing covariates* m*.*

2.4.2 Semantics.

In Section 2.3 we gave a denotational semantics for the simple language in diffeological spaces. This extends to the language in this section, as follows. As before, each type τ is interpreted as a diffeological space, which is a set equipped with a family of plots:

- A variant type $\{\ell_1\,\tau_1 \mid \ldots \mid \ell_{\mathsf{n}}\,\tau_n\}$ is inductively interpreted as a disjoint union and U-plots given by

$$[\![\{\ell_1\,\tau_1 \mid \cdots \mid \ell_{\mathsf{n}}\,\tau_n\}]\!] \stackrel{\text{def}}{=} \biguplus_{i=1}^n [\![\tau_i]\!]$$

$$\mathcal{P}^U_{[\![\{\ell_1\,\tau_1 \mid ... \mid \ell_{\mathsf{n}}\,\tau_n\}]\!]} \stackrel{\text{def}}{=} \left\{ \left[U_j \xrightarrow{f_j} [\![\tau_j]\!] \to \biguplus_{i=1}^n [\![\tau_i]\!] \right]_{j=1}^n \;\middle|\; U = \biguplus_{j=1}^n U_j,\ f_j \in \mathcal{P}^{U_j}_{[\![\tau_j]\!]} \right\}$$

- A list type $\mathbf{list}(\tau)$ is interpreted as a set of lists and U-plots given by

$$[\![\mathbf{list}(\tau)]\!] \stackrel{\text{def}}{=} \biguplus_{i=1}^\infty [\![\tau]\!]^i$$

$$\mathcal{P}^U_{[\![\mathbf{list}(\tau)]\!]} \stackrel{\text{def}}{=} \left\{ \left[U_j \xrightarrow{f_j} [\![\tau]\!]^j \to \biguplus_{i=1}^\infty [\![\tau]\!]^i \right]_{j=1}^\infty \;\middle|\; U = \biguplus_{j=1}^\infty U_j,\ f_j \in \mathcal{P}^{U_j}_{[\![\tau]\!]^j} \right\}$$

The constructors and destructors for variants and lists are interpreted as in the usual set theoretic semantics, showing immediately that the $\beta\eta$-laws are respected by the interpretation. It is routine to show inductively that these interpretations

are smooth. Thus every term $\Gamma \vdash t : \tau$ in the extended language is interpreted as a smooth function $[\![t]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$ between diffeological spaces.

In this section we focused on a language with lists, but other inductive types are easily interpreted in the category of diffeological spaces in much the same way.

Remark 2.4.1 ()**.** *The categorically minded reader may regard this as a consequence of* **Diff** *being a category of concrete sheaves, a particular case of Grothendieck quasitopos (see e.g. Baez and Hoffnung [2011]). Indeed, a Grothendieck quasitopos is complete and cocomplete and so the category of algebras for the endofunctor associated to any algebraic data type has an initial object. The set-like description comes from these categories being concrete.*

2.5 Categorical analysis of forward AD and its correctness

 This section has three parts. First, we give a categorical account of the functoriality of AD (Example 2.5.1). Then we introduce our gluing construction, and relate it to the correctness of AD (Diagram 2.3). Finally, we state and prove a correctness theorem for all first-order types by considering a category of manifolds (Theorem 2.5.6).

2.5.1 Syntactic categories.

$$\Gamma \vdash \mathbf{case}\,\langle t_1,\dots,t_n\rangle\,\mathbf{of}\,\langle x_1,\dots,x_n\rangle \to s = s\big[{}^{t_1}\!/_{x_1},\dots,{}^{t_n}\!/_{x_n}\big] : \tau$$
$$\Gamma \vdash s\big[{}^{t}\!/_{\mathbf{y}}\big] \overset{\#x_1,\dots,x_n}{=} \mathbf{case}\,t\,\mathbf{of}\,\langle x_1,\dots,x_n\rangle \to s\big[{}^{\langle x_1,\dots,x_n\rangle}\!/_{\mathbf{y}}\big] : \tau$$
$$\Gamma \vdash \mathbf{case}\,\ell_{\mathsf{i}}\,t\,\mathbf{of}\,\{\ell_1\,x_1 \to s_1 \;\big|\; \cdots \;\big|\; \ell_{\mathsf{n}}\,x_n \to s_n\} = s_i\big[{}^{t}\!/_{x_i}\big] : \tau$$
$$\Gamma \vdash s\big[{}^{t}\!/_{\mathbf{y}}\big] \overset{\#x_1,\dots,x_n}{=} \mathbf{case}\,t\,\mathbf{of}\,\{\ell_1\,x_1 \to s\big[{}^{\ell_1\,x_1}\!/_{\mathbf{y}}\big] \;\big|\; \cdots \;\big|\; \ell_{\mathsf{n}}\,x_n \to s\big[{}^{\ell_{\mathsf{n}}\,x_n}\!/_{\mathbf{y}}\big]\} : \tau$$
$$\Gamma \vdash \mathbf{fold}\,(x_1,x_2).t\,\mathbf{over}\,[\,]\,\mathbf{from}\,r = r : \tau$$
$$\Gamma \vdash \mathbf{fold}\,(x_1,x_2).t\,\mathbf{over}\,s_1 :: s_2\,\mathbf{from}\,r = t\big[{}^{s_1}\!/_{x_1},{}^{\mathbf{fold}\,(x_1,x_2).t\,\mathbf{over}\,s_2\,\mathbf{from}\,r}\!/_{x_2}\big] : \tau$$
$$\Gamma \vdash u = s\big[{}^{[\,]}\!/_{\mathbf{y}}\big], r\big[{}^{s}\!/_{x_2}\big] = s\big[{}^{x_1::\mathbf{y}}\!/_{\mathbf{y}}\big] \Rightarrow s\big[{}^{t}\!/_{\mathbf{y}}\big] \overset{\#x_1,x_2}{=} \mathbf{fold}\,(x_1,x_2).r\,\mathbf{over}\,t\,\mathbf{from}\,u : \tau$$
$$\Gamma \vdash (\lambda x.t)\,s = t\big[{}^{s}\!/_{x}\big] : \tau$$
$$\Gamma \vdash t \overset{\#x}{=} \lambda x.t\,x : \tau_1 \to \tau_2$$

We write $\overset{\#x_1,\dots,x_n}{=}$ *to indicate that the variables are fresh in the left hand side.*

Figure 2.9: Standard $\beta\eta$-laws (e.g. Pitts [1995]) for products, functions, variants and lists.

Our language induces a syntactic category as follows.

Definition 2.5.1. *Let* **Syn** *be the category whose objects are types, and where a morphism $\tau \to \sigma$ is a term in context $x : \tau \vdash t : \sigma$ modulo the $\beta\eta$-laws (Fig. 2.9). Composition is by substitution.*

For simplicity, we do not impose arithmetic identities such as $x + y = y + x$ in **Syn**. As is standard, this category has the following universal property.

Lemma 2.5.1 (e.g. Pitts [1995])**.** *For every bicartesian closed category* $\mathbf{C}$ *with list objects, and every object $F(\mathbf{real}) \in \mathbf{C}$ and morphisms $F(\underline{c}) \in \mathbf{C}(1, F(\mathbf{real}))$, $F(+), F(*) \in \mathbf{C}(F(\mathbf{real}) \times F(\mathbf{real}), F(\mathbf{real}))$, $F(\varsigma) \in \mathbf{C}(F(\mathbf{real}), F(\mathbf{real}))$ in* $\mathbf{C}$*, there is a unique functor $F : \mathbf{Syn} \to \mathbf{C}$ respecting the interpretation and preserving the bicartesian closed structure as well as list objects. In particular, $F(\{\ell_1\, \tau_1 \mid \ldots \mid \ell_{\mathsf{n}}\, \tau_n\}) = \sum_{1 \leq i \leq n} F(\tau_i)$.*

Proof. The functor $F : \mathbf{Syn} \to \mathbf{C}$ is a canonical denotational semantics for the language, interpreting types as objects of $\mathbf{C}$ and terms as morphisms. For instance, $F(\tau \to \sigma) \stackrel{\text{def}}{=} (F\tau \to F\sigma)$, the function space in the category $\mathbf{C}$, and $F(t\, s)$ is the composite $F(t\, s) \stackrel{\text{def}}{=} (Ft, Fs); \mathit{eval}$.

When $\mathbf{C} = \mathbf{Diff}$, the denotational semantics of the language in diffeological spaces (Sections 2.3, 2.4.2) can be understood as the unique structure preserving functor $[\![-]\!] : \mathbf{Syn} \to \mathbf{Diff}$ satisfying $[\![\mathbf{real}]\!] = \mathbb{R}$, $[\![\varsigma]\!] = \varsigma$ and so on. □

Remark 2.5.1. *More generally, this lemma will generalize without further issues to a language with an arbitrary set of basic types $B \in \mathcal{B}$, for which one must provide $F(B)$. It will also generalize to the language having arbitrary primitives $c : \tau \to \sigma]$ (for constants take $\tau = 1$) for which one must provide $F(c) \in \mathbf{C}(F(\tau), F(\sigma))$. Here, $F(\tau)$ is defined as the structure preserving extension of the assignment F. For instance, if $\tau = B_1 \times B_2$ for two basic types B_i, then $F(\tau) \stackrel{\text{def}}{=} F(B_1) \times F(B_2)$. It will also straightforwardly extend to a language admitting arbitrary algebraic data types. With a little more care, this can also be extended to a language with partiality and recursion [Vákár, 2020a][Section 3.3].*

Example 2.5.1 (Canonical definition of forward-mode AD)**.** *The forward AD macro $\overrightarrow{\mathcal{D}}$ (§2.2,2.4.1) arises as a canonical Cartesian closed functor on* **Syn**. *Consider the unique Cartesian closed functor $F : \mathbf{Syn} \to \mathbf{Syn}$ such that $F(\mathbf{real}) = \mathbf{real} * \mathbf{real}$, and*

$$
\begin{aligned}
&F(\underline{c}) = \overrightarrow{\mathcal{D}}(\underline{c})\\
&F(\varsigma) = x : F(\mathbf{real}) \vdash \overrightarrow{\mathcal{D}}(\varsigma(x)) : F(\mathbf{real})\\
&F(+) = z : F(\mathbf{real}) \times F(\mathbf{real}) \vdash \mathbf{case}\, z\, \mathbf{of}\, \langle x, y\rangle \to \overrightarrow{\mathcal{D}}(x + y) : F(\mathbf{real})\\
&F(*) = z : F(\mathbf{real}) \times F(\mathbf{real}) \vdash \mathbf{case}\, z\, \mathbf{of}\, \langle x, y\rangle \to \overrightarrow{\mathcal{D}}(x * y) : F(\mathbf{real})
\end{aligned}
$$

Then for any type τ, $F(\tau) = \overrightarrow{\mathcal{D}}(\tau)$, and for any term $x : \tau \vdash t : \sigma$, $F(t) = \overrightarrow{\mathcal{D}}(t)$ as morphisms $F(\tau) \to F(\sigma)$ in the syntactic category.

Example 2.5.2 (Soundness of the denotational semantics)**.** *Our interpretation $[\![-]\!] : \mathbf{Syn} \to \mathbf{Diff}$ also arises as a canonical Cartesian closed functor. It is given by $[\![\mathbf{real}]\!] = \mathbb{R}$ and the syntactic primitives are interpreted by their corresponding smooth primitives, e.g. $[\![\varsigma]\!] = \varsigma$. The soundness therefore amounts to* **Diff** *being a bicartesian closed category with list objects.*

2.5.2 Categorical gluing and logical relations.

Gluing is a method for building new categorical models which has been used for many purposes, including logical relations and realizability Mitchell and Scedrov [1992]. Our logical relations argument in the proof of Theorem 2.3.2 can be understood in this setting. In this subsection, for the categorically minded, we explain this, and in doing so we quickly recover a correctness result for the more general language in Section 2.4 and for arbitrary first-order types.

We define a category $\mathbf{Gl}_U$ whose objects are triples (X, X', S) where X and X' are diffeological spaces and $S \subseteq \mathcal{P}^U_X \times \mathcal{P}^U_{X'}$ is a relation between their U-plots. A morphism $(X, X', S) \to (Y, Y', T)$ is a pair of smooth functions $f\colon X \to Y$, $f'\colon X' \to Y'$, such that if $(g, g') \in S$ then $(g; f, g'; f') \in T$. The idea is that this is a semantic domain where we can simultaneously interpret the language and its automatic derivatives.

Proposition 2.5.2. *The category* $\mathbf{Gl}_U$ *is bicartesian closed, has list objects, and the projection functor* $\mathrm{proj} : \mathbf{Gl}_U \to \mathbf{Diff} \times \mathbf{Diff}$ *preserves this structure.*

Proof. The category $\mathbf{Gl}_U$ is a full subcategory of the comma category $\mathrm{id}_{\mathbf{Set}} \downarrow \mathbf{Diff}(U, -) \times \mathbf{Diff}(U, -)$. The result thus follows by the general theory of categorical gluing (e.g. [Johnstone et al., 2007, Lemma 15]). As every category defined by a universal property, $\mathbf{Gl}_U$ is defined up to unique isomorphism. Another way to describe an object in $\mathbf{Gl}_U$ is as a pair (X, S) where X is an object of the product category $\mathbf{Diff} \times \mathbf{Diff}$, i.e. X is a pair (X_1, X_2) of diffeological spaces, and $S \subseteq \mathcal{P}^U_X \times \mathcal{P}^U_{X'}$. For convenience, let us write $\mathbf{Diff}^2 \stackrel{\text{def}}{=} \mathbf{Diff} \times \mathbf{Diff}$. Let us further write $f : U \to \mathbf{Diff}^2(X, Y)$ for a pair of morphisms $f_1 : U \to \mathbf{Diff}(X, Y), f_2 : U \to \mathbf{Diff}(X, Y)$ and we call f a plot if both f_1 and f_2 are.

Then, the function space $[(X, S) \Rightarrow (Y, T)]$ in Gl_U is given by

$$\Big(\mathbf{Diff}^2(X, Y), \{f : U \to \mathbf{Diff}^2(X, Y) \text{ plot} \mid \forall r \in U.\, \forall g \in S.\, \lambda r.\, f(r)(g(r)) \in T\}\Big)$$

The product space $[(X, S) \times (Y, T)]$ is given by

$$\Big(X \times Y, \{f : U \to X \times Y \text{ plot} \mid \forall r \in U.\, \pi_1 f(r) \in S, \pi_2 f(r) \in T\}\Big)$$

The coproduct space $[(X, S) + (Y, T)]$ is given by

$$\Big(X + Y, \{f : U \to X + Y \text{ plot} \mid f = \iota_X \circ g, g \in S \text{ or } f = \iota_Y \circ g, g \in T\}\Big)$$

where ι_X is the inclusion $X \to X + Y$.
The list object space $[\mathbf{list}(X), T]$ on $[X, S]$ is defined recursively by

$$\Big(\mathbf{list}(X), \{f : U \to \mathbf{list}(X) \text{ plot} \mid f = \iota_1 \text{ or } f = \iota_{X \times \mathbf{list}(X)} \circ g, g \in S \times T\}\Big)$$

where by $S \times T$ we mean $\{f : U \to X \times \mathbf{list}(X) \mid \pi_1 \circ f \in S, \pi_2 \circ f \in T\}$. □

We give a semantics $(\!|-|\!) = ((\!|-|\!)_0, (\!|-|\!)_1, S_-)$ for the language in $\mathbf{Gl}_\mathbb{R}$, interpreting types τ as objects $((\!|\tau|\!)_0, (\!|\tau|\!)_1, S_\tau)$, and terms as morphisms. We let $(\!|\mathbf{real}|\!)_0 \stackrel{\text{def}}{=} \mathbb{R}$ and $(\!|\mathbf{real}|\!)_1 \stackrel{\text{def}}{=} \mathbb{R}^2$, with the relation $S_{\mathbf{real}} \stackrel{\text{def}}{=} \{(f, (f, \nabla f)) \mid f : \mathbb{R} \to \mathbb{R} \text{ smooth}\}$. We interpret the constants $\underline{c}$ as pairs $(\!|\underline{c}|\!)_0 \stackrel{\text{def}}{=} \underline{c}$ and $(\!|\underline{c}|\!)_1 \stackrel{\text{def}}{=} (\underline{c}, 0)$, and we interpret $+, \times, \varsigma$ in the standard way (meaning, like $[\![-]\!]$) in $(\!|-|\!)_0$, but according to the derivatives in $(\!|-|\!)_1$, for instance, $(\!|*|\!)_1 : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}^2$ is

$$(\!|*|\!)_1((x, x'), (y, y')) \stackrel{\text{def}}{=} (xy, xy' + x'y).$$

At this point one checks that these interpretations are indeed morphisms in $\mathbf{Gl}_\mathbb{R}$. This amounts to checking that these interpretations are dual numbers representations in the sense of (2.2). The remaining constructions of the language are interpreted using the categorical structure of $\mathbf{Gl}_\mathbb{R}$, following Lemma 2.5.1.

Notice that the diagram below commutes. One can check this by hand or note that it follows from the initiality of **Syn** (Lemma 2.5.1). Indeed, all the functors preserve all the structure, and both paths from **Syn** to **Diff** $\times$ **Diff** agree on base types and primitives. Therefore, both paths from **Syn** to **Diff** $\times$ **Diff** are *the* unique structure-preserving functor with the given choice of interpretation for base types and primitives, and are thus equal.

$$\begin{array}{ccc} \mathbf{Syn} & \xrightarrow{(\mathrm{id}, \overrightarrow{\mathcal{D}}(-))} & \mathbf{Syn} \times \mathbf{Syn} \\ {\scriptstyle (\!|-|\!)}\downarrow & & \downarrow{\scriptstyle [\![-]\!]\times[\![-]\!]} \\ \mathbf{Gl}_\mathbb{R} & \xrightarrow[\mathrm{proj}]{} & \mathbf{Diff} \times \mathbf{Diff} \end{array} \tag{2.3}$$

We thus arrive at a restatement of the correctness theorem (Theorem 2.3.2), which holds even for the extended language with variants and lists, because for any $x_1 : \mathbf{real}, \ldots, x_n : \mathbf{real} \vdash t : \mathbf{real}$, the interpretations $([\![t]\!], [\![\overrightarrow{\mathcal{D}}(t)]\!])$ are in the image of the projection $\mathbf{Gl}_\mathbb{R} \to \mathbf{Diff} \times \mathbf{Diff}$, and hence $[\![\overrightarrow{\mathcal{D}}(t)]\!]$ is a dual number encoding of $[\![t]\!]$.

2.5.3 Correctness at all first-order types, via manifolds.

We now generalize Theorem 2.3.2 to hold at all first-order types, not just the reals.

So far, we have shown that our macro translation (Section 2.6.4) gives correct derivatives to functions of the real numbers, even if other types are involved in the definitions of the functions (Theorem 2.3.2). We can state this formally because functions of the real numbers have well understood derivatives (Section 2.6.1). There are no established mathematical notions of derivatives at higher types, and so we cannot even begin to argue that our syntactic derivatives of functions $(\mathbf{real} \to \mathbf{real}) \to (\mathbf{real} \to \mathbf{real})$ match with some existing mathematical notion.

However, for functions of first-order type, like $\mathbf{list}(\mathbf{real}) \to \mathbf{list}(\mathbf{real})$, there *are* established mathematical notions of derivative, because we can understand $\mathbf{list}(\mathbf{real})$ as the *manifold* of all tuples of reals, and then appeal to the well-known theory of manifolds and jet bundles. We do this now, to achieve a correctness theorem for all first-order types (Theorem 2.5.6). The key high level points are that

- manifolds support a notion of differentiation, and an interpretation of all first-order types, but not an interpretation of higher types;
- diffeological spaces support all types, including higher types, but not an established notion of differentiation in general;
- manifolds and smooth maps embed full and faithfully in diffeological spaces, preserving the interpretation of first-order types, so we can use the two notions together.

We now explain this development in more detail.

For our purposes, a smooth manifold M is a second-countable (4.4.5) Hausdorff (4.4.1) topological space together with a smooth atlas: an open cover $\mathcal{U}$ together with homeomorphisms $\left(\phi_U : U \to \mathbb{R}^{n(U)}\right)_{U \in \mathcal{U}}$ (called charts) such that for all opens $U, V \in \mathcal{U}$ and charts $\phi_U : U \to \mathbb{R}^n, \psi_V : V \to \mathbb{R}^m$, the function $\phi_U^{-1}; \phi_V : \phi_U(U \cap V) \to \mathbb{R}^m$ is smooth. A function $f : M \to N$ between manifolds is smooth if smooth for all charts $\phi_U : U \to \phi_U(U) \subseteq M$ and $\psi_V : V \to \psi_V(V) \subseteq N$ of M and N, respectively, we have that the function $\phi_U^{-1}; f; \psi_V$ is smooth on its domain $(\phi_U)(U \cap f^{-1}(V))$.

Let us write **Man** for this category.

Our manifolds are slightly unusual because different charts in an atlas may have different finite dimension $n(U)$. Thus we consider manifolds with dimensions that are potentially unbounded, albeit locally finite. This does not affect the theory of differential geometry as far as we need it here.

Each open subset of $\mathbb{R}^n$ can be regarded as a manifold. This lets us regard the category of manifolds **Man** as a full subcategory of the category of diffeological spaces. We consider a manifold $(X, \{\phi_U\}_U)$ as a diffeological space with the same carrier set X and where the plots $\mathcal{P}_X^U$ are the smooth functions in $\mathbf{Man}(U, X)$. A function $X \to Y$ is smooth in the sense of manifolds if and only if it is smooth in the sense of diffeological spaces Iglesias-Zemmour [2013].

Remark 2.5.2 ( ). *For the categorically minded reader, this means that we have a full embedding of* **Man** *into* **Diff**. *Moreover, the natural interpretation of the first-order fragment of our language in* **Man** *coincides with that in* **Diff**. *That is, the embedding of* **Man** *into* **Diff** *preserves finite products and countable coproducts (hence initial algebras of polynomial endofunctors).*

Proposition 2.5.3. *Suppose that a type τ is first-order, i.e. it is just built from reals, products, variants, and lists (or, again, arbitrary inductive types), and not function types. Then the diffeological space $[\![\tau]\!]$ is a manifold.*

Proof. This is proved by induction on the structure of types. In fact, one may show that every such $[\![\tau]\!]$ is isomorphic to a manifold of the form $\biguplus_{i=1}^{n} \mathbb{R}^{d_i}$ where the bound n is either finite or ∞, but this isomorphism is typically not an identity function. □

The constraint to first-order types is necessary because, e.g. the space $[\![\mathbf{real} \to \mathbf{real}]\!]$ is not a manifold, because of a Borsuk-Ulam argument.

We recall that the derivative of any morphism $f : M \to N$ of manifolds is given as follows. For each point x in a manifold M, define the *tangent space* $\mathcal{T}_x M$ to be the set $\{\gamma \in \mathbf{Man}(\mathbb{R}, M) \mid \gamma(0) = x\}/\sim$ of equivalence classes $[\gamma]$ of smooth curves γ in M based at x, where we identify $\gamma_1 \sim \gamma_2$ iff $\nabla(\gamma_1; f)(0) = \nabla(\gamma_2; f)(0)$ for all smooth $f : M \to \mathbb{R}$. Observe that the charts around a point x in a manifold M determine the dimension $\dim_x(M)$ of M at x and let $\mathcal{T}_x(M) \stackrel{\text{def}}{=} \mathbb{R}^{\dim_x(M)}$ be the *tangent space* at x. The *tangent bundle* of M is the set $\mathcal{T}(M) \stackrel{\text{def}}{=} \biguplus_{x \in M} \mathcal{T}_x(M)$. The charts of M equip $\mathcal{T}(M)$ with a canonical manifold structure. Then for smooth $f : M \to N$, the derivative $\mathcal{T}(f) : \mathcal{T}(M) \to \mathcal{T}(N)$ is defined as $\mathcal{T}(f)(x, [\gamma]) \stackrel{\text{def}}{=} (f(x), [\gamma; f])$. All told, the derivative is a functor $\mathcal{T} : \mathbf{Man} \to \mathbf{Man}$.

As is standard, we can understand the tangent bundle of a composite space in terms of that of its parts.

Lemma 2.5.4. *There are canonical isomorphisms $\mathcal{T}(\biguplus_{i=1}^{\infty} M_i) \cong \biguplus_{i=1}^{\infty} \mathcal{T}(M_i)$ and $\mathcal{T}(M_1 \times \ldots \times M_n) \cong \mathcal{T}(M_1) \times \ldots \times \mathcal{T}(M_n)$.*

Proof. For products, see e.g. Lee [2013, ex. 3-2]. For disjoint unions, notice that smooth curves into a disjoint union of manifolds always factor over a single inclusion, because $\mathbb{R}$ is connected. □

Lemma 2.5.5. *A smooth function $f : \mathcal{T}(M) \to \mathcal{T}(N)$ is of the form $\mathcal{T}(g)$ for some $g : M \to N$ iff for all smooth curves $\gamma : \mathbb{R} \to M$, we have that $\mathcal{T}(\gamma); f = \mathcal{T}(\gamma; g)$.*

Proof. This is well-known. It follows immediately from the (geometric) definition of $\mathcal{T}(M)$ as equivalence classes $[\gamma]$ of curves γ that we are using. □

We define a canonical isomorphism $\phi_\tau^{\overrightarrow{\mathcal{D}}\mathcal{T}} : [\![\overrightarrow{\mathcal{D}}(\tau)]\!] \to \mathcal{T}([\![\tau]\!])$ for every type τ, by induction on the structure of types. We let $\phi_{\mathbf{real}}^{\overrightarrow{\mathcal{D}}\mathcal{T}} : [\![\overrightarrow{\mathcal{D}}(\mathbf{real})]\!] \to \mathcal{T}([\![\mathbf{real}]\!])$ be given by $\phi_{\mathbf{real}}^{\overrightarrow{\mathcal{D}}\mathcal{T}}(x, x') \stackrel{\text{def}}{=} (x, [t \mapsto x + x't])$. For the other types, we use Lemma 2.5.4. We can now phrase correctness at all first-order types.

Theorem 2.5.6 (Semantic correctness of $\overrightarrow{\mathcal{D}}$ (full))**.** *For any ground τ, any first-order context Γ and any term $\Gamma \vdash t : \tau$, the syntactic translation $\overrightarrow{\mathcal{D}}$ coincides with the tangent bundle functor, modulo these canonical isomorphisms:*

$$\begin{array}{ccc} [\![\overrightarrow{\mathcal{D}}(\Gamma)]\!] & \xrightarrow{[\![\overrightarrow{\mathcal{D}}(t)]\!]} & [\![\overrightarrow{\mathcal{D}}(\tau)]\!] \\ {\scriptstyle \phi_\Gamma^{\overrightarrow{\mathcal{D}}\mathcal{T}}}\downarrow \cong & & \cong \downarrow{\scriptstyle \phi_\tau^{\overrightarrow{\mathcal{D}}\mathcal{T}}} \\ \mathcal{T}([\![\Gamma]\!]) & \xrightarrow[\mathcal{T}([\![t]\!])]{} & \mathcal{T}([\![\tau]\!]) \end{array}$$

Proof. For any curve $\gamma \in \mathbf{Man}(\mathbb{R}, M)$, let $\bar{\gamma} \in \mathbf{Man}(\mathbb{R}, \mathcal{T}(M))$ be the tangent curve, given by $\bar{\gamma}(x) = (\gamma(x), [t \mapsto \gamma(x + t)])$. First, we note that a smooth map $h : \mathcal{T}(M) \to \mathcal{T}(N)$ is of the form $\mathcal{T}(g)$ for some $g : M \to N$ if for all smooth curves $\gamma : \mathbb{R} \to M$ we have $\bar{\gamma}; h = \overline{(\gamma; g)} : \mathbb{R} \to \mathcal{T}(N)$. This generalizes (2.2). Second, for any first-order type τ, $S_{[\![\tau]\!]} = \{(f, \tilde{f}) \mid \tilde{f}; \phi_\tau^{\overrightarrow{\mathcal{D}}\mathcal{T}} = \bar{f}\}$. This is shown by induction on the structure of types. We conclude the theorem from diagram (2.3), by putting these two observations together.

□

2.6 Higher-order AD via Taylor approximations

The source and target languages of the AD macro are the same, so we can always compute higher derivatives by applying the AD macro several times. However, unless further optimisations are performed, this will introduce redundancies.

Notably, as our functions are smooth, partial derivatives commute: $\frac{\partial}{\partial x_i}\frac{\partial}{\partial x_j}f = \frac{\partial}{\partial x_j}\frac{\partial}{\partial x_x}f$ for all smooth function f. The symmetries of the higher-order Jacobian tensor are complex in general, and it is not immediately clear what optimisations would be sufficient to avoid these redundancies.

Alternatively, remember that dual numbers represent the algebra of truncated Taylor series of order 1. By truncating at a higher-order k, we get a more complex algebra, allowing us to compute k-th order derivatives. This is the idea of jet-based AD which we expore in this section. As we will see, even though the combinatorics is much more involved in the new chain-rule, the rest of our analysis from Sections 2.2-2.5 carries through seamlessly in this richer context.

2.6.1 higher-order differentiation: the Faà di Bruno formula and Taylor approximations.

We now generalize the dual number approach (Section 2.2.1) in two directions:

- We look for the best local approximations to f with polynomials of some order R, generalizing the use of linear functions ($R = 1$). This will allow us to compute higher derivatives in one AD pass.

- We can work directly with multivariate functions $\mathbb{R}^k \to \mathbb{R}$ instead of functions of one variable $\mathbb{R} \to \mathbb{R}$ ($k = 1$). This will first allow us both to compute several partial derivatives in one AD pass, but it will also allow us to compute non-diagonal higher-order derivatives in one pass. For instance, with $k = 2$, this will allow us to compute a Hessian term of the form $w^T H v$ by pushing forward two vectors v and w (e.g. hot vectors), instead of merely $v^T H v$.

We will write $\boldsymbol{\alpha}$ for the tuple $(\alpha_1, \ldots, \alpha_k)$.

To make this precise, we recall that, given a smooth function $f : \mathbb{R}^k \to \mathbb{R}$ and a natural number $R \geq 0$, the *R-th order Taylor approximation of f at $a \in \mathbb{R}^k$* is defined in terms of the partial derivatives of f:

$$\begin{array}{ccc} \mathbb{R}^k & \to & \mathbb{R} \\ x & \mapsto & \displaystyle\sum_{\{\boldsymbol{\alpha}\in\mathbb{N}^k \mid \sum_{i=1}^k \alpha_i \leq R\}} \frac{1}{\alpha_1! \cdot \ldots \cdot \alpha_k!} \frac{\partial^{\alpha_1+\ldots+\alpha_k} f}{\partial x_1^{\alpha_1} \cdots \partial x_k^{\alpha_k}}(a) \cdot \prod_{i=1}^k (x_i - a_i)^{\alpha_i} \end{array}$$

The partial derivatives of f are evaluated at a point a, and are therefore simply reals. Overall, the formula is an R-th order real polynomial in the x_i. Similarly to the case of first-order derivatives, we can recover the partial derivatives of f

up to the R-th order from its Taylor approximation by evaluating the series at basis vectors. See Section 2.6.2 below for an example.

In general, we would have k^R R-th order partial derivatives. However, recall that the ordering of partial derivatives does not matter for smooth functions by the Schwartz-Clairaut theorem. So instead, we only need to account for an unordered sequence of natural numbers $\alpha_1, \ldots, \alpha_k$ accounting for the order of each partial derivative. For R-th order partial derivatives, we must have $\sum_i \alpha_i = R$. This is known as the multiset counting number, and there are $\binom{R+k-1}{k-1}$ such tuples. So there will be $\binom{R+k-1}{k-1}$ R-th order partial derivatives, and altogether there are $\sum_{j=0}^{R} \binom{j+k-1}{k-1} = \binom{R+k}{k}$ summands in the R-th order Taylor approximation, by the Hockey-stick identity. Since there are $\binom{R+k}{k}$ partial derivatives of f_i of order $\leq R$, we can store them in the Euclidean space $\mathbb{R}^{\binom{R+k}{k}}$. The same combinatorics shows that this can also be regarded as the space of k-variate polynomials of degree $\leq R$, with the identification $\frac{\partial^\alpha}{\partial x_i^\alpha} \mapsto x_i^\alpha$.

We will use a convention of coordinates

$$(y_{\alpha_1 \ldots \alpha_k} \in \mathbb{R})_{\boldsymbol{\alpha} \in \{(\alpha_1,\ldots,\alpha_k) \in \mathbb{N}^k \mid 0 \leq \alpha_1 + \ldots + \alpha_k \leq R\}}$$

where $y_{\boldsymbol{\alpha}}$ is intended to represent a partial derivative $\frac{\partial^{\alpha_1+\ldots+\alpha_k} f}{\partial x_1^{\alpha_1} \cdots \partial x_k^{\alpha_k}}(a)$ for some function $f : \mathbb{R}^k \to \mathbb{R}$. We will choose these coordinates in lexicographic order of the multi-indices $(\alpha_1, \ldots, \alpha_k)$, that is, the indexes in the Euclidean space $\mathbb{R}^{\binom{R+k}{k}}$ will typically range from $(0, \ldots, 0)$ to $(R, 0, \ldots, 0)$.

The (k, R)-*Taylor representation* of a function $g : \mathbb{R}^n \to \mathbb{R}$ is a function $h : \left(\mathbb{R}^{\binom{R+k}{k}}\right)^n \to \mathbb{R}^{\binom{R+k}{k}}$ that transforms the partial derivatives of $f : \mathbb{R}^k \to \mathbb{R}^n$ of order $\leq R$ under postcomposition with g:

$$\left(\left(\frac{\partial^{\alpha_1+\ldots+\alpha_k} f_j(x)}{\partial x_1^{\alpha_1} \cdots \partial x_k^{\alpha_k}}\right)_{\boldsymbol{\alpha}=(0,\ldots,0)}^{(R,0,\ldots,0)}\right)_{j=1}^{n} ; h = \left(\left(\frac{\partial^{\alpha_1+\ldots+\alpha_k}((f_1,\ldots,f_n);g)(x)}{\partial x_1^{\alpha_1} \ldots \partial x_k^{\alpha_k}}\right)_{\boldsymbol{\alpha}=(0,\ldots,0)}^{(R,0,\ldots,0)}\right)_{j=1}^{n} \tag{2.4}$$

Thus, the Taylor representation generalizes the dual numbers representation ($R = k = 1$). The geometric picture is revised as follows. In the standard dual-number case, we are pushing forward a curve $\gamma : \mathbb{R} \to M$ along a function $M \to N$ (where M, N are smooth manifolds), and from the curve γ we only remember its value and velocity at 0, i.e. $\gamma(0)$ and $\gamma'(0)$. With a higher R, we remember higher derivatives of γ. With a higher k, γ can be seen as a k-dimensional surface on M, and in this case we remember k directional derivatives of γ at 0.

To explicitly calculate the Taylor representation for a smooth function, we recall a generalization of the chain rule to higher derivatives. The chain rule tells us how the coefficients of linear approximations transform under composition of the functions. The *Faà di Bruno formula* [Savits, 2006, Encinas and Masque, 2003, Constantine and Savits, 1996] tells us how coefficients of Taylor approximations – that is, higher derivatives – transform under composition. The formula has a

complex combinatorics, so let us introduce it step by step. First, recall the simple chain rule for functions $f, g : \mathbb{R} \to \mathbb{R}$

$$(f \circ g)' = (f' \circ g) \cdot g'$$

and note that it introduces a product, so that when differentiating once more, by the product rule we get:

$$\begin{aligned}(f \circ g)'' &= ((f' \circ g) \cdot g')' \\ &= (f' \circ g)' \cdot g' + (f' \circ g) \cdot g'' \\ &= (f'' \circ g) \cdot g' \cdot g' + (f' \circ g) \cdot g''\end{aligned}$$

More generally, when differentiating n-times, we will need the general Leibniz rule which generalizes the product rule:

$$(f \cdot g)^{(n)} = \sum_{k_1+k_2=n} \frac{n!}{k_1!k_2!} f^{k_1} \cdot g^{k_2}$$

In fact, as the second order derivative already introduces the product of 3 terms, we will need the general Leibniz rule for m factors:

$$(f_1 \cdot \ldots \cdot f_m)^{(n)} = \sum_{k_1+\ldots+k_m=n} \frac{n!}{k_1! \ldots k_m!} \prod_{1 \le i \le m} f_i^{k_i}$$

Using this, one can show that the higher-order chain rule for functions $f, g : \mathbb{R} \to \mathbb{R}$ is given by

$$(f; g)^{(n)} = \sum_{1 \le k \le n} (f; g^{(k)}) \cdot \sum_{\{\mathbf{e},\mathbf{k} | \sum_i e_i = k, \sum_i e_i k_i = n\}} \prod_{i=1}^{n} \frac{n!}{e_i!} (f^{(k_i)})^{e_i}$$

We can rewrite it in a more compact form as follows:

$$(f; g)^{(n)} = \sum_{1 \le k \le n} G(k, f, g) \cdot F(n, k, f)$$

where $G(k, f, g) = f; g^{(k)}$ and $F(n, k, f) = \sum_{\{\mathbf{e},\mathbf{k} | \sum_i e_i = k, \sum_i e_i k_i = n\}} \prod_{i=1}^{n} \frac{n!}{e_i!} (f^{(k_i)})^{e_i}$.

More generally, the multivariate form from Savits [2006, Theorem 2.1] is given as follows. Given functions $f = (f_1, \ldots, f_l) : \mathbb{R}^k \to \mathbb{R}^l$ and $g : \mathbb{R}^l \to \mathbb{R}$, for $\boldsymbol{\alpha} \in \mathbb{N}^k$ with $\alpha_1 + \ldots + \alpha_k > 0$,

$$\frac{\partial^{\alpha_1+\ldots+\alpha_k}(f; g)}{\partial x_1^{\alpha_1} \cdots \partial x_k^{\alpha_k}} = \sum_{\{\boldsymbol{\beta} \in \mathbb{N}^l \mid 1 \le \sum_i \beta_i \le \sum_i \alpha_i\}} G(\boldsymbol{\beta}, f, g) \cdot F(\boldsymbol{\alpha}, \boldsymbol{\beta}_1, f)$$

where $G(\boldsymbol{\beta}, f, g) = f; \frac{\partial^{\beta_1+\ldots+\beta_l} g}{\partial y_1^{\beta_1} \cdots \partial y_l^{\beta_l}}$ and

$$F(\boldsymbol{\alpha}, \boldsymbol{\beta}, f) = \sum_{\{(\mathbf{e}^1,\ldots,\mathbf{e}^q) \in (\mathbb{N}^l)^q | e_j^1 + \ldots + e_j^q = \beta_j, (e_1^1+\ldots+e_l^1)\cdot\alpha_i^1 + \ldots + (e_1^q+\ldots+e_l^q)\cdot\alpha_i^q = \alpha_i\}} \prod_{r=1}^{q} \prod_{j=1}^{l} \frac{\alpha_1! \cdot \ldots \cdot \alpha_k!}{e_j^r!} \left(\frac{1}{\alpha_1^r! \cdot \ldots \cdot \alpha_k^r!} \frac{\partial^{\alpha_1^r+\cdots+\alpha_k^r} f_j}{\partial^{\alpha_1^r} x_1 \cdots \partial^{\alpha_k^r} x_k} \right)^{e_j^r}$$

where $(\alpha_1^1, \ldots, \alpha_k^1), \ldots, (\alpha_1^q, \ldots, \alpha_k^q) \in \mathbb{N}^k$ are an enumeration of all the vectors $(\alpha_1^r, \ldots, \alpha_k^r)$ of k natural numbers such that $\alpha_j^r \leq \alpha_j$ and $\alpha_1^r + \ldots + \alpha_k^r > 0$ and we write q for the number of such vectors. The details of this formula reflect the complicated combinatorics that arise from repeated applications of the chain and product rules for differentiation that one uses to prove it. Conceptually, however, it is rather straightforward: it tells us that the coefficients of the R-th order Taylor approximation of $f; g$ can be expressed exclusively in terms of those of f and g.

Thus, the Faà di Bruno formula uniquely determines the Taylor approximation $h : \left(\mathbb{R}^{\binom{R+k}{k}}\right)^n \to \mathbb{R}^{\binom{R+k}{k}}$ in terms of the derivatives of $g : \mathbb{R}^n \to \mathbb{R}$ of order $\leq R$, and we can also recover all such derivatives from h.

2.6.2 Example: a two-dimensional second order Taylor series

As an example, we can specialize the Faà di Bruno formula above to the second order Taylor series of a function $f : \mathbb{R}^2 \to \mathbb{R}^l$ and its behaviour under postcomposition with a smooth function $g : \mathbb{R}^l \to \mathbb{R}$:

$$\frac{\partial^2 (f;g)(x)}{\partial x_i \partial x_{i'}}(a) = \sum_{j=1}^{l} \frac{\partial g(y)}{\partial y_j}(f(a)) \frac{\partial^2 f_j(x)}{\partial x_i \partial x_{i'}}(a) + \sum_{j,j'=1}^{l} \frac{\partial^2 g(y)}{\partial y_j \partial y_{j'}}(f(a)) \frac{\partial f_{j'}(x)}{\partial x_i}(a) \frac{\partial f_j(x)}{\partial x_{i'}}(a),$$

where $i, i' \in \{1, 2\}$ might either coincide or be distinct.

Rather than working with the full $(2,2)$-Taylor representation of g, we ignore the non-mixed second order derivatives $y_{02}^j = \frac{\partial^2 f_j(x)}{\partial x_2^2}$ and $y_{20}^j = \frac{\partial^2 f_j(x)}{\partial x_1^2}$ for the moment, and we represent the derivatives of order ≤ 2 of $f_j : \mathbb{R}^2 \to \mathbb{R}$ (at some point a) as the numbers

$$(y_{00}^j, y_{01}^j, y_{10}^j, y_{11}^j) = \left(f_j(a), \frac{\partial f_j(x)}{\partial x_2}(a), \frac{\partial f_j(x)}{\partial x_1}(a), \frac{\partial^2 f_j(x)}{\partial x_1 \partial x_2}(a)\right) \in \mathbb{R}^4$$

and we can choose a similar representation for the derivatives of $(f; g)$. Then, we observe that the Faà di Bruno formula induces the function $h : (\mathbb{R}^4)^l \to \mathbb{R}^4$

$$h((y_{00}^1, y_{01}^1, y_{10}^1, y_{11}^1), \ldots, (y_{00}^l, y_{01}^l, y_{10}^l, y_{11}^l)) =$$
$$\begin{pmatrix} g(y_{00}^1, \ldots, y_{00}^l) \\ \sum_{j=1}^l \frac{\partial g(y^1, \ldots, y^l)}{\partial y_j}(y_{00}^1, \ldots, y_{00}^l) \cdot y_{01}^j \\ \sum_{j=1}^l \frac{\partial g(y^1, \ldots, y^l)}{\partial y_j}(y_{00}^1, \ldots, y_{00}^l) \cdot y_{10}^j \\ \sum_{j=1}^l \frac{\partial g(y^1, \ldots, y^l)}{\partial y_j}(y_{00}^1, \ldots, y_{00}^l) \cdot y_{11}^j + \sum_{j,j'=1}^l \frac{\partial^2 g(y^1, \ldots, y^l)}{\partial y_j \partial y_{j'}}(y_{00}^1, \ldots, y_{00}^l) \cdot y_{10}^j \cdot y_{01}^{j'} \end{pmatrix}.$$

In particular, we can note that

$$h((y_{00}^1, y_{01}^1, y_{10}^1, 0), \ldots, (y_{00}^l, y_{01}^l, y_{10}^l, 0)) = \begin{pmatrix} g(y_{00}^1, \ldots, y_{00}^l) \\ \sum_{j=1}^l \frac{\partial g(y^1, \ldots, y^l)}{\partial y_j}(y_{00}^1, \ldots, y_{00}^l) \cdot y_{01}^j \\ \sum_{j=1}^l \frac{\partial g(y^1, \ldots, y^l)}{\partial y_j}(y_{00}^1, \ldots, y_{00}^l) \cdot y_{10}^j \\ \sum_{j,j'=1}^l \frac{\partial^2 g(y^1, \ldots, y^l)}{\partial y_j \partial y_{j'}}(y_{00}^1, \ldots, y_{00}^l) \cdot y_{10}^j \cdot y_{01}^{j'} \end{pmatrix}.$$

We see that we can use this method to calculate any directional first and second order derivative of g in one pass. For example, if $l = 3$, so $g : \mathbb{R}^3 \to \mathbb{R}$, then the last component of $h((x, x', x'', 0), (y, y', y'', 0), (z, z', z'', 0))$ is the result of taking the first derivative in direction (x', y', z') and the second derivative in direction (x'', y'', z''), and evaluating at (x, y, z).

In the proper Taylor representation, we explicitly include the non-mixed second order derivatives as inputs and outputs, leading to a function $h' : (\mathbb{R}^6)^l \to \mathbb{R}^6$. Above, we have followed a common trick to avoid some unnecessary storage and computation, since these extra inputs and outputs are not required for computing the second order derivatives of g. For instance, if $l = 2$ then the last component of $h((x, 1, 1, 0), (y, 0, 0, 0))$ computes $\frac{\partial^2 g(x,y)}{\partial x^2}(x, y)$.

2.6.3 Example: a one-dimensional second order Taylor series

As opposed to (2,2)-AD, (1,2)-AD computes the first and second order derivatives in the same direction. For example, if $g : \mathbb{R}^2 \to \mathbb{R}$ is a smooth function, then $h : (\mathbb{R}^3)^2 \to \mathbb{R}^3$. An intuition for h can be given in terms of triple numbers. The transformed function operates on triples of numbers, (x, x', x''), and it is common to think of such a triple as $x + x'\epsilon + x''\epsilon^2$ for an 'infinitesimal' ϵ which has the property that $\epsilon^3 = 0$. For instance, we have

$$h((x_1, 1, 0), (x_2, 0, 0)) = (g(x_1, x_2), \frac{\partial g(x, x_2)}{\partial x}(x_1), \frac{\partial^2 g(x, x_2)}{\partial x^2}(x_1))$$
$$h((x_1, 0, 0), (x_2, 1, 0)) = (g(x_1, x_2), \frac{\partial g(x_1, x)}{\partial x}(x_2), \frac{\partial^2 g(x_1, x)}{\partial x^2}(x_2))$$
$$h((x_1, 1, 0), (x_2, 1, 0)) =$$
$$(g(x_1, x_2), \frac{\partial g(x_1, x)}{\partial x}(x_2), \frac{\partial^2 g(x, x_2)}{\partial x^2}(x_1) + \frac{\partial^2 g(x_1, x)}{\partial x^2}(x_2) + 2\frac{\partial^2 g(x, y)}{\partial x \partial y}(x_1, x_2))$$

We see that we directly get non-mixed second-order partial derivatives but not the mixed-ones. We can recover $\frac{\partial^2 g(x,y)}{\partial x \partial y}(x_1, x_2)$ as

$$\frac{1}{2}(h((x_1, 1, 0), (x_2, 1, 0)) - h((x_1, 1, 0), (x_2, 0, 0)) - h((x_1, 0, 0), (x_2, 1, 0)))$$

More generally, if $g : \mathbb{R}^l \to \mathbb{R}$, then $h : (\mathbb{R}^3)^l \to \mathbb{R}^3$ satisfies:

$$h((x_1, x_1', 0), \ldots, (x_l, x_l', 0)) = \begin{pmatrix} g(x_1, \ldots, x_l) \\ \sum_{i=1}^{l} \frac{\partial g(x_1, \ldots, x_l)}{\partial x_i}(x_1, \ldots, x_l) \cdot x_i' \\ \sum_{i,j=1}^{l} \frac{\partial^2 g(x_1, \ldots, x_l)}{\partial x_i \partial x_j}(x_1, \ldots, x_l) \cdot x_i' \cdot x_j' \end{pmatrix}.$$

As we have seen with the example of $\frac{\partial^2 g(x,y)}{\partial x \partial y}(x_1, x_2)$, we can more generally always recover the mixed second order partial derivatives with several calls to h on different arguments (3 in the case of a second-order derivative). This is therefore different from the (2,2) method, which was more direct as it only required one call to h.

$$
\begin{array}{lll}
\overrightarrow{\mathcal{D}}(\tau \to \sigma) & \stackrel{\text{def}}{=} & \overrightarrow{\mathcal{D}}(\tau) \to \overrightarrow{\mathcal{D}}(\sigma) \\
\overrightarrow{\mathcal{D}}(\tau_1 \times \ldots \times \tau_n) & \stackrel{\text{def}}{=} & \overrightarrow{\mathcal{D}}(\tau_1) \times \ldots \times \overrightarrow{\mathcal{D}}(\tau_n) \\
\overrightarrow{\mathcal{D}}(\mathbf{real}) & \stackrel{\text{def}}{=} & \mathbf{real}^{\binom{R+k}{k}} \quad \text{(type of tuples of reals of length } \tbinom{R+k}{k}\text{)} \\
\overrightarrow{\mathcal{D}}(\{\ell_1\, \tau_1 \mid \ldots \mid \ell_{\mathsf{n}}\, \tau_n\}) & \stackrel{\text{def}}{=} & \{\ell_1\, \overrightarrow{\mathcal{D}}(\tau_1) \mid \ldots \mid \ell_{\mathsf{n}}\, \overrightarrow{\mathcal{D}}(\tau_n)\} \\
\overrightarrow{\mathcal{D}}(\mathbf{list}(\tau)) & \stackrel{\text{def}}{=} & \mathbf{list}(\overrightarrow{\mathcal{D}}(\tau))
\end{array}
$$

Figure 2.10: Higher-order AD translation on types

Remark 2.6.1. *In the rest of this section, we study forward-mode (k, R)-automatic differentiation for a language with higher-order functions. The reader may like to fix $k = R = 1$ for the standard automatic differentiation with first-order derivatives, based on dual numbers, that we have seen in Sections 2.2-2.5.*

2.6.4 AD for higher-order derivatives

The aim of higher-order forward-mode AD is to find the (k, R)-Taylor representation of a function by syntactic manipulations, for some choice of (k, R) that we fix. For our simple language, we implement this as the following inductively defined macro $\overrightarrow{\mathcal{D}}_{(k,R)}$ on both types and terms. For the sake of legibility, we simply write $\overrightarrow{\mathcal{D}}_{(k,R)}$ as $\overrightarrow{\mathcal{D}}$ here and leave the dimension k and order R of the Taylor representation implicit. The following definition is for general k and R, but we treat specific cases afterwards in Examples 2.6.1 and 2.6.2.

Here, $(\partial_{\beta_1 \cdots \beta_n} op)(x_1, \ldots, x_n)$ are some chosen terms of type **real** in the language with free variables from $x_1, \ldots, x_n$. We think of these terms as implementing the partial derivative $\frac{\partial^{\beta_1 + \ldots + \beta_n} [\![op]\!](x_1, \ldots, x_n)}{\partial x_1^{\beta_1} \cdots \partial x_n^{\beta_n}}$ of the smooth function $[\![op]\!] : \mathbb{R}^n \to \mathbb{R}$ that *op* implements.

Remark 2.6.2. *As in the $(1, 1)$ case, all the action of the AD transform happens at the level of primitives, and the macro extends in a homomorphic way to the rest of the constructs of the language.*

For example, we could choose the following representations of derivatives of order ≤ 2 of our example operations

$$
\begin{array}{ll}
\partial_{01}(+)(x_1, x_2) = \underline{1} & \partial_{02}(+)(x_1, x_2) = \underline{0} \\
\partial_{10}(+)(x_1, x_2) = \underline{1} & \partial_{11}(+)(x_1, x_2) = \underline{0} \\
\partial_{20}(+)(x_1, x_2) = \underline{0} &
\end{array}
$$

$$
\begin{array}{ll}
\partial_{01}(*)(x_1, x_2) = x_1 & \partial_{02}(*)(x_1, x_2) = \underline{0} \\
\partial_{10}(*)(x_1, x_2) = x_2 & \partial_{11}(*)(x_1, x_2) = \underline{1} \\
\partial_{20}(*)(x_1, x_2) = \underline{0} &
\end{array}
$$

$$
\begin{array}{l}
\partial_1(\varsigma)(x) = \mathbf{let}\, y = \varsigma(x)\, \mathbf{in}\, y * (\underline{1} - y) \\
\partial_2(\varsigma)(x) = \mathbf{let}\, y = \varsigma(x)\, \mathbf{in} \\
\qquad\qquad \mathbf{let}\, z = y * (\underline{1} - y)\, \mathbf{in}\, z * (\underline{1} - \underline{2} * y)
\end{array}
$$

$$
\begin{aligned}
&\overrightarrow{\mathcal{D}}(x) \stackrel{\text{def}}{=} x\\
&\overrightarrow{\mathcal{D}}(\underline{c}) \stackrel{\text{def}}{=} \langle \underline{c}, 0\rangle\\
&\overrightarrow{\mathcal{D}}(\lambda x.t) \stackrel{\text{def}}{=} \lambda x.\overrightarrow{\mathcal{D}}(t)\\
&\overrightarrow{\mathcal{D}}(t\,s) \stackrel{\text{def}}{=} \overrightarrow{\mathcal{D}}(t)\,\overrightarrow{\mathcal{D}}(s)\\
&\overrightarrow{\mathcal{D}}(\langle t_1,\ldots,t_n\rangle) \stackrel{\text{def}}{=} \langle \overrightarrow{\mathcal{D}}(t_1),\ldots,\overrightarrow{\mathcal{D}}(t_n)\rangle\\
&\overrightarrow{\mathcal{D}}(\tau.\ell\,t) \stackrel{\text{def}}{=} \overrightarrow{\mathcal{D}}(\tau).\ell\,\overrightarrow{\mathcal{D}}(t)\\
&\overrightarrow{\mathcal{D}}([\,]) \stackrel{\text{def}}{=} [\,]\\
&\overrightarrow{\mathcal{D}}(t :: s) \stackrel{\text{def}}{=} \overrightarrow{\mathcal{D}}(t) :: \overrightarrow{\mathcal{D}}(s)\\
&\overrightarrow{\mathcal{D}}(\mathbf{case}\,t\,\mathbf{of}\,\langle x_1,\ldots,x_n\rangle \to s) \stackrel{\text{def}}{=} \mathbf{case}\,\overrightarrow{\mathcal{D}}(t)\,\mathbf{of}\,\langle x_1,\ldots,x_n\rangle \to \overrightarrow{\mathcal{D}}(s)\\
&\overrightarrow{\mathcal{D}}(\mathbf{case}\,t\,\mathbf{of}\,\{\ell_1\,x_1 \to s_1 \;\big|\; \ldots \;\big|\; \ell_{\mathsf{n}}\,x_n \to s_n\}) \stackrel{\text{def}}{=}\\
&\qquad \mathbf{case}\,\overrightarrow{\mathcal{D}}(t)\,\mathbf{of}\,\{\ell_1\,x_1 \to \overrightarrow{\mathcal{D}}(s_1) \;\big|\; \ldots \;\big|\; \ell_{\mathsf{n}}\,x_n \to \overrightarrow{\mathcal{D}}(s_n)\}\\
&\overrightarrow{\mathcal{D}}(\mathbf{fold}\,(x_1,x_2).t\,\mathbf{over}\,s\,\mathbf{from}\,r) \stackrel{\text{def}}{=} \mathbf{fold}\,(x_1,x_2).\overrightarrow{\mathcal{D}}(t)\,\mathbf{over}\,\overrightarrow{\mathcal{D}}(s)\,\mathbf{from}\,\overrightarrow{\mathcal{D}}(r)\\
&\overrightarrow{\mathcal{D}}(op(t_1,\ldots,t_n)) \stackrel{\text{def}}{=} \mathbf{case}\,\overrightarrow{\mathcal{D}}(t_1)\,\mathbf{of}\,\langle x^1_{0\ldots0},\ldots,x^1_{R,0\ldots0}\rangle \to\\
&\qquad\qquad\qquad\qquad \vdots\\
&\qquad\qquad\qquad\qquad \mathbf{case}\,\overrightarrow{\mathcal{D}}(t_n)\,\mathbf{of}\,\langle x^n_{0\ldots0},\ldots,x^n_{R,0\ldots0}\rangle \to\\
&\qquad\qquad\qquad\qquad \langle\, D^{0\ldots0}op(x^1_{0\ldots0},\ldots,x^1_{R,0\ldots0},\ldots,x^n_{0\ldots0},\ldots,x^n_{\ldots0}),\\
&\qquad\qquad\qquad\qquad\quad \ldots,\\
&\qquad\qquad\qquad\qquad\quad D^{R\ldots0}op(x^1_{0\ldots0},\ldots,x^1_{R,0\ldots0},\ldots,x^n_{0\ldots0},\ldots,x^n_{R,0\ldots0})\,\rangle
\end{aligned}
$$

where $D^{0\ldots0}op(x^1_{0\ldots0},\ldots,x^1_{R,0\ldots0},\ldots,x^n_{0\ldots0},\ldots,x^n_{R,0\ldots0}) \stackrel{\text{def}}{=} op(x^1_{0\ldots0},\ldots,x^n_{0\ldots0})$

$$
\begin{aligned}
&D^{\alpha_1\ldots\alpha_k}op(x^1_{0\ldots0},\ldots,x^1_{R,0\ldots0},\ldots,x^n_{0\ldots0},\ldots,x^n_{R,0\ldots0}) \stackrel{\text{def}}{=} \qquad (\text{for } \alpha_1+\ldots+\alpha_k>0)\\
&\alpha_1!\cdot\ldots\cdot\alpha_k!\cdot\textstyle\sum_{\{\boldsymbol{\beta}\in\mathbb{N}^l \;\mid\; 1\le\beta_1+\ldots+\beta_l\le\alpha_1+\ldots+\alpha_k\}}\partial_{\boldsymbol{\beta}}op(x^1_{0\ldots0},\ldots,x^n_{0\ldots0})\cdot\\
&\qquad\textstyle\sum_{\{(\mathbf{e}^1,\ldots,\mathbf{e}^q)\in(\mathbb{N}^l)^q \;\mid\; e^1_j+\ldots+e^q_j=\beta_j,(e^1_1+\ldots+e^1_l)\cdot\alpha^1_i+\ldots+(e^q_1+\ldots+e^q_l)\cdot\alpha^q_i=\alpha_i\}}\\
&\qquad\qquad \textstyle\prod_{r=1}^{q}\prod_{j=1}^{l}\frac{1}{e^r_j!}\cdot\left(\frac{1}{\alpha^r_1!\cdot\ldots\cdot\alpha^r_k!}\cdot x^j_{\alpha_1\ldots\alpha_k}\right)^{e^r_j}.
\end{aligned}
$$

Figure 2.11: Higher-order AD translation on terms

Note that our rules, in particular, imply that $\overrightarrow{\mathcal{D}}(\underline{c}) = \langle \underline{c}, \underline{0},\ldots,\underline{0}\rangle$.

We extend $\overrightarrow{\mathcal{D}}$ to contexts: $\overrightarrow{\mathcal{D}}(\{x_1{:}\tau_1,\ldots,x_n{:}\tau_n\}) \stackrel{\text{def}}{=} \{x_1{:}\overrightarrow{\mathcal{D}}(\tau_1),\ldots,x_n{:}\overrightarrow{\mathcal{D}}(\tau_n)\}$.

Example 2.6.1 ($(1,1)$- and $(2,2)$-AD)**.** *Our choices of partial derivatives of the example operations are sufficient to implement (k,R)-Taylor forward AD with $R\le 2$. To be explicit, the distinctive formulas for $(1,1)$- and $(2,2)$-AD methods (specializing*

our abstract definition of $\overrightarrow{\mathcal{D}}_{(k,R)}$ above) are

$$
\begin{aligned}
&\overrightarrow{\mathcal{D}}_{(1,1)}(\mathbf{real}) = \mathbf{(real \times real)}\\
&\overrightarrow{\mathcal{D}}_{(1,1)}(op(t_1,\ldots,t_n)) =\\
&\qquad \mathbf{case}\ \overrightarrow{\mathcal{D}}_{(1,1)}(t_1)\ \mathbf{of}\ \langle x_0^1, x_1^1\rangle \to \ldots \to \mathbf{case}\ \overrightarrow{\mathcal{D}}_{(1,1)}(t_n)\ \mathbf{of}\ \langle x_0^n, x_1^n\rangle \to\\
&\qquad \langle op(x_0^1,\ldots,x_0^n), \textstyle\sum_{i=1}^n x_1^i * \partial_i op(x_0^1,\ldots,x_0^n)\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\overrightarrow{\mathcal{D}}_{(2,2)}(\mathbf{real}) = \mathbf{real}^6\\
&\overrightarrow{\mathcal{D}}_{(2,2)}(op(t_1,\ldots,t_n)) =\\
&\qquad \mathbf{case}\ \overrightarrow{\mathcal{D}}_{(2,2)}(t_1)\ \mathbf{of}\ \langle x_{00}^1, x_{01}^1, x_{02}^1, x_{10}^1, x_{11}^1, x_{20}^1\rangle \to\\
&\qquad \vdots\\
&\qquad \mathbf{case}\ \overrightarrow{\mathcal{D}}_{(2,2)}(t_n)\ \mathbf{of}\ \langle x_{00}^n, x_{01}^n, x_{02}^n, x_{10}^n, x_{11}^n, x_{20}^n\rangle \to\\
&\qquad\qquad \langle op(x_{00}^1,\ldots,x_{00}^n),\\
&\qquad\qquad \textstyle\sum_{i=1}^n x_{01}^i * \partial_{\hat{i}} op(x_{00}^1,\ldots,x_{00}^n),\\
&\qquad\qquad \textstyle\sum_{i=1}^n x_{02}^i * \partial_{\hat{i}} op(x_{00}^1,\ldots,x_{00}^n) + \sum_{i,j=1}^n x_{01}^i * x_{01}^j * \partial_{\widehat{i,j}} op(x_{00}^1,\ldots,x_{00}^n),\\
&\qquad\qquad \textstyle\sum_{i=1}^n x_{10}^i * \partial_{\hat{i}} op(x_{00}^1,\ldots,x_{00}^n),\\
&\qquad\qquad \textstyle\sum_{i=1}^n x_{11}^i * \partial_{\hat{i}} op(x_{00}^1,\ldots,x_{00}^n) + \sum_{i,j=1}^n x_{10}^i * x_{01}^j * \partial_{\widehat{i,j}} op(x_{00}^1,\ldots,x_{00}^n),\\
&\qquad\qquad \textstyle\sum_{i=1}^n x_{20}^i * \partial_{\hat{i}} op(x_{00}^1,\ldots,x_{00}^n) + \sum_{i,j=1}^n x_{10}^i * x_{10}^j * \partial_{\widehat{i,j}} op(x_{00}^1,\ldots,x_{00}^n)\rangle
\end{aligned}
$$

where we informally write $\hat{i}$ for the one-hot encoding of i (the sequence of length n consisting exclusively of zeros except in position i where it has a 1) and $\widehat{i,j}$ for the two-hot encoding of i and j (the sequence of length n consisting exclusively of zeros except in positions i and j where it has a 1 if $i \neq j$ and a 2 if $i = j$)

As noted in Section 2.6.2, it is often unnecessary to include all components of the $(2,2)$-algorithm, for example when computing a second order directional derivative. In that case, we may define a restricted $(2,2)$-AD algorithm that drops the non-mixed second order derivatives from the definitions above and defines $\overrightarrow{\mathcal{D}}_{(2,2)'}(\mathbf{real}) = \mathbf{real}^4$ and

$$
\begin{aligned}
\overrightarrow{\mathcal{D}}_{(2,2)'}(op(t_1,\ldots,t_n)) = \quad & \mathbf{case}\ \overrightarrow{\mathcal{D}}_{(2,2)'}(t_1)\ \mathbf{of}\ \langle x_{00}^1, x_{01}^1, x_{10}^1, x_{11}^1\rangle \to\\
& \vdots\\
& \mathbf{case}\ \overrightarrow{\mathcal{D}}_{(2,2)'}(t_n)\ \mathbf{of}\ \langle x_{00}^n, x_{01}^n, x_{10}^n, x_{11}^n\rangle \to\\
& \langle op(x_{00}^1,\ldots,x_{00}^n),\\
& \textstyle\sum_{i=1}^n x_{01}^i * \partial_{\hat{i}} op(x_{00}^1,\ldots,x_{00}^n),\\
& \textstyle\sum_{i=1}^n x_{10}^i * \partial_{\hat{i}} op(x_{00}^1,\ldots,x_{00}^n),\\
& \textstyle\sum_{i=1}^n x_{11}^i * \partial_{\hat{i}} op(x_{00}^1,\ldots,x_{00}^n) + \sum_{i,i'=1}^n x_{10}^i * x_{01}^i * \partial_{\hat{i}} op(x_{00}^1,\ldots,x_{00}^n)\rangle.
\end{aligned}
$$

Example 2.6.2 (Inner products, continued from 2.2.1)**.** *To illustrate the calculation*

of $\overrightarrow{\mathcal{D}}_{(2,2)'}$*, let us expand (and* β*-reduce)* $\overrightarrow{\mathcal{D}}_{(2,2)'}(t \cdot_2 s)$*:*

$$\begin{aligned}
&\mathbf{case}\, \overrightarrow{\mathcal{D}}_{(2,2)'}(t)\, \mathbf{of}\, \langle z_1, z_2\rangle \to \mathbf{case}\, \overrightarrow{\mathcal{D}}_{(2,2)'}(s)\, \mathbf{of}\, \langle y_1, y_2\rangle \to \\
&\mathbf{case}\, z_1\, \mathbf{of}\, \langle z_1, z'_{1,1}, z'_{1,2}, z''_1\rangle \to \mathbf{case}\, y_1\, \mathbf{of}\, \langle y_1, y'_{1,1}, y'_{1,2}, y''_1, \rangle \to \\
&\mathbf{case}\, z_2\, \mathbf{of}\, \langle z_2, z'_{2,1}, z'_{2,2}, z''_2, \rangle \to \mathbf{case}\, y_2\, \mathbf{of}\, \langle y_2, y'_{2,1}, y'_{2,2}, y''_2, \rangle \to \\
&\langle z_1 * y_1 + z_2 * y_2, \\
&\qquad z_1 * y'_{1,1} + z'_{1,1} * y_1 + z_2 * y'_{2,1} + z'_{2,1} * y_2, \\
&\qquad z_1 * y'_{1,2} + z'_{1,2} * y_1 + z_2 * y'_{2,2} + z'_{2,2} * y_2, \\
&\qquad z''_1 * y_1 + z''_2 * y_2 + y''_1 * z_1 + y''_2 * z_2 + \\
&\quad y'_{1,1} * y'_{1,2} + y'_{2,1} * y'_{2,2} + z'_{1,1} * z'_{1,2} + z'_{2,1} * z'_{2,2}\rangle
\end{aligned}$$

As for the first-order derivatives case, this turns $\overrightarrow{\mathcal{D}}$ into a well-typed, functorial macro in the following sense.

Lemma 2.6.1 (Functorial macro)**.** *If* $\Gamma \vdash t : \tau$ *then* $\overrightarrow{\mathcal{D}}(\Gamma) \vdash \overrightarrow{\mathcal{D}}(t) : \overrightarrow{\mathcal{D}}(\tau)$*.*
If $\Gamma, x : \sigma \vdash t : \tau$ *and* $\Gamma \vdash s : \sigma$ *then* $\overrightarrow{\mathcal{D}}(\Gamma) \vdash \overrightarrow{\mathcal{D}}(t[{}^{s}/_{x}]) = \overrightarrow{\mathcal{D}}(t)[{}^{\overrightarrow{\mathcal{D}}(s)}/_{x}]$*.*

Now, generalising Theorem 2.3.2, we obtain a correctness theorem for higher-order AD:

Theorem 2.6.2 (Semantic correctness of $\overrightarrow{\mathcal{D}}$ (limited))**.** *For any term* $x_1 \colon \mathbf{real}, \dots, x_n \colon \mathbf{real} \vdash t : \mathbf{real}$*, the function* $[\![\overrightarrow{\mathcal{D}}_{(k,R)}(t)]\!]$ *is the* (k, R)*-Taylor representation* (2.4) *of* $[\![t]\!]$*. In detail: for any smooth functions* $f_1 \dots f_n : \mathbb{R}^k \to \mathbb{R}$*,*

$$\left(\left(\frac{\partial^{\alpha_1+\dots+\alpha_k} f_j(x)}{\partial x_1^{\alpha_1}\cdots\partial x_k^{\alpha_k}}\right)_{\boldsymbol{\alpha}=(0,\dots,0)}^{(R,0,\dots,0)}\right)_{j=1}^{n} ; [\![\overrightarrow{\mathcal{D}}_{(k,R)}(t)]\!] = \left(\left(\frac{\partial^{\alpha_1+\dots+\alpha_k}((f_1,\dots,f_n);[\![t]\!])(x)}{\partial x_1^{\alpha_1}\cdots\partial x_k^{\alpha_k}}\right)_{\boldsymbol{\alpha}=(0,\dots,0)}^{(R,0,\dots,0)}\right)_{j=1}^{n}.$$

For instance, if $n = 2$, then $[\![\overrightarrow{\mathcal{D}}_{(1,1)}(t)]\!](x_1, 1, x_2, 0) = \left([\![t]\!](x_1, x_2), \frac{\partial [\![t]\!](x, x_2)}{\partial x}(x_1)\right)$.

Proof. Similarly to the first-order derivatives case, for each type τ, we define logical predicates S_τ between (open) k-dimensional plots in $[\![\tau]\!]$ and (open) k-dimensional plots in $[\![\overrightarrow{\mathcal{D}}_{(k,R)}(\tau)]\!]$, i.e. $S_\tau \subseteq \mathcal{P}^{\mathbb{R}^k}_{[\![\tau]\!]} \times \mathcal{P}^{\mathbb{R}^k}_{[\![\overrightarrow{\mathcal{D}}_{(k,R)}(\tau)]\!]}$, by induction on τ:

$$\begin{aligned}
S_{\mathbf{real}} &\stackrel{\text{def}}{=} \left\{ \left(f, \left(\frac{\partial^{\alpha_1+\dots+\alpha_k} f(x)}{\partial x_1^{\alpha_1}\cdots\partial x_k^{\alpha_k}} \right)_{\boldsymbol{\alpha}=(0,\dots,0)}^{(R,0,\dots,0)} \right) \;\middle|\; f : \mathbb{R}^k \to \mathbb{R} \text{ smooth} \right\} \\
S_{(\tau_1\times\dots\times\tau_n)} &\stackrel{\text{def}}{=} \{((f_1,\dots,f_n),(g_1,\dots,g_n)) \mid (f_1,g_1) \in S_{\tau_1}, \dots, (f_n,g_n) \in S_{\tau_n}\} \\
S_{\tau\to\sigma} &\stackrel{\text{def}}{=} \{(f_1,f_2) \mid \forall (g_1,g_2) \in S_\tau . (x \mapsto f_1(x)(g_1(x)), x \mapsto f_2(x)(g_2(x))) \in S_\sigma\}
\end{aligned}$$

Then, we establish the fundamental lemma of logical relations, proved by induction on the typing derivation of t:

If $x_1{:}\tau_1, ..., x_n{:}\tau_n \vdash t : \sigma$ and, for all $1{\leq}i{\leq}n$, $f_i : \mathbb{R}^k \to [\![\tau_i]\!]$ and $g_i : \mathbb{R}^k \to [\![\overrightarrow{\mathcal{D}}_{(k,R)}(\tau_i)]\!]$ are such that (f_i, g_i) is in S_{τ_i}, then we have that

$$\Big((f_1, \ldots, f_n); [\![t]\!], (g_1, \ldots, g_n); [\![\overrightarrow{\mathcal{D}}_{(k,R)}(t)]\!]\Big)$$

is in S_σ.

We again conclude the theorem from the fundamental lemma by considering the case where $\tau_i = \sigma = \mathbf{real}$, $m = n$ and $s_i = y_i$. □

2.6.5 Correctness of higher-order automatic differentiation

We now generalize Theorem 2.6.2 to hold at all first-order types, not just the reals.

We recall how the Taylor representation of any morphism $f : M \to N$ of manifolds is given by its action on jets [Kolář et al., 1996, Chapter IV]. For each point x in a manifold M, define the (k, R)-*jet space* $\mathcal{T}^{(k,R)}_x M$ to be the set $\{\gamma \in \mathbf{Man}(\mathbb{R}^k, M) \mid \gamma(0) = x\}/\sim$ of equivalence classes $[\gamma]$ of k-dimensional plots γ in M based at x, where we identify $\gamma_1 \sim \gamma_2$ iff all partial derivatives of order $\leq R$ coincide in the sense that

$$\frac{\partial^{\alpha_1+\ldots+\alpha_k}(\gamma_1; f)(x)}{\partial x_1^{\alpha_1} \ldots \partial x_k^{\alpha_k}}(0) = \frac{\partial^{\alpha_1+\ldots+\alpha_k}(\gamma_2; f)(x)}{\partial x_1^{\alpha_1} \ldots \partial x_k^{\alpha_k}}(0)$$

for all smooth $f : M \to \mathbb{R}$ and all multi-indices $\boldsymbol{\alpha} = (0, \ldots, 0) \ldots, (R, 0, \ldots, 0)$. In the case of $(k, R) = (1, 1)$, a (k, R)-jet space is better known as a *tangent space*. The (k, R)-*jet bundle* (a.k.a. *tangent bundle*, in case $(k, R) = (1, 1)$) of M is the set $\mathcal{T}^{(k,R)}(M) \stackrel{\text{def}}{=} \biguplus_{x \in M} \mathcal{T}^{(k,R)}_x(M)$. The charts of M equip $\mathcal{T}^{(k,R)}(M)$ with a canonical manifold structure. The (manifold) diffeology of these jet bundles can be concisely summarized by the plots $\mathcal{P}^U_{\mathcal{T}^{(k,R)}(M)} = \{f : U \to |\mathcal{T}^{(k,R)}(M)| \mid \exists g \in \mathcal{P}^{U\times\mathbb{R}^k}_M . \forall u \in U . (g(u, 0), [v \mapsto g(u, v)]) = f(u)\}$.
Then $\mathcal{T}^{(k,R)}$ acts on smooth maps $f : M \to N$ to give $\mathcal{T}^{(k,R)}(f) : \mathcal{T}^{(k,R)}(M) \to \mathcal{T}^{(k,R)}(N)$ is defined as $\mathcal{T}^{(k,R)}(f)(x, [\gamma]) \stackrel{\text{def}}{=} (f(x), [\gamma; f])$. In local coordinates, this action $\mathcal{T}^{(k,R)}(f)$ is seen to coincide precisely with the (k, R)-Taylor representation of f given by the Faà di Bruno formula [Merker, 2004]. All told, the (k, R)-jet bundle is a functor $\mathcal{T}^{(k,R)} : \mathbf{Man} \to \mathbf{Man}$ [Kolář et al., 1996].

Similarly to the first-order case (Lemma 2.5.4), we can understand the jet bundle of a composite space in terms of that of its parts.

Lemma 2.6.3. *There are canonical isomorphisms* $\mathcal{T}^{(k,R)}(\biguplus_{i=1}^\infty M_i) \cong \biguplus_{i=1}^\infty \mathcal{T}^{(k,R)}(M_i)$ *and* $\mathcal{T}^{(k,R)}(M_1 \times \ldots \times M_n) \cong \mathcal{T}^{(k,R)}(M_1) \times \ldots \times \mathcal{T}^{(k,R)}(M_n)$.

Proof. This is the same as for the tangent bundle functor. For disjoint unions, notice that smooth morphisms from $\mathbb{R}^k$ into a disjoint union of manifolds always factor over a single inclusion, because $\mathbb{R}^k$ is connected. For products, it is well-known that partial derivatives of a morphism $(f_1, \ldots, f_n)$ are calculated component-wise [Lee, 2013, ex. 3-2]. □

Again similarly to the first-order case, we define a canonical isomorphism $\phi_\tau^{\overrightarrow{\mathcal{D}}\mathcal{T}} : [\![\overrightarrow{\mathcal{D}}_{(k,R)}(\tau)]\!] \to \mathcal{T}^{(k,R)}([\![\tau]\!])$ for every type τ, by induction on the structure of types. We let $\phi_{\mathbf{real}}^{\overrightarrow{\mathcal{D}}\mathcal{T}} : [\![\overrightarrow{\mathcal{D}}_{(k,R)}(\mathbf{real})]\!] \to \mathcal{T}^{(k,R)}([\![\mathbf{real}]\!])$ be given by $\phi_{\mathbf{real}}^{\overrightarrow{\mathcal{D}}\mathcal{T}}(x, x') \stackrel{\text{def}}{=} (x, [t \mapsto x + x't])$. For the other types, we use Lemma 2.6.3. We can now phrase an analogue of the correctness at all first-order types (Theorem 2.5.6) to the higher-order setting:

Theorem 2.6.4 (Semantic correctness of $\overrightarrow{\mathcal{D}}_{(k,R)}$ (full))**.** *For any ground τ, any first-order context Γ and any term $\Gamma \vdash t : \tau$, the syntactic translation $\overrightarrow{\mathcal{D}}_{(k,R)}$ coincides with the (k, R)-jet bundle functor, modulo these canonical isomorphisms:*

$$\begin{array}{ccc} [\![\overrightarrow{\mathcal{D}}_{(k,R)}(\Gamma)]\!] & \xrightarrow{[\![\overrightarrow{\mathcal{D}}_{(k,R)}(t)]\!]} & [\![\overrightarrow{\mathcal{D}}_{(k,R)}(\tau)]\!] \\ {\scriptstyle \phi_\Gamma^{\overrightarrow{\mathcal{D}}\mathcal{T}}}\downarrow\cong & & \cong\downarrow{\scriptstyle \phi_\tau^{\overrightarrow{\mathcal{D}}\mathcal{T}}} \\ \mathcal{T}^{(k,R)}([\![\Gamma]\!]) & \xrightarrow[\mathcal{T}^{(k,R)}([\![t]\!])]{} & \mathcal{T}^{(k,R)}([\![\tau]\!]) \end{array}$$

Proof. Again, this is essentially identical to the $(1, 1)$ case we proved in Section 2.5.3, in the proof of Theorem 2.5.6. For any k-dimensional plot $\gamma \in \mathbf{Man}(\mathbb{R}^k, M)$, let $\bar{\gamma} \in \mathbf{Man}(\mathbb{R}^k, \mathcal{T}^{(k,R)}(M))$ be the (k, R)-jet curve, given by $\bar{\gamma}(x) = (\gamma(x), [t \mapsto \gamma(x + t)])$. First, we note that a smooth map $h : \mathcal{T}^{(k,R)}(M) \to \mathcal{T}^{(k,R)}(N)$ is of the form $\mathcal{T}^{(k,R)}(g)$ for some $g : M \to N$ if for all smooth $\gamma : \mathbb{R}^k \to M$ we have $\bar{\gamma}; h = \overline{(\gamma; g)} : \mathbb{R}^k \to \mathcal{T}^{(k,R)}(N)$. This generalizes (2.4). Second, for any first-order type τ, $S_{[\![\tau]\!]} = \{(f, \tilde{f}) \mid \tilde{f}; \phi_\tau^{\overrightarrow{\mathcal{D}}\mathcal{T}} = \bar{f}\}$. This is shown by induction on the structure of types. We conclude the theorem from diagram (2.3), by putting these two observations together. □

2.7 Conclusion

2.7.1 Summary

We have shown that diffeological spaces provide a denotational semantics for a higher-order language with variants and inductive types (Sections 2.3,2.4). We have used this to show correctness of a simple forward-mode AD translation (Theorem 2.3.2, Theorem 2.5.6). The method is not tied to this specific translation, as we illustrated in Section 2.6. We have shown that not only can AD be seen as a functor, it also preserves all the categorical structure that is present in the syntax (Example 2.5.1). Subject to this constraint of being a structure preserving functor, AD is canonically determined once we specify its effect on all primitives operations of the language.

The structure of our elementary correctness argument for Theorem 2.3.2 is a typical logical relations proof. As explained in Section 2.5, this can equivalently be understood as a denotational semantics in a new kind of space obtained by categorical gluing. Our semantics using diffeological spaces is interesting as it simply consists of sets with structure, while its categorical foundation (i.e. being a Grothendieck quasi-topos) makes is suited for the semantics of very expressive languages.

Finally, we have used this denotational semantics to phrase and prove semantic correctness of our AD transform for programs between first-order types, which might have higher-order subprograms. Overall, then, there are two logical relations at play. One is in diffeological spaces, which ensures that all definable functions are smooth. The other is in the correctness proof (equivalently in the categorical gluing), which explicitly tracks the derivative of each function, and tracks the syntactic AD even at higher types.

2.7.2 Related work and context

AD has a long history and many implementations. AD was perhaps first phrased in a functional setting in Pearlmutter and Siskind [2008], and there are now a number of teams working on AD in the functional setting (e.g. Wang et al. [2019], Shaikhha et al. [2019], Elliott [2018]), some providing efficient implementations. Although that work does not involve formal semantics, it is inspired by intuitions from differential geometry and category theory.

We add to a very recent body of work on verified automatic differentiation. Much of this is concurrent with and independent of the work we presented. In the first-order setting, there are recent accounts based on denotational semantics in manifolds [Fong et al., 2019] and based on synthetic differential geometry [Cruttwell et al., 2019], as well as work making a categorical abstraction [Cockett et al., 2020] and work connecting operational semantics with denotational semantics [Abadi and Plotkin, 2020, Plotkin, 2018]. Recently there has also been significant progress at higher types. The work of Brunel et al. gives formal correctness proofs for reverse-mode derivatives on computation graphs [Brunel et al., 2019]. The work of Barthe et al. [2020] provides a general discussion of some new syntactic logical relations arguments including one very similar to our syntactic proof of Theorem 2.3.2.

The differential λ-calculus [Ehrhard and Regnier, 2003] is related to AD, and explicit connections are made in Mak and Ong [2020], Manzyuk [2012]. One difference is that the differential λ-calculus allows the addition of terms at all types, and hence vector space models are suitable, but this appears peculiar with the variant and inductive types that we consider here.

Finally, we emphasise that we have chosen the neural network (2.1) as our running example mainly for its simplicity. There are many other examples of AD outside the neural networks literature: AD is useful whenever derivatives need to be calculated on high dimensional spaces. This includes optimization problems more generally, where the derivative is passed to a gradient descent method (e.g. Robbins and Monro [1951], Kiefer et al. [1952], Qian [1999], Kingma and Ba [2014], Duchi et al. [2011], Liu and Nocedal [1989]). Other applications of AD are in advanced *integration* methods, since derivatives play a role in Hamiltonian Monte Carlo [Neal et al., 2011, Hoffman and Gelman, 2014] and variational inference [Kucukelbir et al., 2017].

2.7.3 Discussion

Derivatives of higher-order functions. In our gluing categories **Gl** (Section 2.5.2), we have avoided the question of what semantic derivatives should be

associated with higher-order functions. The logical relations argument allows any derivative-like for higher-order functions that will preserve the correct-notion of derivative when eventually going back to first-order. This sidesteps the whole idea of defining them as limits, or as tangents in certain function spaces. In fact, higher-order functions will typically have several "derivatives" in our sense. For an intuition, consider a second-order function $f : (\mathbb{R} \to \mathbb{R}) \to \mathbb{R}$. Given that $\mathcal{D}(\mathbb{R} \to \mathbb{R}) = \mathbb{R}^2 \to \mathbb{R}^2$, we will have $df : (\mathbb{R}^2 \to \mathbb{R}^2) \to \mathbb{R}^2$. However, the correctness criterion only forces df to behave in a certain way when given an input of the form $T(f) : \mathbb{R}^2 \to \mathbb{R}^2$. Such $T(f)$ are a strict subset of smooth functions $\mathbb{R}^2 \to \mathbb{R}^2$. This is easily seen as $T(f)$ is linear in its second argument. Therefore, a correct df is in a sense free to choose what it does with functions $(\mathbb{R}^2 \to \mathbb{R}^2)$ which are not of the form $T(f)$. In Vákár et al. [2022], we construct an explicit such definable df that is semantically different from what our AD macro produces (i.e. $\mathcal{D}(f) \neq df$). We see this as a 'feature not a bug', as it is in general very hard to have geometric intuition in such infinite dimensional spaces, and it is therefore unclear whether a tangent space which would be a vector space would even be something reasonable to ask on a function space. As the vast majority of optimisation reduces to a finite dimensional setting, we leave the investigation of canonical ways to do AD on functions spaces for future work. Indeed, while derivatives of higher-order functions are of deep interest and have rightly been studied in their own right in differential geometry, we believe the situation is subtly different in computer science:

- In programming applications, we usually use higher-order programs only to construct the first-order functions that we ultimately end up running and calculating derivatives of. Automatic differentiation methods can exploit this freedom: derivatives of higher-order functions only matter in so far as they can be used to construct the correct derivatives of first-order functions, so we can choose a simple and cheap notion of derivative among the valid options. Therefore, the fact that our semantics does not commit to a single notion of derivative of higher-order functions can be seen as modelling the pragmatics of programming practice.

- While function spaces in differential geometry are typically infinite dimensional objects that are unsuitable for representation in the finite memory of a computer, higher-order functions as used in programming are much more restricted: all they can do is call a function on finitely many arguments and analyse the function outputs. As such, function types in programming can be thought of as (locally) finite dimensional.

We now make some connections to the state of the art in AD implementation. As is common in denotational semantics research, we have here focused on an idealized language and simple translations to illustrate the main aspects of the method. There are a number of points where our approach is simplistic compared to the advanced current practice, as we now explain.

Representation of vectors. In our examples, we have treated n-vectors as tuples of length n. This style of programming does not scale to large n. A

better solution would be to use array types, following Shaikhha et al. [2019]. Our categorical semantics and correctness proofs straightforwardly extend to cover them, similarly to our treatment of lists. What differs is the API of terms, but this is easily accommodated.

Efficient forward-mode AD. For AD to be useful, it must be fast. The syntactic translation $\overrightarrow{\mathcal{D}}$ that we use is the basis of an efficient AD library [Shaikhha et al., 2019]. However, numerous optimizations are needed, ranging from algebraic manipulations, to partial evaluations, to the use of an optimizing C compiler. The resulting implementation is performant in experiments [Shaikhha et al., 2019]. In Shaikhha et al. [2022a], we validated some of these manipulations using our semantics, and showed the correctness of their similar AD macro, using a similar logical relations argument as used here.

Efficient reverse-mode AD. One goal of the simplicity and modularity of our work on forward-mode AD was to inspire other refinement and generalizations. An important one is reverse-mode, where the computation of the derivative goes in the opposite direction as the compute flow for the primal, exploiting the symmetry in the chain rule. This is usually much more efficient for computing the gradient of a function $\mathbb{R}^n \to \mathbb{R}$ on one pass, whereas it would typically require n passes for forward-mode. Speed being the main attraction of reverse-mode AD, its implementations tend to rely on mutable state, control operators and/or staging [Pearlmutter and Siskind, 2008, Carpenter et al., 2015, Wang et al., 2019, Brunel et al., 2019], which we have not considered here. It was noted in Brunel et al. [2019] that using the linear structure of the function space for derivative pat is key to avoid code duplication and unnecessary evaluations. In Wang et al. [2019], some impure features (references) are used to ensure efficiency. In short, extra features accounting for more control in the language seem necessary to accurately model and prove correct a performant reverse AD algorithm. This was more recently completed in a body of work [Radul et al., 2022, Smeding and Vákár, 2022, Krawiec et al., 2022]. Pursuing on this work, we have also investigated an efficient version of reverse-mode that will be purely functional [Huot and Shaikhha, 2022].

Other language features. The idealized languages that we considered so far do not touch on several useful language constructs. For example, the use of functions that are partial (such as division) or partly smooth (such as ReLU); phenomena such as iteration, recursion; and probabilities. As diffeological spaces are based on Euclidean spaces and smooth maps, perhaps we can construct generalized smooth spaces based on a different "first-order category". This will be investigated in detail in the next chapter. Secondly, first-order types are all instances of manifolds with trivial tangent bundle. In practice, types corresponding to more general manifolds like spheres are also used (see constraint types in Stan [Carpenter et al., 2017] and bijectors in TensorFlow [Dillon et al., 2017]). We leave this to future work.

3

Differentiable Programming beyond smoothness

Summary. This chapter introduces a denotational model for a higher-order-recursive language with conditionals, and shows the correctness of the automatic differentiation program transformation in this richer setting, going beyond differentiability. In the first part of this chapter, we first explain the revised first-order setting when there are conditionals in the language (Section 3.2), following Lee et al. [2020]. We then define our recursive higher-order language (Section 3.3) and introduce a novel denotational model for it (Section 3.4). Using a revised version of logical relations for this new setting, we give a revised correctness theorem for automatic differentiation (Section 3.5).

In the second part of this chapter, we use more advanced tools from category theory to prove that the denotational semantics is sound and adequate (Section 3.6), and that automatic differentiation is correct (Section 3.7).

This chapter can be thought of as a natural extension to Chapter 2, which goes beyond differentiability in two ways. First, we introduce conditionals, which can break continuity (and therefore differentiability). Second, we explore an effectful setting with partiality and recursion. Therefore, more sophisticated tools are required when reasoning about effectful recursive languages, and new theory needs to be used for reasoning about functions which are not differentiable everywhere.

In the next chapter (Chapter 4), we will investigate applications of the theory developed in this chapter, and see how the guarantees on automatic differentiation obtained in this chapter affect the correct usage of derivatives obtained by automatic differentiation in other algorithms.

Publication. This chapter is based on joint work with Alex Lew, Sam Staton, and Vikash Mansinghka. It largely extends Lew, Huot, and Mansinghka [2021] with all the categorical machinery needed for the proofs. The proofs and the categorical

machinery are quite simplified compared to our original account, and this is my own work. The technical content of this chapter roughly covers Sections II-III in Huot et al. [2023b], which was recently accepted at LICS 2023. Many of the ideas were developed together in a collaborative process, but, except where explicitly noted, the presentation of the definitions and proofs in this thesis are my own.

3.1 Introduction

3.1.1 Motivation

In this chapter, we explore a richer theory, compared to Chapter 2. We are interested in more expressive functional languages, which include recursion, conditionals, and partiality. Many elementary functions such as tan or division are partial, and AD is often applied to programs containing such partial functions. In addition. it is common for AD to be applied to functions with non-differentiable points, such as ReLU in deep learning. This raises the question of the guarantees of AD in such a setting. Recursion is another common feature, which is used for instance in iterative optimisation methods such as gradient descent or in defining expressive neural networks such as recurrent neural network. Therefore, more generally, we need to go beyond diffeological spaces and develop semantics models to account for these new features. We would also like the new denotational semantics to be quite flexible and powerful, allowing us to interpret a rich language which could easily be extended. Furthermore, we would obviously still need to be able to talk about smoothness in some sense and show some important properties of AD and generalized derivatives. Another application area for AD on languages that need higher-order functions, conditionals and recursion is probabilistic programming languages [Cusumano-Towner et al., 2019, Bingham et al., 2019, Ścibior et al., 2017, 2018]. Inference in probabilistic programming languages is tightly linked to AD, as several inference methods rely on computing deterministic derivatives of programs. Notably, Hamiltonian Monte Carlo (HMC) is an Metropolis-Hasting (MH) algorithm that requires the gradient of the joint density of the probabilistic program, and the Jacobian determinant is used in the change of variable formula.

3.1.2 Overview of the technical results

This chapter is divided into two parts. The first part will not make heavy use of category theory, it presents AD as a program transform on a richer language, and introduces a new semantic model for reasoning about differentiable programming. The second part focuses on the underlying categorical machinery and proofs that underlie this foundation for differentiable programming.

In the first part, we will first recall (Section 3.2) the work of Lee et al. [2020] on a first-order language with conditionals. They introduced an elegant new semantic model in terms of piece-wise analytic functions. After defining a simple AD transform on the language, we will see that AD computes intensional derivatives: they generalize the usual derivative to this new class of functions. A key insight is

that what AD computes is intensional: it does not only depend on the semantics of the program, but also on its representation. Second, we will extend the language of Lee et al. [2020] to a higher-order recursive language, and extend AD to this setting (Section 3.3). We will then introduce a new semantic model, ωPAP (Section 3.4), generalizing the piece-wise analytic functions of Lee et al. [2020] to a much richer setting, that can serve as a sound and adequate model of our language. Finally, we give a correctness theorem for AD (Theorem 3.5.3) in Section 3.5.

To do so, in the second part, we introduce the categorical machinery allowing us to do this. One main idea is that what makes the category of diffeological spaces so well-behaved is that it can be seen as a category of concrete sheaves. We will recall the basic theory of concrete sheaves, and we will build a new model on these principles. In fact, for the semantics model to interpret a language with recursion, we will need some kind of ωcpo-structure on the homsets. To do this, we will reuse the idea of "ω-concrete sheaves" introduced in Vákár et al. [2019] and expanded in Matache et al. [2022]. We will prove that ωPAP is an instance thereof, thus showing that it forms a well-behaved category (Section 3.6). To show properties of the language using logical relations like arguments, we will use the version of gluing developed by Katsumata [2013], called fibrations for logical relations (Section 3.7). They conveniently and elegantly allow us to reason denotationally about expressive functional languages, including with recursion.

To summarize, we make the following key contributions:

- We introduce ωPAP-spaces, an expressive model for a higher-order recursive functional language with smooth primitives.
- We define AD on the language and show that AD computes intensional derivatives, in the sense of Lee et al. [2020].
- We show how the categorical machinery used is suited for studying differentiable programming, but also more generally for the semantics of higher-order recursive functional languages.

Part I

3.2 PAP functions on Euclidean spaces

In practice, AD is often applied to functions with non-differentiable points. For instance, neural networks using $\mathrm{ReLU}(x) = \max(0, x)$ define non-differentiable functions, but the 'derivatives' of losses involving those functions are computed using automatic differentiation systems. A natural question is whether this matters at all because the issues rarely occur (e.g. on a set of Lebesgue measure 0). Lee et al. [2020] showed that the issue is actually subtle, and we will also see in the next chapter more subtleties showing how non-intuitive it can be, depending on the usage of the derivatives produced by AD. This section closely follows the development of Lee et al. [2020], on which our work is based.

First, as the following theorems show, this is bound to be a subtle analysis problem. One may think that functions that are almost everywhere differentiable compose, but this is not always the case.

Proposition 3.2.1 ([Lee et al., 2020]). *There exist functions $f : (0,1) \to (0,1)$ and $g : (0,1) \to [0,1]$ such that f and g are almost everywhere (a.e.) differentiable and continuous, but $g \circ f$ fails to be a.e. differentiable.*

Recall that the chain rule was key to obtaining a compositional AD macro already at first-order. When restricting to almost everywhere differentiable functions whose composition is almost everywhere differentiable, we might expect the chain rule to hold almost everywhere. Lee et al. [2020] also showed that this is not the case either.

Proposition 3.2.2 ([Lee et al., 2020]). *There exist functions $f, g : (0,1) \to (0,1)$ such that f, g and $g \circ f$ are a.e. differentiable and continuous, but such that it is not true that $g'(f(x))$ is defined for almost all $x \in (0,1)$.*

Proposition 3.2.3 ([Lee et al., 2020]). *There exist functions $f, g : (0,1) \to (0,1)$ such that f, g and $g \circ f$ are a.e. differentiable and continuous, but for some measurable subset $A \subseteq (0,1)$ with non-zero measure, we have $f'(x) = 0$ and $(g \circ f)'(x) \neq 0$ for all $x \in A$.*

We also want to match what is done in practice where AD systems do differentiate programs at the points of non-differentiability, going beyond a semantics saying that the program is undefined at those points, as is done e.g. in Abadi and Plotkin [2020]. This justifies introducing a strictly stronger property than almost everywhere (a.e.) differentiability that will be better behaved. The resulting functions are called *PAP functions*.

Definition 3.2.1 (piecewise representation). *Let $U \subseteq \mathbb{R}^n$, $V \subseteq \mathbb{R}^m$, and $f : U \to V$. A piecewise representation of f is a countable family $\{(A_i, f_i)\}_{i \in I}$ such that:*

(1) the sets A_i form a partition of U;

(2) each $f_i : U_i \to \mathbb{R}^m$ is defined on an open domain $U_i \supseteq A_i$;

(3) for all $x \in A_i$, $f_i(x) = f(x)$.

The PAP functions are those with *analytic* piecewise representations:

Definition 3.2.2 (analytic function). *If the Taylor series of a smooth function f converges pointwise to f in a neighborhood around x, we say that f is **analytic** at x. An analytic function is a function that is analytic at every point in its domain.*

Definition 3.2.3 (analytic set). *We call a set $A \subseteq \mathbb{R}^n$ an analytic set iff there exist finite collections $\{g_i^+\}_{i \in I}$ and $\{g_j^-\}_{j \in J}$ of analytic functions into $\mathbb{R}$, with open domains X_i^+ and X_j^-, such that*

$$A = \{x \in (\bigcap_i X_i^+) \cap (\bigcap_j X_j^-) \mid \forall i. g_i^+(x) > 0 \wedge \forall j. g_j^- \leq 0\}$$

That is, analytic sets are subsets of open sets carved out using a finite number of analytic inequalities, strict or not strict.

Definition 3.2.4 (PAP function)**.** *We say f is piecewise analytic under analytic partition (PAP) if there exists a piecewise representation $\{(A_i, f_i)\}_{i\in I}$ of f such that the A_i are analytic sets and the f_i are analytic functions.*
We call such a representation a PAP representation.

Example 3.2.1.
- *The usual elementary operations on reals $+, -, *, \cos, \exp$ are analytic and therefore PAP.*
- *Functions like* $\max, \min$ *are PAP.*
- *Compositions of PAP functions are PAP.*

Lee et al. [2020] use PAP functions to give a semantics to a simple first-order language. Their language has real number constants $\underline{c}$, primitive real-valued functions $f : \mathbb{R}^{n_f} \to \mathbb{R}$, as well as an **if** construct for branching. The language is given in Figure 3.1.

$$
\begin{array}{lll}
\tau & ::= & \text{types} \\
& \mid \ \textbf{real} & \text{real numbers} \\
t & ::= & \text{terms} \\
& \mid \ x & \text{variable} \\
& \mid \ \underline{c} \mid f(t_1, \ldots, t_n) & \text{constants/operations} \\
& \mid \ \textbf{if}\, t_1 > 0\, \textbf{then}\, t_2\, \textbf{else}\, t_3 & \text{conditionals}
\end{array}
$$

$$
\frac{}{\Gamma \vdash x : \tau}((x:\tau) \in \Gamma) \qquad \frac{\forall i = 1 \ldots n \quad \Gamma \vdash t_i : \textbf{real}}{\Gamma \vdash f(t_1, \ldots, t_n) : \textbf{real}}(f \text{ operation})
$$

$$
\frac{}{\Gamma \vdash \underline{c} : \textbf{real}}(c \in \mathbb{R}) \qquad \frac{\Gamma \vdash t_1 : \textbf{real} \quad \Gamma \vdash t_2 : \tau \quad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \textbf{if}\, t_1 > 0\, \textbf{then}\, t_2\, \textbf{else}\, t_3\ : \tau}
$$

Figure 3.1: Types, terms and type system of Lee et al. [2020]'s language

Well-typed expressions t in the language denote functions $[\![t]\!] : \mathbb{R}^n \to \mathbb{R}$ of an *input vector* $\mathbf{x} \in \mathbb{R}^n$ (for some n). We have

$$
\begin{array}{ll}
[\![\underline{c}]\!]\mathbf{x} & = c \\
[\![x_i]\!]\mathbf{x} & = \mathbf{x}[i] \\
[\![f(t_1, \ldots, t_n)]\!]\mathbf{x} & = f([\![t_1]\!]\mathbf{x}, \ldots, [\![t_n]\!]\mathbf{x}) \\
[\![\textbf{if}\, t_1 > 0\, \textbf{then}\, t_2\, \textbf{else}\, t_3\,]\!]\mathbf{x} & = [[\![t_1]\!]\mathbf{x} > 0] \cdot ([\![t_2]\!]\mathbf{x}) + [[\![t_1]\!]\mathbf{x} \leq 0] \cdot ([\![t_3]\!]\mathbf{x})
\end{array}
$$

Proposition 3.2.4 ([Lee et al., 2020])**.** *Constant functions and projection functions are PAP. Suppose that every primitive operation $f : \mathbb{R}^n \to \mathbb{R}$ is PAP. Then all well-typed expressions $\Gamma \vdash t : \textbf{real}$ denote PAP functions.*

PAP functions are not necessarily differentiable everywhere. But Lee et al. [2020] define *intensional derivatives*, which do always exist for PAP functions:

Definition 3.2.5 (Intensional derivative of PAP representation)**.** *Let $\gamma = \{(A_i, f_i)\}_i$ be a PAP representation of some function f. An intensional derivative of γ is the representation $\mathcal{D}\gamma := \{A_i, \mathcal{D}f_i\}_i$ where $\mathcal{D}f_i : U_i \supseteq A_i \to \mathbb{R}^{n\times m}$ is the usual derivative of the analytic function $f_i : U_i \subseteq \mathbb{R}^n \to \mathbb{R}^m$.*

Definition 3.2.6 (intensional derivative)**.** *Let f be PAP. Define $\partial_\bullet f$ to be the set*

$$\partial_\bullet f := \{g \mid \mathcal{D}\gamma \text{ PAP representation of } g, \gamma \text{ PAP representation of } f\}$$

Each $df \in \partial_\bullet f$ is an intensional derivative *of f.*

In the simple case where $f : \mathbb{R} \to \mathbb{R}$ is PAP, this simply says that a function $g : \mathbb{R} \to \mathbb{R}$ is *an intensional derivative* of f if there exists a PAP representation $\{(A_i, f_i)\}_{i\in I}$ of f such that when $x \in A_i$, $g(x) = f_i'(x)$. Unlike traditional derivatives, they are not unique. For instance, for every analytic set $A \subseteq R$, and any $x \in A$, the two sets $A_1 = A \cap (-\infty, x)$ and $A_2 = A \cap (x, +\infty)$ are analytic. Therefore, we can always refine a piecewise representation by changing A into $A_1, A_2, \{x\}$. On the analytic set $\{x\}$, there are infinitely many analytic functions that agree with a PAP function f on this point, but have different derivatives. To see that, define analytic functions $f_c(y) = cy + (f(x) - cx)$ for $c \in \mathbb{R}$ on a neighbourhood of x. We have $f_c(x) = f(x)$ and $f_c'(y) = c$. This gives f infinitely many intensional derivatives, with any possible value for an intensional derivative at x. More generally, this can happen at countably many points.

Still, PAP functions enjoy the following desirable properties:

Proposition 3.2.5 ([Lee et al., 2020])**.**

1. ***Almost-everywhere differentiability.*** *PAP functions are differentiable (in the classical sense) almost everywhere. Any intensional derivative will agree almost-everywhere with the true derivative.*
2. ***Closure under differentiation.*** *Any intensional derivative of a PAP function is a PAP function.*
3. ***Closure under composition.*** *PAP functions compose.*
4. ***Chain rule for intensional derivatives.*** *If df, dg are intensional derivatives of f and g respectively, and if $h := g \circ f$, then $dh(x) := dg(f(x)) \cdot df(x)$ is an intensional derivative of h, where $\cdot$ is matrix composition.*

Similarly to the smooth case, we will use dual-numbers to propagate derivatives as well as primal values to ensure a nice compositionality rule. The following is a slight reformulation from what is exposed in Lee et al. [2020].

Definition 3.2.7. *A function $g : \mathbb{R}^{2n} \to \mathbb{R}^{2m}$ is a dual-number intensional representation of a PAP function $f : \mathbb{R}^n \to \mathbb{R}^m$ if*

$$g(x, dx) = (f(x), df(x) \cdot dx)$$

where $\cdot$ is matrix-vector product, and df is an intensional derivative of f.

Lee et al. [2020] then give a simple AD algorithm $\mathcal{D}$ for their language (Figure 3.2) and show that, when applied to an expression t, it is guaranteed to yield *some* intensional derivative of $[\![t]\!]$ as long as each primitive f comes equipped with a definable intensional derivative. Essentially, this proof is based on an analogue to the chain rule for intensional derivatives. Note that this assumes an extended target language for the AD macro, which supports product types and enough primitives operations to encode valid dual-number intensional derivatives $f_{\mathcal{D}}$ for all primitive operations f. For instance, if $f = \cos$, the target language needs a primitive $\cos_{\mathcal{D}} : \mathbf{real} \times \mathbf{real} \to \mathbf{real} \times \mathbf{real}$. Of course, in practice, $\cos_{\mathcal{D}}$ would itself be implemented as a composition of simpler primitives. Its first component would be cos and its second component could be implemented with $-$ and sin, as $\cos' = -\sin$.

$$
\begin{array}{lcl}
\mathcal{D}[\![\mathbf{real}]\!] & \stackrel{\text{def}}{=} & \mathbf{real} \times \mathbf{real} \\
\mathcal{D}(x) & \stackrel{\text{def}}{=} & x \\
\mathcal{D}(\underline{c} : \mathbf{real}) & \stackrel{\text{def}}{=} & \langle \underline{c}, 0 \rangle \\
\mathcal{D}(f(t_1, \ldots, t_n)) & \stackrel{\text{def}}{=} & f_{\mathcal{D}}(\mathcal{D}(t_1), \ldots, \mathcal{D}(t_n)) \\
\mathcal{D}(\mathbf{if}\, t_1 > 0\, \mathbf{then}\, t_2\, \mathbf{else}\, t_3\,) & \stackrel{\text{def}}{=} & \mathbf{if}\, t_1 > 0\, \mathbf{then}\, \mathcal{D}(t_2)\, \mathbf{else}\, \mathcal{D}(t_3)
\end{array}
$$

Figure 3.2: Simple AD macro

Proposition 3.2.6 ([Lee et al., 2020])**.** *If for all primitive operations f, $\mathcal{D}(f)$ is a dual-number intensional representation for f, then for any well-typed program t, $\mathcal{D}[\![t]\!]$ is a dual-number intensional representation of $[\![t]\!]$.*

3.3 A higher-order language with conditionals and recursion

As we discussed in Section 3.1, we are interested in defining AD on a higher-order recursive language with conditionals, extending both Lee et al. [2020] and our work from Chapter 2. This will introduce several new semantics challenges, including

- **Domain of partial functions.** Programs will now denote partial functions, and we need to make decisions as to what the allowed domains of partial functions should be.
- **Reasoning about recursion.** Our semantics model will need some sort of ωcpo-enrichment to support recursion, and it might not be immediately clear what properties of AD should be satisfied for recursive programs.
- **Reasoning about intensional derivatives.** In the smooth setting, we could at least rely on the tangent bundle functor to provide a semantic account of the syntactic transform $\mathcal{D}$ at first-order. There is no equivalent for intensional derivatives in the literature.

3.3.1 Language

We present our extended language in Figure 3.3. It is parametrized by a set of (partial) operations f on booleans (e.g. $=, \min, \max$), natural numbers (e.g. $+, *$), reals (e.g. $\tan, \log, *$) and mixed (e.g. $>$: $\mathbf{real} \times \mathbf{real} \to \mathbb{B}$). Compared to Section 3.2, we now allow arbitrary terms of type $\mathbb{B}$ in the condition of conditionals. We assume that the real primitives represent partial analytic functions, which, as argued by Lee et al. [2020], covers almost all of the primitives encountered e.g. in machine learning and optimization. We assume constants $\underline{c}$ for all the basic types $\mathbb{B}, \mathbb{N}, \mathbf{real}$. The language is now higher-order and contains lambda abstraction and application. It also contains recursive terms $\mu f.t$. For instance, the factorial function could be written as

$$\mu\ \texttt{fact} : \mathbb{N} \to \mathbb{N}.\lambda n : \mathbb{N}.\mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * \texttt{fact}\ (n-1)$$

Similarly to Section 2.4 in Chapter 2, we can easily extend the language to support algebraic data types without any further complication.

τ	::=	types	
	$\mid$ **real**		real numbers
	$\mid$ $\mathbb{B}$		booleans
	$\mid$ $\mathbb{N}$		natural numbers
	$\mid$ $\tau_1 \times \tau_2$		product
	$\mid$ $\tau_1 \to \tau_2$		function
t	::=	terms	
	$\mid$ x		variable
	$\mid$ $\underline{c}$ $\mid$ $f(t_1, \ldots, t_n)$		constants/operations
	$\mid$ **if** t_1 **then** t_2 **else** t_3		conditionals
	$\mid$ $\lambda x.t$ $\mid$ $t\,s$		function abstraction/app.
	$\mid$ $\mu f.t$		recursive function

$$\frac{\Gamma, f : \tau_1 \to \tau_2 \vdash t : \tau_1 \to \tau_2}{\Gamma \vdash \mu f.t : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash t : \mathbb{B} \quad \Gamma \vdash t_2 : \tau \quad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3\ : \tau}$$

Figure 3.3: Types, terms and type system of our language

Example 3.3.1 (Simple recurrent neural network (RNN))**.** *The basic idea of a recurrent neural net is to feed the output of the layer back to it, along with its input, a certain number of times, before the final result is obtained. This is for instance somewhat natural in natural language processing, where the next input is usually the next word, but the feeding back loop serves for tracking long-distance relationships between words. We can abstractly write it in our language as*

$$\lambda \mathit{input} : \mathbf{real}^k.\lambda \mathit{layer} : \mathbf{real}^k \times \mathbf{real}^k \to \mathbf{real}^k.\mu \mathit{RNN} : \mathbb{N} \to \mathbf{real}^k.\lambda n : \mathbb{N}.$$
$$\mathbf{if}\ n = 0\ \mathbf{then}\ \mathit{input}\ \mathbf{else}\ \mathit{layer}\ (\mathit{input}, \mathit{RNN}\ (n-1))$$

where `layer` *represents one layer on the recurrent neural network (which is usually a whole neural network), and* `input` *represents the input to the neural network, which in this case is assumed to be given as a tuple of reals, as is often the case in practice.*

3.3.2 AD macro

We now present the AD transform on the types and terms of our extended language in Figure 3.4. Similarly to all the AD macros we previously defined in this dissertation (Figures 2.5,2.8,2.11), the transformation is homomorphic in all the constructs of the language, and the non-trivial action happens at the primitives, i.e. AD is structure-preserving on the types and terms of our language. For each type τ, we define a dual number type $\mathcal{D}[\![\tau]\!]$, by induction on types. The AD macro translates terms of type τ into terms of type $\mathcal{D}(\tau)$. For all the primitives f of the language, we assume given primitives $f_{\mathcal{D}}$ that are the AD translation of the primitives, with the appropriate type. For instance, as $>$: $\mathbf{real} \times \mathbf{real} \to \mathbb{B}$, we have $>_{\mathcal{D}}$: $(\mathbf{real} \times \mathbf{real}) \times (\mathbf{real} \times \mathbf{real}) \to \mathbb{B}$. As said in Section 3.2, this is not a limitation as in practice all these primitives can be implemented as composition of simpler primitives, but this choice simplifies the presentation. For instance, if a primitive f represents a partial function $\mathbb{R}^n \to \mathbb{R}$, then we could assume that for every $1 \leq i \leq n$, we have the primitive $\partial_i f$ representing the i-th partial (intensional) derivative of f. In this case, $f_{\mathcal{D}}$ can be implemented as

$$f_{\mathcal{D}} \stackrel{\text{def}}{=} \lambda dx_1, \ldots \lambda dx_n.\mathbf{case}\, dx_1\, \mathbf{of}\, \langle x_1, \delta x_1\rangle \to \ldots \to \mathbf{case}\, dx_n\, \mathbf{of}\, \langle x_n, \delta x_n\rangle \to \\ \langle f(x_1, \ldots, x_n), \partial_1 f(x_1, \ldots, x_n) * \delta x_1 + \ldots + \partial_n f(x_1, \ldots, x_n) * \delta x_n\rangle$$

As in Sections 2.2.3 and 2.6 in Chapter 2, our macro in functorial, i.e. it preserves typing, substitution, and $\beta\eta$-equality.

Lemma 3.3.1 (Functoriality of AD)**.** *For all well-typed terms t_1, t_2 of the language, we have:*

- *If $\Gamma \vdash t_1 : \tau$, then $\mathcal{D}(\Gamma) \vdash \mathcal{D}(t_1) : \mathcal{D}(\tau)$.*
- *$\mathcal{D}(t_1[t_2/x]) = \mathcal{D}(t_1)[\mathcal{D}(t_2)/x]$.*
- *If $t_1 \stackrel{\beta\eta}{=} t_2$, then $\mathcal{D}(t_1) \stackrel{\beta\eta}{=} \mathcal{D}(t_2)$.*

3.3.3 Operational semantics

We now define a standard call-by-value big-step operational semantics for our language. The values are given by the following grammar

$$v ::= x \mid c \mid f \mid \lambda x.t \mid \mu f.t \mid \langle v_1, v_2\rangle$$

The operational semantics $\Downarrow$ is given in Figure 3.5.

$$\mathcal{D}[\![\mathbf{real}]\!] = \mathbf{real} \times \mathbf{real}$$
$$\mathcal{D}[\![\mathbb{B}]\!] = \mathbb{B}$$
$$\mathcal{D}[\![\mathbb{N}]\!] = \mathbb{N}$$
$$\mathcal{D}[\![\tau_1 \times \tau_2]\!] = \mathcal{D}[\![\tau_1]\!] \times \mathcal{D}[\![\tau_2]\!]$$
$$\mathcal{D}[\![\tau_1 \to \tau_2]\!] = \mathcal{D}[\![\tau_1]\!] \to \mathcal{D}[\![\tau_2]\!]$$

$$\begin{array}{lcl}
\mathcal{D}(x) & \stackrel{\text{def}}{=} & x \\
\mathcal{D}(\underline{c} : \mathbf{real}) & \stackrel{\text{def}}{=} & \langle \underline{c}, 0 \rangle \\
\mathcal{D}(\underline{c} : \mathbb{B}) & \stackrel{\text{def}}{=} & \underline{c} \\
\mathcal{D}(\underline{c} : \mathbb{N}) & \stackrel{\text{def}}{=} & \underline{c} \\
\mathcal{D}(f(t_1, \ldots, t_n)) & \stackrel{\text{def}}{=} & f_{\mathcal{D}}(\mathcal{D}(t_1), \ldots, \mathcal{D}(t_n)) \\
\mathcal{D}(\lambda x.t) & \stackrel{\text{def}}{=} & \lambda x.\mathcal{D}(t) \\
\mathcal{D}(t\, s) & \stackrel{\text{def}}{=} & \mathcal{D}(t)\, \mathcal{D}(s) \\
\mathcal{D}(\langle t_1, t_2 \rangle) & \stackrel{\text{def}}{=} & \langle \mathcal{D}(t_1), \mathcal{D}(t_2) \rangle \\
\mathcal{D}(\mathbf{case}\, t\, \mathbf{of}\, \langle x_1, x_2 \rangle \to s) & \stackrel{\text{def}}{=} & \mathbf{case}\, \mathcal{D}(t)\, \mathbf{of}\, \langle x_1, x_2 \rangle \to \mathcal{D}(s) \\
\mathcal{D}(\mu f.t) & \stackrel{\text{def}}{=} & \mu f.\mathcal{D}(t) \\
\mathcal{D}(\mathbf{if}\, t_1\, \mathbf{then}\, t_2\, \mathbf{else}\, t_3\,) & \stackrel{\text{def}}{=} & \mathbf{if}\, \mathcal{D}(t_1)\, \mathbf{then}\, \mathcal{D}(t_2)\, \mathbf{else}\, \mathcal{D}(t_3)
\end{array}$$

Figure 3.4: AD macro for our higher-order recursive language

$$\frac{}{\underline{c} \Downarrow \underline{c}} \qquad \frac{t_2 \Downarrow \lambda x.t \quad t_1 \Downarrow v_1 \quad t[v_1/x] \Downarrow v_2}{t_2 t_1 \Downarrow v_2}$$

$$\frac{\forall i, t_i \Downarrow v_i \quad t \Downarrow f}{t(t_1, \ldots, t_n) \Downarrow \underline{f(v_1, \ldots, v_n)}}$$

$$\frac{t_1 \Downarrow \mathsf{True} \quad t_2 \Downarrow v}{\mathbf{if}\, t_1\, \mathbf{then}\, t_2\, \mathbf{else}\, t_3 \Downarrow v}$$

$$\frac{t_1 \Downarrow \mathsf{False} \quad t_3 \Downarrow v}{\mathbf{if}\, t_1\, \mathbf{then}\, t_2\, \mathbf{else}\, t_3 \Downarrow v}$$

$$\frac{t \Downarrow \langle v_1, v_2 \rangle \quad t_2[v_1/x_1, v_2/x_2] \Downarrow v}{\mathbf{case}\, t\, \mathbf{of}\, \langle x_1, x_2 \rangle \to t_2 \Downarrow v}$$

$$\frac{t[\mu f.t/f] \Downarrow v}{\mu f.t \Downarrow v}$$

Figure 3.5: Big step operational semantics

3.4 ωPAP spaces

Recall that **CartSp** is not Cartesian closed. For the same reason, the category where objects are natural numbers n and morphisms $n \to m$ are PAP functions $\mathbb{R}^n \to \mathbb{R}^m$ is also not Cartesian closed. In addition, lacking an obvious ωcpo-structure, it would not support the interpretation of recursion either. We introduce a new convenient setting for differentiable programming, that generalizes PAP functions, with a sheaf-like flavour, like diffeological spaces [Iglesias-Zemmour, 2013] or quasi-Borel spaces [Heunen et al., 2017].

3.4.1 ωPAP

We first define PAP spaces, which are like diffeological spaces except that the plots are now PAP functions.

Definition 3.4.1 (c-analytic set)**.** *A set $A \subseteq \mathbb{R}^n$ is c-analytic if it is a countable union of analytic subsets of $\mathbb{R}^n$.*

Definition 3.4.2 (PAP space)**.** *A PAP space X is a tuple $(|X|, \mathcal{P}_X)$ of a carrier set $|X|$ with collections $\mathcal{P}_X^A \subseteq |X|^A$, called plots in X, for each c-analytic set A, satisfying:*

1. *Every map from the one point of $\mathbb{R}^0$ to $|X|$ is a plot in X.*
2. *If $\phi \in |X|^A$ is a plot in X and $f : A' \to A$ is a PAP function between c-analytic sets A' and A, then $\phi \circ f$ is a plot in X.*
3. *Suppose the c-analytic sets $A_j \subseteq A$ form a countable partition of the c-analytic set A, with inclusions $i_j : A_j \to A$. If $\phi \circ i_j$ is a plot in X for every j, then ϕ is a plot in X.*

Definition 3.4.3 (PAP homomorphism)**.** *A homomorphism of PAP spaces $f : X \to Y$ is a map between their carrier sets $|X|$ and $|Y|$ with the property that $f \circ \phi$ is a plot in Y whenever ϕ is a plot in X.*

Definition 3.4.4. *An ωcpo is a partially ordered set closed under least upper bounds of countable chains. A continuous function between ωcpo's is a monotone function that preserves least upper bounds.*

Example 3.4.1. *$(\mathbb{N} \cup \{\infty\}, \leq)$ is an ωcpo where every element is smaller than ∞ and otherwise it's the natural order on $\mathbb{N}$. A continuous function $f : (\mathbb{N} \cup \{\infty\}, \leq) \to (\mathbb{N} \cup \{\infty\}, \leq)$ is an increasing function $f : \mathbb{N} \to \mathbb{N}$ and such that $f(\infty) = \infty$.*

Example 3.4.2 (Discrete-order)**.** *Every set X can be made into an ωcpo with the discrete order $=$.*

Example 3.4.3. *Every set X can be made into an ωcpo, by adding a new least element $\perp$, and with the discrete order $=$ on X.*

More generally, every ωcpo $(X, \leq)$, $X \sqcup \{\perp\}$ can be made into an ωcpo where the order on X is the same, and $\perp \leq x$ for every $x \in X$.

ωPAP-spaces are PAP with an extra ωcpo-structure and an extra closure condition for the plots.

Definition 3.4.5 (ωPAP space)**.** *An ωPAP space consists of a triple $(|X|, \mathcal{P}_X, \leq_X)$ such that $(|X|, \mathcal{P}_X)$ is a PAP space and $(|X|, \leq_X)$ is an ωcpo, satisfying the following additional condition. Whenever $(\alpha_i)_i$ is an ω-chain in $\mathcal{P}_X^A$ under the pointwise order, $(\bigvee_{i\in\mathbb{N}} \alpha_i)(x) := \bigvee_{i\in\mathbb{N}}(\alpha_i(x))$ defines a plot $\bigvee_{i\in\mathbb{N}} \alpha_i \in \mathcal{P}_X^A$.*

Homomorphisms of ωPAP spaces $X \to Y$ are defined to be functions from $|X|$ to $|Y|$ that are both PAP morphisms and ωcpo morphisms. We write ωPAP for the category of these spaces and homomorphisms.

Definition 3.4.6 (ωPAP)**.** *We call ωPAP the category whose objects are ωPAP spaces and whose morphisms are ωPAP morphisms between them.*

3.4.2 Examples of ωPAP spaces

We now look at several examples of ωPAP-spaces. The proof that all of these examples are indeed ωPAP-spaces, satisfying the usual universal properties, is postponed to Section 3.6 in Part II.

Example 3.4.4 (PAP-spaces)**.** *Every PAP-space $X := (|X|, \mathcal{P}_X)$ can be seen as a ωPAP-space where the order is given by equality.*

Example 3.4.5. *Any analytic set A can be given the structure of an ωPAP-space $(A, \mathcal{P}_A, =)$ where for every c-analytic set C, the plots $\mathcal{P}_A^C$ are given by*

$$\mathcal{P}_A^C := \{f : C \to A \mid f \text{ is a PAP function}\}$$

More generally, any c-analytic set can be given the structure of ωPAP-space $(A, \mathcal{P}_A, =)$ in the same way, with

$$\mathcal{P}_A^C := \{f : C \to A \mid f \text{ is a PAP function}\}$$

With the example above, we see that PAP functions $f : A \to B$ between c-analytic sets A, B can be seen as ωPAP-morphisms between ωPAP-spaces. More generally, arbitrary subsets of $\mathbb{R}^n$ can be equipped with an ωPAP-structure.

Example 3.4.6 (Arbitrary subsets of $\mathbb{R}^n$.)**.** *As in the previous example, we equip sets $U \subseteq \mathbb{R}^n$ with the discrete partial order and take as plots $\mathcal{P}_U^A$ the PAP functions from A to U.*

Example 3.4.7 (Domain restriction)**.** *If Y is a ωPAP space, $X \subseteq Y$ a subspace, and $f : Y \to Z$ is a ωPAP-morphism, then $f|_X : X \to Z$, the restriction of f to X, is a ωPAP-morphism.*

Example 3.4.8 (Codomain restriction)**.** *If Y is a ωPAP space, $f : Y \to Z$ is a ωPAP-morphism, $X \subseteq Z$ a subspace containing $f(Y)$, then $f : Y \to X$, the corestriction of f to X, is a ωPAP-morphism.*

Example 3.4.9 (Products)**.** *Given ωPAP spaces X and Y, define $X \times Y$ to be their product as ωcpo's, with plots $\langle f, g\rangle \in \mathcal{P}^A_{X\times Y}$ whenever $f \in \mathcal{P}^A_X$ and $g \in \mathcal{P}^A_Y$.*

More generally, given ωPAP spaces X_i for $i \in I$, we can define $\prod_{i\in I} X_i$ by their product as ωcpos, with plots $f \in \mathcal{P}^A_{\prod_{i\in I} X_i}$ whenever each projection $\pi_i \circ f \in \mathcal{P}^A_{X_i}$.

Example 3.4.10 (Coproducts)**.** *Let I be a countable index set, and X_i a ωPAP space for each $i \in I$. Then $\bigsqcup_i X_i$ is again an ωPAP space, with carrier set $\bigsqcup_i |X_i|$, and the partial order inherited from each X_i. A function $f : A \to \bigsqcup_i |X_i|$ is a plot if there exists a countable partition $\{A_j\}_j$ of A into analytic sets, such that for each j, there is a plot $f_j : U_j \supseteq A_j \to X_{k_j}$, where $k_j \in I$, such that on A_j, $f(x) = \mathbf{in}_{k_j} f_j(x)$, i.e., if the preimage of each X_i under f is a countable union of disjoint analytic sets for each i.*

Example 3.4.11 (Exponentials)**.** *Let X and Y be ωPAP spaces. Then Y^X is again a ωPAP space, with carrier set $\omega PAP(X, Y)$. The partial order is the pointwise partial order. A function $f : A \to |Y^X|$ is a plot if $\boldsymbol{uncurry}\, f : A \times X \to Y$ is an ωPAP morphism.*

Example 3.4.12 (Algebraic data types)**.** *If X is a ωPAP-space, then $\mathbf{list}(X)$, the set of finite lists of elements of X, is a ωPAP-space, where the set is given as in the category of sets and functions, and where the plots are given inductively as follows:*

$$\mathcal{P}^A_{\mathbf{list}(X)} = \{f : A \to \mathbf{list}(X) \mid f = \iota_1 \text{ or } f = \iota_{X\times\mathbf{list}(X)} \circ \langle f_1, f_2\rangle, f_1 \in \mathcal{P}^A_X, f_2 \in \mathcal{P}^A_{\mathbf{list}(X)}\}$$

More generally, the same idea will work for any algebraic data type.

3.4.3 A partiality monad on ωPAP

To interpret functions which can be partial, and recursion, we need a way to encode partiality semantically. This is a non-trivial challenge, and it is possible to formulate reasonable-sounding solutions that turn out not to work. For instance, we might think that partial functions $f : U \rightharpoonup V$ definable using recursion are almost PAP in the following sense: there exists an analytic partition $\{A_i\}_{i\in I}$ of U, and analytic functions $\{f_j\}_{j\in J}$ for some subset of indices $J \subseteq I$, such that f is defined exactly on $\bigcup_{j\in J} A_j$ and $f(x) = f_j(x)$ whenever $x \in A_j$. This is the definition we would obtain if a partial function $f : U \rightharpoonup V$ is characterized by a total function $U \to V + 1$ where $+$ is the coproduct of ωPAP-spaces. However, consider the program `cantor` in Figure 3.6. It denotes a partial function that is undefined outside of $[0, 1]$, and also on the $\frac{1}{3}$-Cantor set. This region *cannot* be expressed as a countable union of analytic sets $\cup_{i\in I\setminus J} A_i$, and therefore it cannot denote a ωPAP-function $\mathbb{R} \to \mathbb{R} + 1$ (see Proposition 3.4.1). Therefore, this candidate notion of PAP partial function is too restrictive: some recursive programs do not satisfy it.

```
def cantor (x : R) : R
    if x <= 1/3 then
        return cantor(3*x)
    elif x <2/3 then
        return x*x
    else
        return cantor(3*x -2)
```

Figure 3.6: This program can be seen as a piecewise function, constantly $\bot$ on the $\frac{1}{3}$-Cantor set, but defined elsewhere in $(0, 1)$.

Proposition 3.4.1. *The $\frac{1}{3}$-Cantor set cannot be expressed as a countable union of analytic sets.*

Proof. Each analytic set on $\mathbb{R}$ is an open set subjected to finitely many analytic inequalities. The zero set of a real-analytic function on $\mathbb{R}$ is either countable or whole of $\mathbb{R}$ (by Theorem of isolated zeros). Thus, these finitely many inequalities must together carve out countably many isolated points (e.g., $-sin(x) <= -c$ and $sin(x) <= c$ carve out a countable collection of x's). But the Cantor set is made of uncountably many isolated points, and thus cannot be obtained in this way. □

Instead, we will only force the domain of the partial function to be a c-analytic set, but not the complement of its domain. That is, a partial function $X \to Y$ will be represented by a total function $X \to \mathbf{L}Y$, where $\mathbf{L}Y$ is defined as follows.

- The underlying set is $|\mathbf{L}Y| := |Y| \sqcup \{\bot\}$.
- The order structure is given by $\forall x \in |Y|, \bot \leq_{\mathbf{L}Y} x$ and $\forall x, x' \in |Y|, x \leq_{\mathbf{L}Y} x'$ iff $x \leq_Y x'$.
- Plots are given by

$$\mathcal{P}^A_{\mathbf{L}Y} := \Big\{ g : |A| \to |Y| \sqcup \{\bot\} \mid \exists B \text{ c-analytic s.t. } g^{-1}(|X|) = B \text{ and } g|_B \in \mathcal{P}^B_Y \Big\}$$

This definition is inspired by similar constructs present in Vákár et al. [2019], Vákár [2020a], and we will see in Part II that they are all instances of a more general categorical construction.

We also have ωPAP-morphisms

- $\mathtt{return}_X : X \to \mathbf{L}X$ given by $\lambda x.\mathbf{inl}\ x$
- $\gg\!=_{X,Y} : \mathbf{L}X \times (X \to \mathbf{L}Y) \to \mathbf{L}Y$ given by $\lambda(x, g).\mathbf{match}\, x\, \mathbf{with}\, \mathbf{inr}\, \bot \mapsto \mathbf{inr}\, \bot, \mathbf{inl}\, x \mapsto g(x)$

where $\mathbf{inl}, \mathbf{inr}$ are respectively left and right injections (as set functions: they are not ωPAP-morphisms).

$\mathbf{L}$ extends to ωPAP-morphisms f by $\mathbf{L}f(x) = f(x)$ and $\mathbf{L}f(\bot) = \bot$. Overall, we have a (commutative) monad $\mathbf{L} : \omega PAP \to \omega PAP$:

Proposition 3.4.2. *$\mathbf{L} : \omega PAP \to \omega PAP$ is a (strong, commutative) monad.*

The proof is postponed to Section 3.6 in Part II.

Given a ωPAP-morphism $f : A \to \mathbf{L}B$ where A, B are c-analytic sets, we write $\widetilde{f} : Dom(f) \to B$ for its corresponding partial function, which is PAP.

3.4.4 Denotational semantics

We can give a denotational semantics to our language using ωPAP-spaces and morphisms. A program $\Gamma \vdash t : \tau$ is interpreted as a ωPAP-morphism $[\![t]\!] : [\![\Gamma]\!] \to \mathbf{L}[\![\tau]\!]$, where $\mathbf{L}$ now indicates that programs represent partial functions. In particular, first-order programs t are interpreted as partial PAP functions, which generalizes the semantics from the previous section. Substitution is no longer given by composition in the category, but by composition in the Kleisli category, which is the usual composition of effectful programs.

The semantics of types is given by

$$\begin{array}{lcl}
[\![\mathbf{real}]\!] & \stackrel{\text{def}}{=} & (\mathbb{R}, PAP(-,\mathbb{R}), =) \\
[\![\mathbb{B}]\!] & \stackrel{\text{def}}{=} & 1+1 \\
[\![\mathbb{N}]\!] & \stackrel{\text{def}}{=} & \sum_{i\in\mathbb{N}} 1 \\
[\![(\tau_1 \times \ldots \times \tau_n)]\!] & \stackrel{\text{def}}{=} & \prod_{i=1}^n [\![\tau_i]\!] \\
[\![\tau \to \sigma]\!] & \stackrel{\text{def}}{=} & \omega PAP([\![\tau]\!], \mathbf{L}[\![\sigma]\!])
\end{array}$$

A context $\Gamma = (x_1 \colon \tau_1 \ldots x_n \colon \tau_n)$ is interpreted as the product space $[\![\Gamma]\!] \stackrel{\text{def}}{=} \prod_{i=1}^n [\![\tau_i]\!]$. The semantics of well-typed terms is given by

$$\begin{array}{lcl}
[\![x]\!](\rho) & \stackrel{\text{def}}{=} & \mathbf{return}\ \rho(x) \\
[\![\underline{c}]\!](\rho) & \stackrel{\text{def}}{=} & \mathbf{return}\ c \\
[\![f(t_1 \ldots t_n)]\!](\rho) & \stackrel{\text{def}}{=} & [\![t_1]\!](\rho) \gg\!= (\lambda x_1.[\![t_2]\!](\rho) \\
 & & \quad \gg\!= \ldots \gg\!= (\lambda x_n. f(x_1, \ldots, x_n))) \\
[\![\langle t_1, t_2\rangle]\!](\rho) & \stackrel{\text{def}}{=} & [\![t_1]\!](\rho) \gg\!= (\lambda x.[\![t_2]\!](\rho) \gg\!= (\lambda y.\mathbf{return}\ (x,y))) \\
[\![\lambda x : \tau.t]\!](\rho) & \stackrel{\text{def}}{=} & \mathbf{return}\ \lambda v.[\![t]\!](\rho[x \mapsto v]) \\
[\![\texttt{case}\ t_1 \texttt{of}\ \langle x_1, x_2\rangle \to t_2]\!] & \stackrel{\text{def}}{=} & [\![t_1]\!](\rho) \gg\!= (\lambda v.[\![t_2]\!](\rho[x \mapsto v])f) \\
[\![t_1\, t_2]\!](\rho) & \stackrel{\text{def}}{=} & [\![t_1]\!](\rho) \gg\!= (\lambda f.[\![t_2]\!](\rho) \gg\!= f) \\
[\![\texttt{if}\ t_1 \texttt{then}\ t_2 \texttt{else}\ t_3]\!](\rho) & \stackrel{\text{def}}{=} & [\![t_1]\!](\rho) \gg\!= \lambda b.\mathbf{if}\ b\ \mathbf{then}\ [\![t_2]\!](\rho) \mathbf{else}\ [\![t_3]\!](\rho) \\
[\![\mu f.\lambda x.t]\!](\rho) & \stackrel{\text{def}}{=} & \mathbf{return}\ \bigvee_{i\in\mathbb{N}} f_i
\end{array}$$

where $f_0 = \lambda x.\mathbf{inr}\ \bot$ and $f_{i+1} = \lambda v.[\![t]\!](\rho[f \mapsto f_i, x \mapsto v])$.

We can give a standard call-by-value evaluation to our language, e.g. following Pitts [2012]. Then, our denotational semantics in ωPAP is sound and adequate.

Theorem 3.4.3. *The denotational semantics in ωPAP of our language is sound and adequate w.r.t. the operational semantics (Section 3.3.3).*

The proof will be given in Section 3.6.

Definition 3.4.7. *A ground type is a type from the grammar of our language (Figure 3.3) which does not involve arrow types ($\tau_1 \to \tau_2$). When we further restrict the type to not contain $\mathbb{B}$ or $\mathbb{N}$, we call it a real ground type. A (real) ground context is a context where all the variables have (real) ground type.*

In addition to giving a sound and adequate semantics to our language, ωPAP-spaces are conservative over PAP functions defined on c-analytic sets (see Theorem 3.6.2). This implies the following result

Proposition 3.4.4. *Let $\Gamma \vdash t : \tau$ be a well-typed term, where Γ is a real ground context and τ a real ground type. Then $[\![t]\!] : [\![\Gamma]\!] \to \mathbf{L}[\![\tau]\!]$ represents a PAP function $f : A \to [\![\tau]\!]$ defined on a c-analytic set A. It is in particular almost-everywhere differentiable.*

The proof of Proposition 3.4.4 will also be given in Section 3.6.

3.5 Correctness of AD

To show the correctness of our AD macro, we first need to refine Definition 3.2.7 to account for partial functions defined on a c-analytic set A instead of the whole of $\mathbb{R}^n$.

Definition 3.5.1 (Dual-number intensional representation)**.** *Let $A \subseteq \mathbb{R}^n$ and $B \subseteq \mathbb{R}^m$ be c-analytic sets. A function $g : A \times \mathbb{R}^n \to B \times \mathbb{R}^m$ is a dual-number intensional representation of a PAP function $f : A \to B$ if for all $x \in A, dx \in \mathbb{R}^n$ we have*

$$g(x, dx) = (f(x), df(x) \cdot dx)$$

where $\cdot$ is matrix-vector product, and df is an intensional derivative of f.

Similarly to intensional derivatives, dual-number intensional representations compose:

Proposition 3.5.1. *Let $f_1 : A \to B$ and $f_2 : B \to C$ be PAP, and g_1, g_2 be dual-number intensional representations of f_1, f_2 respectively. Then, $g_2 \circ g_1$ is a dual-number intensional representation of $f_2 \circ f_1$.*

In the smooth case in the previous chapter, we proved that AD is correct in the sense that on every first-order term t (i.e. a term with a ground type in a ground context), AD produces a term $\mathcal{D}(t)$ that computes a dual-number representation of t. The relied on the fact on the fact that the input and output terms of AD represent smooth functions. In this generalized setting, the analogue definition of correctness for AD is that it computes *some* dual-number intensional representation of t.

Theorem 3.5.2 (Semantic correctness of $\mathcal{D}$ (limited))**.** *For any term* $x_1 : \mathbf{real}, \ldots, x_n : \mathbf{real} \vdash t : \mathbf{real}$*, the PAP function* $[\![\overrightarrow{\mathcal{D}}(t)]\!] : A \times \mathbb{R}^n \to \mathbb{R} \times \mathbb{R}$ *is a dual-number intensional representation of* $[\![t]\!] : A \subseteq \mathbb{R}^n \to \mathbb{R}$*.*

Theorem 3.5.2 will be proved as a special case of the following Theorem 3.5.3. More generally, this will hold for every first-order term as for any real ground context Γ, $[\![\Gamma]\!]$ is isomorphic to $\mathbb{R}^n$ for some n, and likewise, for every real ground type τ, $[\![\tau]\!]$ is isomorphic to $\mathbb{R}^m$ for some m.

Theorem 3.5.3 (Semantic correctness of $\mathcal{D}$)**.** *For any term well-typed term* $\Gamma \vdash t : \tau$ *of real ground type* τ *in a real ground context* Γ*, the PAP function* $[\![\overrightarrow{\mathcal{D}}(t)]\!] : A \times \mathbb{R}^n \to \mathbb{R}^m \times \mathbb{R}^m$ *is a dual-number intensional representation of* $[\![t]\!] : A \subseteq \mathbb{R}^n \to \mathbb{R}^m$*.*

In particular, $\mathcal{D}$ produces correct gradients almost-everywhere. To visualize this with a diagram, we define a binary relation $\mathcal{R}$ such that $f \mathcal{R} g$ if f is PAP and g a dual-number intensional representation of f. Recall that in the smooth case, this binary relation was functional, with $f \mathcal{R} T(f)$, and T could even be made into a functor. Using the notation $f \leftsquigarrow g$ if $f \mathcal{R} g$, Theorem 3.5.3 can be summarised as the following diagram:

$$\begin{array}{ccc} \Gamma \vdash t : \tau & \xrightarrow{\quad \mathcal{D}(-) \quad} & \mathcal{D}\Gamma \vdash \mathcal{D}t : \mathcal{D}\tau \\ {\scriptstyle [\![-]\!]}\downarrow & & \downarrow{\scriptstyle [\![-]\!]} \\ [\![\Gamma \vdash t : \tau]\!] & \leftsquigarrow & ([\![\mathcal{D}\Gamma \vdash \mathcal{D}t : \mathcal{D}\tau]\!]) \end{array}$$

The proof of Theorem 3.5.3 is delayed until Section 3.7.

Part II 

As discussed in Section 3.1, in this part we first develop the categorical machinery allowing us to show that

(i) ωPAP forms a very well-behaved category, giving a sound and adequate denotational semantics to our higher-order recursive language (Section 3.6).

(ii) Our AD transform (Figure 3.4) is correct (Theorem 3.5.3), i.e. it computes dual-number intensional representations (Section 3.7).

3.6 ωPAP as a category of ω-concrete sheaves

In this section, we will recall some of the formalism developed in Matache, Moss, and Staton [2022] on categories of concrete sheaves with recursion. They form very-well behaved categories that give sound and adequate models for fine-grained call-by-value PCF languages. We will show that ωPAP is an instance of such a category, and gives a sound and adequate semantics to our language (Figure 3.3).

3.6.1 Background: Categories of concrete sheaves with recursion

A category of sheaves is a well-behaved subcategory of a category of presheaves. In particular, they are locally-presentable and so complete and cocomplete [Adámek and Rosicky, 1994]. The intuition is that they are well-chosen presheaves that 'believe' in certain colimits in the base category. For instance, it is well-known that the Yoneda embedding $\mathcal{Y} : \mathbf{C} \to \widehat{\mathbf{C}}$ from a small category $\mathbf{C}$ to its category of presheaves $\widehat{\mathbf{C}}$ does not preserve all colimits that $\mathbf{C}$ may have. This means for instance that for objects A, B of $\mathbf{C}$ for which the coproduct exists, there is a presheaf P such that $\widehat{\mathbf{C}}(\mathcal{Y}(A+B), P) \ncong \widehat{\mathbf{C}}(\mathcal{Y}(A), P) \times \widehat{\mathbf{C}}(\mathcal{Y}(B), P)$. One can e.g. take $Pc \stackrel{\text{def}}{=} \mathcal{Y}(A)c \sqcup \mathcal{Y}(B)c$. Intuitively, P 'does not perceive' that $\mathcal{Y}(A+B)$ is the coproduct of $\mathcal{Y}(A)$ and $\mathcal{Y}(B)$. If, on the other hand, we restrict to the full subcategory $\widetilde{\mathbf{C}}$ of $\widehat{\mathbf{C}}$ of objects for which the canonical map $\widehat{\mathbf{C}}(\mathcal{Y}(A), P) \times \widehat{\mathbf{C}}(\mathcal{Y}(B), P) \to \widehat{\mathbf{C}}(\mathcal{Y}(A+B), P)$ is an isomorphism, then $\mathcal{Y} : \mathbf{C} \to \widetilde{\mathbf{C}}$ will preserve coproducts. The theory of Grothendieck sheaves is the generalization of this idea, ensuring that $\mathcal{Y}$ preserves certain colimits and that $\widetilde{\mathbf{C}}$ is a well-behaved category. When further restricting to a subcategory of sheaves satisfying a concreteness condition, the objects will behave like sets with structure [Baez and Hoffnung, 2011]. One can show that e.g. diffeological spaces [Iglesias-Zemmour, 2013], Quasi-Borel spaces [Heunen et al., 2017], and PAP-spaces (Definitions 3.4.2 and 3.4.3) are of this form.

3.6.1.1 Concrete sites and sheaves

In this dissertation, we are only interested in concrete sheaves, and will not need the full theory of sheaves. Instead, we recall the minimum setting from Matache et al. [2022].

Definition 3.6.1. *A concrete category is a category* $\mathbf{C}$ *with a terminal object* $\star$ *such that the functor* $\mathbf{C}(\star, -) : \mathbf{C} \to \mathbf{Set}$ *is faithful.*

Example 3.6.1. *Let* **OpenCont** *be the category whose objects are open subsets* U *of* $\mathbb{R}^n$ *(for all* $n \in \mathbb{N}$*), and whose morphisms* $U \to V$ *are continuous functions.* $\mathbb{R}^0$ *is a terminal object, and* $\mathbf{OpenCont}(\mathbb{R}^0, -)$ *simply sends an open set to its underlying set, and sends a continuous function to its underlying function. Therefore,* $\mathbf{OpenCont}(\mathbb{R}^0, -)$ *is faithful and* **OpenCont** *is concrete.*

Definition 3.6.2. *A concrete site* $(\mathbf{C}, \mathcal{J})$ *is a small concrete category* $\mathbf{C}$ *with an initial object* 0*, together with a coverage* $\mathcal{J}$*, which specifies for each object* c *a set* $\mathcal{J}(c)$ *of families of maps with codomain* c *. We call such a family* $\{f_i : c_i \to c\}_{i \in I} \in \mathcal{J}(c)$ *a covering family and say it covers* c*. The coverage must satisfy the following axioms.*

1. *For every map* $h : d \to c$ *in* $\mathbf{C}$*, if* $\{f_i : c_i \to c\}_{i \in I}$ *covers* c*, then there is a covering family* $\{g_j : d_j \to d\}_{j \in I'}$ *of* d *such that every* $h \circ g_j$ *factors through some* f_i*.*
2. *If* $\{f_i : c_i \to c\}_{i \in I}$ *covers* c*, then* $\bigcup_{i \in I} Im(|f_i|) = |c|$*.*
3. *The initial object* 0 *is covered by the empty set.*
4. *The identity is always covering.*
5. *If* $\{f_i : c_i \to c\}_{i \in I} \in \mathcal{J}(c)$ *and* $\{g_{ij} : c_{ij} \to c_i\}_{j \in J_i} \in \mathcal{J}(c_i)$ *for each* i*, then* $\{f_i \circ g_{ij} : c_{ij} \to c\}_{i \in I, j \in J_i} \in \mathcal{J}(c)$*.*

Example 3.6.2 (Cartesian spaces and smooth maps)**. CartSp** *is a concrete site, where* $\{f_i : U_i \to V\}_i$ *is a covering family if* $\bigcup_i f_i(U_i) = V$*, i.e. the images of the* f_i *cover the codomain* V *in the usual sense.*

Example 3.6.3. *Similarly to* **CartSp**, **OpenCont** *is a concrete site, where* $\{f_i : U_i \to V\}_i$ *is a covering family if* $\bigcup_i f_i(U_i) = V$*, i.e. the images of the* f_i *cover the codomain* V *in the usual sense. Its initial object is the empty set.*

Example 3.6.4 (Standard Borel spaces)**.** *The category* **Sbs** *has objects the Borel subsets of* $\mathbb{R}$ *and morphisms the measurable functions between these objects. It is a concrete site where the coverage contains the countable sets of inclusions functions* $\{U_i \to U\}_i$ *such that* $U = \bigcup_i U_i$ *and the* U_i *are disjoint.*

Definition 3.6.3. *A concrete sheaf* X *on a concrete site* $(\mathbf{C}, \mathcal{J})$ *is a set* $|X|$*, together with, for each object* $c \in \mathbf{C}$*, a set* $\mathcal{P}_X^c$ *of functions* $|c| \to |X|$*, such that*

- *each* $\mathcal{P}_X^c$ *contains all the constant functions;*
- *for any map* $h : d \to c \in \mathbf{C}$*, and any* $g \in \mathcal{P}_X^c$*, the composite function* $g \circ |h| : |d| \to |X|$ *is in* $\mathcal{P}_X^d$*;*
- *for each function* $g : |c| \to |X|$ *and each covering family* $\{f_i : c_i \to c\}_{i \in I}$*, if each* $g \circ |f_i| \in \mathcal{P}_X^{c_i}$*, then* $g : |c| \to |X| \in \mathcal{P}_X^c$*.*

Definition 3.6.4. *A morphism $\alpha : X \to Y$ between concrete sheaves is a function $\alpha : |X| \to |Y|$ that preserves the structure, namely if $g \in \mathcal{P}_X^c$, then $\alpha \circ g \in \mathcal{P}_Y^c$.*

Example 3.6.5. *Let* **Cont** *be the category defined as follows. Its objects are pairs $(X, \mathcal{P}_X)$ where for each open $U \subseteq \mathbb{R}^n$, $\mathcal{P}_X^U \subseteq X^U$ is closed under the following axioms:*

- *Constant functions are in $\mathcal{P}_X^U$.*
- *If $f \in \mathcal{P}_X^U$ and $g : V \to U$ is continuous, then $f \circ g \in \mathcal{P}_X^V$.*
- *If $\{U_i\}_{i\in\mathbb{N}}$ is an open cover of an open U, and for all i, $f_i \in \mathcal{P}_X^{U_i}$, such that for all $i, j \in \mathbb{N}^2$, f_i and f_j agree on $U_i \cap U_j$, then $f : U \to X$ defined by $x \in U_i \mapsto f_i(x)$, is an element of $\mathcal{P}_X^U$.*

Its morphisms $X \to Y$ are functions $\alpha : X \to Y$ such that $\alpha \circ f \in \mathcal{P}_Y^U$ whenever $f \in \mathcal{P}_X^U$. Then, **Cont** *is a category of concrete sheaves on the concrete site* **OpenCont**.

Example 3.6.6. *Diffeological spaces and Quasi-Borel spaces are examples of categories of concrete sheaves and morphisms between concrete sheaves. Other examples are given in Baez and Hoffnung [2011].*

3.6.1.2 ω-Concrete sheaves

To interpret recursion, the concrete-sheaf structure will not be sufficient in general. Instead of evoking what is to my knowledge the yet-to-be-formalised theory of enriched sheaves over the category of *ωcpo*'s, we continue with the setting from Matache et al. [2022].

Definition 3.6.5. *An ω-concrete sheaf on a site $(\mathbf{C}, \mathcal{J})$ is a concrete sheaf X together with an ordering $\leq_X$ on $|X|$ that equips X with the structure of ωcpo, such that each $\mathcal{P}_X^c$ is closed under pointwise suprema of chains with respect to the pointwise ordering. A morphism $\alpha : X \to Y$ of ω-concrete sheaves is a continuous function between ωcpo's, $\alpha : |X| \to |Y|$, that is also a morphism of concrete sheaves.*

ω-concrete sheaves form a category $\omega Conc(\mathbf{C}, \mathcal{J})$, which is a Cartesian-closed category with binary coproducts [Matache et al., 2022].

Example 3.6.7. *The category of ω-diffeological spaces [Vákár, 2020a] and of ω-Quasi-Borel spaces [Vákár et al., 2019] are examples of categories of ω-concrete sheaves.*

3.6.1.3 Partiality: admissible monos and a partiality monad

To model recursion, we first need to define a (strong) partiality monad $\mathbf{L}$ on $\omega Conc(\mathbf{C}, \mathcal{J})$ (Section 3.4.3). There are many choices for the lifting of the plots, so we parametrize the definition of partiality monad by a class $\mathcal{M}$ of monomorphisms from the site $(\mathbf{C}, \mathcal{J})$, which we call admissible monos.

Recall that, in any category, monos with the same codomain are preordered. If $m_1 : d_1 \rightarrowtail c, m_2 : d_2 \to c$ then $m_1 \leq m_2$ iff there exists $f : d_1 \to d_2$ such that $m_2 \circ f = m_1$ We write $Sub(c)$ for the poset quotient of the set of monos with codomain c. For any class $\mathcal{M}$ of monos, we write $Sub_{\mathcal{M}}(c)$ for the full subposet of $Sub(c)$ whose elements have representatives in $\mathcal{M}$.

Definition 3.6.6. *A class $\mathcal{M}$ of admissible monos for a concrete site $(\mathbf{C}, \mathcal{J})$ consists of, for each object $c \in \mathbf{C}$, a set of monos $\mathcal{M}(c)$ with codomain c satisfying the following conditions*

1. *For all $c \in \mathbf{C}, 0 \to c \in \mathcal{M}(c)$.*
2. *$\mathcal{M}$ contains all isomorphisms.*
3. *$\mathcal{M}$ is closed under composition.*
4. *All pullbacks of $\mathcal{M}$-maps exist and are again in $\mathcal{M}$ (This makes $Sub_{\mathcal{M}}$ a functor $\mathbf{C}^{op} \to \mathbf{Set}$).*
5. *For each c, the function $Sub_{\mathcal{M}}(c) \to \mathbf{Set}(|c|, \{0, 1\})$ is componentwise injective and order-reflecting, and the image of $Sub_{\mathcal{M}}$ is closed under suprema of ω-chains.*
6. *Given an increasing chain in $Sub_{\mathcal{M}}(c)$, $(c_n \rightarrowtail c)_{n\in\mathbb{N}}$, denote its least upper bound by $c_\infty \rightarrowtail c$. Then the closure under precomposition (with any morphism) of the set $\{c_n \rightarrowtail c_\infty\}_{n\in\mathbb{N}}$ contains a covering family of c_∞.*

Now, given a class $\mathcal{M}$ of admissible monos, we can define a partiality monad $L_{\mathcal{M}}$ on the category of ω-concrete sheaves:

Definition 3.6.7. *We define the (strong) partiality monad $\mathbf{L}_{\mathcal{M}}$ associated to the class of admissible monos $\mathcal{M}$ as*

- $|\mathbf{L}_{\mathcal{M}}X| = |X| \sqcup \{\perp\}$
- $\forall x \in |X|, \perp \leq_{\mathbf{L}_{\mathcal{M}}X} x$
- $\forall x, x' \in |X|, x \leq_{\mathbf{L}_{\mathcal{M}}X} x'$ *iff* $x \leq_X x'$
- $$\mathcal{P}^c_{\mathbf{L}_{\mathcal{M}}X} \stackrel{\text{def}}{=} \Big\{ g : |c| \to |X| \sqcup \{\perp\} \mid \exists c' \rightarrowtail c \in \mathcal{M}(c) \text{ s.t. } g^{-1}(|X|) = Im(|c'|) \\ \text{and } g|_{Im(|c'|)} \in \mathcal{P}^{c'}_X \Big\}$$

The strong monad structure is exactly the same as the maybe monad on **Set**.

Proposition 3.6.1 ([Matache et al., 2022])**.** $\mathbf{L}_{\mathcal{M}} : \omega Conc(\mathbf{C}, \mathcal{J}) \to \omega Conc(\mathbf{C}, \mathcal{J})$ *is a strong monad.*

3.6.1.4 Conservativity result

This is best summarised by quoting Matache et al. [2022]:

> Suppose we write programs in a language with higher-order recursion and a type **real** of real numbers. If all the primitive functions are continuous, does that mean that the definable functions **real** → **real** are all continuous? In general, suppose we have some set X and a class $\mathcal{C}$ of operations over it. If we write programs over X in a language with higher-order recursion, are the definable functions **real** → **real** in the class $\mathcal{C}$? In a language with recursion, the definable functions need not terminate and so might be partial. Thus, more precisely, we should investigate the definable partial functions **real** → **real**. We would characterize these partial functions, and their domains of definition. For example, if all the primitive operations are continuous, then we might prove that the definable functions **real** → **real** are partial continuous functions whose domain is an open set. The theory of ω-concrete sheaves is a good setting for this.

Definition 3.6.8 (Subcanonical site)**.** *A site $(\mathbf{C}, \mathcal{J})$ is subcanonical if every representable presheaf $\mathbf{C}(-, A)$ is a concrete sheaf.*

Example 3.6.8. CartSp, Sbs, OpenCont *are all subcanonical. This boils down to the following facts. All constant functions are smooth/measurable/continuous respectively. Smooth/measurable/continuous functions compose. Smoothness/continuity are local properties, and measurable functions are stable under countable gluing.*

Definition 3.6.9. *Let $\mathbf{C}$ be any category of $\mathcal{M}$ any class of monos containing the isomorphisms, closed under composition, and all of whose pullbacks exist and are again in $\mathcal{M}$. Then $p\mathbf{C}_{\mathcal{M}}$, the category of $\mathcal{M}$-partial maps in $\mathbf{C}$, has the same objects as $\mathbf{C}$ but morphisms $c \to b$ are equivalence classes of pairs $(m : d \rightarrowtail c, f : d \to b)$ with $m \in \mathcal{M}$, where $(m_1 : d_1 \rightarrowtail c, f_1 : d_1 \to b)$ is equivalent to $(m_2 : d_2 \rightarrowtail c, f_2 : d_2 \to b)$ iff there exists an isomorphism $i : d_1 \to d_2$ such that $m_2 \circ i = m_1$ and $f_2 \circ i = f_1$. Composition is given by pullback.*

We denote by $Kl_T(\mathbf{C})$ the Kleisli category for the monad T on the category $\mathbf{C}$.

Theorem 3.6.2 ([Matache et al., 2022])**.** *If $(\mathbf{C}, \mathcal{J})$ is a subcanonical concrete site with an admissible class of monos $\mathcal{M}$, then the functors $\mathbf{C} \to \omega Conc(\mathbf{C}, \mathcal{J})$ and $p\mathbf{C}_{\mathcal{M}} \to Kl_{\mathbf{L}_{\mathcal{M}}}(\omega Conc(\mathbf{C}, \mathcal{J}))$ are full and faithful.*

3.6.1.5 Soundness and adequacy

Matache et al. [2022] defined a language PCF_v, a call-by-value variant of PCF. It has product, sum types, and a recursion operator similar to ours, and is presented as a fine-grained call-by-value language. This language is essentially equivalent to our language (Figure 3.3), except that we have extra base types **real** and $\mathbb{B}$, and more basic primitives involving these types. Our conditional is definable using their

sum types. They give their language a standard call-by-value operational semantics, which translates directly to an operational semantics for our language. In summary, even though their theorem is stated for PCF_v, it directly applies to our language, without needing further changes or extra proofs, as long as we also interpret our base types and primitives in the model, as is standard Moggi [1991].

Theorem 3.6.3 ([Matache et al., 2022])**.** *A concrete site with a class of admissible monos,* $(\mathbf{C}, \mathcal{J}, \mathcal{M})$*, presents a sound and adequate model of PCF$_v$ in* $\omega Conc(\mathbf{C}, \mathcal{J})$*.*

3.6.2 ωPAP: a category of ω-concrete sheaves

Having recalled the general setting of ω-concrete sheaves, we now show how ωPAP is an instance of this general construction. To do so, we will define the category $cPAP$ and show it is a concrete-site (which is the one for ωPAP) in Section 3.6.2.2. We will define a set of admissible monos for $cPAP$ in Section 3.6.2.3. In order to achieve this, we will need a key technical proposition (Proposition 3.6.4) and a few other technical lemmas, proved in Section 3.6.2.1.

3.6.2.1 C-analytic sets

The definition of c-analytic sets does not require that the analytic sets that make up a c-analytic set be disjoint, but it turns out that it is always possible to partition a c-analytic set into countably many pairwise disjoint analytic sets.

Proposition 3.6.4. *For any c-analytic set A, there exists a countable collection of pairwise disjoint analytic sets $\{A_i\}_{i\in\mathbb{N}}$ such that $A = \cup_{i\in\mathbb{N}} A_i$.*

This is an important property, and most of the rest of this section is devoted to proving the proposition. The proof is due to Alex Lew.

Let $\{A_i\}_{i\in\mathbb{N}}$ be a countable set of analytic subsets of $\mathbb{R}^n$.

Each analytic set A_i is associated with an open domain U_i and a finite number J_i of analytic functions $\{f_{ij}\}_{j\in[1,\ldots,J_i]}$, so that $A_i = \{x \in U_i \mid \forall j. f_{ij} \leq 0\}$. Because any open set is a countable union of open balls in $\mathbb{R}^n$, we can assume without loss of generality in the definition of a c-analytic set A that the U_i are open balls.

Definition 3.6.10. *A simple analytic set is a set $A = \{x \in \mathcal{B}_\epsilon(x_0) \mid \forall i \leq I. g_i^+(x) > 0 \wedge \forall j \leq J. g_i^-(x) \leq 0\}$, for natural numbers I and J and some $0 < \epsilon \leq \infty$ and $x_0 \in \mathbb{R}^n$.*

Lemma 3.6.5. *The complement of any simple analytic set is a finite union of pairwise disjoint simple analytic sets.*

Proof. A point x can fail to be in a simple analytic set A for $I + J + 1$ mutually exclusive reasons, each of which applies to a simple analytic set of points:

1. If $\epsilon < \infty$, it can fail to lie in $\mathcal{B}_\epsilon(x_0)$. The set of points outside $\mathcal{B}_\epsilon(x_0)$ is simple analytic: $h_1^-(x) := \epsilon - ||x - x_0||_2^2$ is real analytic, and consider $\{x \in \mathbb{R}^n \mid h_1^-(x) \leq 0\}$.

2. (For each $1 \leq i \leq I$.) It can lie within $\mathcal{B}_\epsilon(x_0)$, and satisfy $g_n^+(x) > 0$ for all $n < i$, but fail to satisfy $g_i^+(x) > 0$. Let $h_n^+(x) = g_n^+(x)$ for $n < i$, and $h_1^-(x) = g_i^+(x)$. Then the set of all such points is $\{x \in \mathcal{B}_\epsilon(x_0) \mid \forall n < i.h_n^+(x) > 0 \wedge h_1^-(x) \leq 0\}$.

3. (For each $1 \leq j \leq J$.) It can lie within $\mathcal{B}_\epsilon(x_0)$, and satisfy $g_i^+(x) > 0$ for all $i \leq I$, and satisfy $g_n^-(x) \leq 0$ for all $n < j$, but fail to satisfy $g_j^-(x) \leq 0$. Let $h_i^+(x) = g_i^+(x)$ for all $i \leq I$, $h_n^-(x) = g_n^-(x)$ for all $n < j$, and $h_{I+1}^+(x) = g_j^-(x)$. Then the set of all points to which this reason applies is $\{x \in \mathcal{B}_\epsilon(x_0) \mid \forall i \leq I + 1.h_i^+(x) > 0 \wedge \forall n < j.h_n^-(x) \leq 0\}$.

The union of these simple analytic sets is the complement of A. □

Corollary 3.6.5.1. *If $A_1, \ldots, A_n$ are simple analytic sets, then $\overline{\cup_{i \leq n} A_i}$ is a finite union of disjoint analytic sets.*

Proof. We have $\overline{\cup A_i} = \cap \overline{A_i} = \cap(\cup_{m \leq M_i} B_{im})$, where $\{B_{im}\}_{m \leq M_i}$is a representation of A_i's complement as a disjoint union of finitely many simple analytic sets. Distributing, this is in turn equal to $\cup_{m_1 \leq M_1} \cdots \cup_{m_n \leq M_n} (\cap_{i \leq n} B_{im_i})$. Each element of this large union is analytic, because it is the intersection of n analytic sets. Furthermore, these analytic sets are pairwise disjoint: any pair will disagree on m_i for some i, and so the intersection will include disjoint sets B_{im_i} and $B_{im_i'}$, for $m_i \neq m_i'$. Because these two sets are disjoint, the overall intersections will be disjoint. □

Corollary 3.6.5.2. *If $A_1, \ldots, A_n$ are simple analytic sets, then $A_n \setminus \cup_{i<n} A_i$ is a finite union of disjoint analytic sets, each of which is a finite intersection of simple analytic sets.*

Proof. $A_n \setminus \cup_{i<n} A_i = A_n \cap (\overline{\cup_{i<n} A_i})$, and by the previous corollary, the right-hand term can be rewritten as a finite union of *disjoint* analytic sets $\cup_{i<m} B_i$. The intersection is then equal to $\cup_{i<m} A_n \cap B_i$. Each element of this finite union is analytic, because analytic sets are closed under intersection. They are also disjoint, since the B_i are disjoint. □

Lemma 3.6.6. *Let $A = \cup_{n \in \mathbb{N}} A_n$ be a countable union of simple analytic sets A_n. Then there are countably many pairwise disjoint analytic sets $\{B_m\}_{m \in \mathbb{N}}$ such that $A = \cup_{m \in \mathbb{N}} B_m$.*

Proof. Consider the countably many disjoint sets $A_k' = A_k \setminus (\cup_{i<k} A_i)$. Each A_k' can itself be expressed as a countable union of pairwise disjoint sets, by the previous corollary. □

Corollary 3.6.6.1. *Every c-analytic set A can be written as a countable disjoint union of analytic sets.*

Proof. Any open domain U can be expressed as a countable union of open balls. Therefore, in a countable union of *arbitrary* analytic sets, any analytic set A with a complicated open domain U can be replaced by countably many analytic sets with open-ball domains (and the same defining inequalities as A). We conclude with Lemma 3.6.6, using the fact that a countable family of countable families is a countable family. □

Next, we show a corollary that will be useful in the next section.

Corollary 3.6.6.2. *An inclusion $U \subseteq V$ of c-analytic sets U, V is a PAP function $U \to V$.*

Proof. By Corollary 3.6.6.1, we can write $U = \sqcup_{i\in\mathbb{N}} A_i$ and $V = \sqcup_{j\in\mathbb{N}} B_j$ where the A_i and B_j are analytic sets. Analytic sets are closed under intersection, so $\{A_i \cap B_j\}_{i\in\mathbb{N},j\in\mathbb{N}}$ is a countable partition of U into analytic sets. Hence, $\{(A_i \cap B_j, id)\}_{i\in\mathbb{N},j\in\mathbb{N}}$ is a piecewise representation of the inclusion $U \rightarrowtail V$. □

Finally, we show a few closure properties of c-analytic sets.

Lemma 3.6.7. *If $f : U \to V$ is analytic and $A \subseteq V$ is analytic, then $f^{-1}(A)$ is analytic.*

Proof. By definition, $A = \{x \in (\bigcap_i X_i^+) \cap (\bigcap_j X_j^-) \mid \forall i. g_i^+(x) > 0 \wedge \forall j. g_j^-(x) \leq 0\}$ for some analytic functions g_i^+, g_j^-. Thus,

$$
\begin{aligned}
&f^{-1}(A) \\
&= \{x \in f^{-1}\Big((\bigcap_i X_i^+) \cap (\bigcap_j X_j^-)\Big) \mid \forall i. g_i^+(f(x)) > 0 \wedge \forall j. g_j^-(f(x)) \leq 0\} \\
&= \{x \in (\bigcap_i f^{-1}(X_i^+) \cap (\bigcap_j f^{-1}(X_j^-)) \mid \forall i. (g_i^+ \circ f)(x) > 0 \wedge \forall j. (g_j^- \circ f)(x) \leq 0\}
\end{aligned}
$$

As f is analytic, it is continuous and so the sets $f^{-1}(X_i^+)$ and $f^{-1}(X_j^-)$ are open. And again, as f is analytic, so are the functions $g_i^+ \circ f$ and $g_j^- \circ f$. □

Corollary 3.6.7.1. *If $f : U \to V$ is PAP and $A \subseteq V$ is c-analytic, then $f^{-1}(A)$ is c-analytic.*

Proof. By Corollary 3.6.6.1, we can write $A = \sqcup_i A_i$ for analytic sets A_i. Thus, $f^{-1}(A) = f^{-1}(\sqcup_i A_i) = \sqcup_i f^{-1}(A_i)$ and it is sufficient to show that each $f^{-1}(A_i)$ is c-analytic.

f is PAP so there exists a partition $U = \sqcup_i B_i$ where B_i are analytic sets and analytic functions $f_i : U_i \to R^n$ such that $f|_{B_i} = f_i$. Therefore, $f^{-1}(A) = \sqcup_i f_i^{-1}(A) \cap B_i$. Each $f_i^{-1}(A)$ is analytic by Lemma 3.6.7, and so each $f_i^{-1}(A) \cap B_i$ is analytic. This means $f^{-1}(A)$ is indeed c-analytic. □

3.6.2.2 Category cPAP

Definition 3.6.11 (cPAP)**.** *We call cPAP the category whose objects are c-analytic sets and whose morphisms are PAP functions between them. Composition is given by the usual composition of functions.*

Lemma 3.6.8. *cPAP is a concrete category.*

Proof.

- cPAP **has a terminal object.** The analytic set $\mathbb{R}^0$ is terminal.
- **The functor hom**$(1, -) : cPAP \to$ **Set is faithful.** The functor is the identity on morphisms.

□

Proposition 3.6.9. *cPAP is a concrete site, where the coverings for a c-analytic set U are given by countable c-analytic partitions of U, i.e., $\{(U_i)_i \mid \cup_i U_i = U \wedge \forall i \neq j, U_i \cap U_j = \emptyset\}$.*

Proof. First, note that our definition for the coverings is a well-defined, as inclusions are PAP functions by Corollary 3.6.6.2.

We now show that the given coverings satisfy the 5 axioms of a concrete site.

1. Suppose we have a cPAP morphism $g : C \to D$, and a c-analytic partition $\{D_i\}_{i\in I}$ of D. Then we wish to find a partition of C so that g can be represented as a piecewise gluing of functions with codomains D_i. First, note that since they are objects of cPAP, each D_i can be further partitioned into countably many distinct analytic subsets $\{D_{ij}\}_{j\in J_i}$, $D_{ij} \subseteq D_i$. Furthermore, because g is a morphism, it has a PAP representation $\{(C_k, g_k)\}_{k\in K}$ for analytic subsets $C_k \subseteq C$ and analytic functions g_k. For each i, j, k define $C_{ijk} = \{c \in C_k \mid g_k(c) \in D_{ij}\}$: since g_k is an analytic function and D_{ij} is an analytic set, C_{ijk} is analytic. Define g^*_{ijk} to be the restriction of g_k to C_{ijk}. Then $\{(C_{ijk}, g^*_{ijk})\}_{i\in I, j\in J_i, k\in K}$ is a piecewise representation of g in which each piece's codomain is D_i for some i.

2. Let $\{f_i : c_i \to c\}_{i\in I}$ be a covering of c. Then $\bigcup_{i\in I} Im(|f_i|) = |c|$. This follows from the definition of *partition*.

3. The initial object 0 is covered by the empty set.

4. The identity is always covering.

5. Let $\{f_i : c_i \to c\}_{i\in I}$ be a covering of c and $\{g_{ij} : c_{ij} \to c_i\}_{j\in J_i}$ be a cover of c_i for each i. Then $\{f_i \circ g_{ij} : c_{ij} \to c\}_{i\in I, j\in J_i} \in \mathcal{J}(c)$. This also follows from the definition of *partition*, and the fact that a countable union of countable sets is countable.

□

Lemma 3.6.10. *cPAP is a subcanonical site.*

Proof. Let X be a representable presheaf on cPAP. Then, up to natural isomorphism, X maps a c-analytic set A to $\mathbf{cPAP}(A, B)$ for some fixed c-analytic set B. Consider a covering family $\{(A_i)\}_{i\in I}$ for A, and a compatible collection of plots $\phi_i \in X(A_i)$, which can be identified (via the natural isomorphism) with cPAP morphisms $\phi_i : A_i \to B$. Then we must show that there is a unique $\phi \in X(A)$ whose restriction to each A_i is ϕ_i. To see this, note that each ϕ_i has a PAP representation $\{(A_{ij}, \phi_{ij})\}_{j\in J_i}$. By the definition of a covering family on cPAP, the A_i are disjoint, and so $\{(A_{ij}, \phi_{ij})\}_{i\in I, j\in J_i}$ is a PAP representation; the ϕ we are looking for is the function it represents. □

3.6.2.3 Admissible monos for cPAP

As an admissible set of monos, we choose those given by the monos defined on an c-analytic subset of their domain of definition. More formally, we define $\mathcal{M}_{PAP}$ for cPAP as follows:

$$\mathcal{M}_{PAP}(B) = \{m : A \rightarrowtail B \text{ PAP} \mid A \overset{\text{PAP}}{\cong} A', A' \text{ is a c-analytic subset of } B\}$$

Proposition 3.6.11. *$\mathcal{M}_{PAP}$ is an admissible class of monos.*

Proof.

1. For all $c \in \mathbf{C}, 0 \to c \in \mathcal{M}_{PAP}(c)$: the empty-set is c-analytic.
2. $\mathcal{M}_{PAP}$ contains all isomorphisms: clear.
3. $\mathcal{M}_{PAP}$ is closed under composition: clear.
4. All pullbacks of $\mathcal{M}_{PAP}$-maps exist and are again in $\mathcal{M}_{PAP}$: this amounts to showing that the preimage $B := f^{-1}(A)$ for a PAP function f and a c-analytic set A is c-analytic. This is true by Corollary 3.6.7.1.
5. For each c, the function $Sub_{\mathcal{M}}(c) \to \mathbf{Set}(|c|, \{0,1\})$ is componentwise injective and order-reflecting, and the image of $Sub_{\mathcal{M}}$ is closed under suprema of ω-chains: the function is the identity on sets, and is thus injective and order-reflecting. Given an omega chain $\{A_i \subseteq B\}_{i\in\mathbb{N}}$, we have $i < j \Rightarrow A_i \subseteq A_j$. Let $A := \bigcup_{i\in\mathbb{N}} A_i \subseteq B$. It is a countable union of c-analytic sets and is thus a c-analytic set.
6. Given an increasing chain in $Sub_{\mathcal{M}}(c)$, $(c_n \rightarrowtail c)_{n\in\mathbb{N}}$, denote its least upper bound by $c_\infty \rightarrowtail c$. Then the closure under precomposition (with any morphism) of the set $(c_n \rightarrowtail c_\infty)_{n\in\mathbb{N}}$ contains a covering family of c_∞: as shown in the previous point, c_∞ is c-analytic and by Corollary 3.6.6.1, $c_\infty = \sqcup A_i$ where A_i are analytic sets. Thus, $(c_n \cap A_i \rightarrowtail c_\infty)_{n\in\mathbb{N},i\in\mathbb{N}}$ is a covering family of c_∞ that is contained in the closure by precomposition of $(c_n \rightarrowtail c_\infty)_{n\in\mathbb{N}}$.

□

3.6.2.4 Soundness, adequacy, and conservativity

We can now prove Proposition 3.4.4 and Theorem 3.4.3.

Proposition 3.4.4. Let $\Gamma \vdash t : \tau$ be a well-typed term, where Γ is a real ground context and τ a real ground type. Then $[\![t]\!] : [\![\Gamma]\!] \to \mathbf{L}[\![\tau]\!]$ represents a PAP function $f : A \to [\![\tau]\!]$ defined on a c-analytic set A. It is in particular almost-everywhere differentiable.

Proof. As we have shown that $cPAP$ is a subcanonical (Proposition 3.6.10) concrete site (Proposition 3.6.9) with an admissible class of monos (Proposition 3.6.11), by Theorem 3.6.2 we have that the embeddings of PAP functions into ωPAP, and the one from partial PAP functions to the Kleisli category for $\mathbf{L}$ are full and faithful. In particular, this means that a term $\Gamma \vdash t : \tau$ of ground type in a ground context

whose semantics is in the Kleisli category for $\mathbf{L}$, can in fact be seen as a partial PAP function defined on a c-analytic set. □

Theorem 3.6.12. *ωPAP has products, coproducts, exponentials and $\mathbf{L} : \omega PAP \to \omega PAP$ is a strong lifting monad.*

Proof. It follows directly from the fact that we have seen that ωPAP is a category of ω-concrete sheaves for the concrete site $cPAP$ (Proposition 3.6.9), and $\mathbf{L}$ is obtained from an admissible class of monos on $cPAP$ (Propositions 3.6.11 and 3.6.1), by Proposition 7.5 in Matache et al. [2022]. □

Theorem 3.4.3. The denotational semantics in ωPAP of our language is sound and adequate w.r.t. the operational semantics (Section 3.3.3).

Proof. The result is now immediate by applying Theorem 3.6.3 to the concrete site $(PAP, \mathcal{J}_{PAP}, \mathcal{M}_{PAP})$ with the admissible class of monos $\mathcal{M}_{PAP}$. □

3.7 Logical relations arguments via categorical gluing

The goals of this sections are threefold:

- Introduce fibrations for logical relations [Katsumata, 2013], a version of categorical glueing that is particularly well-suited for logical relations arguments on rich effectful languages, as a replacement for the argument using syntactic categories from Section 2.5.
- Use this framework to prove the correctness of AD for PAP functions, i.e. that AD computes intensional derivatives (Theorem 3.5.2).
- Demonstrate the elegance of a fully categorical reasoning, in the case of AD.

Compared to the argument in Section 2.5, the situation here is more involved, for several technical reasons. First, our language (Figure 3.3) is more expressive than in Chapter 2, and in particular we have to reason with recursion. This makes it challenging to adapt the argument using syntactic categories that we had in Section 2.5. Second, the logical relation itself is not as simple and well-behaved as in Section 2.5, as PAP functions have several intensional derivatives. Third, the use of conditionals and non-smooth operations in the language implies that we have to be careful with the logical relations on types which are interpreted as coproducts, i.e. booleans $[\![\mathbb{B}]\!] = 1 + 1$ in our case. Very similar challenges appear are also explored and explained in depth in Vákár [2020a].

Our solution to these problems is a particularly nice setting of categorical gluing, introduced in Katsumata [2013], of fibrations for logical relations, which we recall in Section 3.7.1. In Section 3.7.2 we then use their framework to prove Theorem 3.5.2. This latter section is fairly technical and separated in three parts. Section 3.7.2.1 constructs a nice setting for reasoning about the correctness of AD

by constructing a convenient gluing category. This plays the role of setting the right framework for reasoning. In the proof of correctness of AD in Section 2.5, the analogous role was played by reasoning with Cartesian-closed categories and syntactic categories. Section 3.7.2.2 then plays the role of constructing the specific logical relations needed for our correctness argument. The analogous role in the proof in Section 3.7.1 was the construction of the functor $(\!|-|\!) : \mathbf{Syn} \to \mathbf{Gl}_{\mathbb{R}}$. Finally, Section 3.7.2.3 shows how to prove Theorem 3.5.2.

3.7.1 Fibrations for logical relations

We recall the bare minimum of fibration theory that is required for our purpose. Roughly speaking, fibrations offer a useful point of view as the functor $p : Gl \to \mathbf{C}$ from a gluing category (sometimes called a scone or subscone [Mitchell and Scedrov, 1992]) to the base category is an example fibration, with particularly nice properties. This was explored in detail in Katsumata [2013], Katsumata [2005], and we follow closely their development.

3.7.1.1 Fibrations

We first recall some basic facts from the theory of fibrations. See Jacobs [1999] for the standard textbook reference on the subject. Fibrations are particular kinds of functors that generalise the case of a forgetful functor from a category of predicates on objects and predicate-preserving morphism to the underlying category of objects and morphisms.

Definition 3.7.1. *Let $p : \mathbb{E} \to \mathbb{B}$ be a functor. A morphism h in $\mathbb{E}$ is said to be above a morphism w in $\mathbb{B}$ if $p(h) = w$. A morphism $f : X \to Y \in \mathbb{E}$ is **Cartesian over** $u : I \to J \in \mathbb{B}$ if $pf = u$ and every $g : Z \to Y \in \mathbb{E}$ for which $pg = u \circ w$ for some $w : pZ \to I$, there is a unique $h : Z \to X \in \mathbb{E}$ above w with $f \circ h = g$.*

Definition 3.7.2. *A **fibration** is a functor $p : \mathbb{E} \to \mathbb{B}$ such that for every $Y \in \mathbb{E}$ and $u : I \to pY \in \mathbb{B}$, there is a Cartesian morphism $f : X \to Y \in \mathbb{E}$ above u. It is also called **fibred category** or **category (fibred) over** $\mathbb{B}$.*

Example 3.7.1. *Let* **Pred** *be the category whose objects are pairs (X, S) of a set X and a subset $S \subseteq X$, and morphisms $(X, S) \to (Y, T)$ are functions $f : X \to Y$ such that $f(S) \subseteq T$. Let $p :$ **Pred** $\to$ **Set** be the functor that forgets the subsets S. It is a fibration.*

A very standard way to construct new fibrations from known ones is by the technique of 'change-of-base'.

Proposition 3.7.1. *(Change-of-base) Let $p : \mathbb{E} \to \mathbb{B}$ be a fibration and $K : \mathbb{A} \to \mathbb{B}$ be a functor. Form the pullback in **Cat**:*

$$\begin{array}{ccc} \mathbb{A} \times_{\mathbb{B}} \mathbb{E} & \longrightarrow & \mathbb{E} \\ {\scriptstyle K^*(p)}\downarrow & & \downarrow{\scriptstyle p} \\ \mathbb{A} & \xrightarrow[K]{} & \mathbb{B} \end{array}$$

In this situation, the functor $K^*(p)$ is also a fibration.

Example 3.7.2. *Consider the functor $K : \mathbf{Set} \times \mathbf{Set} \to \mathbf{Set}$ that sends pairs of sets to their product. In this case,*

$$\begin{array}{ccc} \mathbf{BRel} & \longrightarrow & \mathbf{Pred} \\ {\scriptstyle K^*(p)}\downarrow & \lrcorner & \downarrow{\scriptstyle p} \\ \mathbf{Set} \times \mathbf{Set} & \xrightarrow[K]{} & \mathbf{Set} \end{array}$$

BRel *is a category of functions that preserve a binary relation. An object in* **BRel** *is a binary relation $S \subseteq X \times Y$, and a morphism $S \subseteq X \times Y \to R \subseteq X_2 \times Y_2$ is a function $f : X \times Y \to X_2 \times Y_2$ such that for all $(x, y) \in S, f(x, y) \in R$.*

Proposition 3.7.2. *(Composition) Let $p : \mathbb{E} \to \mathbb{B}$ and $r : \mathbb{B} \to \mathbb{A}$ be fibrations. Then $rp : \mathbb{E} \to \mathbb{A}$ is also a fibration, in which $f \in \mathbb{E}$ is a Cartesian morphism iff f is a Cartesian morphism for p and $p(f)$ is a Cartesian morphism for r. For each $I \in \mathbb{A}$ one obtains a fibration $p_I : \mathbb{E}_I = (rp)^{-1}(I) \to \mathbb{B}_I = r^{-1}(I)$ by restriction.*

Definition 3.7.3. *Given a fibration $p : \mathbb{E} \to \mathbb{B}$, the fibre category over $I \in \mathbb{B}$ is the subcategory $\mathbb{E}_I$ of $\mathbb{E}$ whose objects are above I and morphisms above id_I.*

An intuition for the Cartesian-morphism in fibrations is that of weakest-precondition. In the example of $p : \mathbf{Pred} \to \mathbf{Set}$, given a function $f : S \to T$ and a predicate $A \subseteq T$, the Cartesian lifting of p can be chosen to be $f : (f^{-1}(A), S) \to (A, T)$ and $f^{-1}(A)$ is indeed the smallest set B such that for all $x \in B, f(x) \in A$. This analogy is explored in greater detail in Melliès and Zeilberger [2015]. By contrast, opfibrations can be given an analogy in terms of strongest post-condition.

Definition 3.7.4. *$p : \mathbb{E} \to \mathbb{B}$ is an opfibration if $p^{op} : \mathbb{E}^{op} \to \mathbb{B}^{op}$ is a fibration.*

Example 3.7.3. *$p : \mathbf{Pred} \to \mathbf{Set}$ is an opfibration.*

3.7.1.2 Fibrations for logical relations for effectful languages

As we discussed, a view on fibrations is a generalized forgetful functors from categories of predicates on the objects of a category to the underlying category, the simplest example being $p : \mathbf{Pred} \to \mathbf{Set}$, which is both a fibration and an opfibration, and a morphism $f : (A, S) \to (B, T)$ in **Pred** is determined by its type and the underlying morphism $p(f) : S \to T$. Katsumata [2013] considers the class of fibrations that will behave similarly to $\mathbf{Pred} \to \mathbf{Set}$ as fibration for logical relations.

Definition 3.7.5. *A partial order bifibration with fibrewise small products is a faithful functor $p : \mathbb{E} \to \mathbb{B}$ such that*

- *p is a fibration*
- *p is an opfibration*
- *each fibre category is a partial order*

- *each fibre category has meets, and the inverse image functors preserve them*

Definition 3.7.6 ([Katsumata, 2013])**.** *A fibration for logical relations over a bi-CCC $\mathbb{B}$ is a partial order bifibration $p : \mathbb{E} \to \mathbb{B}$ with fibrewise small products such that $\mathbb{E}$ is a bi-CCC and p strictly preserves the bicartesian-closed structure.*

One very convenient of fibrations for logical relations is that, similarly to fibrations, they are closed under change-of-base.

Proposition 3.7.3 ([Katsumata, 2013])**.** *The pullback of a fibration for logical relations along a finite product preserving functor is a fibration for logical relations.*

In particular, the usual subscone fibration [Mitchell and Scedrov, 1992] is recovered as a change of base along the functor $\mathbb{B}(1, -) : \mathbb{B} \to \mathbf{Set}$.

Definition 3.7.7 ([Katsumata, 2013])**.** *Let $p : \mathbb{E} \to \mathbb{B}$ be a functor. Given $X, Y \in \mathbb{E}$, and $f : pX \to pY$, we write $f : X \overset{\bullet}{\to} Y$ to denote the following proposition: $\exists \overset{\bullet}{f}\colon X \to Y.p(\overset{\bullet}{f}) = f$. We say that f has a lift (in $\mathbb{E}$).*

The setting of Katsumata [2013] works for fairly general languages with effects. One way to describe them is in terms of algebraic operations, or equivalently in terms of generic effects.

Definition 3.7.8 ([Plotkin and Power, 2003])**.** *Let $\mathbf{C}$ be a category and T a strong monad on it. Given objects C and D of $\mathbf{C}$, a (D, C) algebraic operation or generic effect for is a morphism $C \to TD$.*

A lambda-calculus with effects is parametrised by a set of base types and (effectful) primitives, each having an arity and coarity, describing the number and types of arguments and return values. This is encapsulated as a signature:

Definition 3.7.9 ([Katsumata, 2013])**.** *A λ_c-signature Σ is a tuple $(B, K, O, ar, car : K \bigcup O \to GType(B))$ where B is a set of base types, K a set of effect-free constants, O a set of algebraic operations, and $GType(B)$ the set of ground types on B. ar, car are the arity and co-arity functions.*

Definition 3.7.10 ([Katsumata, 2013])**.** *Let Σ be a signature. A $\lambda_c(\Sigma)$-structure is a tuple $A = (\mathbb{B}, T, A, a)$ where $\mathbb{B}$ is a bi-CCC, T a strong monad on $\mathbb{B}$, A a functor $B \to \mathbb{B}$.*

In other words, a $\lambda_c(\Sigma)$-structure gives an interpretation of the base types and primitives of the language. Note that we can always extend A to a structure preserving functor $A[\![-]\!] : Gtype(B) \to \mathbb{B}$. A property is the choice for any base type $b \in B$ of a predicate Vb in the category of predicates $\mathbb{E}$ over values of type b, and of a predicate Cb over computations $T(Ab)$ of type b. In more detail

Definition 3.7.11 ([Katsumata, 2013])**.** *A property over a $\lambda_c(\Sigma)$-structure A is a pair (V, C) of functors $V, C : B \to \mathbb{E}$ such that $p \circ V = A$ and $p \circ C = T \circ A$*

The question of interest is then whether all programs preserve the property, which is answered by the following theorem.

Theorem 3.7.4 (Logical relations, [Katsumata, 2013]). *Let Σ be a signature, $A = (\mathbb{B}, T, A, a)$ be an $\lambda_c(\Sigma)$-structure, $p : \mathbb{E} \to \mathbb{B}$ be a fibration for logical relations and (V, C) be a property over A. If the property satisfies the following conditions*

- *for all $b \in B$, `return` has a lift*
- *for all $k \in K$, $A[\![k]\!]$ has a lift*
- *for all $o \in O$, $[\![o]\!]$ has a lift*

then for all well-typed terms $x_1 : b_1, \ldots, x_n : b_n \vdash t : b$, $A[\![t]\!]$ lifts to the total category.

3.7.1.3 Fibrations for logical relations for effectful languages with recursion

So far, we have reviewed the way to use fibrations for logical relations for higher-order effectful languages, but not for languages with recursion. We review this next, still closely following Katsumata [2013]. The main new ingredient that is required is an ωcpo-structure with some compatibility conditions, which we review next.

Definition 3.7.12 ([Katsumata, 2013]). *An ωcpo-enriched bi-CCC is a bi-CCC $\mathbf{C}$ such that each homset is an ωcpo, and the composition, tupling $(-,-)$, projections, cotupling $[-,-]$, injections, currying $\lambda(-)$ and application of the bi-CC structure on $\mathbf{C}$ are all monotone and continuous.*

Definition 3.7.13 ([Katsumata, 2013]). *A pseudo-lifting monad on an ωcpo-enriched bi-CC category $\mathbb{B}$ is a monad on the underlying non-enriched category, with a distinguished morphism at each component I, $bt_I : 1 \to TI$, that is the least morphism.*

Definition 3.7.14 ([Katsumata, 2013]). *An ωcpo-enriched $\lambda_c(\Sigma)$-structure is a tuple $(\mathbb{B}, T, A, a)$ such that $\mathbb{B}$ is an ωcpo-enriched bi-CCC, T is a pseudo-lifting monad over $\mathbb{B}$ and $(\mathbb{B}, T, A, a)$ is a $\lambda_c(\Sigma)$-structure.*

Definition 3.7.15 ([Katsumata, 2013]). *We call $X \in \mathbb{E}$ above $TI \in \mathbb{B}$ admissible if*

- $\bot_{pX} : 1 \xrightarrow{\bullet} X$
- *for all $Y \in \mathbb{E}$ and ω-chain $f_i \in \mathbb{B}(pY, pX)$ such that $f_i : Y \xrightarrow{\bullet} X$, we have $\bigsqcup_{i=0}^{\infty} f_i : Y \xrightarrow{\bullet} X$.*

We can now phrase an analogue of Theorem 3.7.4 for a language with recursion.

Theorem 3.7.5 (Logical relations for language with recursion, [Katsumata, 2013]). *Let Σ be a signature, $A = (\mathbb{B}, T, A, a)$ an ωcpo-enriched $\lambda_c(\Sigma)$-structure, $p : \mathbb{E} \to \mathbb{B}$ a fibration for logical relations and (V, C) be a property over A. If the property satisfies*

- *the conditions of Theorem 3.7.4,*
- *for all $b \in \mathbb{B}, Cb$ is admissible.*

Then for all well-typed terms of the language with iteration, $x_1 : b_1, \ldots, x_n : b_n \vdash t : b$ we have $A[\![t]\!] : \prod_i Vb_i \xrightarrow{\bullet} Cb$.

3.7.2 Correctness of AD

As we discussed in the beginning of Section 3.7, this section is divived in three parts.

First, in Section 3.7.2.1, we construct a *suitable* category of predicates on $\omega PAP \times \omega PAP$ (i.e. a fibration for logical relations). The emphasis on suitable comes from the following point. In the case **Pred** $\to$ **Set**, a predicate is a (class of) momomorphism $A \rightarrowtail S$, which are morphisms in **Set**. We could hope that in our case, our logical predicate of interest $\mathcal{R}_\tau$ would be a monomorphism $\mathcal{R}_\tau \rightarrowtail [\![\tau]\!] \times [\![\mathcal{D}(\tau)]\!]$ in ωPAP, relating pairs of curves as in the proof in Section 2.5. However, it turns out that the relevant $\mathcal{R}_\tau$ we came up with is not a *concrete* sheaf, i.e. not an object in ωPAP, as it does not satisfy the concreteness axiom. We instead develop an alternate route by showing it is a sheaf in a certain category of sheaves **Sh**(**Inj**). We obtain our fibration for logical relations on $\omega PAP \times \omega PAP$ by change-of-base from the one obtained by a category of predicates on **Sh**(**Inj**).

Second, in Section 3.7.2.1, we construct a property on the fibrations for logical relations on ωPAP we constructed in Section 3.7.2.1 that will mimic correct dual-numbers (Equation 2.2), but adapted to work for PAP functions. A first key change compared to the dual-number case is that derivatives are not unique any more, and this makes this logical relation more complex. The second key change is that the logical relation has a definition for values and computations, as is typical for effectful languages.

Lastly, we will show the correctness of AD (Theorem 3.5.2) by combining these elements with Theorem 3.7.5.

3.7.2.1 A fibration for logical relations on $\omega PAP \times \omega PAP$

We first define a specific fibration for logical relations based on a category of predicates on a category of Grothendieck sheaves. Its site is defined as follows.

Definition 3.7.16. *We define the category* **Inj***, a subcategory of cPAP, whose objects are c-analytic sets and morphisms are PAP injections.*

Lemma 3.7.6. Inj *is a site, with coverage $\mathcal{J}$ where coverings are given as in cPAP.*

Proof. The proof for the case of cPAP carries through to this restricted setting. In particular, in cPAP the important stability axiom only used inclusions which are PAP injections, and is therefore still valid. □

We now define the category of sheaves **Sh**(**Inj**):

Definition 3.7.17. *Let* **Sh**(**Inj**) *be the category of sheaves on the site* **Inj***.*

Lemma 3.7.7. Sh(**Inj**) *is a Grothendieck topos, and in particular it is Cartesian-closed, complete and co-complete.*

Proof. This is standard sheaf theory, see e.g. MacLane and Moerdijk [2012]. □

We now define the category of predicates on **Sh**(**Inj**). By a subsheaf P of F in **Sh**(**Inj**), we mean a sheaf F such that for all c-analytic set A, $P(A) \subseteq F(A)$, and $P(f)$ is the restriction of $F(f)$ for all morphisms f.

Definition 3.7.18. *We denote by* $\mathbf{Sub}(\mathbf{Sh}(\mathbf{Inj}))$ *the category whose objects are pairs of sheaves* (P, F) *of* $\mathbf{Sh}(\mathbf{Inj})$*, where* P *is a subsheaf of* F*, and morphisms* $(P, F) \to (P', F')$ *are natural transformations* $F \to F'$ *that restrict to the subsheaves.*

Lemma 3.7.8. *The second projection* $(P, F) \mapsto F$ *is a fibrations for logical relations* $p : \mathbf{Sub}(\mathbf{Sh}(\mathbf{Inj})) \to \mathbf{Sh}(\mathbf{Inj})$.

Proof. As $\mathbf{Sh}(\mathbf{Inj})$ has pullbacks, p is a fibration. It is clearly faithful. By the theory of gluing ([Carboni and Johnstone, 1995, Johnstone et al., 2007, Johnstone, 2002, Mitchell and Scedrov, 1992]), as $\mathbf{Sh}(\mathbf{Inj})$ is a CCC and has an epi-mono factorisation system, $\mathbf{Sub}(\mathbf{Sh}(\mathbf{Inj}))$ is a CCC, has finite colimits, and p preserves the CCC-structure. In addition, each fibre category is a partial order and has small products. It remains to show that *cod* is an opfibration and that p preserves coproducts. By Lemma 4.6 in Kammar and McDermott [2018], it suffices to show that monos are closed under coproducts, as we already have the other conditions for having a factorization system for logical relations. The arrow category on $\mathbf{Sh}(\mathbf{Inj})$ is cocomplete, as $\mathbf{Sh}(\mathbf{Inj})$ is (it is a topos). As $\mathbf{Sh}(\mathbf{Inj})$ is a topos, it has an epi-mono factorization system, and therefore it has image factorization. Therefore, the category of monos is a reflective subcategory of the arrow category. As such, it is cocomplete, and in particular it has coproducts. So monos are closed under coproducts, and we conclude by Lemma 4.6 in Kammar and McDermott [2018], as said above. □

Using the fibration for logical relations $p : \mathbf{Sub}(\mathbf{Sh}(\mathbf{Inj})) \to \mathbf{Sh}(\mathbf{Inj})$, we now construct a new one on $\omega PAP \times \omega PAP$ by change of base. Let $F : \omega PAP \times \omega PAP \to \mathbf{Sub}(\mathbf{Sh}(\mathbf{Inj}))$ be the functor given by $F(X, Y) = X \times Y^{\mathbb{N}}$, where $X \times Y^{\mathbb{N}}$ forgets that it is an ω-concrete sheaf on the site $cPAP$, and only remembers that it is in particular a sheaf on the site $\mathbf{Inj}$. As $Y \to Y^{\mathbb{N}}$ and $(X, Y) \to X \times Y$ are limits, they are product-preserving functors. The restriction functor $\omega PAP \to \mathbf{Sh}(\mathbf{Inj})$ is product-preserving, and therefore F is a composition of product-preserving functors and is product preserving. Therefore, by Proposition 3.7.3, the pullback of p along F is a fibration for logical relations. We denote this pullback by $\pi : \mathbf{Gl} :\to \omega PAP \times \omega PAP$. An object in $\mathbf{Gl}$ is a triple (X_1, X_2, S) where X_1 and X_2 are objects of ωPAP and S is a subsheaf of $X_1 \times X_2^{\mathbb{N}}$ seen as sheaf in $\mathbf{Sh}(\mathbf{Inj})$. A morphism $f : (X_1, X_2, S) \to (Y_1, Y_2, T)$ is a pair of ωPAP morphisms $(f : X_1 \to Y_1, g : X_2 \to Y_2)$ such that for all c-analytic set A and plot $\phi : A \to X_1 \times X_2^{\mathbb{N}}$ respecting the predicate S (i.e. $\phi \in S(A)$), the plot $(f \times (\mathbb{N} \Rightarrow g)) \circ \phi : A \to Y_1 \times Y_2^{\mathbb{N}}$ respects the predicate T, i.e. is an element of the set $T(A)$. Unveiling the definition a bit more, ϕ consists of a plot $\phi_1 : A \to X_1$ and a family of plots $(\phi_{2,i} : A \to X_2)_{i \in \mathbb{N}}$, and (f, g) is a morphism in $\mathbf{Gl}$ if for all such $\left(\phi_1 : A \to X_1, (\phi_{2,i} : A \to X_2)_{i \in \mathbb{N}}\right) \in S(A)$, $\left(f \circ \phi_1, (g \circ \phi_{2,i} : A \to X_2)_{i \in \mathbb{N}}\right) \in T(A)$.

Remark 3.7.1. *Before moving on, let us say a few words on this category of predicates* $\mathbf{Gl}$*. We could have chosen the category* $\mathbf{Sh}(\mathbf{Inj})$ *to be sheaves on the site* $cPAP$ *instead. This would not change much, but this is a bit overkill for our purpose and simply adds some unnecessary complexity. The property we will define for the correctness of AD will require to be indexed by all c-analytic sets* A*, instead*

of simply the one object $\mathbb{R}$, as it was the case in Section 2.5. However, to do so it will be sufficient to consider **Inj** *as we did. One main reason we have partial functions which force the definition of a lift for $\mathbf{L}\mathbb{R}$, as well as for $\mathbb{R}$. A second reason is that the non-uniqueness of intensional derivatives creates a technical complication. Compared to the case of true derivatives, knowing a correct intensional derivative at a single point is essentially useless information, as intensional derivatives are always free to take an arbitrary value on a countable set of points. More generally, it is unclear whether a dual-number intensional representation $h : \mathbb{R}^{2n} \to \mathbb{R}$ for a PAP function $f : \mathbb{R}^n \to \mathbb{R}$ is determined by a property like Equation 2.2. That is, the problem is that the input space $\mathbb{R}$ is smaller than $\mathbb{R}^n$, and the quantification only shows that h is 'correct' on a set of at most the size of $\mathbb{R}$ inside $\mathbb{R}^n$, which is Lebesgue-negligible. This problem disappears e.g. when we can show that h is correct on each open of an open cover of its domain.*

3.7.2.2 A property for the correctness of AD

We now define the property we will use for the correctness of AD, that is the analogue in the setting of fibrations for logical relations [Katsumata, 2013] to the usual logical predicates.

Definition 3.7.19. *Let $f : A \subseteq \mathbb{R}^n \to \mathbb{R}$ be a PAP function. Given any intensional derivative $g : A \to \mathbb{R}^n$ of f (Definition 3.2.6), we call $\pi_i \circ g : A \to \mathbb{R}$ an i-th partial intensional derivative of f.*

We write hot_i for the vector in $\mathbb{R}^n$ that consist only of zeros, except it has a single 1 at the i-th position. Given any dual-number intensional representation $g : A \times \mathbb{R}^n \to \mathbb{R}^2$ of f, define the PAP function $g_i : A \to \mathbb{R}^2$ given by

$$g_i(x) = g(x, hot_i)$$

g_i is called an i-th partial dual-number intensional representation of f, and its second component is an i-th partial intensional derivative of f. We write $\partial^i_{DNR} f$ for the set of i-th partial dual-number intensional representations of f.

We extend the definition of $\partial^i_{DNR} f$ to all natural numbers i by setting $\partial^i_{DNR} f = \{\lambda x.0\}$ for any $i > n$.

Definition 3.7.20. *Let $V(\mathbb{R}) := (\mathbb{R}, (\mathbb{R}^2)^{\mathbb{N}}, V_{\mathbb{R}}) \in \mathbf{Gl}$ where for each c-analytic set A, we have*

$$V_{\mathbb{R}}(A) = \{(f : A \to \mathbb{R}, (g_i : A \to \mathbb{R}^2)_{i \in \mathbb{N}}) \mid f, g\ \omega PAP, \exists\ h\ \omega PAP \\ \text{and linear in its second argument}, \forall i \in \mathbb{N}, h(-, hot_i) = g_i \in \partial^i_{DNR} f\}$$

Likewise, let $C(\mathbb{R}) := (\mathbf{L}\mathbb{R}, \mathbf{L}(\mathbb{R}^2)^{\mathbb{N}}, C_{\mathbb{R}}) \in \mathbf{Gl}$ where for each c-analytic set A, we have

$$C_{\mathbb{R}}(A) = \{(f : A \to \mathbf{L}\mathbb{R}, g : A \to \mathbf{L}(\mathbb{R}^2)^{\mathbb{N}} \mid Dom(f) = Dom(g) \text{ and} \\ \forall i.(\widetilde{f}, \widetilde{g_i}) \in V_{\mathbb{R}}(Dom(f))\}$$

Let $V(\mathbb{B}) = (\mathbb{B}, \mathbb{B}^{\mathbb{N}}, V_{\mathbb{B}}) \in \mathbf{Gl}$ *where for each c-analytic set* A*, we have*

$$V_{\mathbb{B}}(A) = \{f : A \to \mathbb{B}, g : A \to \mathbb{B}^{\mathbb{N}} \mid \forall i \in \mathbb{N}, g_i = f\}$$

Let $C(\mathbb{B}) = (\mathbf{L}\mathbb{B}, \mathbf{L}\mathbb{B}^{\mathbb{N}}, C_{\mathbb{B}}) \in \mathbf{Gl}$ *where for each c-analytic set* A*, we have*

$$C_{\mathbb{B}}(A) = \{(f : A \to \mathbf{L}\mathbb{B}, g : A \to \mathbf{L}\mathbb{B}^{\mathbb{N}}) \mid Dom(f) = Dom(g) \text{ and } \forall i \in \mathbb{N}.(\widetilde{f}, \widetilde{g_i}) \in V_{\mathbb{B}}(Dom(f))\}$$

Lemma 3.7.9. *We have defined a property for our language.*

Proof. The only thing to show is that $V_{\mathbb{R}}, C_{\mathbb{R}}, V_{\mathbb{B}}, C_{\mathbb{B}}$ are subsheaves of $\mathbb{R}\times(\mathbb{R}^2)^{\mathbb{N}}, \mathbf{L}\mathbb{R}\times \mathbf{L}(\mathbb{R}^2)^{\mathbb{N}}, \mathbb{B} \times \mathbb{B}^{\mathbb{N}}, \mathbf{L}\mathbb{B} \times \mathbf{L}\mathbb{B}^{\mathbb{N}}$ respectively. It is immediate for the C's once we have shown the result for the V's. For $V_{\mathbb{R}}$, it amounts to showing two things. First, whether partial intensional derivatives restrict to a subset of the domain of a function, which is evidently true. Second, whether partial intensional derivatives glue if they agree on their intersection, which as for usual derivatives, is true as well. For $V_{\mathbb{B}}$, it is immediate as it is an equalizer of $\mathbb{B} \times \mathbb{B}^{\mathbb{N}}$, which exists in any topos and the canonical map $V_{\mathbb{B}} \rightarrowtail \mathbb{B} \times \mathbb{B}^{\mathbb{N}}$ is a mono. □

Let us now look a bit more into what morphisms preserving the property look like.

Definition 3.7.21. *Let* $V(\mathbb{R}^k) := (\mathbb{R}^k, ((\mathbb{R}^2)^{\mathbb{N}})^k, V_{\mathbb{R}^k})$ *where for each c-analytic set* A*, we have*

$$V_{\mathbb{R}^k}(A) = \{(f : A \to \mathbb{R}^k, g : A \to ((\mathbb{R}^2)^{\mathbb{N}})^k) \mid \forall 1 \leq j \leq k, (\pi_j f, \pi_j g) \in V_{\mathbb{R}}(A)\}$$

Let $f : \mathbb{R}^n \to \mathbf{L}\mathbb{R}$ *be a PAP function. We define* $V(f)$ *to be pairs of functions* (f, h)*, where* $h : ((\mathbb{R}^2)^{\mathbb{N}})^k \to \mathbf{L}(\mathbb{R}^2)^{\mathbb{N}}$ *is* ωPAP*, and such that for any c-analytic set* A*, and any* $(g_1, g_2) \in V_{\mathbb{R}^k}(A)$*,* $(f \circ g_1, h \circ g_2) \in C_{\mathbb{R}}(A)$*.*

This means that given a PAP functions $g_1, \ldots, g_n : A \to \mathbb{R}$, h will send i-th partial intensional representations of each of the g_i to i-th partial intensional representations of $f \circ \langle g_1, \ldots, g_n\rangle$. Note that if $A \subseteq \mathbb{R}^k$, this will be trivially satisfied for any $i > k$.

Let us now see how we will interpret and a lift primitive $t : \mathbf{real}^2 \to \mathbf{real}$ of our language. This interpretation will only be required for the correctness proof. We set $[\![t]\!]_{\omega PAP\times\omega PAP} = ([\![t]\!], \prod_{i\in\mathbb{N}}[\![\mathcal{D}(t)]\!])$. In other words, we make countably many copies of the semantics of the AD-translation of t. We can now more generally show that every primitive of the language, whose semantics is interpreted in $\omega PAP \times \omega PAP$ following the example above, will lift to **Gl**.

Finally, to satisfy the conditions of Theorem 3.7.5, we need to show that for each base type B, $C(B)$ is admissible (Definition 3.7.15).

Lemma 3.7.10. $C(\mathbb{B})$ *is admissible.*

Proof. This means we need to find a lift for $\bot : 1 \to \mathbf{L}\mathbb{B}$ and a lift for the sup $\bigvee_{i\in\mathbb{N}}(f_i, g_i)$ of an omega chain $(f_i, g_i) : (X, Y) \to (\mathbf{L}\mathbb{B}, \mathbf{L}(\mathbb{B}))$ of morphisms which each have a lift.

- lift for $\bot : 1 \to \mathbf{L}\mathbb{B}$. We chose $(\bot_{\mathbb{B}}, \bot_{\mathbb{B}}, (\bot, \bot))$ where $\bot : A \to \mathbf{L}\mathbb{B}$ represents the function with empty domain.

- lift for the sup $\bigvee_{i\in\mathbb{N}}(f_i, g_i)$. Assume that for all $i \in \mathbb{N}$, (f_i, g_i): $(X, Y, R) \to C(\mathbb{B})$. This means that for all c-analytic set A, and all $r \in R(A) \subseteq \mathcal{P}^A_{X\times Y}$, we have $(f_i \times g_i)(r) \in C_{\mathbb{B}}(A)$. Let $r \in R(A)$. We need to show that $(\bigvee_{i\in\mathbb{N}}(f_i \times g_i))(r) \in C_R(A)$. By hypothesis, $(f_i \times g_i)(r)$ is just a pair of identical morphisms $B_i \to 1 + 1$. As composition is continuous and lubs are computed point-wise, we have $(\bigvee_{i\in\mathbb{N}}(f_i \times g_i))(r) = \bigvee_{i\in\mathbb{N}}(f_i \times g_i)(r)$ which is a pair of identical morphisms $B \to 1 + 1$, which is an element of $C_{\mathbb{B}}(A)$.

□

Lemma 3.7.11. *$C(\mathbb{R})$ is admissible.*

Proof. This means we need to find a lift for $\bot : 1 \to \mathbf{L}\mathbb{R}$ and a lift for the sup $\bigvee_{i\in\mathbb{N}}(f_i, g_i)$ of an omega chain $(f_i, g_i) : (X, Y) \to (\mathbf{L}\mathbb{B}, \mathbf{L}(\mathbb{R}))$ of morphisms which each have a lift.

- Lift for $\bot : 1 \to \mathbf{L}\mathbb{R}$. We chose $(\bot_{\mathbb{R}}, \bot_{\mathbb{R}\times\mathbb{R}}, (\bot, (\bot, \bot)))$ where $\bot : A \to \mathbf{L}\mathbb{R}$ is the nowhere defined function.

- Lift for the sup $\bigvee_{i\in\mathbb{N}}(f_i, g_i)$. Assume (f_i, g_i): $(X, Y, R) \to C(\mathbb{R})$. This means that every all c-analytic set A, and every $r \in R(A) \subseteq \mathcal{P}^A_X \times \mathcal{P}^A_Y$, we have $(f_i \times g_i)(r) \in C_R(A)$. Let $r \in R(A)$. As lubs are computed pointwise, we need to show that $(\bigvee_{i\in\mathbb{N}}(f_i \times g_i))(r) \in C_R(A)$. By hypothesis, for every $i \in \mathbb{N}$, $(f_i \times g_i)(r) \in C_R(A)$ and we can write it as $(f_i \times g_i)(r) = (k_i : B_i \subseteq A \subseteq \mathbb{R}^n \to \mathbb{R}, (l_{i,j} : B_i \to \mathbb{R}^2)_{j\in\mathbb{N}})$ such that there exists h_i ωPAP and linear in its second argument such that $\forall j \in \mathbb{N}, h_i(-, hot_j) = l_{i,j} \in \partial^j_{DNR} k_i$. Each $h_i : B_i \times \mathbb{R}^n \to \mathbb{R}^2$ is a dual-number intensional representation of k_i. Let $B = \bigvee_{i\in\mathbb{N}} B_i$. B is c-analytic. We show that the $(h_i)_{i\in\mathbb{N}}$ form an ω-chain. As $B_i \subseteq B_{i+1}$, $Dom(h_i) \subseteq Dom(h_{i+1})$. For any $x \in B_i, v \in \mathbb{R}^n$, we have, by definition and right linearity of the h_i:

$$\begin{aligned}
h_{i+1}(x, v) &= \sum_{1\leq j\leq n} h_{i+1}(x, v \cdot hot_j) \\
&= \sum_{1\leq j\leq n} v_i \cdot h_{i+1}(x, hot_j) \\
&= \sum_{1\leq j\leq n} v_i \cdot l_{i+1,j}(x) \\
&= \sum_{1\leq j\leq n} v_i \cdot l_{i,j}(x) \\
&= \sum_{1\leq j\leq n} v_i \cdot h_i(x, hot_j) \\
&= \sum_{1\leq j\leq n} h_{i+1}(x, v \cdot hot_j) \\
&= h_i(x, v)
\end{aligned}$$

and therefore $h_i \leq h_{i+1}$ as the order on functions $A \to \mathbf{L}\mathbb{R}^2$ is pointwise. Let $h = \bigvee_{i\in\mathbb{N}} h_i : B \times \mathbb{R}^n \to \mathbb{R}^2$. h is ωPAP. We show that this is a dual-number intensional representation of $k := \bigvee_{i\in\mathbb{N}} k_i$.

For each $i \in \mathbb{N}$, let $B_i = \bigcup_{j\in\mathbb{N}} C_{i,j}$ be a representation of B_i as a countable union of simple analytic sets (Definition 3.6.10) $C_{i,j}$, and $(C_{i,j} \times \mathbb{R}^n, h_{i,j})_{j\in\mathbb{N}}$ be a PAP representation for h_i. By construction, $B = \bigcup_{i,j\in\mathbb{N}} C_{i,j}$ is a representation of B as a countable union of simple analytic sets. Let $\phi : \mathbb{N} \times \mathbb{N}^2$ be a bijection. For every $k \in \mathbb{N}$, let $C'_k = C_{\phi(k)} \setminus \bigcup_{p<k} C_{\phi(p)}$. Then C'_k is a finite union of disjoint analytic sets (Corollary 3.6.5.2), and are thus analytic sets. Furthermore, the $(C'_k)_{k\in\mathbb{N}}$ are disjoint (by construction), and form a partition of B. Therefore, $B = \bigcup_{k\in\mathbb{N}} C'_k$ is a c-analytic partition of B. Let $h'_k : C'_k \times \mathbb{R}^n \to \mathbb{R}^2$ be given by $h'_k(x) = h_{\phi(k)}(x)$. The $(h'_k)_{k\in\mathbb{N}}$ are real analytic function and thus $(C'_k \times \mathbb{R}^n, h'_k)_{k\in\mathbb{N}}$ is a PAP representation for h. In addition, on each C'_k, we have $h(x, dx)|_{C'_k} = h_{\phi(k)}(x, dx) = (k_{\phi(k)}(x), dk_{\phi(k)}(x) \cdot dx) = (k(x), dk(x) \cdot dx)$, using the definition of h, $h_{\phi(k)}$ being a dual-number intensional representation of $k_{\phi(k)}$, and the definition of k. The last equality also uses the fact that $(C'_j, k_j)_{j\in\mathbb{N}}$ is a PAP representation of k, and therefore that $(C'_j, dk_j)_{j\in\mathbb{N}}$ is an intensional derivative of k. This means that h is a dual-number intensional representation of k.

Finally, note that $\forall j \in \mathbb{N}, l_j := h(-, hot_j) = \bigvee_{i\in\mathbb{N}} h_i(-, hot_j) = \bigvee_{i\in\mathbb{N}} l_{i,j}$. Therefore, $(\bigvee_{i\in\mathbb{N}}(f_i \times g_i))(r) = (\bigvee_{i\in\mathbb{N}} k_i, (\bigvee_{i\in\mathbb{N}} l_{i,j})_{j\in\mathbb{N}}) = (k, (l_j)_{j\in\mathbb{N}}) \in C_R(A)$.

□

3.7.2.3 Correctness of AD

Lemma 3.7.12. *ωPAP is an ωcpo-enriched bi-CCC.*

Proof. Each homset is an ωcpo by forgetting the PAP-structure. For composition and coproducts, it is immediate as lubs are taken point-wise. For products, if $f = \bigvee_{i\in\mathbb{N}} f_i$ and $g = \bigvee_{i\in\mathbb{N}} g_i$, then $(f, g) = (\bigvee_{i\in\mathbb{N}} f_i, \bigvee_{i\in\mathbb{N}} g_i) = \bigvee_{i\in\mathbb{N}}(f_i, g_i)$. Finally, if $f_i : X \times Y \to Z$, for currying we have $(\bigvee_{i\in\mathbb{N}} f_i)(x, y) = \bigvee_{i\in\mathbb{N}} f_i(x, y) = \bigvee_{i\in\mathbb{N}} \lambda(f_i)(x)(y)$. As lubs are taken pointwise, we have $\bigvee_{i\in\mathbb{N}} \lambda(f_i)(x)(y) = (\bigvee_{i\in\mathbb{N}} \lambda(f_i)(x))(y)$. This means that $\lambda(\bigvee_{i\in\mathbb{N}} f_i) = \bigvee_{i\in\mathbb{N}} \lambda(f_i)$. The monotonicity part is similar.

In summary, it follows from the fact that the forgetful functor $\omega PAP \to \omega\mathbf{CPO}$ preserves and reflects limits, coproducts, exponentials, and the ωcpo-structure. □

Lemma 3.7.13. *Every primitive of the language has a lift in* **Gl***, where the semantics in $\omega PAP \times \omega PAP$ is given by $[\![t]\!]_{\omega PAP\times\omega PAP} := ([\![t]\!], \mathbf{L}\mathbb{N} \Rightarrow [\![\mathcal{D}(t)]\!])$.*

Proof. By construction, $\mathcal{D}$ sends a PAP primitive f to a dual-number intensional derivative of f. Suppose $f : \mathbb{R} \to \mathbf{L}\mathbb{R}$. Let $(g : A \to \mathbb{R}, (g_i : A \to \mathbb{R}^2)_{i\in\mathbb{N}}) \in V_{\mathbb{R}}(A)$. There exists h ωPAP such that for all $i \in \mathbb{N}$, $h(-, hot_i) = g_i$. Then, $[\![f]\!]_{\omega PAP\times\omega PAP}(g, (g_i)_{i\in\mathbb{N}}) = ([\![f]\!] \circ g, ([\![\mathcal{D}(f)]\!] \circ g_i)_{i\in\mathbb{N}})$. Thus, for all $i \in \mathbb{N}$, $[\![\mathcal{D}(f)]\!] \circ g_i = [\![\mathcal{D}(f)]\!] \circ h(-, hot_i) \in \partial^i_{DNR}([\![f]\!] \circ g)$. Therefore, $[\![f]\!]_{\omega PAP\times\omega PAP} \in V([\![f]\!])$ and the semantics of f has a lift. A similar argument shows that the same holds for $f : \mathbb{R}^n \to \mathbf{L}\mathbb{R}$.

It remains to show that primitives like $>: \mathbf{real} \times \mathbf{real} \to \mathbb{B}$ lift. This is clear, however, as $\mathcal{D}(>)$ simply returns $>$, and the condition in $C_{\mathbb{B}}$ directly holds. □

Theorem 3.7.14. *For any $x_1 : \mathbf{real}, \dots, x_n : \mathbf{real} \vdash t : \mathbf{real}$, $[\![t]\!]_{\omega PAP \times \omega PAP}$ has a lift in* **Gl**.

Proof. We have done all of the work in the previous subsections. We simply check the condition to apply Theorem 3.7.5.

First, note that we indeed have a semantics in $\omega PAP \times \omega PAP$, given by $[\![t]\!]_{\omega PAP \times \omega PAP} := ([\![t]\!], \mathbf{L}\mathbb{N} \Rightarrow [\![\mathcal{D}(t)]\!])$, extending what we have seen in Section 3.7.2.2. Then,

- We have defined a fibrations for logical-relations $\pi : \mathbf{Gl} \to \omega PAP \times \omega PAP$ in Section 3.7.2.1.
- We have defined a property (V, C) in **Gl** in Section 3.7.2.2.
- We check that $x \mapsto \texttt{return}(x)$ lifts, but this is immediate.
- Each primitive has a lift, by Lemma 3.7.13.
- ωPAP is an ωcpo-enriched bi-CCC by Lemma 3.7.12.
- **L** is a pseudo-lifting monad. It means that we have an algebraic element $\bot_A : 1 \to \mathbf{L}A$ that is interpreted in $\mathbf{L}A$ as the bottom element. This is true by construction.
- $C(\mathbb{R})$ is admissible, by Lemma 3.7.11.
- $C(\mathbb{B})$ is admissible, by Lemma 3.7.10.

□

Corollary 3.7.14.1 (Correctness of AD (limited))**.** *For any term $x_1 : \mathbf{real}, \dots, x_n : \mathbf{real} \vdash t : \mathbf{real}$, the PAP function $[\![\overrightarrow{\mathcal{D}}(t)]\!] : A \times \mathbb{R}^n \to \mathbb{R} \times \mathbb{R}$ is a dual-number intensional representation of $[\![t]\!] : A \subseteq \mathbb{R}^n \to \mathbb{R}$.*

Proof. Let $x_1 : \mathbf{real}, \dots, x_n : \mathbf{real} \vdash t : \mathbf{real}$. By Theorem 3.7.14, $[\![t]\!]_{\omega PAP \times \omega PAP}$ has a lift. Consider the c-analytic set $\mathbb{R}^n$ and the pair

$$(id_{\mathbb{R}^n}, \lambda x : \mathbb{R}^n. \prod_{i \leq k \leq n} (\pi_k(x), [i = k]\pi_i'(x))_{i \in \mathbb{N}})$$

where, by convention, $\pi_i(x) = \pi_i'(x) = 0$ whenever $i > n$. We use Iverson brackets, $[i = k]$, which equals to 1 when the condition is satisfied, and 0 otherwise. This pair is an element of $V_{\mathbb{R}^n}(\mathbb{R}^n)$. This means that

$$([\![t]\!], (\mathbf{L}(\mathbb{N} \Rightarrow [\![\mathcal{D}(t)]\!])) \circ (\lambda x : \mathbb{R}^n. \prod_{i \leq k \leq n} (\pi_k(x), [i = k]\pi_i'(x))_{i \in \mathbb{N}})) \in C_{\mathbb{R}}(\mathbb{R}^n)$$

Unraveling the definition of $C_{\mathbb{R}}(\mathbb{R}^n)$, this means that for all $i \in \mathbb{N}$, $[\![\mathcal{D}(t)]\!](x, hot_i) = g(x, hot_i)$ is an i-th intensional partial representation of $[\![t]\!]$, for some PAP function g that is linear in its second argument. As $[\![\mathcal{D}(t)]\!]$ is also linear in its second argument, we have that $[\![\mathcal{D}(t)]\!] = g$ is a dual-number intensional representation of $[\![t]\!]$. □

With the same proof, Theorem 3.7.14 gives us directly the stronger result at all ground types:

Corollary 3.7.14.2 (Correctness of AD (full))**.** *For any term well-typed term* $\Gamma \vdash t : \tau$ *of ground type* τ *in a ground context* Γ*, the PAP function* $[\![\overrightarrow{\mathcal{D}}(t)]\!] : A \times \mathbb{R}^n \to \mathbb{R} \times \mathbb{R}$ *is a dual-number intensional representation of* $[\![t]\!] : A \subseteq \mathbb{R}^n \to \mathbb{R}$*.*

3.8 Conclusion and Related work

3.8.1 Summary

We introduced ωPAP-spaces (Section 3.4) as a model for a higher-order recursive language, with conditionals and partial piece-wise analytic primitives (Section 3.3). We have seen that they correspond to ω-concrete sheaves (Section 3.6), a similar construction to diffeological spaces, that is also endowed with the structure of ωcpo, enabling the interpretation of recursion. Morphisms of ωPAP-spaces generalize PAP functions (Section 3.2), which are piece-wise analytic functions, stable under the language construct of conditionals. They are almost-everywhere differentiable and enjoy many desirable properties. In general, they have infinitely many intensional derivatives, and each agrees almost-everywhere with the true derivative. We have seen that intensional derivatives is what AD systems compute (Theorem 3.5.3), thus providing some correctness guarantee on what AD produces, even on such an expressive language. To show this, we used the setting of fibrations for logical relations (Section 3.7), a categorical version for doing logical relation arguments for effectful recursive languages. We have seen that the fact that the derivatives are intensional, and therefore non-unique, made more complex the logical relations.

3.8.2 Related work and context

Non smooth differentiation. Optimisation of non-smooth functions in machine learning has been studied extensively [Bolte et al., 2021, Lee et al., 2018, Duchi et al., 2011, Khan and Barton, 2015, Shamir and Zhang, 2013, Astorino and Fuduli, 2007]. de Amorim and Lam [2022] develop a calculus and denotational semantics based on Schwartz distributions, to study AD in a non-smooth setting. Their calculus has desirable properties but does not seem to be easily implementable, and is not easily linked to current practices in machine learning. Optimisation has been studied in non-smooth settings for a long time, see e.g. Lewis and Overton [2013].

Guarantees on AD beyond the smooth setting. Beck and Fischer [1994] studied the problem of conditionals for AD in a first-order setting, getting results which are essentially subsumed by Lee et al. [2020]. Ablin et al. [2020] studied AD for functions defined as a minimum. Kakade and Lee [2018] showed a correctness argument and a cheap gradient principle [Griewank and Walther, 2008] for AD in a non-smooth setting, where AD computes subgradients. Griewank and Walther [2008] study AD in a non-smooth setting in their Chapter 14. Bolte and Pauwels [2020]

give a non-smooth calculus adapted for AD in machine learning, in a restricted first-order setting that covers most of deep-learning architectures, and prove a result of convergence of stochastic gradient descent with AD-computed derivatives. To do so, they rely on the work developed in [Bolte and Pauwels, 2021], where a theory generalized derivatives is developed, called conservative field, that are set-valued. Sherman et al. [2021] provide a higher-order calculus and AD, gives a denotational in a category of sheaves on a site of smoothish functions. Lee et al. [2023] considers a restricted first-order language, and studies the correctness of AD for neural-networks with machine-representable parameters.

Other logical relations proofs for AD. The logical relations argument we presented in Chapter 2 inspired several other recent work Lew et al. [2023], Lew et al. [2021], Vákár [2020a], Vákár [2020b], Krawiec et al. [2022], Lucatelli Nunes et al. [2022], Vákár and Smeding [2022], who used similar logical relations arguments, and a similar logical relations argument was found around the same time [Barthe et al., 2020]. This seems to indicate a strong versatility of logical relations for reasoning about differentiable programming, and in particular for the correctness of AD. It would be interesting if similar logical relations could be developed for other properties that are related to differentiability, such as Clarke subdifferentials.

3.8.3 Discussion

Higher-order derivatives. Similarly to what we presented in Section 2.6, we could define jet-based AD for computing higher-order intensional derivatives in one pass of an AD macro. This should not present any additional challenges in terms of semantics. Likewise, whether one wishes to do efficient implementations of AD, for instance using reverse-mode, correctness arguments for reverse-mode in a smooth setting might be adapted to the ωPAP-setting, as PAP functions obey a chain-rule. We see this investigation of efficient implementations as orthogonal to the semantic considerations of what differentiation even means in a non-differentiable setting, though it is obviously of key importance in practice.

Other language features and primitives. We have studied a higher-order recursive language with PAP primitives, but these primitives were first-order operations. Even though our language is very expressive, some differentiable functions can only be represented approximately by programs, and naively differentiating the approximation will usually not lead to an approximation of the derivative [Griewank and Walther, 2008][Chapter 15]. This is for instance the case for ordinary differential equation solvers and other functions defined implicitly by equations, e.g. by the implicit function theorem. In such a case, one may wonder whether we can treat such higher-order constructs as primitives, equipped with custom derivatives, that will extend the language and AD in a sound way. This will be explored in the next Chapter.

4

Some applications of the ωPAP framework

Summary. The goal of this chapter is to explore some of the applications of the ωPAP-framework we developed in the Chapter 3.

As we have discussed in Chapter 1, AD is often called by other algorithms, which usually expect certain mathematical guarantees on what AD computes. These might not always be satisfied: AD may incorrectly differentiate a program when differentiating its code, e.g. when the differentiability of the program is not immediately obvious when looking at its code. This is for instance the case when ignoring probabilistic effects [Lew, Huot, Staton, and Mansinghka, 2023]. In addition, naively differentiating the code can yield correct but with suboptimal computational performance [Chen et al., 2018]. These programs are then better considered as black-box primitives that need to be equipped with custom derivatives.

In this chapter, we then contribute to the following. From the programming side, we show how to extend AD to new expressive types and primitives. We recover some constructs from the literature and give a general recipe for doing so (Section 4.1). We then investigate the usage of AD for computing derivatives, as needed in certain other algorithms. We first look at AD-computed derivatives in the context of optimization, and more specifically for gradient descent (Section 4.2). We show that gradient descent may fail when the gradient is computed by AD, even when the initialization or the learning rate is randomized, but we prove that it almost surely converges when both are randomized (Theorem 4.2.5). Finally, we investigate the interaction between PAP functions and measures. To do so, we first introduce a monad of measure on ωPAP (Section 4.3). We then prove a new result about the pushforward of the Lebesgue measure by PAP functions (Theorem 4.4.15 in Section 4.4). After giving a semantics to an extension to our language (Figure 3.3) with standard probabilistic constructs as found in probabilistic programming, we use this result to prove a novel characterization theorem about the posterior distribution of closed probabilistic programs (Theorem 4.4.18).

Together, these results demonstrate the value of denotational reasoning with

ωPAP, a model that is general enough to interpret very rich language features and allows to reason about differentiability, yet specific enough to reject many pathological and non-computable examples.

Publication. This chapter is based on joint work with Alex Lew, Sam Staton, and Vikash Mansinghka. Section 4.1 is partly based on material from Lew, Huot, Staton, and Mansinghka [2023], which appeared at POPL 2023. Section 4.4 is based on material from Huot, Lew, Staton, and Mansinghka [2023a]. The technical content of this chapter roughly covers Sections IV-V in Huot et al. [2023b], which was recently accepted at LICS 2023. Many of the ideas were developed together in a collaborative process, but, except where explicitly noted, the presentation of the definitions and proofs in this thesis are my own.

4.1 Modular extensions to the language

In this section, we show that the expressive power of ωPAP-spaces seamlessly supports other powerful languages extensions. We first give a simple, general recipe that does not explicitly requires category theory to soundly extend forward-mode AD to new types and primitives. This is an equivalent formulation to the one using gluing or logical relations directly, but I thought it might give a clearer picture in the particular context of AD. I demonstrate the recipe on several examples, some coming from the recent literature.

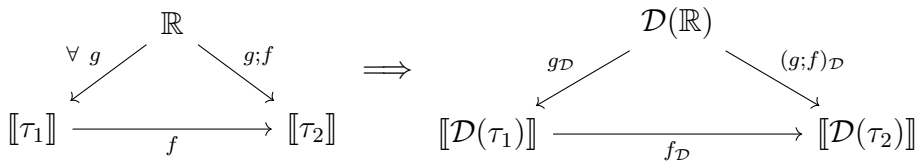
4.1.1 General recipe

To add new base types and primitives that are correctly supported by AD, our method involves three simple key steps:

1. **Define the translation $\mathcal{D}(\tau)$ for each new type τ.**

2. **Define correctness at type $\mathbb{R} \to [\![\tau]\!]$.** For each type τ, we define a notion of derivative for $\mathbb{R} \to [\![\tau]\!]$ functions: a relation between an $\mathbb{R} \to [\![\tau]\!]$ function and an $\mathbb{R} \to [\![\mathcal{D}(\tau)]\!]$ function encoding what it means to be a derivative.

3. **For every primitive f involving τ, provide correctness-preserving translations $f_{\mathcal{D}}$.** For arbitrary functions $f : [\![\tau_1]\!] \to [\![\tau_2]\!]$, we define correctness as the *preservation* of this relationship. As a pair of commutative diagrams:



For Sections 4.1.1.1 and 4.1.1.2, the underlying semantics is in **Diff**. Let us first demonstrate how this recipe recovers the base case of AD for smooth primitives $\mathbb{R}^n \to \mathbb{R}$.

4.1.1.1 Forward-mode AD on primitives $\mathbb{R} \to \mathbb{R}$

Assume that we start with a simply-typed lambda calculus whose only ground types are non-continuous types, such as $\mathbb{N}$ or $\mathbb{B}$. Suppose that we enrich the language by introducing the type **real** and new first-order primitives using this type (e.g. the usual elementary functions). We are interested in defining AD on the extended language, and by design the non-trivial part will happen on the new type **real**.

First, the goal is now to define AD on this new type **real**, which we can do as follows:

Definition 4.1.1 ($\mathcal{D}(-)$ on type $\mathbb{R}$)**.**

$$\mathcal{D}(\mathbf{real}) := \mathbf{real} \times \mathbf{real}$$

Second, we define correctness at type $\mathbb{R} \to [\![\mathbf{real}]\!]$ as follows:

Definition 4.1.2 (Correct dual-number derivative at type **real**)**.** *Let $f : \mathbb{R} \to [\![\mathbf{real}]\!]$, $g : \mathbb{R} \to [\![\mathbf{real}]\!] \times [\![\mathbf{real}]\!]$ be smooth. We say that g is a correct dual-number derivative of f if for all reals θ, we have $g(\theta) = (f(\theta), f'(\theta))$.*

Expectedly, Definition 4.1.2 recovers the usual formula for dual numbers. Thirdly, we need to show that every new primitive f involving the type **real** – e.g. elementary functions $\cos, \sin, \exp, +, *, \ldots$ – are correctness-preserving.

Definition 4.1.3 (Correctness preserving translation)**.** *Let $f : \mathbb{R} \to \mathbb{R}$ be a differentiable function. Then $f_{\mathcal{D}} : \mathbb{R} \times \mathbb{R} \to \mathbb{R} \times \mathbb{R}$ is a correctness preserving translation of f if for all differentiable $h : \mathbb{R} \to \mathbb{R}$,*

$$f_D(h(\theta), h'(\theta)) = ((f \circ h)(\theta), (f \circ h)'(\theta))$$

This is simply a restatement of Equation 2.2. In other words, we have two ways of looking at correct derivatives at play here. One is the usual notion of derivatives as limits, seen as an abstract property. The second way is as functions preserving this property on pairs of functions (f, g) where g has the property of being the derivative of f, by sending such pairs of functions to some other such pair of functions. We can now define what it means for the syntactic transformation $\mathcal{D}(-)$ to be semantically correct:

Definition 4.1.4 (Correctness of $\mathcal{D}(\cdot)$)**.** *The AD macro $\mathcal{D}(\cdot)$ is correct if, for all closed terms $\vdash t : \mathbf{real} \to \mathbf{real}$, $[\![\vdash \mathcal{D}(t) : \mathbb{R} \times \mathbb{R} \to \mathbb{R} \times \mathbb{R}]\!]$ is a correctness preserving translation of $[\![\vdash t : \mathbf{real} \to \mathbf{real}]\!]$.*

We now look at two other examples.

4.1.1.2 Adding a type real* for reals that do not need to be used smoothly

Similarly to types such as $\mathbb{B}$ or $\mathbb{N}$ for which we do not track any sort of differentiability or derivative, we could track that a function $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is only differentiable in its

first argument, and have AD ignore the second one. To do this, a way is to add a new type $\mathbf{real}^*$ of reals that do not need to be used smoothly. The semantics of **real** and $\mathbf{real}^*$ have the same underlying set, but different diffeologies (Definition 2.3.1). More precisely, $[\![\mathbf{real}^*]\!] = \bigsqcup_{r\in\mathbb{R}} 1$ is an uncountable coproduct of singletons, i.e. we have

$$\mathcal{P}^{\mathbb{R}}_{[\![\mathbf{real}^*]\!]} = \{f : \mathbb{R} \to \mathbb{R} \mid f \text{ is constant}\}$$

The AD macro $\mathcal{D}(\cdot)$ does not attach dual numbers to non-smooth values, i.e. we define

$$\mathcal{D}(\mathbf{real}^*) := \mathbf{real}^*$$

From a user's perspective, values of non-smooth type are *allowed* to be used non-smoothly, whereas values of smooth type *must* be used smoothly. We can reflect this in the type of new primitives $\leq\colon \mathbf{real}^* \times \mathbf{real}^* \to \mathbb{B}$. In addition to $\leq$, we introduce a coercion $\lfloor\cdot\rfloor : \mathbf{real}^* \to \mathbf{real}$ from a non-smooth type to a smooth type, but not in the other direction. The intuition is that you are always allowed to promise (unnecessarily) to use a value smoothly, but not to go back on your promise. Smooth and non-smooth types can be mixed to create functions whose types perform fine-grained tracking of *which* arguments they are differentiable with respect to. For example, a term $\vdash t : \mathbb{R} \times \mathbb{R}^* \to \mathbb{R}$ is guaranteed to have a denotation differentiable with respect to its first argument, but not its second. The key feature unlocked by this fine-grained tracking is that we can now assign more permissive types to some terms.

We define correctness at type $\mathbb{R} \to [\![\mathbf{real}^*]\!]$ as follows. We say that $g : \mathbb{R} \times \mathbb{R} \to [\![\mathbf{real}^*]\!]$ is a correct dual-number derivative of $f : \mathbb{R} \to [\![\mathbf{real}^*]\!]$ if for all reals $\theta, d\theta$, we have $g(\theta, d\theta) = f(\theta)$.

We define $\mathcal{D}(\leq) := \leq$ and $[\![\mathcal{D}(\lfloor - \rfloor)]\!](\theta) := (\theta, 0)$. One quickly verifies that they are correctness-preserving. Another way to phrase it is that no information can 'leak' from the smooth world to the non-smooth world.

4.1.2 Adding new higher-order primitives

Sometimes it is not clear how differentiating the code of a program, that we abstractly know represents a differentiable function, would lead to the correct derivative. This is for instance the case for iterative methods. The simplest example is perhaps finding the root of a smooth function. For instance, it is known that the roots of a polynomial vary smoothly with the coefficients, but by Galois theory there is no closed-form formula for such roots for polynomials of degree 5 and beyond. To compute these roots approximately, standard procedures involve variants of Newton's method.

Another iterative method used in numerical computing is the CORDIC (for coordinate rotation digital computer) algorithm. It is used to compute efficiently trigonometric functions and their inverses, and can also be used to compute square roots, multiplication, division, logarithms and exponentials. It is a simple algorithm that can be implemented in a few lines of code, and that is used in many embedded systems, typically computing one bit of precision at each iteration. Note that the CORDIC algorithm is not a numerical method, but a method that computes

exactly the result of the operation it is designed for, up to a fixed number of bits of precision. Naively differentiating the code of an implementation of CORDIC would not lead to the correct derivative, and in fact would typically return a derivative of 0. This reminds us that the correctness of AD is not a trivial property, as it depends on the representation of the code of the function being differentiated. Thus, if we wanted to use AD to compute the derivative of a higher order function that implements CORDIC, we would need to add it as a new primitive to the language, and equip it with a custom derivative.

This idea is explored in the case of AD for computable reals in the thesis of Sherman [2020]. We show how to interpret such primitives and their custom derivatives in **Diff**. For each primitive, its derivative satisfies the logical relations for the correctness AD. We treat separately the case of an ODE Solver in Section 4.1.3, which is not discussed in Sherman [2020].

integral01 : (**real** → **real**) → **real.** This primitive can be given a semantics both in **Diff** or in ωPAP. It takes a smooth function f and returns $\int_0^1 f(x)dx$. The tangent part of its AD translation returns $\int_0^1 f'(x)dx$. In more detail, we can first show that $[\![\mathbf{integral01}]\!] : (\mathbb{R} \to \mathbb{R}) \to \mathbb{R}$ is a morphism in **Diff**. Let $f : \mathbb{R} \to (\mathbb{R} \to \mathbb{R})$ be a plot, i.e. a smooth function $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. We have to show that $u \mapsto \int_0^1 f(u, x)dx$ is smooth. This is well-known, e.g. by the Leibniz integral rule. We now define $[\![\mathbf{integral01}_{\mathcal{D}}]\!] : (\mathbb{R} \times \mathbb{R} \to \mathbb{R} \times \mathbb{R}) \to \mathbb{R} \times \mathbb{R}$ by

$$[\![\mathbf{integral01}_{\mathcal{D}}]\!](df) = \int_0^1 df(x, 0)dx$$

By also using the Leibniz integral rule, this shows that we have a morphism in **Diff**. It is immediate that **integral01** preserves the logical relations. More generally, we can add a primitive **integral** : $\mathbf{real}^* \times \mathbf{real}^*$ → (**real** → **real**) → **real** (see Section 4.1.1.2 for the definition $\mathbf{real}^*$) such that

$$[\![\mathbf{integral}]\!](a, b)(f) = \int_a^b f(x)dx$$

Instead, we may also allow the integration bounds to smoothly depend on the parameters from the context. In this case we would have a primitive **integral** : **real** × **real** → (**real** → **real**) → **real** such that the tangent part will be given by the full Leibniz integral rule:

$$\pi_2[\![\mathbf{integral}_{\mathcal{D}}]\!](da, db)(df) = \pi_2 \int_a^b df(x, 0)dx \ + \ \pi_1\Big(df(db) *_{\mathcal{D}} db \ -_{\mathcal{D}} \ df(da) *_{\mathcal{D}} da\Big)$$

where $-_{\mathcal{D}}, *_{\mathcal{D}}$ are the dual-number translations for $-, *$ respectively.

argmax : (**real** × **real** → **real**) → (**real** → **real**)**.** This primitive takes a smooth function $g(x, y)$ and returns $h(x) := \mathrm{argmax}_{y \in \mathbb{R}} g(x, y)$. Of course, g might not have a unique maximum in general, and in fact the algorithms implementing **argmax** may also fail to find it for various reasons. We can encode this information by making **argmax** a partial function, defined on a reasonable subset of $(\mathbb{R} \times \mathbb{R} \to \mathbb{R})$.

Assume that g is smooth and $g(x,-)$ has a unique maximum for all x. Furthermore, assume that the second order derivative $\partial_2^2 g(x,y) < 0$ at the maximum $y = h(x)$, where we use ∂_i for the i^{th} partial derivative. At the maximum, the partial derivative must vanish, hence $\partial_2 g(x, h(x)) = 0$. Differentiating w.r.t. x, we obtain

$$\partial_1\partial_2 g(x,h(x)) + h'(x)\partial_2^2 g(x,y) = 0$$

This simply means that $h'(x) = -\frac{\partial_1\partial_2 g(x,h(x))}{\partial_2^2 g(x,y)}$. This can be viewed as a corollary of the implicit function theorem. We define the set

$$X := \{g : \mathbb{R}\times\mathbb{R} \to \mathbb{R} \mid g \text{ is smooth}, \forall x, g(x,-) \text{ has a unique maximum}, \partial_2^2 g(x,y) < 0\}$$

X can be given the structure of a diffeological space by restriction of the one on the function space $\mathbb{R}\times\mathbb{R} \to \mathbb{R}$. Therefore, we can interpret $[\![\mathbf{argmax}]\!] : X \to (\mathbb{R} \to \mathbb{R})$ as a morphism in **Diff**. We can also reflect this syntactically as follows. We can introduce a new type χ to the language, and primitives $\mathbf{argmax} : \chi \to (\mathbf{real} \to \mathbf{real})$, $\lfloor - \rfloor : \chi \to (\mathbf{real} \times \mathbf{real} \to \mathbf{real})$ and $\mathbf{assert} : (\mathbf{real} \times \mathbf{real} \to \mathbf{real}) \to \chi$. $[\![\lfloor - \rfloor]\!]$ is just an inclusion map, whereas $[\![\mathbf{assert}]\!] : (\mathbb{R}\times\mathbb{R} \to \mathbb{R}) \to [\![\chi]\!]_\perp$ will be a partial function (for the partiality monad $(-)+1$ on **Diff**). That way, a user can assert that a function satisfies the condition required for **argmax** to produce a result. Should the given program g not satisfy the conditions, the semantics of **argmax** g will simply be $\perp$. One can for instance extend AD with $\mathcal{D}(\chi) = \mathcal{D}(\mathbf{real} \times \mathbf{real} \to \mathbf{real})$. The definition for correct dual-number derivatives at type χ is simply the obvious one inherited from restricting correctness at the type $(\mathbf{real} \times \mathbf{real} \to \mathbf{real})$.

firstroot01 : $(\mathbf{real} \to \mathbf{real}) \to \mathbf{real}$**.** This primitive takes a smooth function f and returns the smallest $x \in [0,1]$ such that $f(x) = 0$. The tangent part of the AD translation is essentially another application of the implicit function theorem, and works as follows. First, recall that the input function $f : \mathbb{R} \to \mathbb{R}$ may also depend on the context Γ. For simplicity in the exposition, assume $\Gamma = \{x : \mathbb{R}\}$. Therefore, f is really a smooth function $\mathbb{R}\times\mathbb{R} \to \mathbb{R}$, and we are interested in $f(x,y) = 0$. The implicit function theorem in this case states that if $\frac{\partial f}{\partial y}(x_0,y_0) \neq 0$, then on an open $U \subseteq \mathbb{R}$ containing x_0, there exists a smooth function $g : U \to \mathbb{R}$ such that $g(x_0) = y_0$ and $f(x, g(x)) = 0$ for all $x \in U$. Moreover, we have

$$\frac{\partial g}{\partial x}(x) = -\frac{\frac{\partial f}{\partial x}(x,g(x))}{\frac{\partial f}{\partial y}(x,g(x))} \tag{4.1}$$

In our case, note that the output of $x : \mathbf{real} \vdash \mathbf{firstroot01}\ f$ will be $g(x)$, and thus its derivative should return $g'(x)$ as computed in Equation 4.1. The more general case involves the Jacobian matrix of f. Similarly to the case of **argmax**, we add a new type $\mathcal{Y}$, encoding the restrictions on the valid inputs to **firstroot01**. We define

$$[\![\mathcal{Y}]\!] = \{f : \mathbb{R} \to \mathbb{R} \ \mid\ f \text{ smooth}, \forall x \in \mathbb{R}, f'(x) \neq 0, \exists x \in [0,1], f(x) = 0\}$$

This is given the structure of diffeological space by restriction of the one on $\mathbb{R} \Rightarrow \mathbb{R}$. $[\![\mathbf{firstroot01}]\!] : [\![\mathcal{Y}]\!] \to \mathbb{R}$ is a morphism in **Diff**. Indeed,let $f : U \to [\![\mathcal{Y}]\!]$ be a

plot, i.e. a smooth function $f : U \times \mathbb{R} \to RR$ such that for all x, y, $\frac{\partial f}{\partial y}(x, y) \neq 0$. Then, $[\![\textbf{firstroot01}]\!] \circ f$ is the smooth function $g : U \to \mathbb{R}$ such that for all $x \in U$, $f(x, g(x)) = 0$, which is a plot. Similarly to the case of **argmax** again, we can equivalently see $[\![\textbf{firstroot01}]\!]$ as a morphism $(\mathbb{R} \to \mathbb{R}) \to \mathbb{R} + 1$.

4.1.3 Differentiating an ODE solver

As a last example of primitive we could add to the language, we add an ODE solver and show how to do differentiation of ODE solutions, somewhat following Chen et al. [2018].

Somewhat similarly to what we have seen with implicit differentiation, an unknown function can be uniquely defined as satisfying an ordinary differential equation (ODE) with initial condition, which in its simplest form is given by

$$\frac{dh(t)}{dt} = f(h(t), t, \theta) \qquad h(0) = c$$

where c is the fixed initial condition, h the unknown function of interest, f a fixed function, and θ a parameter of interest, w.r.t. which we want to differentiate h. One intuition for the situation can be given as follows. If t represents time, let $h(t)$ represent the position of a boat at time t, with a known initial position c at time 0. f can then describe wind conditions, i.e. following Newton's laws, the forces acting on the boat determine the evolution of its velocity and position. In this situation, the derivative we are interested in captures the change in the final position of the boat after time t has passed when we slightly modify the wind conditions.

In the simplest case, where $\theta, t \in \mathbb{R}$, and if there is a unique solution to the problem (e.g. if f is locally-Lipschitz in its second variable, by the Cauchy-Lipschitz theorem), we therefore obtain a partial function $h : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, and we wish to compute $\frac{\partial h(t,\theta)}{\partial \theta}$.

In general, neither h nor its derivative will admit analytic formulas. More precisely, the solution can be expressed as an integral, which is typically intractable. Instead, h will usually be well approximated by ODE solvers, which use iterative methods such as Euler's or Runge-Kutta's. We can add a higher-order primitive $\textbf{ODESolve} : \textbf{real}^* \to \textbf{real}^* \to \textbf{real}^* \to (\textbf{real} \times \textbf{real} \to \textbf{real}) \to \textbf{real}$ such that $[\![\textbf{ODESolve}]\!](t_0, t_1, c, f)$ returns the unique solution h to the ODE:

$$\frac{dh(t)}{dt} = f(h(t), t) \qquad h(t_0) = c$$

at the value t_1, which can be shown to be formally given by

$$h(t_1) := \int_{t_0}^{t_1} f(h(t), t)dt$$

When giving a semantics to **ODESolve** in **Diff**, we must again restrict its domain appropriately. For instance, if f is locally Lipschitz in its first argument and Lipschitz in its second, we are ensured that a unique global solution h defined on all of $\mathbb{R}$ exists.

One main idea from Chen et al. [2018] is that one can compute the gradient of h using the adjoint method [Pontryagin, 1987]. One can show that the gradient of h

will also satisfy an ODE. Both the value and the derivative can then be computed by a single call to an ODE solver, but for an augmented ODE ([Chen et al., 2018]). Therefore, we can define a new primitive $\mathbf{ODESolve}_{\mathcal{D}}$ such that $\mathcal{D}(\mathbf{ODESolve}) = \mathbf{ODESolve}_{\mathcal{D}}$, and such that $\mathbf{ODESolve}_{\mathcal{D}}$ implements the augmented ODE solver which returns $(h(t,\theta), \frac{\partial h(t,\theta)}{\partial \theta})$ (remember that θ comes from the context Γ, which every term is allowed to depend on). This primitive is a very powerful abstraction, and Chen et al. [2018] showed that it can be used to replace residual or recurrent networks for supervised learning, by simply considering a neural-network followed by an ODE solver, through which one does backpropagation for the learning phase.

4.2 Automatic Differentiation in Optimisation

In this section, we will first recall the most standard setting in which derivative of functions are used for optimisation: the algorithm of gradient descent, and its theoretical guarantees (Section 4.2.1). Then, we will see how this algorithm can unfortunately fail, in spite of a programs' semantics verifying the condition, when the function's derivative is computed by AD (Section 4.2.2). Fortunately, we will give a sufficient condition to ensure the almost-sure convergence of the algorithm when the derivatives are computed with AD (Section 4.2.3).

4.2.1 Quick recap

In optimisation, a very common technique for minimizing or maximizing a function $f : \mathbb{R}^d \to \mathbb{R}$ is gradient descent/ascent. We review the basic theory [Polyak, 1987]. The problem is finding the minimum of a function

$$x^* = \text{argmin}_{x \in \mathbb{R}^d} f(x)$$

To do so, the method of gradient descent uses a greedy approach by locally following the steepest descent, which can be shown to be given by the gradient:

$$x^{n+1} := x^n - \epsilon \nabla f(x^n) \tag{4.2}$$

When ϵ is too big, or when the function does not have a mimimum (e.g. when it is not lower-bounded), then we cannot hope to converge to a minimum. These are the two main technical requirements for convergence:

Definition 4.2.1. *Let $L \in \mathbb{R}_{>0}$. A differentiable function $f : \mathbb{R}^d \to \mathbb{R}$ is L-smooth if its gradients are Lipschitz continuous, i.e for all $x, y \in \mathbb{R}^d$*

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|$$

Definition 4.2.2. *A function $f : \mathbb{R}^d \to \mathbb{R}$ is bounded below if there exists $R > 0$ such that for all $x \in \mathbb{R}^d$, $f(x) \geq R$.*

Theorem 4.2.1. *Let f be L-smooth, bounded below, and $0 < \epsilon < \frac{2}{L}$. Then, for any initial $x^0 \in \mathbb{R}^d$, following the method from Equation 4.2, the gradient of f tends to 0:*

$$\lim_{t \to \infty} \nabla f(x^n) = 0$$

and the function $f(x)$ monotonically decreases on values of the sequence $\{x^n\}_{n\in\mathbb{N}}$, that is for all $n \in \mathbb{N}$, $f(x^{n+1}) \leq f(x^n)$.

Note that we are actually not guaranteed to converge to a minimal point. A particular nice setting is that of strictly convex functions, which always have a unique global minimum:

Definition 4.2.3. *A function $f : \mathbb{R}^d \to \mathbb{R}$ is convex if for all $x < y \in \mathbb{R}^d, 0 < \lambda < 1$,*

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$$

f is strictly convex if the inequality is strict whenever $x \neq y$.

The convergence of gradient descent to a minimum point can be ensured in the stronger setting when f is *strongly* convex, which implies it is also strictly convex:

Definition 4.2.4. *A function $f : \mathbb{R}^d \to \mathbb{R}$ is strongly convex if there exists $m > 0$ such that for all $x, y \in \mathbb{R}^d, 0 < \lambda < 1$,*

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y) + \frac{m}{2}\lambda(1-\lambda)\|x-y\|_2^2$$

Theorem 4.2.2. *Let f be as in Theorem 4.2.1 and strongly convex. Then, for $0 < \epsilon < \frac{2}{L}$ and any initial $x^0 \in \mathbb{R}^d$, the method from Equation 4.2 converges to the unique global minimum x^* of f, with the rate of geometric progression:*

$$\|x^n - x^*\| \leq cq^n \qquad 0 \leq q < 1$$

Similarly, there are convergence theorems for the case where the gradient is computed with stochastic error, which is often called stochastic gradient descent (see e.g. [Polyak, 1987], Chapter 4). We will not need to review the many variations of all these theorems, it suffices to say that this is a well-studied topic with many subtle variations, and as more generally in analysis, counterexamples are numerous and not always intuitive. We will only be interested in gradient descent where the stochasticity comes from randomizing the starting rate ϵ or the initial point x^0, but where the function being optimized and its gradient are non-probabilistic. We will now see this in action, with new results based on PAP functions.

4.2.2 Failure of Gradient descent with intensional derivatives

PAP functions are in particular almost everywhere differentiable. We have also seen that for differentiable functions, AD produces the correct derivative almost everywhere (Theorem 3.5.3). As the gradient descent algorithm only requires finitely many steps, the call to AD to compute gradients will only be at finitely many points x^n, which represent a negligible subset of the points. It is therefore tempting to believe that an analogue of the theorem of convergence of gradient descent should hold for PAP functions: the set of potential problems is negligible, so maybe it will be fine. We show that these intuitions are wrong: it is not sound in general to use

intensional derivatives as a substitute for proper gradients in a gradient descent algorithm, even in the nicest settings where the program represents a smooth convex function. Even when randomizing the initial point x^0 in the GD algorithm, we might fail to reach a critical point with good probability.

Proposition 4.2.3. *Let $\epsilon > 0$. There exists a program P such that $[\![P]\!] : \mathbb{R} \to \mathbb{R}$ satisfies the condition of Theorem 4.2.2, and yet, for all $x^0 \in \mathbb{R}$, the gradient descent method (Equation 4.2) diverges to ∞, when the gradients are computed with AD (as in Figure 3.4).*

Furthermore, this remains true for more sophisticated versions of gradient descent, e.g. when the rate ϵ is changing at each step.

Proof. Let $\epsilon > 0$. Let P be the closed program defined by the following Python code, where lr is our chosen rate ϵ.

```
# Recursive helper
def g(x, n):
    if x > 0:
        return ((x - n) * (x - n)) / (lr * 2)
    if x == 0:
        return x / lr + n*n / (2*lr)
    else:
        return g(x+1, n+1)

def P(x) = g(x,0)
```

This can easily be written in our language with recursion and conditionals (Figure 3.3), but it would slightly impact readability.

$[\![P]\!]$ is PAP, and by inspection, we see that $[\![P]\!] = x \mapsto \frac{x^2}{2\epsilon}$. This is proved by simple induction on $\mathbb{N}$ in the recursive function g defining P. Now, let us show that $f := [\![P]\!]$ satisfies the hypothesis of Theorem 4.2.2.

- f is strongly convex: this is well-known for the x^2 function
- f is bounded below: $f \geq 0$
- f is L-smooth for some L:

$$\nabla f(x) - \nabla f(y) = \frac{2x}{2\epsilon} - \frac{2y}{2\epsilon} = \frac{1}{\epsilon}(x - y)$$

  And therefore f is $\frac{1}{\epsilon}$-smooth.

This means that for all $0 < \epsilon' < \frac{2}{\frac{1}{\epsilon}} = 2\epsilon$, gradient descent on f converges to the global minimum 0. Now, let us see that this is not the case when we use the intensional derivatives for f computed by AD, in the case $\epsilon' := \epsilon$.

Applying our standard AD macro to P, we obtain a PAP function P' that returns $\frac{x}{\epsilon}$ when $x \in \mathbb{R}_{>0}$, and $\frac{(x+\lfloor|x|\rfloor+1)-(\lfloor|x|\rfloor+1)}{\epsilon}$ when $x \in \mathbb{R}_{<0} - \{-n \mid n \in \mathbb{N}\}$, and $\frac{1}{\epsilon}$ for $x \in \{-n \mid n \in \mathbb{N}\}$.

Let $x^0 \in \mathbb{R}$ be any initialization for the gradient descent algorithm. Then, after one step of gradient descent, we obtain $x^1 := x^0 - \epsilon \times P'(x^0)$. If $x^0 > 0$, then $[\![P'(x^0)]\!] = \frac{x^0}{\epsilon}$ and thus $x^1 = 0$. When $x^0 < 0$ is not an integer, $[\![P'(x^0)]\!] = \frac{x^0}{\epsilon}$ as thus $x^1 = 0$ as well. Finally, when $x^0 < 0$ is an integer, $[\![P'(x^0)]\!] = \frac{1}{\epsilon}$ and therefore $x^1 = x^0 - 1$. In any case, x^1 is a non-positive integer, and for x^2 we will always have $x_2 = x_1 - 1$. Therefore, $x^n \to -\infty$ as $t \to \infty$.

A similar program P can be constructed when the learning rate is non-fixed. The recursive program inside P's definition simply calls $\epsilon(n)$ instead of using ϵ, and the analysis will be very similar, except that x^1 will not necessarily be an integer when x^0 is a non-positive integer, but this happens for almost no x^0. □

The proof applies to an idealized setting where gradient descent is run with real numbers, rather than floating-point numbers, but in our experiment (Figure 4.1 and 4.2), PyTorch did diverge on this program. This particular example and the PyTorch implementation is due to Alex Lew.

```
import torch

# Recursive helper
def g(x, lr, n):
    if x > 0:
        return ((x - n) * (x - n)) / (lr * 2)
    if x == 0:
        return x / lr + n*n / (2*lr)
    else:
        return g(x+1, lr, n+1)

# Create a pathological loss function for a particular learning rate
def make_loss(lr):
    def l(x):
        return g(x, lr, 0)
    return l

# Randomly initialize a position for SGD
position = torch.rand(1, requires_grad=True)

# Set up SGD and loss function
optim = torch.optim.SGD([position], lr=0.1, momentum=0)
l = make_loss(0.1)

# Run SGD
for i in range(100):
    optim.zero_grad()
    loss = l(position)
    loss.backward()
    optim.step()
    print(position, loss)
```

Figure 4.1: PyTorch implementation of the example of failure of gradient descent with intensional derivatives.

```
tensor([0.], requires_grad=True) tensor([0.1078], grad_fn=<DivBackward0>)
tensor([-1.], requires_grad=True) tensor([0.], grad_fn=<AddBackward0>)
tensor([-2.], requires_grad=True) tensor([5.], grad_fn=<AddBackward0>)
tensor([-3.], requires_grad=True) tensor([20.], grad_fn=<AddBackward0>)
tensor([-4.], requires_grad=True) tensor([45.], grad_fn=<AddBackward0>)
tensor([-5.], requires_grad=True) tensor([80.], grad_fn=<AddBackward0>)
tensor([-6.], requires_grad=True) tensor([125.], grad_fn=<AddBackward0>)
tensor([-7.], requires_grad=True) tensor([180.], grad_fn=<AddBackward0>)
tensor([-8.], requires_grad=True) tensor([245.], grad_fn=<AddBackward0>)
tensor([-9.], requires_grad=True) tensor([320.], grad_fn=<AddBackward0>)
tensor([-10.], requires_grad=True) tensor([405.], grad_fn=<AddBackward0>)
tensor([-11.], requires_grad=True) tensor([500.], grad_fn=<AddBackward0>)
tensor([-12.], requires_grad=True) tensor([605.], grad_fn=<AddBackward0>)
tensor([-13.], requires_grad=True) tensor([720.], grad_fn=<AddBackward0>)
tensor([-14.], requires_grad=True) tensor([845.], grad_fn=<AddBackward0>)
```

Figure 4.2: A run of the program from Figure 4.1. The loss function is everywhere equal to $\frac{x^2}{2lr}$. The value of the position keeps decreasing by 1 each iteration, and the loss itself keeps growing. We only show the first few steps as the program diverges.

Similar counterexamples can be constructed for some variants of the gradient descent algorithm, e.g. when the rate is ϵ is random but the initial point x^0 is fixed.

Proposition 4.2.4. *Let x^0 be a fixed initial point. Then there exists a program P such that $[\![P]\!] : \mathbb{R} \to \mathbb{R}$ satisfies the conditions of Theorem 4.2.1, and $\nabla[\![P]\!](x^0) \neq 0$, and yet, for all $\epsilon > 0$, the gradient descent method yields $x^n = x^0$ for all $n \in \mathbb{N}$, when the gradients are computed with AD.*

Proof. Simply choose $P(x) = \textbf{if } x = x^0 \textbf{ then } (x^0)^2 \textbf{ else } x^2$, for which AD will give the intensional derivative 0 at x^0. □

The counterexamples constructed for the proofs of the theorems above are all specifically tied to a specific learning rate. It is therefore reasonable to think that by also randomizing the initial learning rate, one might be able to avoid such counter-examples with probability 1. This is actually the case, as we will now show.

4.2.3 Almost-sure Convergence of Gradient Descent with intensional derivatives

Theorem 4.2.5 (Convergence of Gradient Descent with intensional derivatives)**.** *Let f be differentiable and PAP, bounded below, L-smooth. Let (ϵ, x_0) be drawn uniformly randomly on $(0, \frac{2}{L}) \times \mathbb{R}^d$. Then, when gradient descent computes gradients using AD, the true gradient tends to 0 with probability 1:*

$$\lim_{t\to\infty} \nabla f(x^n) = 0$$

Furthermore, with probability 1, the function $f(x)$ monotonically decreases: $f(x^{n+1}) \leq f(x^n)$.

If f is additionally strongly convex, then, with probability 1, gradient descent converges to the global minimum of f.

Proof. Given a fixed intensional derivative ∇f of f, let $F_{int}(\epsilon, x) = (\epsilon, x - \epsilon \nabla_x f)$. Let $F_{true}(\epsilon, x) = (\epsilon, x - \epsilon \nabla^*_x f)$ where $\nabla^*_x f$ is the true gradient of the differentiable function f. We restrict the domain of F_{true} and F_{int} to $(0, \frac{2}{L}) \times \mathbb{R}^d$. For $\delta > 0$ and $n \in \mathbb{N}$, we define

$$X^*_{n,\delta} := \{(\epsilon, x^0) \in (0, \frac{2}{L}) \times \mathbb{R}^d \mid |\nabla^* f(\pi_2 F^n_{true}(\epsilon, x^0))| \leq \delta\}$$

where g^n is the n-fold composition of a function g with itself. As f is L-smooth, $\nabla^* f$ is locally L-Lipschitz and thus continuous. Therefore, by the first conclusion of Theorem 4.2.1, for any $\delta > 0$,

$$\bigcup_{t=0}^{\infty} X^*_{n,\delta} = (0, \frac{2}{L}) \times \mathbb{R}^d$$

We define similarly, for any $\delta > 0$,

$$X_{n,\delta} := \{(\epsilon, x^0) \in (0, \frac{2}{L}) \times \mathbb{R}^d \mid |\nabla^* f(\pi_2 F^n_{int}(\epsilon, x^0))| \leq \delta\}$$

We will show by induction on $n \in \mathbb{N}$ that $X_{n,\delta}$ and $X^*_{n,\delta}$ only differ by a null-set, i.e. that there exists null-sets $N_{n,\delta}$ and $M_{n,\delta}$ such that $X_{n,\delta} \cap N^c_{n,\delta} = X^*_{n,\delta} \cap M^c_{n,\delta}$. This implies that

$$\begin{aligned}
\bigcup_{n=0}^{\infty} X_{n,\delta} &\supseteq \bigcup_{t=0}^{\infty} X_{n,\delta} \cap N_{n,\delta} \\
&= \bigcup_{n=0}^{\infty} X^*_{n,\delta} \cap M_{n,\delta} \\
&\supseteq (\bigcup_{n=0}^{\infty} X^*_{n,\delta}) \cap ((0, \frac{2}{L}) \times \mathbb{R}^d - \bigcup_{n=0}^{\infty} M_{n,\delta}) \\
&= (0, \frac{2}{L}) \times \mathbb{R}^d \cap ((0, \frac{2}{L}) \times \mathbb{R}^d - \bigcup_{n=0}^{\infty} M_{n,\delta}) \\
&= (0, \frac{2}{L}) \times \mathbb{R}^d - \bigcup_{n=0}^{\infty} M_{n,\delta}
\end{aligned}$$

As a countable union of null-sets is a null-set, this means that $\bigcup_{n=0}^{\infty} X_{n,\delta}$ equals $(0, \frac{2}{L}) \times \mathbb{R}^d$ up to a null-set, as desired.

By induction on $n \in \mathbb{N}$, we show the following property:

$$\begin{gathered}
\forall A \subseteq (0, \frac{2}{L}) \times \mathbb{R}^d \text{ measurable of measure 0,} \\
\{(\epsilon, x) \mid F^n_{true}(\epsilon, x) \in A\} \text{ is measurable with measure 0}
\end{gathered}$$

If $n = 0$, F^n_{true} is the identity and it's obvious. Let $n \in \mathbb{N}$ such that the property holds. Let $A \subseteq (0, \frac{2}{L}) \times \mathbb{R}^d$ be of measure 0. To use the induction hypothesis, it

suffices to show that $F_{true}^{-1}(A)$ has measure 0. The Jacobian matrix of F_{true} is an $(n+1) \times (n+1)$ matrix, given by

$$J(F_{true})_{\epsilon,x} = \begin{pmatrix} 1 & 0_n^T \\ J_x f^T & I_n - \epsilon H_x f \end{pmatrix}$$

where I_n is the $n \times n$ identity matrix, 0_n the zero vector in $\mathbb{R}^n$, $J_x f$ is the Jacobian of f at x, $H_x f$ its Hessian matrix at x, and $(-)^T$ is the transpose operation.

The rank of $J(F_{true})_{\epsilon,x}$ is lower bounded by one plus the rank of $I_n - \epsilon H f_x$. The latter is strictly less than n exactly when $\frac{1}{\epsilon}$ is an eigenvalue of $H f_x$. For a fixed x, this can only occur for finitely many ϵ (at most n). Thus, $J(F_{true})_{\epsilon,x}$ has max rank $n+1$ almost everywhere. Denote by R the set on which F_{true} has max rank. f being PAP, F is PAP as well. Therefore, F_{true} is almost everywhere real analytic. Denote by C the set on which F_{true} is real analytic. Let $N = R \cap C$. Note that N^c is a null-set. On N, F is $F \in \mathcal{C}^1$ and its Jacobian is invertible. Therefore, by the implicit function theorem, the result is locally true around any point $x \in N$. This means, for all $x \in N$, there exists a neighborhood U_x of x such that $F_{true}^{-1}(A) \cap U_x$ is a null-set. We can use a countable cover $\{U_i\}_{i \in \mathbb{N}}$ of N of such neighborhoods, and thus

$$\begin{aligned} F_{true}^{-1}(A) &= F_{true}^{-1}(A) \cap (N \cup N^c) \\ &\subseteq F_{true}^{-1}(A) \cap \Big[(\bigcup_{i \in \mathbb{N}} U_i) \cup N^c \Big] \\ &\subseteq \Big[\bigcup_i F_{true}^{-1}(A) \cap U_i \Big] \cup Z \end{aligned}$$

$F_{true}^{-1}(A)$ is measurable as A is, and as null-sets are closed under countable unions, this shows that $F_{true}^{-1}(A)$ has measure 0, which concludes the induction.

Now, we show by induction on $n \in \mathbb{N}$ that for almost all $(\epsilon, x^0) \in (0, \frac{2}{L}) \times \mathbb{R}^d$,

$$F_{true}^n(\epsilon, x^0) = F_{int}^n(\epsilon, x^0)$$

Once again, the base case is immediate. Let $n \in \mathbb{N}$ such that the property holds. As ∇f is an intensional derivative of f, for almost all $x \in \mathbb{R}$, we have that $\nabla_x f = \nabla_x^* f$. Therefore, for almost all (ϵ, x), we have that $F_{true}(\epsilon, x) = F_{int}(\epsilon, x)$. Let A be the set of points on which they differ. By the previous proposition, $F_{true}^{-1}(A)$ has measure 0. Therefore, F_{true}^{n+1} and F_{int}^{n+1} may differ only on the set $F_{true}^{-1}(A)$, which has measure 0. This concludes the induction.

As a corollary of what we have just shown, we have that for almost all (ϵ, x^0), x^n is the same, whether it was obtained from true or AD-computed gradients. Indeed, x^n is either $\pi_2 F_{true}^n(\epsilon, x^0)$ or $\pi_2 F_{int}^n(\epsilon, x^0)$, and we have shown that $F_{true}^n(\epsilon, x^0)$ and $F_{true}^n(\epsilon, x^0)$ agree almost everywhere. Therefore $f(x^{n-1}) \leq f(x^n)$ remains true almost-everywhere for all n, as a countable union of null-sets is a null-set. Finally, if f is strongly convex, then $\{x^n\}_{n \in \mathbb{N}}$ will reach the global minimum. As x^n is the same, whether it was obtained from true or AD-computed gradients, for almost all (ϵ, x^0), this means that gradient-descent with AD-computed gradients will also almost surely converge to the global minimum. □

This theorem ensures that gradient descent will converge as expected when intensional gradients are used, at the reasonable cost of only getting almost-sure convergence. This makes a precise statement for the informal and slightly erroneous argument that can be found in the literature, e.g. in Mazza and Pagani [2021].

4.3 A monad on ωPAP

In this section, we construct a commutative monad $T : \omega PAP \to \omega PAP$. We will use it in the next section for giving a semantics to an extension to our language (Figure 3.3) with probabilistic constructs, closely following Vákár et al. [2019]. In such a case, terms will denote certain unnormalized probability kernels, and in particular closed terms $\vdash t : \tau$ will denote measures on $[\![\tau]\!]$. In the case of $\tau = \mathbf{real}^n$, closed terms will be s-finite measures [Staton, 2017], and our main result in the next Section (Theorem 4.4.18) will be a more precise characterization of definable measures.

The construction of the monad relies on two main ingredients: Theorem 4.3.1 from Kammar and McDermott [2018] which is based on factorization systems (Section 4.3.1), and the construction in Section 4.3.5 which follows closely ideas developed in Vákár et al. [2019]. The construction builds a natural transformation $\mathbb{E} : RX \to \mathcal{J}X$, where RX is a "randomization functor" (Section 4.3.4), which can be seen as a source of randomness for probabilistic computations. $\mathbb{E}$ sends such a source of randomness to a measure μ, seen as an expectation operator $f \mapsto \int f d\mu$. Expectation operators can be seen to live within a continuation monad $\mathcal{J}X$, and our commutative monad T (Theorem 4.3.11 in Section 4.3.6) is extracted as a commutative submonad of the continuation monad, with the help of Theorem 4.3.1.

4.3.1 Background: Factorizations systems

Definition 4.3.1. *A **weak factorization system** in a category* $\mathbf{C}$ *is a pair* (L, R) *of classes of morphisms of* $\mathbf{C}$ *such that:*

1. *Both* L *and* R *are closed under composition and contain all isomorphisms*
2. *every morphism* f *factors:*
$$\begin{array}{ccc} X & \xrightarrow{\forall f \in \mathbf{C}} & Y \\ & {\scriptstyle \exists l \in L} \searrow \quad \nearrow {\scriptstyle \exists r \in R} & \\ & Z & \end{array}$$
3. *lifting property for every commuting square* $\begin{array}{ccc} A & \xrightarrow{\forall u} & B \\ {\scriptstyle \forall l \in L}\downarrow & & \downarrow{\scriptstyle \forall r \in R} \\ C & \xrightarrow[\forall v]{} & D \end{array}$ *, there exists a morphism* $d : C \to B$ *making the following diagram commute:* $\begin{array}{ccc} A & \xrightarrow{u} & B \\ {\scriptstyle l}\downarrow & \overset{d}{\nearrow} & \downarrow{\scriptstyle r} \\ C & \xrightarrow[v]{} & D \end{array}$

Definition 4.3.2. *An **orthogonal factorization system** is a weak factorization system (L, R) in which solutions to lifting problems are unique.*

Definition 4.3.3. *An **(epi,mono)-factorization** is an orthogonal factorization system (L, R) in which the left class are epis and the right class are monos. In this case, we will suggestively write the classes $(\mathcal{E}, \mathcal{M})$.*

Definition 4.3.4. *A epi e is a strong epi if in each square* $\begin{array}{ccc} P & \xrightarrow{f} & A \\ {\scriptstyle e}\downarrow & \overset{\exists d}{\nearrow} & \downarrow{\scriptstyle m} \\ Q & \xrightarrow[g]{} & B \end{array}$ *whenever m is mono.*

Definition 4.3.5. *A dense function is a continuous function $e : X \twoheadrightarrow Y$ between ωcpo's such that the smallest ω-chain-closed subset $U \subseteq Y$ with $e[X] \subseteq U$ is Y itself.*

Definition 4.3.6. *A full function is a continuous function $m : X \rightarrowtail Y$ between ωcpo's such that $m(x) \leq m(x')$ implies $x \leq x'$ for each $x, x' \in X$.*

Definition 4.3.7. *A full mono between categories of ω-concrete sheaves is a full mono between them as ωcpos.*
A densely strong epi $e : X \twoheadrightarrow Y$ between ω-concrete sheaves is an ω-concrete sheaf morphism that maps the plots $\mathcal{P}_X^U$ into a Scott dense subset of $\mathcal{P}_Y^U$ w.r.t the pointwise order.

f $(\mathcal{E}, \mathcal{M})$ is a factorization system on $\mathbf{C}$, then for every morphism $f : A \to B$ we denote by (f^e, f^m) the pair of morphisms obtained from a factorization of f. We denote by $f[A]$ the image of A by f, i.e. $f^e : A \to f[A]$ and $f^m : f[A] \to B$.

Definition 4.3.8 ([Kammar and McDermott, 2018])**.** *We say that $(\mathcal{E}, \mathcal{M})$ is closed under an endofunctor S on $\mathbf{C}$ if for every $e : A \to B \in \mathcal{E}$, we have $Se : SA \to SB \in \mathcal{E}$.*

Theorem 4.3.1 ([Kammar and McDermott, 2018])**.** *Let $\mathbf{C}$ be a category, $(\mathcal{E}, \mathcal{M})$ a factorization system, S and T monads over $\mathbf{C}$, and $m : S \to T$ a monad morphism.*

- *If $(\mathcal{E}, \mathcal{M})$ is closed under S then $m[S]$ is a monad, and m^e, m^m are monad morphisms. As a consequence, every algebra for $m[S]$ induces an algebra for S.*
- *If, moreover, $(\mathcal{E}, \mathcal{M})$ is closed under products, S and T are strong monads, and m is a strong monad morphism, then $m[S]$ is a strong monad and m^e, m^m are strong monad morphisms*
- *If the right class of $(\mathcal{E}, \mathcal{M})$ consists of monos, then S need only be a monad structure.*

4.3.2 Background: transfinite induction

Definition 4.3.9. *A well-order on a set S is a total order on S with the property that every non-empty subset of S has a least element in this ordering.*

Definition 4.3.10. *The Principle of Transfinite Induction. is defined as follows. Let C be any well-ordered set. Let P be a property with*

$$\forall \alpha \in C, (\forall \beta \in C, \beta < \alpha \to P(\beta)) \to P(\alpha)$$

Then, the following holds:

$$\forall \alpha \in C, P(\alpha)$$

Using it is similar to normal proof by induction with case analysis.

Example 4.3.1. *Let us prove the classic result that every infinite ordinal can be written uniquely as the sum of a limit ordinal and a finite ordinal.*

We do this with a proof by transfinite induction. If α is a limit ordinal then $\alpha = \alpha + 0$. If $\alpha = \beta + 1$ and $\beta = \gamma + \delta$ then $\alpha = \gamma + \delta + 1$ where γ is a limit and $\delta + 1$ is a finite ordinal, as wanted. The case where $\alpha = 0$ is not needed because this holds vacuously for finite ordinals.

To see that this sum is unique, suppose this is true for β, and let $\alpha = \beta + 1$. Then $\beta = \gamma + \delta$ and $\alpha = \gamma + (\delta + 1)$. If $\alpha = \gamma' + \delta'$, where γ' is a limit ordinal, then $\delta' > 0$ because α is a successor and so $\beta = \gamma' + (\delta' - 1)$. Since this sum is unique for β, we have $\gamma = \gamma'$ and $\delta' - 1 = \delta$ as wanted. For a limit ordinal, this is immediate as $\gamma + \delta$ is not a limit ordinal if $\delta \neq 0$. So $\alpha = \alpha + 0 = \gamma + \delta$, which implies $\delta = 0$ and $\alpha = \gamma$, as desired.

4.3.3 Orthogonal factorization system in ωPAP

We wish to prove that densely strong epis and full monos form an orthogonal factorization system on ωPAP. The proofs in this section are due to Alex Lew.

First, we prove a useful lemma:

Lemma 4.3.2. *Let X be an ωPAP space and let $S_0 \subseteq |X|^U$ be a set of functions. Then the following transfinite recursion is well-defined, and converges to $S = Cl_{U,\omega}^{|X|} S_0$, the closure of S_0 under lubs of increasing ω-chains:*

- *At step 0, return S_0.*
- *At step $\alpha = \beta + 1$, return $S_\alpha = S_\beta \bigcup \{\bigvee_{i \in \mathbb{N}} f_i \mid \forall i \in \mathbb{N}, f_i \in S_\beta, (f_i)_{i \in \mathbb{N}}\ \omega\text{-chain}\}$.*
- *At step λ, where λ is a limit ordinal, return $S_\lambda = \bigcup_{\beta < \lambda} S_\beta$.*

Proof. First, note that $S_\alpha \subseteq S_\beta$ whenever $\beta \geq \alpha$. We show this by transfinite induction on β:

- If $\beta = \alpha$, then $S_\beta = S_\alpha$.

- If $\beta = \gamma + 1 > \alpha$, then $S_\alpha \subseteq S_\gamma \subseteq S_\beta$, where the first inclusion holds by the inductive hypothesis and the second by the definition of $S_\beta = S_{\gamma+1}$.
- If $\beta > \alpha$ is a limit ordinal, then it explicitly is a union of sets, including S_α.

Furthermore, we now show by transfinite induction that if $S_\alpha = S_\beta$ for some $\beta > \alpha$, then $S_\gamma = S_\alpha$ for all $\gamma > \alpha$:

- If $\alpha < \gamma < \beta$, then the result is immediate: we know $S_\alpha \subseteq S_\gamma \subseteq S_\beta$, and so when $S_\alpha = S_\beta$, all of the set inclusions must be equalities.
- If $\alpha < \beta$, then either $\beta = \alpha + 1$ or $\alpha + 1 < \beta$. So, by the last bullet, $S_{\alpha+1}$ in particular is equal to S_α, and thus we know that the 'step' operator that adds all lubs of chains in S_α leaves S_α unchanged. If $\alpha + 1 < \beta$ we then conclude by using the induction hypothesis.
- Now suppose $\gamma > \beta$. Let $\alpha = \lambda + n$, $\gamma = \delta + m$.
  - If $\lambda = \delta$, then we're done: the set does not change under our 'step' operation, and so $S_{\lambda+k} = S_\alpha$ for all $k \geq n$.
  - If $\lambda \neq \delta$, then $S_\delta = (\bigcup_{\epsilon \leq \alpha} S_\epsilon) \bigcup (\bigcup_{\delta > \epsilon > \alpha} S_\epsilon)$. The second of these, by induction, is equal to S_α, and the first must be equal to S_α by the subset property proven above. Because the 'step' operation leaves S_α unchanged, this also proves $S_{\delta+m} = S_\alpha$.

Assuming the axiom of choice, the set $|X|^U$ can be well-ordered and therefore has an initial ordinal α. Let α^+ be the initial ordinal of the successor cardinal to $|\alpha|$ (i.e. the first cardinal strictly greater than $|\alpha| = ||X|^U|$). We now show that $S_{\alpha^+} = S_{\alpha^+ + 1}$ and therefore that $S_{\alpha^+} = S_\beta$ for all $\beta > \alpha^+$.

Suppose by way of contradiction that $S_{\alpha^+} \neq S_{\alpha^+ + 1}$. Then, by the point above, for all $\gamma < \delta \leq \alpha^+$, we have $S_\gamma \neq S_\delta$. We can now define an injective mapping from S_{α^+} to $|X|^U$. For each input $\beta < \alpha^+$, we choose an element of $S_{\beta+1} \setminus S_\beta$. That is:

- For $\beta = 0$, choose any element of $S_1 \setminus S_0$.
- For $\beta = \gamma + 1$, choose any element of $S_{\gamma+2} \setminus S_{\gamma+1}$, which must be non-empty by assumption.
- For β a limit ordinal, choose any element of $S_{\beta+1} \setminus S_\beta$, which must be non-empty.

So this shows that $|S_{\alpha^+}| \leq ||X|^U| = |S_\alpha| < S_{\alpha^+}$, where the last inequality comes from the hypothesis. This is a contradiction, and therefore $S_\alpha = S_{\alpha^+}$. We now know that the process converges in at most α steps.

We finally need to show that $S_\alpha = Cl^{|X|}_{U,\omega} S_0$. By a clear inductive argument, we have that $S_\beta \subseteq Cl^{|X|}_{U,\omega} S_0$ for all β. Furthermore, let $(f_i)_{i \in \mathbb{N}}$ be an ω-chain in S_α. Then, take the supremum ordinal $\beta = \bigvee_{i \in \mathbb{N}} \beta_i$, where β_i is the minimum ordinal where $f_i \in S_{\beta_i}$. By construction, S_β must contain f, the lub of $(f_i)_{i \in \mathbb{N}}$. As $S_\beta \subseteq S_\alpha$, this shows that S_α is closed under lubs of omega-chains and contains S_0, and

thus contains the smallest such closure $Cl^{|X|}_{U,\omega} S_0$, therefore finally concluding that $S_\alpha = Cl^{|X|}_{U,\omega} S_0$. □

As a result, we can reason about sets of the form $Cl^{|X|}_{U,\omega} S_0$ using transfinite induction. Spelling out what that means in our case, for a claim of the form $\forall x \in Cl^{|X|}_{U,\omega} S_0, P(x)$, we must prove:

- Base case: $\forall x \in S_0, P(x)$.
- Inductive case: for any ordinal α, assuming P holds for all $x' \in \bigcup_{\beta<\alpha} S_\beta$, prove that for all $x \in S_\alpha$, $P(x)$ holds.

But note that in our setting, for limit ordinals α, the inductive case is trivial, because $S_\alpha = \bigcup_{\beta<\alpha} S_\beta$. Thus, it suffices to consider the successor case where $\alpha = \beta + 1$: in this case, the new elements of S_α for which we must prove P are just lubs of elements from S_β (for all of which P holds by induction). This yields a simpler inductive principle:

- Base case: $\forall x \in S_0, P(x)$.
- Inductive case: suppose $(f_i)_{i\in\mathbb{N}}$ is an ω-chain of $U \to |X|$ functions, and let $f = \bigvee_{i\in\mathbb{N}} f_i$. Then, show that if $P(f_i)$ holds for all $i \in \mathbb{N}$, then $P(f)$ holds.

We can now begin in earnest, with the three requirements for a weak factorization system.

Lemma 4.3.3. *The classes $\mathcal{M}$ of full monos and $\mathcal{E}$ of densely strong epis in ωPAP are closed under composition.*

Proof. Let $f : X \rightarrowtail Y, g : Y \rightarrowtail Z$ be full monos in ωPAP. Suppose $(f;g)(a) \leq (f;g)(b)$ for some $a, b \in |X|$. Rewriting, we have that $g(f(a)) \leq g(f(b))$, and since g is a full mono, this implies that $f(a) \leq f(b)$. Because f is a full mono, we have that $a \leq b$, as required.

Now consider densely strong epis $f : X \twoheadrightarrow Y, g : Y \twoheadrightarrow Z$. We wish to show that their composition $f;g$ is densely strong. Clearly $f;g$ is a morphism, so the main task is to show that $(f;g)[\mathcal{P}^U_X]$ is dense in $\mathcal{P}^U_Z$, i.e., that $\mathcal{P}^U_Z = Cl^{|Z|}_{U,\omega}\left((f;g)[\mathcal{P}^U_X]\right)$.

First, we know that $Cl^{|Z|}_{U,\omega}\left((f;g)[\mathcal{P}^U_X]\right) \subseteq \mathcal{P}^U_Z$: by the definition of an ωPAP morphism, $(f;g)[\mathcal{P}^U_X] \subseteq \mathcal{P}^U_Z$, and plots are closed under taking lubs. For the inclusion in the other direction, it suffices to show that $g[\mathcal{P}^U_Y] \subseteq Cl^{|Z|}_{U,\omega}\left((f;g)[\mathcal{P}^U_X]\right)$, because g is a densely strong epi: taking the lub-closure on both sides, we get $\mathcal{P}^U_Z \subseteq Cl^{|Z|}_{U,\omega}\left((f;g)[\mathcal{P}^U_X]\right)$.

Now consider an arbitrary element $g \circ p \in g[\mathcal{P}^U_Y]$. Since f is a densely strong epi, we know $p \in Cl^{|Y|}_U(f[\mathcal{P}^U_X])$. We prove, by induction on the transfinite process of taking this closure, that $g \circ p \in Cl^{|Z|}_U\left((f;g)[\mathcal{P}^U_X]\right)$:

- (Step 0.) Let $p = f \circ q$ for some $q \in \mathcal{P}^U_X$. Then $g \circ p = (f;g) \circ q \in (f;g)[\mathcal{P}^U_X] \subseteq Cl^{|Z|}_U\left((f;g)[\mathcal{P}^U_X]\right)$.

- (Step α.) Let $p = \bigvee_{i\in\mathbb{N}} p_i$ for a chain of plots $(p_i)_{i\in\mathbb{N}}$, where p_i is introduced at a step $\alpha_i < \alpha$. Then, by the induction hypothesis, $g \circ p_i \in Cl_U^{|Z|}\left((f;g)[\mathcal{P}_X^U]\right)$ for all $i \in \mathbb{N}$. Then, $g \circ p = g \circ \bigvee_{i\in\mathbb{N}} p_i = \bigvee_{i\in\mathbb{N}}(g \circ p_i)$ is also in the closure.

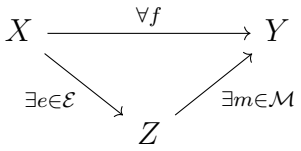
This completes the proof. □

We must also prove that $\mathcal{M}$ and $\mathcal{E}$ contain all isomorphisms.

Lemma 4.3.4. *The full monos and densely strong epis contain all isos.*

Proof. Let $f : X \to Y$ be an iso, and $g : Y \to X$ its inverse. First, we show that f is a full mono. If $f(x) \leq f(y)$, then $x = g(f(x)) \leq g(f(y)) = y$, by monotonicity of g. Second, we show that f is also a densely strong epi; in fact, $f[\mathcal{P}_X^U] = \mathcal{P}_Y^U$. As f is a morphism, it guarantees that $f[\mathcal{P}_X^U] \subseteq \mathcal{P}_Y^U$. Furthermore, any plot $p \in \mathcal{P}_Y^U$ is the image of $g \circ p \in \mathcal{P}_X^U$ under f. □

Lemma 4.3.5. *Every ωPAP morphism $f : X \to Y$ factors through $\mathcal{E}$ and $\mathcal{M}$:*



Proof. Let $f : X \to Y$ and let Z be the following ωPAP space:

- carrier set: $|Z| = Cl_\omega^Y f[|X|]$, the lub-closure of $f[|X|] \subseteq |Y|$ under the order $\leq_Y$.
- order: $a \leq_Z b$ iff $a \leq_Y b$.
- plots: $\mathcal{P}_Z^U = Cl_{U,\omega}^{|Z|}(f[\mathcal{P}_X^U])$.

Let's first check that this is an ωPAP space, by checking the axioms:

- **All constant maps are plots.** Let $z \in |Z|$. If $z = f(x)$ for some $x \in |X|$, then the constant map $\lambda u.z = f \circ (\lambda u.x)$ is a plot because $\lambda u.x$ is. Otherwise, by transfinite induction on the lub-closure process, $z = \bigvee_{i\in\mathbb{N}} z_i$, where, by the induction hypothesis, $\lambda u.z_i \in \mathcal{P}_Z^U$ for all $i \in \mathbb{N}$ Then, as lubs in $\mathcal{P}_Z^U$ are computed pointwise, we have $\lambda u.z = \bigvee_{i\in\mathbb{N}} \lambda u.z_i \in \mathcal{P}_Z^U$.
- **Plots are closed under precomposition with PAP functions.** We again proceed by transfinite induction on the plots $p \in \mathcal{P}_Z^U$. In the base case, $p = f \circ q$ for some plot $q \in \mathcal{P}_X^U$. Letting $g : V \to U$ be any PAP function, we have that $g; q \in \mathcal{P}_X^V$ and therefore $g; p = f \circ (g; q) \in f[\mathcal{P}_X^V] \subseteq \mathcal{P}_Z^V$. In the inductive case, $p = \bigvee_{i\in\mathbb{N}} p_i$ for plots $p_i \in \mathcal{P}_Z^U$, where, by the induction hypothesis, when $g : V \to U$ is PAP, $g; p_i \in \mathcal{P}_Z^V$. Then, as lubs are taken pointwise, $g; p = g; \bigvee_{i\in\mathbb{N}} p_i = \bigvee_{i\in\mathbb{N}}(g; p_i) \in \mathcal{P}_Z^V$.

- **Plots are closed under gluing along an analytic partition.** Let $\{A_i\}_{i\in\mathbb{N}}$ be an analytic partition of U, and $p_i \in \mathcal{P}_Z^{A_i}$ be plots. We proceed by transfinite induction: the inductive hypothesis is that for all collections $\{p_i\}_{i\in\mathbb{N}}$ in which each plot p_i was introduced at a step with ordinal β or lower, the piecewise gluing p of the p_i's along the A_i's is a plot in $\mathcal{P}_Z^U$. In the base case, $\beta = 0$ and we have that every $p_i = f \circ q_i$ for some $q_i \in \mathcal{P}_X^{A_i}$. The gluing of the p_i along A_i yields $f \circ q$, where $q \in \mathcal{P}_X^U$ is the gluing of the q_i along A_i. Since $q \in \mathcal{P}_X^U$, $f \circ q \in f[\mathcal{P}_X^U] \subseteq \mathcal{P}_Z^U$, as desired. In the inductive case, the $(p_i)_{i\in\mathbb{N}}$ are lubs, $p_i = \bigvee_{j\in\mathbb{N}} p_i^j$, where $p_i^j \in \mathcal{P}_Z^{A_i}$ is a plot of order lower than β. Now consider $p^j := \bigvee_{i\in\mathbb{N}} p_i^j$, the gluing of p_i^j along A_i, which by induction is a plot in $\mathcal{P}_Z^U$. Lubs act pointwise, so we can compute lubs separately on each component of the partition. Doing so, we find that $\bigvee_{j\in\mathbb{N}} p^j = p$, the piecewise gluing of the p_i. Thus $p \in \mathcal{P}_Z^U$.
- **Closure under lubs.** this is true by construction: $\mathcal{P}_Z^U = Cl_{U,\omega}^{|Z|}(f[\mathcal{P}_X^U])$.

We can now consider the epi-mono factorization of f. Let $f_e : X \twoheadrightarrow Z$ be the function taking x to $f(x)$ (viewed as an element of $|Z|$), and $f_m : Z \rightarrowtail Y$ be the inclusion of $|Z|$ in $|Y|$. Clearly $f = f_e; f_m$, so all that remains is to show that f_e is a densely strong epi and f_m is a full mono. f_e is an epi as its underlying function is an epi in **Set**. That f_e is a densely strong epi is a direct consequence of our definition of Z's plots: we define them to be the lub-closure of $f[\mathcal{P}_X^U] = f_e[\mathcal{P}_X^U]$. That f_m is a full mono is also immediate: $f_m(x) \leq_Y f_m(y)$ if and only if $x \leq_Z y$, because $f_m(x) = x$, $f_m(y) = y$, and $\leq_Y = \leq_Z$ on their shared domain. □

Now, the final functoriality property.

Lemma 4.3.6. *In the following diagram, the morphism d making the diagram commute is unique.*

$$\begin{array}{ccc} A & \xrightarrow{\forall u} & B \\ {\scriptstyle f\in\mathcal{E}}\downarrow & \overset{\exists d}{\nearrow} & \downarrow{\scriptstyle g\in\mathcal{M}} \\ C & \xrightarrow[\forall v]{} & D \end{array}$$

Proof. By Kammar and McDermott [2018], this is equivalent to the following functoriality condition. Whenever we have a commuting square:

$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ {\scriptstyle g_1}\downarrow & & \downarrow{\scriptstyle g_2} \\ X' & \xrightarrow[f']{} & Y' \end{array}$$

Then, given epi-mono factorizations (e, m) and (e', m') of f and f' respectively, there is a unique morphism h, making the following diagram commute:

$$\begin{array}{ccccc} X & \xrightarrow{e\in\mathcal{E}} & A & \xrightarrow{m\in\mathcal{M}} & Y \\ {\scriptstyle g_1}\downarrow & & \vdots{\scriptstyle \exists h} & & {\scriptstyle g_2}\downarrow \\ X' & \xrightarrow{e'\in\mathcal{E}} & A' & \xrightarrow{m'\in\mathcal{M}} & Y' \end{array}$$

We prove this alternative formulation of the condition. The existence and uniqueness of a Scott-continuous map $h : A \to A'$ comes from the orthogonal

factorization system of full monos and dense epis in ωcpo. So we only need to show that this morphism is a PAP-morphism. Let $p \in \mathcal{P}_A^U$. Because e is a densely strong epi, such a plot is in the closure $Cl_{U,\omega}^{|A|}(e[\mathcal{P}_X^U])$. We proceed by transfinite induction on the plot p.

- Let $p = e \circ q$ for some plot $q \in \mathcal{P}_X^U$. Then, $p; h = q; e; h = q; g_1; e'$. As $g_1; e'$ is a morphism, its composition with q yields a plot.
- In the inductive case, we assume $p = \bigvee_{i\in\mathbb{N}} p_i$ for plots p_i introduced at some earlier step. By the induction hypothesis, $p_i; h$ is a plot in A'. Then, by monotonicity of h, $h(p(a)) = h(\bigvee_{i\in\mathbb{N}} p_i(a)) = \bigvee_{i\in\mathbb{N}} h(p_i(a))$, and since $p_i; h$ are plots by the induction hypothesis, so is their lub.

□

Therefore, combining Lemmas 4.3.3, 4.3.4, 4.3.5 and 4.3.6, we have shown:

Theorem 4.3.7. *The full monos and densely strong epis form an orthogonal factorization system in ωPAP.*

4.3.4 A randomization monad structure

Definition 4.3.11 ([Kammar and McDermott, 2018])**.** *A strong monad structure $\underline{T}$ over a Cartesian closed category $\mathbf{C}$ is a triple $(T, \texttt{return}, \gg\!=)$ consisting of an assignment of an object TX and a morphism $\texttt{return}_X : X \to TX$ for every object X, and an assignment of a morphism $\gg\!=_{X,Y} : TX \times (TX)^X \to TY$.*

Definition 4.3.12. *A strong monad is a strong monad structure $\underline{T}$ satisfying the monad laws, expressed in the Cartesian closed internal language as:*
$(\texttt{return}\, x) \gg\!= f = f(x) \quad a \gg\!= \texttt{return} = a \quad (a \gg\!= f) \gg\!= g = a \gg\!= (\lambda x. f(x) \gg\!= g)$

Definition 4.3.13. *We define the space of random seeds $\Omega \in \omega PAP$ as follows.*

$$\Omega = \bigsqcup_{i\in\mathbb{N}} \mathbb{R}^i$$

As $\mathbb{R}^i$ are measurable spaces, Ω inherits the structure of a measurable space. The Lebesgue measure $\lambda_{\mathbb{R}^i}$ on $\mathbb{R}^i$ extends to a measure μ on Ω given by $\mu(A) = \sum_{i\in\mathbb{N}} \lambda_{\mathbb{R}^i}(A \cap \mathbb{R}^i)$.

We will often write λ for the Lebesgue measure on $\mathbb{R}^i$, and by slight abuse of notation, this extended Lebesgue measure on Ω.

Definition 4.3.14. *We define the following functor R: $\omega PAP \to \omega PAP$, which we call a randomisation functor, by $R := \Omega \Rightarrow \mathbf{L}(-)$.*

The randomisation functor R has the following monad structure.

- $\texttt{return}_X = \lambda x. \lambda r. \mathbf{match}\, r\, \mathbf{with}\, [] \mapsto x; \_ \mapsto \bot : X \to RX$

- $\gg=_{X,Y} = \lambda\mu.\lambda k.\lambda r.\mathbf{match}\, r\, \mathbf{with}\, s : xs \mapsto \mathbf{do}_\perp \{xs_1 \leftarrow \mathbf{take}\, \lfloor s \rfloor\, xs; x \leftarrow \mu(xs_1); xs_2 \leftarrow \mathbf{drop} \lfloor s \rfloor\, xs; k(x)(xs_2)\}; \_ \to \perp : RX \times (X \Rightarrow RY) \to RY$. $\lfloor - \rfloor : \mathbb{R} \to \mathbb{N}$ is the usual floor function, and the **take** and **drop** partial functions fail on negative arguments, or if their first argument exceeds the length of their second argument. In other cases, $\mathbf{case}\, n\, xs$ returns the first n elements of the list xs, and $\mathbf{drop}\, n\, xs$ returns xs missing its first n elements.

4.3.5 An expectation operator

We first show how every ωPAP space X can be seen as a measurable. First, we show how to do it for c-analytic sets.

Lemma 4.3.8.
- *c-analytic sets are Borel sets.*
- *cPAP-morphisms are Borel measurable.*

Proof.
- Let $A = \bigcup_{i\in\mathbb{N}} A_i$ be a c-analytic set, where the A_i are analytic sets. An analytic function $f : U \to \mathbb{R}$ is continuous and so Borel measurable. $(0,\infty), (-\infty, 0]$ are measurable, and so are $f^{-1}((0,\infty)), f^{-1}((-\infty,0])$. Measurable sets are closed under finite intersection, so each A_i is measurable. Measurable sets are also closed under countable union, so A is measurable.
- Let $f : A \to B$ be a cPAP-morphism. It is in particular PAP. Let $A = \bigsqcup_{i\in\mathbb{N}} A_i$ be a c-analytic partition of A such that $(A_i, f_i)_{i\in\mathbb{N}}$ is a representation of f. The $(f_i)_{i\in\mathbb{N}}$ are analytic functions, and they are therefore continuous and thus measurable. The restriction of a measurable function to a measurable subset is measurable, so the $(f_i|_{A_i})_{i\in\mathbb{N}}$ are measurable. Measurable functions are closed under countable gluing, and so f is measurable.

□

Therefore, c-analytic sets A inherit the structure of measurable space from the Borel one of $\mathbb{R}^n$, and PAP morphisms $A \to B$ are measurable. Next, every ωPAP space X can be seen as a measurable space where the measurable sets are the finest Σ algebra such that for every c-analytic set A, and every plot $p \in \mathcal{P}_X^A$, p is measurable.

Let $\mathbb{W}$ be the set of extended non-negative real weights $([0,\infty], \mathcal{P}_{[0,\infty]}, \leq_{[0,\infty]})$ where $\mathcal{P}^A_{[0,\infty]} = \mathbf{Meas}(A, [0,\infty])$, and $\leq_{[0,\infty]}$ is the usual linear order. $\mathbf{Meas}(A, [0,\infty])$ represents the set of measurable functions from the Borel set A to the Borel set $[0,\infty]$. Note that every morphism $f \in \omega PAP(X, \mathbb{W})$ is measurable. Indeed, let $B \subseteq \mathbb{W}$ be measurable. Then, $f^{-1}(B)$ is measurable iff for every plot $p \in \mathcal{P}_X^A$, $p^{-1}(f^{-1}(B))$ is measurable. However, $p^{-1}(f^{-1}(B)) = (f \circ p)^{-1}(B)$ is the preimage of the measurable set B by the plot $f \circ p$ which is a measurable function by construction, and is therefore measurable.

A randomization $\alpha \in RX$ represents an intensional description of a measure on X by pushing forward the (extended) Lebesgue measure λ on Ω. The undefined part of α shaves some of the measure. Similarly to Vákár et al. [2019], $Dom(\alpha) := \alpha^{-1}[x \in |\mathbf{L}X| \mid x \neq \perp]$ is a c-analytic set, and for every weighting function $w : X \to \mathbb{W}$, the composition $w \circ \alpha : Dom(\alpha) \to \mathbb{W}$ is an ωPAP-morphism (and

by construction, all these functions are measurable). Let $\mathcal{J} := ((-) \Rightarrow \mathbb{W}) \Rightarrow \mathbb{W}$ be the continuation monad.

Definition 4.3.15. *We define an expectation operator* $\mathbb{E} : RX \to \mathcal{J}X$ *by*

$$\mathbb{E}_\alpha[w] := \int_{Dom(\alpha)} \lambda(dr) w(\alpha(r))$$

Proposition 4.3.9. *The expectation operator above is well-defined, an ωPAP-morphism, and a strong monad structure morphism.*

Proof. The proof follows the following three steps:

1. $\mathbb{E}$ is a well-defined function
2. $\mathbb{E}$ is an ωPAP morphism
3. $\mathbb{E}$ is a strong monad structure morphism

$\mathbb{E}$ is a well-defined function. We need to show that $\mathbb{E}_\alpha[w]$ is well-defined, and that $\mathbb{E}_\alpha \in \mathcal{J}X$, i.e. that it is an ωPAP morphism $(X \Rightarrow \mathbb{W}) \to \mathbb{W}$.

1. $\mathbb{E}_\alpha[w]$ is well-defined. Note that, by the universal property of the coproduct, we can write α as a coproduct $\alpha = [\alpha_i]_{i\in\mathbb{N}}$ where $\alpha_i : \mathbb{R}^i \to \mathbf{L}X$. Therefore, $\int_{Dom(\alpha)} \lambda(dr) w(\alpha(r)) = \sum_{i\in\mathbb{N}} \int_{Dom(\alpha_i)} \lambda(dr) w(\alpha_i(r))$. By Lemma 4.3.8, $Dom(\alpha_i)$ is a Borel set and $w \circ \alpha_i$ a Borel measurable function, so each integral $\int_{Dom(\alpha_i)} \lambda(dr) w(\alpha_i(r))$ is well-defined and valued in $[0, \infty]$, and therefore so is the countable sum of integrals. Thus, $\mathbb{E}_\alpha[w]$ is a well-defined expression.

2. $\mathbb{E}_\alpha$ is ωPAP for every random element $\alpha \in RX$. Let $\phi \in \mathcal{P}^A_{X\to\mathbb{W}}$ be a plot, i.e. a ωPAP-morphism $\phi : X \times A \to \mathbb{W}$. We need to show that $\lambda a. \int_{Dom(\alpha)} \lambda(dr) \phi(a, \alpha(r)) \in \mathcal{P}^A_{\mathbb{W}}$, i.e. is measurable. Write α as a coproduct by the universal property of coproducts: $\alpha = [\alpha_i]_{i\in\mathbb{N}}$. Let $g_i : Dom(\alpha_i) \times A \to \mathbb{W}$ be defined by $g(\omega, a) = \phi(\alpha_i(\omega), a)$. It is an ωPAP-morphism as it is the composition of the PAP morphisms ϕ and $\alpha_i \times id$. It is therefore measurable. Therefore, the question is the measurability of $a \mapsto \sum_{i\in\mathbb{N}} \int_{Dom(\alpha_i)} g(a, r)\lambda(dr)$. By the Fubini-Tonelli theorem for the σ-finite measure λ, for every $i \in \mathbb{N}$, $a \mapsto \int_{Dom(\alpha_i)} g(a, r)\lambda(dr)$ is measurable. Pointwise limits of measurable functions are measurable, and therefore $a \mapsto \sum_{i\in\mathbb{N}} \int_{Dom(\alpha_i)} g(a, r)\lambda(dr)$ is measurable.

$\mathbb{E}$ is an ωPAP morphism. Let $\phi \in \mathcal{P}^A_{RX}$, i.e. a ωPAP-morphism $\phi : A \times \Omega \to \mathbf{L}X$. We have to show that $\lambda a.\mathbb{E}_{\phi(a,-)} \in \mathcal{P}^A_{\mathcal{J}X}$, i.e. that $\lambda a.\lambda w.\mathbb{E}_{\phi(a,-)}[w] \in \omega PAP(A \times (X \to \mathbb{W}), \mathbb{W})$. Let $(\psi, w) \in \mathcal{P}^B_{A\times(X\to\mathbb{W})}$. We therefore need to show that $\lambda b.\mathbb{E}_{\phi(\psi(b),-)}[w(b)] \in \mathcal{P}^B_{\mathbb{W}}$, i.e. is measurable.

$$\lambda b.\mathbb{E}_{\phi(\psi(b),-)}[w(b)] = \lambda b. \int_{Dom(\phi(\psi(b),-))} \lambda(dr) w(b, \phi(\psi(b), r))$$

The only change from the argument used in showing that $\mathbb{E}_\alpha$ is ωPAP is that now the integration bounds can also depend on the external parameter b. $\phi(\psi(-), =) : B \times$

$\Omega \to \mathbf{L}X$ is a partial function, so its domain is of the form $\bigsqcup_{i\in\mathbb{N}} B_i \times C_i$ for c-analytic sets $B_i \subseteq B, C_i \subseteq \mathbb{R}^i$. For each $i \in \mathbb{N}$, $\lambda b : B_i. \int_{C_i} \lambda(dr) w(b, \phi(\psi(b), r))$ is measurable by the argument in the previous point ($\mathbb{E}_\alpha$ is ωPAP). Measurable functions are stable under countable gluing, and therefore $\lambda b. \int_{Dom(\phi(\psi(b),-))} \lambda(dr) w(b, \phi(\psi(b), r))$ is measurable.

$\mathbb{E}$ is a strong monad structure morphism. We need to show that $\mathbb{E}$ preserves return and bind, and that $\mathbb{E}$ is a natural transformation.

- $\mathbb{E}_{return} = \lambda x.\lambda w. \int_{\mathbb{R}^0} \lambda_{\mathbb{R}^0}(dr) w(x) = \lambda x.\delta_x = return$, where $w(x)$ appears by using the definition of $return_X$ and δ is the dirac measure.

- The question is whether the following equality holds: $\lambda\alpha : RX.\lambda f : X \to RY.\ \mathbb{E}_\alpha \gg\!= (\lambda x.\mathbb{E}_{f(x)}) = \lambda\alpha : RX.\lambda f : X \to RY.\ \mathbb{E}_{(\alpha \gg\!= f)}$. Expanding the LHS and RHS, we get, for all $\alpha : RX, f : X \to RY, w : Y \to \mathbb{W}$:

$$
\begin{aligned}
&(\mathbb{E}_\alpha \gg\!= (\lambda x.\mathbb{E}_{f(x)}))[w] \\
&= \mathbb{E}_\alpha[\lambda x.\mathbb{E}_{f(x)}[w]] \\
&= \int_{Dom(\alpha)} \lambda(dr) \int_{Dom(f(\alpha(r)))} \lambda(dr') w(f(\alpha(r))(r')) \\
&= \sum_{i\in\mathbb{N}} \sum_{j\in\mathbb{N}} \int_{\mathbb{R}^i \cap Dom(\alpha)} \lambda(dr) \int_{\mathbb{R}^j \cap Dom(f(\alpha(r)))} \lambda(dr') w(f(\alpha(r))(r')) \\
&= \sum_{n\in\mathbb{N}} \sum_{p\in\mathbb{N}} \int_{\mathbb{R}^p \cap Dom(\alpha)} \lambda(dr) \int_{\mathbb{R}^{n-p} \cap Dom(f(\alpha(r)))} \lambda(dr') w(f(\alpha(r))(r')) \\
&= \sum_{n\in\mathbb{N}} \int_{\mathbb{R}} \lambda(ds).1_{0\le s\le n} \int_{\mathbb{R}^{\lfloor s\rfloor} \cap Dom(\alpha)} \lambda(dr_1) \int_{\mathbb{R}^{n-\lfloor s\rfloor} \cap Dom(f(\alpha(r_1)))} \lambda(dr_2) w\Big(f(\alpha(r_1))(r_2)\Big) \\
&= \sum_{n\in\mathbb{N}} \int_{\mathbb{R}^{n+1} \cap Dom(\alpha \gg\!= f)} \lambda(dr).\Big(\mathbf{match}\, r\, \mathbf{with}\, s : xs \mapsto xs_1 = \mathbf{take}\, \lfloor s\rfloor\, xs; \\
&\qquad\qquad xs_2 = \mathbf{drop} \lfloor s\rfloor\, xs; w\Big(f(\alpha(xs_1))(xs_2)\Big)\Big) \\
&= \int_{Dom(\alpha \gg\!= f)} \lambda(dr).\Big(\mathbf{match}\, r\, \mathbf{with}\, s : xs \mapsto xs_1 = \mathbf{take}\, \lfloor s\rfloor\, xs; \\
&\qquad\qquad xs_2 = \mathbf{drop} \lfloor s\rfloor\, xs; w\Big(f(\alpha(xs_1))(xs_2)\Big)\Big) \\
&= \int_{Dom(\alpha \gg\!= f)} \lambda(dr).\Big(\mathbf{match}\, r\, \mathbf{with}\, s : xs \mapsto \mathbf{do}_\perp \{xs_1 \leftarrow \mathbf{take}\, \lfloor s\rfloor\, xs; \\
&\qquad\qquad x \leftarrow \alpha(xs_1); xs_2 \leftarrow \mathbf{drop} \lfloor s\rfloor\, xs; w(f(x)(xs_2))\}\Big) \\
&= \mathbb{E}_{\lambda r.\mathbf{match}\, r\, \mathbf{with}\, s:xs\mapsto \mathbf{do}_\perp \{xs_1\leftarrow \mathbf{take}\, \lfloor s\rfloor\, xs; x\leftarrow \alpha(xs_1); xs_2\leftarrow \mathbf{drop} \lfloor s\rfloor\, xs; f(x)(xs_2)\}; \_\to\perp}[w] \\
&= \mathbb{E}_{(\alpha \gg\!= f)}[w]
\end{aligned}
$$

where we used, in order: the definition of $\gg\!=^{\mathcal{J}}$, the definition of $\mathbb{E}$ twice, splitting the domain of α and $f(\alpha(r))$ as well as Fubini-Tonelli to swap sum and integral, the bijective change of variable $(n, p) = (i + j, i)$, express the sum on p as an integral on s, refactor the triple integral as one, the fact that $\mathbb{R}^0$ is not part of $Dom(\alpha \gg\!= f)$, un-splitting the domain of $Dom(\alpha \gg\!= f)$,

adding monadic composition (which is fine as we are restricted to the domain of $Dom(\alpha \gg\!= f)$), the definition of $\mathbb{E}$, the definition of $\gg\!=^R$.

- For every morphism $f : X \to Y$, we have

- For every morphism $f : X \to Y$, we have

$$\begin{aligned}
\mathbb{E}_{Rf} &= \lambda\alpha.\lambda w.\mathbb{E}_{f\circ\alpha}[w] \quad \text{def. } Rf \\
&= \lambda\alpha.\lambda w.\int_{Dom(f\circ\alpha)} \lambda(dr) w \circ (f \circ \alpha)(r) \quad \text{def. } \mathbb{E} \\
&= \lambda\alpha.\lambda w.\int_{Dom(\alpha)} \lambda(dr)(w \circ f\circ)\alpha(r) \quad f \text{ total, } \circ \text{ associative} \\
&= \lambda\alpha.\lambda w.\mathbb{E}_{\alpha}[w \circ f] \quad \text{def. } \mathbb{E} \\
&= \mathcal{J}f \circ \mathbb{E} \quad \text{def. } \mathcal{J}f
\end{aligned}$$

□

We have constructed a monad structure of randomisable elements, and defined an expectation operator $\mathbb{E}$. We now we need to show that the image of $\mathbb{E}$ in $\mathcal{J}X$, which we denote by TX, defines a commutative monad T.

4.3.6 A ωPAP monad of measures

Besides the orthogonal factorization system, we also need to prove the following result, in order to show that the factorization system is closed under the randomization monad structure and apply Theorem 4.3.1.

Proposition 4.3.10. *The densely strong epis are closed under*

- *countable products: for every countable $I \subseteq \mathbb{N}$ and every I-indexed family $(e_i : A_i \twoheadrightarrow B_i)_{i\in I}$ of densely strong epis, their Cartesian product is also a densely strong epi.*

- *the lifting monad $\boldsymbol{L}$: if $e : A \twoheadrightarrow B$ is a densely strong epi, then $\mathbf{L}(e) : \mathbf{L}A \twoheadrightarrow \mathbf{L}B$ is also a densely strong epi*

- *exponentiation with U: if $e : A \twoheadrightarrow B$ is a densely strong epi, then for every representable U, the exponential $e^U : A^U \twoheadrightarrow B^U$ is also a densely strong epi.*

Proof. 1. **Closure under countable products.** Let $e_i : X_i \to Y_i$ be densely strong epis. We need to show that $\prod_{i\in\mathbb{N}} e_i : \prod_{i\in\mathbb{N}} X_i \to \prod_{i\in\mathbb{N}} Y_i$ is a densely strong epi, i.e. that for every family of plots $(p_i : U \to Y_i)_{i\in\mathbb{N}} \in \mathcal{P}^U_{\prod_{i\in\mathbb{N}} Y_i}$ arises as the image of a plot in $\mathcal{P}^U_{\prod_{i\in\mathbb{N}} X_i}$ or as a lub of such plots. It is known that lubs in ωcpo commute with countable products, i.e. $\bigvee_{i\in\mathbb{N}} \prod_{j\in\mathbb{N}}(p^i_j) = \prod_{j\in\mathbb{N}} \bigvee_{i\in\mathbb{N}}(p^i_j)$.

Therefore, if for each $j \in \mathbb{N}$, the plot p_j is a limit of plots p_j^i which are in the image of e_j, i,e, $p_j = \bigvee_{i\in\mathbb{N}} p_j^i$, then

$$p = \prod_{j\in\mathbb{N}} p_j = \prod_{j\in\mathbb{N}} \bigvee_{i\in\mathbb{N}} p_j^i = \bigvee_{i\in\mathbb{N}} \prod_{j\in\mathbb{N}} (p_j^i)$$

and therefore p is the lub of $(\prod_{j\in\mathbb{N}}(p_j^i))_{i\in\mathbb{N}}$, and each of which is in the image of $\prod_{i\in\mathbb{N}} e_i$ by assumption on the p_j^i.

2. **Closure under the lifting monad L.** Given a densely strong epi $e : X \to Y$, recall that we have $\mathbf{L}e : \mathbf{L}X \to \mathbf{L}Y = \lambda[\uparrow x.e(x), \bot.\bot]$. Every plot $p : U \to \mathbf{L}Y$ of $\mathbf{L}Y$ corresponds to some plot $p : V \subseteq U \to Y$ in Y. In addition, by the densely strong epi property of e, every plot $p : V \to Y$ of Y arises either as the image under e of some plot in X, or as a lub of 'simpler' plots in Y. That is every plot p of Y can be written as $p = \bigvee_{i\in\mathbb{N}} e \circ q_i$ where the q_i are plots in X. By transfinite induction, we show that plots in $\mathbf{L}Y$ are images of lubs of images of plots of $\mathbf{L}X$ by $\mathbf{L}e$:

   - If p is the image of some plot in $\mathcal{P}_X^U$ under e, then for all c-analytic sets V containing U, p is also a plot in $\mathcal{P}_{\mathbf{L}Y}^V$ and arises as the image under $\mathbf{L}e$ of some plot in $\mathcal{P}_{\mathbf{L}X}^V$ with the same domain U.
   - Let $p = \bigvee_{i\in\mathbb{N}} p_i$ where the plots $(p_i)_{i\in\mathbb{N}}$ in $\mathcal{P}_Y^U$ are such that the property already holds for each p_i by the induction hypothesis. Then, for any V containing U, p_i are plots in $\mathcal{P}_{\mathbf{L}Y}^V$ whose lub in $\mathcal{P}_{\mathbf{L}Y}^V$ is p, which is therefore in the closure of $\mathbf{L}e$'s image.

3. **Closure under exponentiation with representables.** Recall that given a densely strong epi $e : X \to Y$, and a representable U, we have $U \Rightarrow e : (U \to X) \to (U \to Y) = \lambda g.\lambda r.e(g(r))$. Now consider a plot in $U \to Y$, i.e., the currying of some morphism $p : V \times U \to Y$. We need to show that it is of the form $\lambda(v, u).e(g(v, u))$ for some $g : V \times U \to X$, or in the lub closure of such morphisms. As e is a densely strong epi, the plots in $\mathcal{P}_Y^W$ are in the image $e[\mathcal{P}_X^W]$, or lubs of them. This is in particular true for $W = U \times V$, and we are done.

□

Theorem 4.3.11. *T defines a locally continuous commutative submonad of $\mathcal{J}$.*

Proof. By Theorem 4.3.7, we have an orthogonal factorization system on ωPAP. By Proposition 4.3.10, if e_i are densely strong epis, then so are $\mathbf{L}(e), \mathbb{R}^i \Rightarrow e$ and $\prod_i i \in \mathbb{N} e_i$. Therefore, so is $R(e) = \bigcup_{i\in\mathbb{N}} \mathbb{R}^i \Rightarrow \mathbf{L}e$. By Section 4.3.4, R is a monad structure, and the continuation monad $\mathcal{J}$ is known to be a strong-monad. Furthermore, by Proposition 4.3.9, $\mathbb{E}$ is a strong monad structure morphism. Therefore, by Theorem 4.3.1, the image of $\mathbb{E}$ induces a strong monad T on ωPAP. Similarly to Vákár et al. [2019],

For commutativity, the proof is the same as for the case in Vákár et al. [2019], and essentially comes from the fact that our measures are s-finite, and thus enjoy a Fubini theorem [Staton, 2017]. In more detail, the following argument is a simple

adaptation of the proof of the same result in Vákár et al. [2019], whose credit therefore goes to these authors.

$$C_{X,Y} := \{\langle \mu, \nu\rangle \in TX \times TY \mid \mu \gg\!= \lambda x.\nu \gg\!= \lambda y.\mathtt{return}(x,y) = \nu \gg\!= \lambda y.\mu \gg\!= \lambda x.\mathtt{return}(x,y)\}$$

Note that $C_{X,Y}$ is closed under suprema of ω-chains. Indeed, let $\mu \in TX$ and $\nu \in TY$. Then, there exist ω-chains $\mu_n \in RX$ and $\nu_m \in RY$ such that $\mu = \sup_n \mu_n$ and $\nu = \sup_m \nu_m$. Then,

$$\begin{aligned}
&\mu \gg\!= \lambda x.\nu \gg\!= \lambda y.\mathtt{return}(x,y)\\
&= (\sup_n \mu_n) \gg\!= \lambda x.(\sup_m \nu_m) \gg\!= \lambda y.\mathtt{return}(x,y)\\
&= \sup_n \sup_m (\mu_n \gg\!= \lambda x.\nu_m \gg\!= \lambda y.\mathtt{return}(x,y)) \quad \text{by continuity of bind}\\
&= \sup_n \sup_m (\nu_m \gg\!= \lambda y.\mu_n \gg\!= \lambda x.\mathtt{return}(x,y)) \quad \text{by our assumptions}\\
&= (\sup_m \nu_m) \gg\!= \lambda y.(\sup_n \mu_n) \gg\!= \lambda x.\mathtt{return}(x,y) \quad \text{by continuity of bind}\\
&= \nu \gg\!= \lambda y.\mu \gg\!= \lambda x.\mathtt{return}(x,y).
\end{aligned}$$

Therefore, it follows that T is a commutative monad if we can show that $RX \times RY \subseteq C_{X,Y}$ for all ωPAP spaces X, Y.

Let $\mu \in RX$ and $\nu \in RY$. There exists ωPAP morphims f, g such that $\mu = f_*\lambda$ and $\nu = g_*\lambda$. By the universal property of coproducts, there exists ωPAP-morphisms f_i, g_i such that $\mu = \sum_{i\in\mathbb{N}} f_{i,*}\lambda$ and $\nu = \sum_{i\in\mathbb{N}} g_{i,*}\lambda$. Then, for every measurable $h : X \times Y \to \mathbb{W}$, we have

$$\begin{aligned}
&\mu \gg\!= \lambda x.\nu \gg\!= \lambda y.h(x,y)\\
&= \int_X f_*\lambda(dx) \int_Y g_*\lambda(dy) h(x,y) \quad \text{def. bind}\\
&= \int_\Omega \lambda(dr) \int_\Omega \lambda(dr') h(f(r), g(r')) \quad \text{property pushforward measure}\\
&= \sum_{i\in\mathbb{N}} \int_{\mathbb{R}^i} \lambda(dr) \sum_{j\in\mathbb{N}} \int_{\mathbb{R}^j} \lambda(dr') h(f_i(r), g_j(r')) \quad \text{def } \mu, \nu \text{ above}\\
&= \sum_{i\in\mathbb{N}} \sum_{j\in\mathbb{N}} \int_{\mathbb{R}^i} \lambda(dr) \int_{\mathbb{R}^j} \lambda(dr') h(f_i(r), g_j(r')) \quad \text{Lebesgue integral of function} \geq 0\\
&= \sum_{i\in\mathbb{N}} \sum_{j\in\mathbb{N}} \int_{\mathbb{R}^j} \int_{\mathbb{R}^i} \lambda(dr)\lambda(dr') h(f_i(r), g_j(r')) \quad \text{Fubini's theorem for } \sigma\text{-finite measures}\\
&= \sum_{i\in\mathbb{N}} \int_{\mathbb{R}^j} \lambda(dr') \sum_{j\in\mathbb{N}} \int_{\mathbb{R}^i} \lambda(dr) h(f_i(r), g_j(r')) \quad \text{Lebesgue integral of function} \geq 0\\
&= \int_\Omega \lambda(dr') \int_\Omega \lambda(dr) h(f(r), g(r')) \quad \text{def } \mu, \nu\\
&\nu \gg\!= \lambda x.\mu \gg\!= \lambda y.h(x,y) \quad \text{def. bind}
\end{aligned}$$

□

Remark 4.3.1. *It might be confusing that we call monad of measures our monad of expectation operators. However, on $\mathbb{R}^n$ and c-analytic subsets A of $\mathbb{R}^n$ (and in particular on submanifolds of $\mathbb{R}^n$, see Lemma 4.3.12 below), every element $\mathbb{E}_\alpha \in TA$ defines a linear map on the σ-compact topological space A. Therefore, by the Riesz representation theorem ([Kowalski, 2011], Theorem 5.2.1), every such expectation operator comes from a unique measure μ on the measurable space A. In this case, it is therefore justified to call measures our expectation operators, implicitly applying the bijection $\mu \mapsto \lambda f. \int f d\mu$ when necessary.*

Lemma 4.3.12. *Every manifold $M \subseteq \mathbb{R}^n$ is a c-analytic subset of $\mathbb{R}^n$.*

Proof. As is recalled in the proof of Theorem 4.4.19, local charts $(\phi_i)_i$ of an embedded d-dimensional smooth manifold $M \subseteq \mathbb{R}^n$ can be chosen to be analytic functions $\phi_i : V_i \subseteq M \to \mathbb{R}^n$. As $\mathbb{R}^d$ is an analytic set, we deduce that each $V_i \overset{\phi_i}{\cong} \mathbb{R}^d$ is an analytic set, and as a countable family of charts $\{V_i\}_{i\in\mathbb{N}}$ covers the manifold M, $M = \bigcup_{i\in\mathbb{N}} V_i$ is c-analytic. □

4.4 Differentiability and measures: pushforward of PAP functions

Probabilistic programming languages enable the representation of probability distributions and general measures by adding support for random sampling and soft conditioning to deterministic languages. These languages are often used to model real-world phenomena, and questions of interest can be framed in terms of expectations under these models. Inference algorithms are typically used to estimate posterior distributions, and many probabilistic programming languages offer automated tools to run these algorithms. However, these tools typically assume certain properties of the programs written by the user, such as having a density with respect to a well-behaved reference measure, typically the Lebesgue measure, or the density being differentiable.

We study verifying properties that hold for all programs in expressive probabilistic languages. We introduce a monad on ωPAP that interprets closed programs as s-finite measures Staton [2017], following the approach of Vákár et al. [2019]. By interpreting probabilistic programs as ωPAP measures, we prove that all closed programs of type $\mathbb{R}^n$ denote distributions supported on a countable union of smooth submanifolds of $\mathbb{R}^n$. Consequently, these programs admit convenient densities with respect to a particular class of reference measures (Theorem 4.4.18). This result demonstrates the extensional properties of the measures that probabilistic programs represent and provides a foundation for reasoning about these programs.

The result is relevant for the automatic computation of densities or Radon-Nikodym derivatives of probabilistic programs, key ingredients in higher-level algorithms such as importance sampling and MCMC. Suppose a user has constructed two closed probabilistic programs of type $\mathbf{real}^n$, p and q, and wishes to use q as an importance sampling proposal for p. If $[\![p]\!]$ is absolutely continuous with respect to $[\![q]\!]$ ($[\![p]\!] \ll [\![q]\!]$), such an importance sampler does exist, but implementing it

requires computing importance weights: this means that at a point $x \sim [\![q]\!]$, we must compute $\frac{d[\![p]\!]}{d[\![q]\!]}(x)$. Now if $[\![p]\!]$ and $[\![q]\!]$ are both absolutely continuous with respect to the Lebesgue measure on $\mathbb{R}^n$, we can compute probability density functions ρ_p and ρ_q for $[\![p]\!]$ and $[\![q]\!]$ respectively, and then compute the importance weight as the ratio of these densities. However, not all programs denote measures that have densities with respect to the Lebesgue measure. Indeed, consider, the program that samples $u \sim \mathit{Unif}(0,1)$, and returns $(u,u) \in \mathbb{R}^2$. Because it is supported only on a 1-dimensional line segment within the plane, it has no density function with respect to the Lebesgue measure in the plane. That is, there is no nonnegative function ρ such that the probability of an event $E \subseteq \mathbb{R}^2$ is equal to $\iint_{\mathbb{R}^2} 1_E(x,y) \cdot \rho(x,y)dxdy$.

Our result gives us an alternative to the Lebesgue measure: we can compute the density of $[\![p]\!]$ (and $[\![q]\!]$) with respect to the Hausdorff measures over the manifolds on which they are supported. Computing densities with respect to Hausdorff measures has previously been proposed by Radul and Alexeev [2021]. Our result is that, unlike the Lebesgue base measure, this choice does not *exclude* any possible programs a user might write, because *every* program is absolutely continuous with respect to such a base measure.

We first briefly recall the setting of probabilistic programming (Section 4.4.1). Then, we recall some background in differential geometry that will be needed to phrase the theorem and key in the proofs (Section 4.4.2). Following this, we prove Theorem 4.4.15 in Section 4.4.5. It gives a characterization of the pushforward of the Lebesgue measure on $\mathbb{R}^n$ by PAP functions. They represent measures with a density w.r.t. some well-behaved class of measures, which we call s-Hausdorff. They represent a natural generalization of the Lebesgue measure on curved spaces. Finally, we prove the main theorem of this section (Theorem 4.4.18) in Section 4.4.6 about the posterior distributions of closed probabilistic programs.

4.4.1 Probabilistic programming

To reason about probabilistic programs, we extend our core calculus (Figure 3.3) with the standard effectful operations exposed by probabilistic programming languages (PPLs): `sample` samples a real from the uniform distribution on $(0,1)$, and `score` r reweights program traces (to implement soft constraints). See Staton [2017] for an introduction, and similar design is underlying many PPLs Cusumano-Towner et al. [2019], Carpenter et al. [2015], Goodman and Stuhlmüller [2014], Goodman et al. [2012], Dash et al. [2022] Their typing rules are given in Figure 4.3.

$$\frac{\Gamma \vdash e : \mathbf{real}}{\Gamma \vdash \texttt{score}\ e : 1} \qquad \frac{}{\Gamma \vdash \texttt{sample} : \mathbf{real}}$$

Figure 4.3: Type system of the probabilistic part of the language

A standard semantics for first-order probabilistic programs is as unnormalized probabilistic kernels. Programs are interpreted as functions sending values for variables in the context to measures, where the measure represents the posterior

distribution of possible return values of the program. For a higher-order recursive language such as ours, a measure-based semantics is given by Vákár et al. [2019] in the category of ω-Quasi-Borel spaces. They are an analogue of ωPAP where the plots are obtained from measurable functions instead of PAP functions.

We revise the interpretation of the deterministic part (Figure 3.3) of the language, replacing the partiality monad $\mathbf{L}$ by our monad of measures T. The semantics of types is mostly unchanged, except $[\![\tau \to \sigma]\!] := \omega PAP([\![\tau]\!], T[\![\sigma]\!])$. A term $\Gamma \vdash t : \tau$ is now interpreted as a morphism $[\![\Gamma]\!] \to T[\![\tau]\!]$. The interpretation of terms using `return` and $\gg=$ are now those of the monad T. We need to adapt the semantics of recursion as $[\![\mu f.\lambda x.t]\!](\rho) = \mathbf{return}\ \bigvee_{i\in\mathbb{N}} f_i$, where $f_0 = \lambda x.\lambda r.\mathbf{inr}\ \bot, f_{i+1} = \lambda v.[\![t]\!](\rho[f \mapsto f_i, x \mapsto v])$. The only other change is in the semantics of primitive operations representing partial functions $f : \mathbf{real}^n \rightharpoonup \mathbf{real}$. In this case, if $Dom(f) = A \subseteq \mathbb{R}^n$, then $[\![f]\!] : \mathbb{R}^n \to T\mathbb{R}$ is given by $\lambda x : \mathbb{R}^n.(\mathbf{if}\ x \in A\ \mathbf{then}\ [\![\texttt{score}\ 0]\!]\ \mathbf{else}\ [\![\texttt{score}\ 1]\!]); \texttt{return}\ f(x)$, where ; is the Kleisli composition, and we now define $[\![\texttt{score}]\!]$. The semantics of the new constructs `score` and `sample` are given respectively by

$$[\![\texttt{score}\ t]\!](\rho) = \lambda f.[\![t]\!](\rho)(\lambda w.\max(0,w)\cdot f(\star))$$
$$[\![\texttt{sample}]\!](\rho) = \lambda f.\int_{[0,1]} f(x)dx$$

where $\star$ is the only element of the terminal object 1.

Staton [2017] showed that closed first-order programs do not denote arbitrarily measures, but only s-finite ones. An s-finite measure is a countable sum of finite measures, and they enjoy many desirable properties. For instance, a version of Fubini's theorem holds for s-finite measures, giving a semantic justification for the reordering of two independent probabilistic computations. We will show that if we further restrict the primitives to be PAP, instead of merely measurable, we can prove a much stronger restriction on the class of definable measures.

The proof of Theorem 4.4.15 is fairly technical, and requires more concepts from analysis and differential geometry, which we will recall in Section 4.4.2. In Section 4.4.3, we define s-Hausdorff measures and give some basic properties on them. Finally, in Section 4.4.5 we will introduce and prove the necessary lemmas building up to Theorem 4.4.15.

4.4.2 Background: differential geometry

In this section, we quickly recall some basic facts about topology, smooth manifolds, Sard's theorem, and the co-area formula from differential geometry.

4.4.2.1 General Topology

A general topological space can be extremely poorly behaved, and look nothing like our intuitive geometric notion of space. A fairly common restriction is to ask for the space to be Hausdorff. See e.g. Kelley [2017] for a general reference on topology.

Definition 4.4.1. *A Hausdorff space X is a topological space where for any two distinct points $x, y \in X$, there exist neighborhoods U_x, U_y of x and y respectively, such that $U_x \cap U_y = \emptyset$.*

A classic counterexample to Hausdorff-spaces is a real line with 2 distinct points, both representing 0. There are several useful ways to quantify whether a topological space is made of 'one block'.

Definition 4.4.2. *A topological space is connected if it cannot be written as a disjoint union of two or more open subsets.*

A stronger notion of connectedness is path-connected:

Definition 4.4.3. *A topological space X is path-connected if for every pair of points $x, y \in X$, there exists a continuous function $c : [0, 1] \to X$ such that $c(0) = x$ and $c(1) = y$.*

It is well-known that a path-connected space is connected, but that the reciprocal is not true in general. A classic example of a connected yet not path-connected space is the curve of $\sin(\frac{1}{x})$ to which one adjoins the vertical closed interval $(-1, 1)$ centered on 0. This interval is connected to the rest of the curve, but any path from a point on the rest of the curve to this interval will have infinite length, and so cannot be the image of a continuous function from the closed interval $(0, 1)$ to this space. However, the converse is true in the case of subspaces of Euclidean spaces.

Proposition 4.4.1. *If U is an open and connected subset of $\mathbb{R}^n$ (with the usual topology), then U is path-connected.*

Likewise, certain topological are in the sense too big for our geometric intuitions to usually carry to such settings. Separable spaces are a restriction making topological spaces better behaved.

Definition 4.4.4. *A topological space is separable if it contains a countable, dense subset.*

Euclidean spaces and smooth manifolds (Section 4.4.2.2) are separable. For instance, rational points $\mathbb{Q}^n \subset \mathbb{R}^n$ form a dense subset of $\mathbb{R}^n$. The space of bounded continuous functions $\mathbb{R} \to \mathbb{R}$, on the other hand, is not separable. To see this, there is an uncountable subset I of such functions such that for all $f, g \in I$, $\|f - g\| \geq 1$ whenever $f \neq g$. Therefore, given a countable subset C of the set of bounded continuous functions $\mathbb{R} \to \mathbb{R}$, there is a function $f \in I$ that is at distance at least $\frac{1}{2}$ from every element of C, and therefore C is not dense in the set of bounded continuous functions $\mathbb{R} \to \mathbb{R}$. This used a standard technique to show that a space is not separable: showing that it has a uncountable set of disjoint opens.

Definition 4.4.5. *A topological space X is second-countable when its topology has a countable base. More precisely, there exists a countable set $\{U_i\}_{i \in \mathbb{N}}$ of opens U_i of X such that every open of X can be written as a union of a subfamily of $\{U_i\}_{i \in \mathbb{N}}$.*

Proposition 4.4.2. *A second-countable space is separable.*

Finally, Lindelöf spaces have a particularly nice property that will be useful in some of the proofs.

Definition 4.4.6. *A topological space is a Lindelöf space when every open cover has a countable subcover.*

Proposition 4.4.3. *A separable space is Lindelöf.*

Corollary 4.4.3.1. *Open subsets of $\mathbb{R}^n$ are Lindelöf.*

4.4.2.2 Smooth manifolds

Smooth manifolds (Section 2.5.3) are a well-studied notion of curved spaces that generalize Euclidean spaces. Common examples include lines, spheres, and tori, and are closed under finite products or countable unions. They will form the basis for the support of our base measures.

Definition 4.4.7. *A smooth manifold M is a second-countable Hausdorff topological space together with a smooth atlas: an open cover $\mathcal{U}$ together with homeomorphisms $(\phi_U : U \to \mathbb{R}^n)_{U\in\mathcal{U}}$ called charts such that $\phi_V \circ \phi_U^{-1}$ is smooth on its domain of definition for all $U, V \in \mathcal{U}$. A function $f : M \to N$ between manifolds is smooth if $\phi_V \circ f \circ \phi_U^{-1}$ is smooth for all charts ϕ_U, ϕ_V of M and N.*

As we have justified in the previous section, Hausdorff second-countable spaces will behave more like the intuitive notion of space we have from elementary geometry. A smooth-manifold is such a space that moreover locally looks locally like a Euclidean space. An obvious example is a sphere, that looks locally like a plane, when one zooms in enough. When there exists a global bound K on the local dimensions n of a smooth manifold, it is well known by a theorem of Whitney ([Whitney, 2012]) that the manifold can be embedded as a submanifold of a Euclidean space, that is its topology is given by restricting the standard one from the surrounding Euclidean space.

4.4.2.3 Hausdorff measure

There is a very natural distribution on smooth manifolds that generalizes the Lebesgue measure on Euclidean spaces and matches our intuitive notions of area and volume for curved spaces.

Definition 4.4.8. *The k-Hausdorff measure of a metric space (X, d) is defined as*

$$H^k(S) := \lim_{\delta\to 0} \inf \left\{ \sum_i \operatorname{diam}(U_i)^k \mid S \subseteq \bigcup_i U_i, \operatorname{diam}(U_i) < \delta \right\}$$

where $\operatorname{diam}(U) := \sup \{d(x, y) \mid x, y \in U\}$ *and* $\operatorname{diam}(\emptyset) := 0$.

Intuitively, it means that the d-dimensional volume of S is defined as the smallest d-dimensional volume of a ball-cover of S.

Definition 4.4.9. *The Hausdorff dimension is then defined as* $\dim_{Haus}(S) := \inf\{k \mid H^k(S) = 0\}$.

The Hausdorff measure computes the sizes of sets S in a dimension-dependent way:

- The 0-Hausdorff measure counts the points in S.
- The 1-Hausdorff measure sums the lengths of curves within a set. It will be infinite on surfaces, and 0 on sets of isolated points.
- The 2-dimensional Hausdorff measure sums the areas of surfaces, and will be infinite on volumes and 0 on curves. More generally, the k-Hausdorff measure will quantify k-dimensional volumes.

For smooth manifolds, the Hausdorff dimension matches the intuitive notion of dimension, e.g. a sphere in $\mathbb{R}^3$ has Hausdorff dimension 2 , a path on such a sphere will have Hausdorff dimension 1, and an open of $\mathbb{R}^n$ has dimension n.

4.4.2.4 Sard's theorem

We recall the Morse–Sard theorem [Morse, 1939, Sard, 1942], a standard result in analysis.

Definition 4.4.10. *Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be differentiable. The critical set X of f is defined as $X := \{x \in \mathbb{R}^n \mid \operatorname{rank}(J_x f) < m\}$.*

Theorem 4.4.4 ([Sard, 1942])**.** *Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be C^k, where $k \geq \max\{n-m+, 1\}$. Let $X \subseteq \mathbb{R}^n$ be the critical set of f. Then the image $f(X)$ has Lebesgue measure 0 in $\mathbb{R}^m$.*

We will use a generalization of this theorem:

Theorem 4.4.5 ([Sard, 1965])**.** *Let $f : N \to M$ be C^k between smooth manifolds of dimensions n and m respectively, where $k \geq \max\{n - m+, 1\}$. Let $X \subseteq \mathbb{R}^n$ be the critical set of f. Then the image $f(X)$ has m-Hausdorff measure 0 in M.*

4.4.2.5 Area and Co-area formulas

The area formula generalizes the well-known change-of-variable formula for a diffeomorphism between opens of Euclidean spaces, and the co-area formula generalizes the Fubini-Tonelli theorem. The general setting in which they can be defined is in the context of geometric measure theory. Just as the Hausdorff measure on manifolds generalizes the Lebesgue measure on Euclidean spaces, this theory extends to an even slightly more general setting, based on rectifiable sets (Definition 4.4.11). We recall some basic definitions and facts of the theory. We will not quite require the full power and generality of this theory, but this version will be convenient for us. See Simon [1983] for an introduction to geometric measure theory.

Definition 4.4.11. *A set $M \subseteq \mathbb{R}^{n+k}$ is said to be countably n-rectifiable if $M \subseteq M_0 \cup (\bigcup_{j=1}^{\infty} F_j(\mathbb{R}^n))$ where $H^n(M_0) = 0$ and $F_j : \mathbb{R}^n \to \mathbb{R}^{n+k}$ are Lipschitz functions.*

There are several equivalent characterisations, we give 2 additional ones.

Lemma 4.4.6.
- *M is countably n-rectifiable iff $M = M_0 \bigcup (\bigcup_{j=1}^{\infty} F_j(A_j^n))$ where $H^n(M_0) = 0$ and $F_j : A_j \to \mathbb{R}^{n+k}$ are Lipschitz, $A_j \subseteq \mathbb{R}^n$.*
- *M is countably n-rectifiable iff $M \subseteq \bigcup_{j=0}^{\infty} N_j$ where $H^n(N_0) = 0$ and $N_j, j \geq 1$ are n-dimensional embedded C^1 submanifolds of $\mathbb{R}^{n+k}$.*

We can define the gradient of functions defined on countably n-rectifiable sets.

Definition 4.4.12. *Let $f : U \to \mathbb{R}^m$ be a locally Lipschitz function, where $U \subseteq \mathbb{R}^{n+k}$ is an open containing the n-rectifiable set M. We define the gradient of f, $\nabla^M f$, H^n-a.e. on M according to $\nabla^M f(x) = \nabla^{N_j} f(x)$ whenever $x \in N_j$ and $f|_{N_j}$ is differentiable (which is true H^n-a.e. by Rademacher's theorem and N_j being C^1).*

This means $\nabla^M f(x) \in T_x M$ for H^n-a.e. $x \in M$, and is up a H^n-measure zero in M, independent of the decomposition $\bigcup_{j=0}^{\infty} N_j$ used. We have not defined formally defined the generalized tangent space for n-rectifiable sets, but suffices to know it agrees H^n-a.e. with the one on the manifolds N_j. Similarly, for the differential at x $d^M f_x : T_x M \to \mathbb{R}^m$.

When $f : \mathbb{R}^n \to \mathbb{R}^n$ is differentiable, we have the following Jacobian term Jf in the change of variable formula:

$$Jf(x) = |\det df_x|$$

where $df_x : \mathbb{R}^n \to \mathbb{R}^n$ is the linear function for the derivative of f at x. This formula is only valid when the input and output spaces have the same dimension n, but it admits the following generalizations:

Definition 4.4.13. *If $n > m$, we can define the Jacobian $J_M f(x)$ for H^n-a.e. $x \in M$ as*

$$J_M f(x) = \sqrt{\det(d^M f_x^* \circ d^M f_x)}$$

where $d^M f_x^* : \mathbb{R}^m \to T_x M$ is the adjoint of $d^M f_x$.

Definition 4.4.14. *Similarly if $n < m$, we can define the Jacobian $J_M f(x)$ for H^n-a.e. $x \in M$ as*

$$J_M f(x) = \sqrt{\det(d^M f_x \circ d^M f_x^*)}$$

Theorem 4.4.7 (Area formula)**.**

$$\int_A J_M f \; dH^n = \int_{\mathbb{R}^n} H^0(A \cap f^{-1}(y)) dH^n(y)$$

for any H^n-measurable set $A \subseteq M$.

And if g is any non-negative H^n-measurable function on M, we get the more general

$$\int_M g \; J_M f \; dH^n = \int_{\mathbb{R}^n} \left(\int_{f^{-1}(y)} g \; dH^0 \right) dH^n(y)$$

In case f is one-to-one, this becomes the classic formula of change-of-variables, stated in a slightly more general setting:

$$\int_M g \; J_M f \; dH^n = \int_{\mathbb{R}^n} g \circ f^{-1} dH^n$$

In addition, there is another very useful formula that generalizes the Fubini-Tonelli theorem, called the co-area formula:

Theorem 4.4.8 (Co-Area formula)**.** *Let $f : A \subseteq \mathbb{R}^n \to \mathbb{R}^m$ be Lipschitz on A measurable. Let $g \geq 0$ be a measurable function on E, or gJ_Mf be integrable. If $n \geq m$, then*

$$\int_A g\ J_M f\ dH^n = \int_{\mathbb{R}^m} \left(\int_{f^{-1}(y)} g\ dH^{n-m} \right) dH^m(y)$$

To see how this is a generalization of the Fubini-Tonelli theorem, we can apply the co-area formula in two ways. Take $A = \mathbb{R}^n$ in both cases, and f the projection onto the first m (resp. $n-m$) coordinates. In any case, $Jf(x) = 1$ and the Hausdorff measure reduces to the Lebesgue measure λ^n, and we have

$$\int_{\mathbb{R}^n} g\ d\lambda^n = \int_{\mathbb{R}^m} \left(\int_{\mathbb{R}^{n-m}} g\ d\lambda^{n-m} \right) d\lambda^m = \int_{\mathbb{R}^{n-m}} \left(\int_{\mathbb{R}^m} g\ d\lambda^m \right) d\lambda^{n-m}$$

We will often write λ instead of λ^n for the Lebesgue measure on $\mathbb{R}^n$ or an a open subset of $\mathbb{R}^n$.

4.4.3 s-Hausdorff measures

This part is novel. We will characterize the pushforward of the Lebesgue measure by PAP functions as those having a density w.r.t. some measure in a set of natural measures on $\mathbb{R}^m$. These new measures extend the well-known Hausdorff measure on a manifold to a generalization that includes a countable sum of these base measures, which we call s-Hausdorff measures.

Definition 4.4.15 (s-Hausdorff measure on $\mathbb{R}^m$)**.** *For each $0 \leq k \leq m$, let $\{M_i^k\}_{i\in\mathbb{N}}$ be a countable family of k dimensional smooth submanifolds of $\mathbb{R}^m$. Let $M^k := \bigcup_{i\in\mathbb{N}} M_i^k$ and equip M^k with the k-Hausdorff measure. This measure extends to a measure μ^k on $\mathbb{R}^m$ by assigning measure 0 to every (Borel) measurable set outside M^k. Let $\mu := \sum_{0\leq k\leq m} \mu^k$. We call such a μ an s-Hausdorff measure.*

Example 4.4.1. *1. The Lebesgue-measure on $\mathbb{R}^m$ is s-Hausdorff.*

2. *The k-Hausdorff measure on an k-dimensional manifold $M \subseteq \mathbb{R}^m$ is s-Hausdorff.*

3. *The sum of finitely many s-Hausdorff measures with disjoint support is s-Hausdorff.*

4. *The Dirac measure δ_x at a point $x \in \mathbb{R}^m$ is s-Hausdorff.*

5. *More generally the measure given by a countable sum of 0-dimensional Hausdorff measure on 0-dimensional submanifolds of $\mathbb{R}^m$ is the counting measure, and is s-Hausdorff.*

Indeed, we can see it as follows:

1. *$\mathbb{R}^m$ is a smooth submanifold of $\mathbb{R}^m$ of dimension m, and it is well-known that H^m coincides with the Lebesgue measure in this case, and is therefore s-Hausdorff.*
2. *Is a special case of the definition of s-Hausdorff for the case of a family of manifolds consisting of only one manifold*
3. *This follows from the fact that a finite union of countable sets is countable.*
4. *A single point $x \in \mathbb{R}^m$ is a 0-dimensional submanifold of $\mathbb{R}^m$. $H^0|_{\{x\}}$ extended to $\mathbb{R}^m$ by assigning measure 0 to every measurable set outside of $\{x\}$ coincides with the Dirac measure δ_x, which is therefore s-Hausdorff.*
5. *Follows from the previous point and the definition of s-Hausdorff measure allowing a countable union of manifolds.*

4.4.4 Infinite densities

Unfortunately, it is very easy to write a closed probabilistic program $\vdash t : \mathbb{R}^m$ for which the support of its posterior distribution is a k dimensional submanifold of $\mathbb{R}^m$ but where its density w.r.t. the k-Hausdorff measure on that manifold would be infinite. The simplest example is a program sampling from the Lebesgue measure on $\mathbb{R}$, which can be simulated using `sample` and `score`, and returns the constant real 1. Its posterior distribution will be supported on the 0-dimensional manifold $\{1\}$, but will have infinite density w.r.t. the reference s-Hausdorff measure δ_1, and yet it is absolutely continuous with respect to it.

More generally, it is easy for a PAP function $f : A \to \mathbb{R}$ for which $f(A)$ is a manifold M, and yet $f_*\lambda$ will have infinite density w.r.t. that manifold Hausdorff measure $H^k(M)$. Indeed, let M be a smooth submanifold of $\mathbb{R}^n$. Let $A = M \times \mathbb{R}$ and $f := \pi_1 : A \to \mathbb{R}^n$. Then $f(A) = M$ and $f_*\lambda$ is absolutely continuous w.r.t. the Lebesgue measure on $'RR^n$, with infinite density as every point.

We will still show that $f_*\lambda \ll H^k(M)$ always holds in such cases, and in this sense the infinity is unavoidable, but benign. By adding the convention that $\infty.0 = 0$ and $\infty.r = \infty$ for any other $r \in (0, \infty]$, we can allow densities to be valued in $[0, \infty]$ without much further trouble. Dealing with infinite densities is made precise in our case in the following way: all the measures that we will consider are s-finite [Staton, 2017], and Vákár and Ong [2018] extends the theory of s-finite kernels by showing various characterisations of s-finite measures and kernels, a Radon-Nikodým theorem, and Lebesgue decomposition, disintegration and randomisation theorems. These results are stated for potentially infinite densities, assuming the same convention about the interaction of 0 and ∞ as we did. For us, therefore, $f : X \to [0, \infty]$ is a density for a measure μ on X w.r.t. a measure ν on X if for every measurable set $A \subseteq X$, we have

$$\mu(A) = \int_A f(x)\nu(dx)$$

In brief, the usual results and intuitions that hold for densities $X \to [0, \infty)$ extend to the case of densities $X \to [0, \infty]$.

4.4.5 Pushforwards of PAP functions

We first prove the following theorem, which is the core technical result required for proving our main result (Theorem 4.4.18) on the support of probabilistic programs.

Theorem 4.4.9. *Let $f : U \subseteq \mathbb{R}^n \to \mathbb{R}^m$ be real analytic (Definition 3.2.2) on a connected open U. Then there is an integer k such that $f(U)$ is contained is a countable union $M := \bigcup_{i\in\mathbb{N}} M_i$ of smooth submanifolds M_i of $\mathbb{R}^m$ of dimension k, up to a set of k-Hausdorff measure 0. Furthermore, $f_*\lambda$ has a density w.r.t. the k-Hausdorff measure on M.*

Before proving the theorem, we need a few other technical lemmas.

Lemma 4.4.10. *Let $\mu = \sum_{0\leq k\leq m}\sum_{i\in\mathbb{N}} f_i^k.\mu_i^k \in \mathcal{M}\mathbb{R}^m$ for some non-negative measurable functions $f_i^k : \mathbb{R}^m \to [0,\infty]$, and such that μ_i^k is the k-Hausdorff measure on some k-dimensional submanifold M_i^k of $\mathbb{R}^m$. Let B be the s-Hausdorff measure given by the (μ_i^k). Then μ has a density w.r.t. B.*

Proof. Let $X_k = \bigcup_i M_i^k$ and μ_k be the k-Hausdorff measure on X_k. Let $g_k = \sum_{i\in\mathbb{N}} 1_{x\in M_i^k}.f_i^k$. As the f_i^k are non-negative and can take the value ∞, the limit exists and therefore the g_k are measurable. By construction, we have the equality $\sum_{i\in\mathbb{N}} f_i^k.\mu_i^k = g_k.\mu^k$ for all $0 \leq k \leq m$. This follows from the simpler equality $f_1.\nu|_A + f_2.\nu|_B = (f_1+f_2).\nu|_{A\cup B}$ which is also valid for countable unions, given that the μ_i^k are all restrictions of the k-Hausdorff measure. Therefore $\mu = \sum_{0\leq k\leq m} g_k.\mu_k$.

Let $g = g_0 + 1_{x\notin X_0}(g_1 + 1_{x\notin X_1}(g_2 + 1_{x\notin X_2}(\ldots + 1_{x\notin X_{m-1}}g_m)))$. g is clearly measurable and non-negative (with the convention that $0.\infty = 0$ here). For all $i < k$, X_i is H^k negligible. Thus $g_k.\mu_k = g_k.\mu_k|_{X_k - X_i} = (1_{x\notin X_i}g_k).\mu_k$. From this fact, we deduce that $\sum_{0\leq k\leq m} g_k.\mu_k = g.B$, and therefore conclude that $\mu = g.B$, i.e. that μ has a density w.r.t. B. □

Lemma 4.4.11. *Let $f : U \subseteq \mathbb{R}^n \to \mathbb{R}^m$ be real analytic on a connected open U. Let k be the maximal rank of f on U. Then $V := \{x \in U \mid rank(f(x)) = k\}$ is*

- *open,*
- *dense in U,*
- *and $\lambda(U - V) = 0$.*

Proof. We first show that V is open. Given a coordinate system for U and V, for any $x \in V$, denote by $G(x)$ the matrix for $T_x f$ in these bases. By hypothesis, $G(x)$ has at least an invertible $k \times k$ sub-matrix, and this is a characterisation of rank. Then denote by $H : M_{n\times m}(\mathbb{R}) \to \mathbb{R} : A \mapsto \sum_{B\ k\times k\ submatrix\ of\ A} |det(B)|$. H is continuous as the determinant is, and $V = (HG)^{-1}(\mathbb{R} - \{0\})$ is thus open. Second, we show that V is dense. Note that this is not true in general for smooth functions. Pick a base for U and V again, and let $G(x)$ the matrix for $T_x f$ in these bases. Thus $x \mapsto G(x)$ is also real analytic.

Let $p \in \bar{V} - V$. Pick a $k \times k$ submatrix B of matrices in $M_{n\times m}(\mathbb{R})$, then let $H_B : M_{n\times m}(\mathbb{R}) \to \mathbb{R}$ be the determinant of that submatrix. Assume $H_B G$ is 0 on a

open neighbourhood W_B of p for all B. Then $W = \cap W_B$ is a finite intersection of opens and thus is open, and $W \cap V \neq \emptyset$ as by assumption p is in the boundary of V. But this is a contradiction as we know that for some B and any x in the intersection, $H_B G(x) \neq 0$. Therefore, there is a B such that $H_B G$ is not 0 at some point x in a neighbourhood W_B on p. As f is analytic, so is df, and so is G. The determinant is a polynomial function of its arguments, it is thus real analytic, and thus so is H_B. As $H_B G$ is analytic and non-identically 0 on W_B, it must be non-zero on a dense subset of W_B. Let us assume that the closure $\bar{V}$ is not all of U. We obtain a contradiction by picking a point close to $\bar{V}$ that would end up in one of the W_B. In more detail, assume by ways of contradiction that $U - \bar{V} \neq \emptyset$. Let $p \in U - \bar{V}$. Let $g : U \to \mathbb{R}^+$ be defined as $g(x) = d(x, \bar{V})$. By assumption, as $\bar{V}$ is closed, we have $g(p) > 0$. Also, note that g is continuous. As k is the maximal ranked reached by f, $\bar{V} \neq \emptyset$. Thus, there exists $y \in \bar{V}$ such that $g(y) = 0$. As U is connected (and thus path connected by Theorem 4.4.1), there is a continuous path $c : [0, 1] \to U$ such that $c(0) = y$ and $c(1) = p$. The composition $g \circ c$ is continuous and so there a point $x \in \bar{V} - V$ such that $c(t) = x$ and $g(x) = 0$ for some $t_0 < 1$. By compactness of $[0, 1]$, we choose the biggest such t_0, which exists as $c(1) = p \notin \bar{V}$. For all $t > t_0$, we therefore have $g(c(t)) > 0$. However, by the result above, there is a dense open subset W of a W_B, a neighbourhood of x, such that $W \subseteq V$. By density of W, this means that $c(t) = 0$ on $c^{-1}(W_B) \cap [t_0, 1] \neq \emptyset$, a contradiction.

We will now prove that $X := U - V$ has Lebesgue measure 0. For a fixed $k \times k$ matrix B of any $n \times m$ matrix, consider the same analytic function H_B as above. Then $g_B := G; H_B : U \to \mathbb{R}$ is analytic. By Mityagin [2015], it is either the 0 function or the preimage of 0 has Lebesgue measure 0. We can write $X = \bigcap_{B\ k\times k\ submatrix} g_B^{-1}(\{0\})$. As f has rank k somewhere, not every g_B is the 0 analytic function, and thus X is contained in a Lebesgue measure 0 set, and therefore has Lebesgue measure 0. □

Lemma 4.4.12. *Let $f : A \to C$ where $A \subseteq \mathbb{R}^n$ is λ-measurable, and let $U \subseteq A$ be λ-measurable. Assume that $\lambda(A - U) = 0$. Then $f_*\lambda = (f|_U)_*\lambda$.*

Proof.

$$\begin{aligned}
f_*\lambda(S) &= \lambda(f^{-1}(S)) \\
&= \lambda(f^{-1}(S) \cap A) \\
&= \lambda(f^{-1}(S) \cap (A - U + U)) \\
&= \lambda((f^{-1}(S) \cap (A - U)) \cup (f^{-1}(S) \cap U)) \\
&\leq \lambda(f^{-1}(S) \cap (A - U)) + \lambda(f^{-1}(S) \cap U) \\
&\leq \lambda(A - U) + \lambda|_U(f^{-1}(S) \\
&= 0 + \lambda|_U(f^{-1}(S)) \\
&= (f|_U)_*\lambda(S)
\end{aligned}$$

As the other inequality is obvious, we showed that $f_*\lambda(S) = (f|_U)_*\lambda(S)$. □

Lemma 4.4.13. *Let $f : U \subseteq \mathbb{R}^n \to \mathbb{R}^m$ be real analytic on a connected open U. Further, assume f has constant rank. Then $f(U)$ is a countable union of manifolds.*

Proof. As f has constant rank, by the constant rank theorem, f is locally of the form $(x_1, \dots, x_n) \mapsto (x_1, \dots, x_k, 0, \dots, 0)$. More precisely, around any $x \in U$, there is a neighborhood W of x and smooth diffeomorphisms ϕ, ψ such that $g := \phi \circ f|_W \circ \psi(x_1, \dots, x_n) = (x_1, \dots, x_k, 0, \dots, 0)$. As g is a projection, it is an open map. As ϕ, ψ are diffeomorphisms, $f|_W$ is also an open map. This means that $f(W)$ is a k-dimensional submanifold of $\mathbb{R}^m$ (technically it is direct if $k = m$, otherwise we need an extra argument where we post-compose with a projection first, obtain the result above, and then use the submersion theorem to obtain that the 0 level of the projection is a manifold, which is $f(W)$).

Now consider the open cover of U given by $\{W_x \mid x \in U\}$ where W_x is a neighbourhood of x as above. By Corollary 4.4.3.1, we can extract a countable open sub-cover of U. Thus, $F(V)$ is a countable union of k-dimensional manifolds. □

Proof of Theorem 4.4.9. Let k be the maximal rank of f on U. Let $V := \{x \in U \mid rank(f(x)) = k\}$. Let us first show that it is sufficient to show the result for $f|_V$.

By Lemma 4.4.11, V is open and dense in U, and $U - V$ has Lebesgue measure 0. By Lemma 4.4.12, $f_*\lambda = (f|_V)_*\lambda$, so we reduced the second statement to showing it for $(f|_V)$. In addition, by Sard's theorem, $f(U - V)$ has k-Hausdorff measure 0. Therefore, it is sufficient to prove the theorem for $f|_V$, which we call f again.

Now, write $V = \bigsqcup_i V_i$ where the V_i are the connected components of V. On each connected component V_i, $f|_{V_i}$ verifies the hypothesis of Lemma 4.4.13. Thus $f(V_i)$ is a countable union of k-dimensional manifolds for each V_i. As Euclidean spaces are locally connected spaces with a countable basis, the same holds for V. Therefore, V has at most countably many connected components V_i. A countable union of countable sets is countable, and we finished proving the first part of the theorem.

Finally, it remains to show that $f_*\lambda$ has a density w.r.t. the k-Hausdorff measure on $f(V)$. By Lemma 4.4.10, it is sufficient to show it for each $(f|_W)_*\lambda, f(W)$, where W is as constructed above. First, let us show that it is also sufficient to restrict to the case where W is contained in a compact set. There exists a countable cover K_i of compact sets of $\mathbb{R}^n$ that cover W (e.g. by taking closed spheres of radius 1 at points with rational coordinates). Each $W_i := K_i \cap W$ is thus compact in W. Assume for now that we have proved that $(f|_{W_i})_*\lambda$ has a density w.r.t. $f(W)$, i.e. that $(f|_{W_i})_*\lambda(A) = \int_A h_i(x) dH^k(x)$ for some measurable function $h_i : f(W) \to [0, \infty]$. Then,

$$\begin{aligned} f_*\lambda(A) &= \sum_i (f|_{W_i})_*\lambda(A) \\ &= \sum_i \int_A h_i(x) dH^k(x) \\ &= \int_A (\sum_i h_i(x)) dH^k(x) \end{aligned}$$

where the last equality holds by the Fubini-Tonelli theorem as $h_i \geq 0$. $h := \sum_i h$ is measurable, and we are done.

So we can now assume that W is contained in a compact set. As f is analytic, it is locally Lipschitz and is thus Lipschitz on W, as W is contained in a compact set. Furthermore, As f has rank k, its Jacobian matrix J_x at every point x has rank k. A standard result in linear algebra asserts that $J_x J_x^T$ also has rank k and therefore the correction term $J_W f(x)$ is non-zero for all x. Let $g(x) := \frac{1}{J_W f(x)}$ is therefore measurable and non-negative. By the co-area formula, we have

$$\begin{aligned} f_*\lambda(A) &= \lambda(f^{-1}(A)) \\ &= \int_{f^{-1}(A)} 1.d\lambda \\ &= \int_{f^{-1}(A)} g(x) J_W f(x).d\lambda \\ &= \int_{f(W)} \Big(\int_{f^{-1}(y)} g(x) dH^{n-k}(x) \Big) dH^k(y) \\ &= \int_{f(W)} h(y) dH^k(y) \end{aligned}$$

where $h(y) := \int_{f^{-1}(y)} g(x) dH^{n-k}(x)$ is measurable and non-negative, yet possibly infinite.

□

Lemma 4.4.14. *Let A be an analytic set. There is an open $U \subseteq A$ such that $\lambda(A - U) = 0$.*

Proof. Let $A = \cap_i X_i^+ \cap \cap_j X_j^-$ as in the definition of analytic sets. Write $X_j^- = X_j \bigsqcup Z_j$ where $Z_j = (g_j^-)^{-1}(\{0\})$. Without loss of generality, assume that none of the g_j^- are the constant 0 function. Then, $U := \cap_i X_i^+ \cap \cap_j X_j$ is clearly open and $U \subseteq A$. Let us now show that $A - U$ has Lebesgue measure 0. It suffices to show that this is the case for each Z_j, as they cover $A - U$. However, by Mityagin [2015], g_j^- is either the 0 function or the preimage of 0 has Lebesgue measure 0. This shows that Z_j have Lebesgue measure 0, as desired. □

Theorem 4.4.15. *Let $f : A \to \mathbb{R}^m$ be PAP (Definition 3.2.4). Then $f_*\lambda$ has a density w.r.t. an s-Hausdorff measure μ on $\mathbb{R}^m$.*

Proof. Let $f : A \to \mathbb{R}^m$ be PAP. A is c-analytic, so we can write it as $A = \bigsqcup_i A_i$ where each A_i is analytic and (f_i, A_i) is a PAP representation for f. By Lemma 4.4.14, there is an open $U_i \subseteq A_i$ such that $\lambda(A_i - U_i) = 0$. By Lemma 4.4.12, it is thus sufficient to consider the case where f_i is defined on an open set U_i. Any open $V \subseteq \mathbb{R}^n$ can always be written as $V = \bigsqcup_i V_i$ for a countable family of connected opens V_i. Applying this to the U_i, we have a countable family of analytic functions $f_{i,j} : U_{i,j} \to \mathbb{R}^m$ defined on connected opens $U_{i,j}$, where $f_{i,j}$ is the restriction of f_i to $U_{i,j}$. By theorem 4.4.9, for each (i, j), there exists an integer $0 \leq k_{i,j} \leq m$ such that $(f_{i,j})_*\lambda = \sum_{l \in \mathbb{N}} f_l^{k_{i,j}} m_l^{k_{i,j}}$, where each $f_l^{k_{i,j}} : \mathbb{R}^m \to [0, \infty]$ is measurable, and each

$m_l^{k_{i,j}}$ is the $k_{i,j}$-Hausdorff measure on some $k_{i,j}$-submanifold $M_l^{i,j}$ of $\mathbb{R}^m$. Therefore,

$$\begin{aligned} f_*\lambda &= \sum_{i,j}(f_{i,j})_*\lambda \\ &= \sum_{i,j}\sum_{l\in\mathbb{N}} f_l^{k_{i,j}} m_l^{k_{i,j}} \\ &= \sum_{0\le k\le m}\sum_{\{(i,j)\ \mid\ k_{i,j}=k\}}\sum_{l} f_l^{k_{i,j}} m_l^{k_{i,j}} \\ &= \sum_{0\le k\le m}\sum_{n} f_n^k m_n^k \end{aligned}$$

where in the last equality we simply re-index the countable sum. We can now conclude by Lemma 4.4.10 that $f_*\lambda$ has a density w.r.t. some s-Hausdorff measure. □

Lemma 4.4.16. *Let ρ be the density obtained from Theorem 4.4.15 for $f_*\lambda$ w.r.t. the s-Hausdorff measure μ. Let S be the support of μ. We can assume that $\rho|_S > 0$.*

Proof. By inspection in the construction of the s-Hausdorff measure in Theorem 4.4.9 and Theorem 4.4.15, it is straightforward to see that the constructed density is positive. □

4.4.6 Support of posterior distributions of probabilistic programs

In the ω-QBS semantics, a closed probabilistic program $\vdash t : \mathbf{real}\times\ldots\times\mathbf{real}$ is interpreted as a measure on $\mathbb{R}^n$, and it is in fact obtained as the pushforward of the Lebesgue measure on Ω by a partial ω-QBS morphism $\Omega \rightharpoonup \mathbb{R}^n$. Because our primitives are PAP, we can refine this result as follows:

Lemma 4.4.17. *Let $\vdash t : \mathbf{real}\times\cdots\times\mathbf{real}$ be a closed probabilistic program. Then, there exists a partial ωPAP function $f_t : \Omega \rightharpoonup \mathbb{R}^n$ such that $[\![t]\!] = f_{t*}\lambda$.*

Proof. By construction, $[\![t]\!] \in T\mathbb{R}^n$. We now show that $[\![t]\!]$ factors through $\mathbb{E}$, that is there exists $f_t \in R\mathbb{R}^n$ such that $[\![t]\!] = \mathbb{E}_{[}f_t]$. This is sufficient as f_t is a partial ωPAP function $\Omega \rightharpoonup \mathbb{R}^n$, and verifies $[\![t]\!] = f_{t*}\lambda$ by definition of $\mathbb{E}$. We show the more general for every term $\Gamma \vdash t : \tau$, $[\![t]\!]$ factors through $\mathbb{E}$, by induction on terms:

- By construction, this is true for primitives operations and constants, as well as for `score` and `sample`.
- This is true for variables, as $return^{TX} = \mathbb{E}\circ return^{RX}$ (Proposition 4.3.9)
- If $[\![t_i]\!] = \mathbb{E}\circ f_{t_i}$, then $[\![t_1]\!] \gg\!= [\![t_2]\!] = (\mathbb{E}\circ f_{t_1}) \gg\!= (\mathbb{E}\circ f_{t_i}) = \mathbb{E}\circ (f_{t_1}\ bind f_{t_2})$ as $\mathbb{E}$ is a strong monad structure morphism (Proposition 4.3.9). Similarly, $return\ (\mathbb{E}\circ[\![t_1]\!]) = \mathbb{E}\circ(return\ [\![t_1]\!])$ as $\mathbb{E}$ is a strong monad structure morphism (Proposition 4.3.9). This argument shows that the induction extends to conditionals, application, pair-elimination, operation application, pairs, and lambdas.

- Finally, for a recursive term $\mu f/\lambda x.t$, by the induction hypothesis on t and a simple induction, we can write $f_{i+1} = \mathbb{E} \circ f_{t,i+1}$, where the f_i are those appearing the definition of $[\![\mu f.\lambda x.t]\!] = return \bigvee_{i\in\mathbb{N}} f_i$. This is also the case for $f_0 = \mathbb{E}_{\lambda\omega.\bot}$, where $\lambda\omega.\bot : \Omega \to \mathbf{L}[\![\tau]\!]$ is the nowhere defined function. In addition, just like the $(f_i)_{i\in\mathbb{N}}$, the $(f_{t,i})_{i\in\mathbb{N}}$ form an ω-chain. Therefore, by continuity of $\circ$, and as $\mathbb{E}$ is a strong monad structure morphism, we have

$$\begin{aligned}
[\![\mu f.\lambda x.t]\!](\rho) &= return \bigvee_{i\in\mathbb{N}} f_i \\
&= return \bigvee_{i\in\mathbb{N}} \mathbb{E} \circ f_{t,i} \\
&= return\ \mathbb{E} \circ (\bigvee_{i\in\mathbb{N}} f_{t,i}) \\
&= \mathbb{E} \circ return \bigvee_{i\in\mathbb{N}} f_{t,i}
\end{aligned}$$

□

That is, any probabilistic program returning reals must arise as a well-behaved transformation of "input randomness" represented by Ω. This is not particularly surprising, of course, because a probabilistic program is *defined* by specifying an (ωPAP) transformation of input randomness, but it highlights the way will use the property of being PAP to refine to prove stronger guarantees of what measures of closed probabilistic programs can be.

Using Theorem 4.4.15, we obtain our main result in this section: a construction that assigns an s-Hausdorff measure $B(t)$ to every closed probabilistic program $\vdash t : \mathbf{real} \times \cdots \times \mathbf{real}$, such that $[\![t]\!]$ has a density w.r.t. $B(t)$. We write $\mu \ll \nu$ to mean that μ is absolutely continuous w.r.t. ν.

Theorem 4.4.18. *Let $\vdash t : \mathbf{real} \times \cdots \times \mathbf{real}$ be a closed probabilistic program. There exists an s-Hausdorff measure $B(t)$ on $\mathbb{R}^n$ such that*

- *$[\![t]\!]$ has a density (possibly infinite) ρ_t with respect to $B(t)$;*
- *if μ is another s-Hausdorff measure w.r.t. which $[\![t]\!]$ has a density, then that density is $[\![t]\!]$-almost-everywhere equal to ρ_t.*
- *if t_2 has the same type and $[\![t_1]\!] \ll [\![t_2]\!]$ then $B(t_1) \ll B(t_2)$; and*
- *there is a set $A \subseteq \mathbb{R}^n$ such that $\mathbf{1}_A$ is a density of $B(t_1)$ with respect to $B(t_2)$.*

Proof. Let $\vdash t : \mathbf{real}^n$. By Lemma 4.4.17, there exists an ωPAP morphism $f : \Omega \to \mathbf{L}\mathbb{R}^n$ such that $f_*\lambda = [\![t]\!]$. As Ω is a coproduct, by the universal property of the coproduct, f decomposes as a family of ωPAP morphisms $f_i : \mathbb{R}^i \to \mathbf{L}\mathbb{R}^n$. By definition of the lifting monads, the f_i represent PAP functions $f_i : A_i \to \mathbb{R}^n$ where each A_i is c-analytic. By Theorem 4.4.15, each $f_{i*}\lambda$ has a density w.r.t. an s-Hausdorff measure μ_i on $\mathbb{R}^n$, i.e. $f_{i*}\lambda = g_i.\mu_i$ for some non-negative measurable function $g_i : \mathbb{R}^n \to [0,\infty]$. Therefore

$$[\![t]\!] = f_*\lambda = \sum_{i\in\mathbb{N}} f_{i*}\lambda = \sum_{i\in\mathbb{N}} g_i.\mu_i$$

And we conclude that this last formula has a density w.r.t. some base measure on $\mathbb{R}^n$ by Lemma 4.4.10. We proved the first point in the theorem.

For the second point, let g be the density of $[\![t]\!]$ w.r.t. μ, and f the density of $[\![t]\!]$ w.r.t. $B(t)$. Write $B(t) = \sum_d H^d(- \cap M^d)$.

By induction on d in $[0, n]$, we show that $g(x) = f(x)$ for all $x \in M^d$, except for a H^d-null set.

- Base case $d = 0$: Let $x \in M^0$.

$$g(x) = (g.\mu)(\{x\}) = [\![t]\!](\{x\}) = (f.B(t))(\{x\}) = f(x)$$

- Inductive case $n \geq d > 0$.

  1. For all $0 \leq i < d$, M^i is a H^d-null set, and therefore it suffices to show the result for $S^d := M^d - (\bigcup_{i<d} M^i)$.
  2. For any H^d measurable $A \subseteq M^d$, we have:

$$\int_{A \cap S^d} f(x) H^d(dx) = f.B(t)(A) = [\![t]\!](A) = g.\mu(A) = \int_{A \cap S^d} g(x) H^d(dx)$$

  Therefore the integrators (and measures by Risez theorem) $\int_{\_} g(x) H^d(dx)$ are equal as measures on S^d, and therefore $f(x) = g(x)$ for all x on S^d except on a H^d null set.

  Let $Z = (\bigcup_{0 \leq d \leq n} M^d)^c$. Then$[\![t]\!](Z) = f.B(t)(Z) = 0$. Let $C = \bigcup_{0 \leq d \leq n} C_d$ where C_d is the H^d-null set in S^d on which f and g disagree. Then $[\![t]\!](C) = 0$ and thus $[\![t]\!](Z \cup C) = 0$. As the set of points on which f and g disagree is contained in $Z \cup C$, we are done.

For the third point, by definition of base measures, we can write $B(t)$ as $B(t) = \sum_{0 \leq d \leq n} H^d(- \cap M^d)$ for some countable unions of d-dimensional manifolds M^d. By the way we constructed $B(t)$, the density ρ_t satisfies $\rho_t(x) > 0$ for every x in M^d (Lemma 4.4.16), except at an H^d-null set. Let A be such that $B(t_2)(A) = 0$. Then, by the positive density ρ_{t_2} of $[\![t_2]\!]$ w.r.t. the base measure $B(t_2)$, we also have $[\![t_2]\!](A) = 0$. Therefore, $[\![t_1]\!](A) = 0$ using the assumption of absolute continuity. Finally, using the first point in Theorem 4.4.18, we conclude that $B(t_1)(A) = 0$.

For the fourth point, this follows from general properties of s-Hausdorff measures. Let $\mu_1 = \sum_{0 \leq d \leq n} H^d(- \cap M^d)$ and $\mu_2 = \sum_{0 \leq d \leq n} H^d(- \cap M'^d)$ be two s-Hausdorff measure such that $\mu_1 \ll \mu_2$. As they are σ-finite measures, by the Radon-Nikodym theorem we can write $\mu_1 = g.\mu_2$. Let $S^d := M^d \backslash \bigcup_{i<d}(M'^i \cup M^i)$ and $A := \bigcup_d S^d$. Let $B \subseteq A \cap M^d$. We have $\mu_1(B) = H^d(B \cap M^d)$, as for each $0 \leq i < d, H^i(B \cap M^i) \leq H^i(A \cap M^d \cap M^i) = H^i(\emptyset) = 0$, and for each $i > d, H^i(B \cap M^i) \leq H^i(M^d) = 0$. Likewise, $\mu_2(B) = H^d(B \cap M'^d)$. As $\mu_1(B) = g.\mu_2(B)$, we have proved that the measures $H^d(- \cap M^d)$ and $g.H^d(- \cap M'^d \cap A)$ are the same measures when restricted to $A \cap M^d$, and therefore $g(x) = 1$ for H^d almost all $x \in M^d \cap A$. We conclude that 1_A is a density for μ_1 w.r.t. μ_2. □

In particular, this means that a sensible design decision for a PPL is to always compute densities of probabilistic programs p with respect to an s-Hausdorff base measure $B(p)$. In order to compute Radon-Nikodym derivatives between multiple programs, the theorem implies we can separately compute their densities and then take the ratio, so long as we also include a "support check" (of the sort described by [Radul and Alexeev, 2021]).

Furthermore, the following result shows that the class of s-Hausdorff measures seems appropriate to serve as base measures, as for each s-Hausdorff measure, there is a closed probabilistic program whose posterior distribution has a strictly positive density w.r.t that measure. This means that we cannot "carve out" any mass from the s-Hausdorff measure, and that the underlying supporting manifolds faithfully represent the possible supports of posterior distributions of probabilistic programs.

Theorem 4.4.19. *Assume that the language has all partial PAP functions as primitives. Then, for every $n \in \mathbb{N}$ and every s-Hausdorff measure μ, there exists a program $\vdash t : \mathbf{real}^n$ such that $[\![t]\!]$ and μ are mutually absolutely continuous.*

Proof. By a theorem of Whitney (refined in Grauert [1958], Morrey [1958]), all (finite-dimensional, second-countable, Hausdorff, without boundary) smooth manifolds admit a (unique) analytic structure, and two smooth manifolds are diffeomorphic iff they are analytic-equivalent. Our case is even simpler, as we consider embedded submanifolds of $\mathbb{R}^n$. Let M be a smooth submanifold of $\mathbb{R}^n$. There exists a tubular envelope $U \subseteq \mathbb{R}^n$ of M, and a smooth projection $p : U \to \mathbb{R}^n$ such that $M = p(U)$. This means that $p_*\lambda|_U$ and H^k on M are mutually absolutely continuous. By reasoning in local coordinates, using analytic charts, we see that p is locally a linear projection, and is thus analytic, and therefore PAP. Now let μ be an arbitrary s-Hausdorff measure, and $\{M_i^k\}_{i\in\mathbb{N},0\leq k\leq n}$ be a support for μ. Let $p_{i,k} : U_{i,k} \to \mathbb{R}^n$ be analytic projections such that $p(U_{i,k}) = M_i^k$, where $i \in \mathbb{N}, 0 \leq k \leq n$. Every open set $U \subseteq \mathbb{R}^n$ is diffeomorphic to a bounded open V of radius at most 1, and the diffeomorphism and its inverse can be chosen to be analytic functions. Let $\phi_{i,k} : V_{i,k} \to U_{i,k}$ be such functions, where $i \in \mathbb{N}, 0 \leq k \leq n$. Let $i \in \mathbb{N}, 0 \leq k \leq n$. Let $W_{i,k}$ be the same open as $V_{i,k}$, but translated such that the ($W_{i\in\mathbb{N},0\leq k\leq n}$ are disjoint, which is always possible as the $(V_{i,k})_{i\in\mathbb{N},0\leq k\leq n}$ have radius at most 1 and there are countably many of them. Let $t_{i,k} : W_{i,k} \to V_{i,k}$ be the associated translation, which is linear and therefore analytic. Let $W = \bigsqcup_{i\in\mathbb{N},0\leq k\leq n} W_{i,k}$. Then the function $f : W \to \mathbb{R}^n$ defined by $x \in W_{i,j} \mapsto t_{i,k}; \phi_{i,k}; p_{i,k}$ is analytic, and $f_*\lambda$ and μ are mutually absolutely continuous. □

Finally, note that we have not claimed that the density is PAP (it is ωPAP, and plots of $\mathbb{W}$ are merely measurable functions), and in fact even if the density does not take the value ∞, it may fail to be PAP.

Theorem 4.4.20. *There exists a probabilistic program $\vdash t : \mathbf{real}$ such that $[\![t]\!]$ has a density ρ w.r.t. the Lebesgue measure on $\mathbb{R}$ and ρ is nowhere real analytic.*

Proof. Consider the following program P, which we again write using Python syntax for clarity:

```
def geom(x):
    y = sample()
    if y < x
        return 0
    else
        return 1+geom(x)

def P(x):
    k = geom(.5)
    score(2**k * exp(- sqrt(2**k)) * abs(cos(2**k * x)))
    return x
```

Given k, the conditional density of $[\![P]\!]$ w.r.t. Lebesgue is given by $2^k e^{-\sqrt{2^k}} |\cos(2^k x)|$. As the probability that $geom(.5) = k$ is $\frac{1}{2^k}$, the unconditional density ρ of $[\![P]\!]$ is given by

$$\rho(x) = \sum_{k \in \mathbb{N}} e^{-\sqrt{2^k}} * |\cos(2^k x)|$$

The function $f(x) = \sum_{k \in \mathbb{N}} e^{-\sqrt{2^k}} * \cos(2^k x)$ is a classic example of function that is smooth and yet nowhere analytic. This is shown to be smooth by showing the uniform convergence of each series of derivatives. This is then shown not to be analytic at each dyadic number by showing that the series of derivatives have a radius of convergence of 0 at each dyadic number. As analycity of a function is an open set property, this implies that f is nowhere analytic. By the same reasoning, ρ is a well-defined function but is nowhere analytic. □

Corollary 4.4.20.1. *There exists a PAP function $f : \mathbb{R} \to \mathbb{R}$ such that $f_*\lambda$ has a finite density w.r.t. the Lebesgue measure that is not PAP.*

Proof. By Theorem 4.4.20, $[\![P]\!]$ is PAP and its density ρ w.r.t. the Lebesgue measure is nowhere analytic. If ρ was PAP, it would be real analytic almost everywhere, a contradiction. □

More generally, using a construction similar to the one used in the Stone-Weierstrass theorem, it might be possible to construct densities representing arbitrary continuous functions, and so we leave restrictions on the language to ensure good properties of the obtained density functions as future work.

4.5 Conclusion and Related work

4.5.1 Summary

We have presented several applications of the ωPAP framework. First, we have shown how the methodology of logical relations can be used to extend the language with expressive higher-order primitives, and how to extend AD in a sound way (Section 4.1). Next, we showed that gradient descent on programs implementing differentiable functions can be soundly used with intensional derivatives, as long as

both the learning rate and the initialization are randomized (Section 4.2). Finally, we looked at an application in probabilistic programming by proving that all programs denote measures with densities with respect to some s-Hausdorff measures (Section 4.4). As such, we argued that the s-Hausdorff measures form a set of convenient base measures for density computations in Monte Carlo inference, and showed that every closed probabilistic program has a density w.r.t. some s-Hausdorff measure. These results demonstrate the value of denotational reasoning with a model that is general enough to be extensible and support the semantics of an expressive language, and yet specific enough to be able to obtain non-trivial guarantees on the denotation of programs and their usage in applications.

4.5.2 Related work and context

Semantics of probabilistic programming. Our commutative monad of measures takes clear inspiration from Staton [2017] and Vákár et al. [2019], adding an extra step in the recent search of semantic models for higher-order probabilistic programming [Heunen et al., 2017, Ehrhard et al., 2017]. In particular, our work refines the model of Vákár et al. [2019] by restricting to PAP functions, instead of merely measurable ones, but keeping its essential good properties for interpreting probabilistic programs. By doing so, our work is closer in spirit to Freer and Roy [2012]'s study of computable properties for higher-order probabilistic languages. Our monad for tracking traces of probabilistic programs is similar to what is presented in Lew et al. [2019], giving a denotational version of similar work that focus on operational semantics, such as Mak et al. [2021]. Our result about densities and s-Hausdorff measures has some of its foundations based on the careful study of s-finite measures and kernels from Vákár and Ong [2018]. It would be interesting to see if the work of Lew et al. [2023] could be extended to a non-differentiable setting with PAP primitives, proving that an AD algorithm on a probabilistic language yields unbiased gradient estimates, perhaps using the estimator derived in Lee et al. [2018] for non-smooth functions.

Disintegration and base measure. Our result on base measures for probabilistic programs is related to the literature on symbolic disintegration [Shan and Ramsey, 2017, Cho and Jacobs, 2019, Narayanan and Shan, 2020], which has also had to wrestle with the problem of finding base measures with respect to which expressive programs have densities. Our approach builds on the idea presented by Radul and Alexeev [2021]: we extend the Hausdorff base measures to s-Hausdorff base measures, and prove that they suffice for ensuring all programs have densities. We leave open the question of characterizing exactly the class of densities on s-Hausdorff measures that arise from probabilistic programs, and the investigation of the closure of these measures under least upper bounds.

5 Conclusion and Discussion

This chapter is divided into three parts. First, we give an overall summary and recap of the main results we have seen. Second, we discuss some related work I contributed to during my PhD but which were not included in this dissertation. Third, we lay out some directions for future work that I find particularly interesting.

5.1 Conclusion

We first give an overall short summary of what has been covered in this dissertation, followed by a summary of the main contributions, and conclude with some thoughts on some of the questions raised in the introduction.

5.1.1 General Summary

Chapter 1. After justifying the need for differentiable programming, we raised intuitive and elementary questions for which we did not have good answers. We further argued that the approach from the PL community would help answer such questions.

Chapter 2. We have investigated AD as a typed program transformation on a simply-typed higher-order lambda calculus. We used diffeological spaces, a generalization of Euclidean spaces and smooth maps fitted for higher-order languages, to give a denotational semantics to the language. Furthermore, we proved the correctness of the AD macro via gluing, a categorical version of logical relations. We have shown that AD has a canonical status by showing that it satisfies a universal property, and we exemplified the generality of the development by showing how only little change is required to adapt to AD computing higher-order derivatives.

Chapter 3. We argued that the semantics for differentiable programming for expressive languages need to go beyond differentiability, even in the first-order case. We introduced ωPAP spaces as a replacement for diffeological spaces to give a denotational semantics to a language further extended with conditionals and recursion. The correctness of AD is then necessarily weakened and reveals that AD computes an intensional derivative, which we showed using the framework of fibrations for logical relations.

Chapter 4. We looked at several applications of the framework set up in Chapter 3. First, we showed how to modularly and provably further extend the language and AD to new types and new primitives. Second, we looked at the behaviour of intensional derivatives when used in a gradient descent algorithm. In particular, we proved that gradient descent converges almost surely when the derivatives are computed by AD, as long as we randomize both the step size and the initialization. Third, we looked at an application of the ωPAP framework in probabilistic programming. We gave a characterization of the posterior distributions of closed probabilistic programs of type $\mathbb{R}^n$, when the deterministic language lives in ωPAP, as those having a density w.r.t. an s-Hausdorff measure on a countable union of manifolds.

5.1.2 Recap of main contributions

We can now compactly summarize some of the main results presented in this dissertation through the following diagrams.

Correctness of AD. Let Γ be a ground context, τ be a ground type, and **GSyn** be the syntactic category for the first-order language. We have the following commuting squares

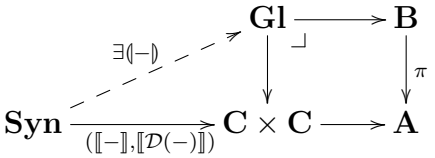
$$\begin{array}{ccc} \mathbf{Syn} & \longleftarrow & \mathbf{GSyn} \\ {\scriptstyle [\![-]\!]}\downarrow & & \downarrow{\scriptstyle [\![-]\!]} \\ \mathbf{Diff} & \longleftarrow & \mathbf{Man} \end{array} \qquad \begin{array}{ccc} \Gamma \vdash t : \tau & \xrightarrow{\mathcal{D}(-)} & \mathcal{D}\Gamma \vdash \mathcal{D}t : \mathcal{D}\tau \\ {\scriptstyle [\![-]\!]}\downarrow & & \downarrow{\scriptstyle [\![-]\!]} \\ [\![\Gamma \vdash t : \tau]\!] & \xrightarrow[\mathcal{T}(-)]{} & \mathcal{T}([\![\Gamma \vdash t : \tau]\!]) \end{array}$$

A version of this square for higher-order derivatives also holds, replacing $\mathcal{D}(-)$ by $\overrightarrow{\mathcal{D}}_{(k,R)}(-)$, and $T(-)$ by $\mathcal{T}^{(k,R)}(-)$.

When we add conditionals, we no longer have a functional relation $f \mapsto f'$ which we could wrap in an endofunctor such as T. We can still wrap the result as an informal commuting square:

$$\begin{array}{ccc} \mathbf{Syn} & & \Gamma \vdash t : \tau & \xrightarrow{\mathcal{D}(-)} & \mathcal{D}\Gamma \vdash \mathcal{D}t : \mathcal{D}\tau \\ {\scriptstyle (\!|-|\!)}\downarrow & & {\scriptstyle [\![-]\!]}\downarrow & & \downarrow{\scriptstyle [\![-]\!]} \\ \mathbf{Gl} & & [\![\Gamma \vdash t : \tau]\!] & \rightsquigarrow & ([\![\mathcal{D}\Gamma \vdash \mathcal{D}t : \mathcal{D}\tau]\!]) \end{array}$$

Proof technique for AD correctness and extensions. We have seen that the proving properties of the language, e.g. correctness of the AD transform in various setting, can be summarised by showing the existence of a lift of a functor to an appropriate category of predicates **Gl**. In the case of AD, we obtain **Gl** as an appropriate pullback, and it lives on top of the product $\mathbf{C} \times \mathbf{C}$, where we interpret our language in **C**.



We have seen in Section 4.1 that this boils down to showing that every primitive of the language preserves a logical relation predicate, and we have shown how to extend it to new types and new primitives.

Posterior Distributions of probabilistic programs. Using the framework of ωPAP, I showed that every closed probabilistic program $\vdash t : \mathcal{M}\mathbb{R}^n$ (in a higher-order language where the deterministic primitives are partial piece-wise analytic functions) has a density w.r.t. some s-Hausdorff measure.

5.1.3 Discussion on the questions from the introduction

Differentiating a function w.r.t. a function. We sidestepped a direct answer. There are certain known definitions for some function spaces. For instance, one can be found in functional analysis (Fréchet differential of a function $f : H \to \mathbb{R}$ where H is a Hilbert space, some of which are functions spaces). Another one is the functional derivative is the calculus of variations. However, there is no good answer as to what the derivative in a more general function space should be. In such infinite dimensional spaces, one quickly loses any geometric intuition and there is no naive version of doing some sort of gradient descent in an arbitrary function space for higher-order functions. Even in diffeological spaces, there are several non-equivalent ways to conservatively extend the tangent bundle functor, well-defined on its subcategory of smooth manifolds, depending for instance on whether one forces the tangent space to be a vector space or not. Worse, the possible extensions appearing in the literature on diffeological spaces can all be given some sort of canonical status, as each satisfies a universal property. For the vast majority of applications, however, we want to do optimization in a possibly large but finite dimensional space, and those cases are covered by our proof of correctness of AD. My current take is therefore that unless one has a particular application in mind for which one can tinker with the mathematics enough to end up in a reasonable function space (again such as in the calculus of variations), it is sufficient to consider any *conservative* extension of the tangent bundle functor to function spaces.

What AD computes. We have seen that in general, AD computes intensional derivatives, not necessarily the true derivative, even when the program represents a

differentiable function. We have also seen that this is due to conditionals in the language, and we can imagine DSLs preventing the use of conditionals where AD would compute actual derivatives, such as the language described in Chapter 2. However, another way to look at it is that AD computing intensional derivatives is a feature. In several applications, computing the gradient is possible but costly. One can instead use in certain cases a 'wrong derivative' that is cheaper to compute, and still obtain a correct final result. The first example that comes to mind is minibatch in deep learning. There, a loss function is often of the form $l(\theta) = \sum_{1 \leq i \leq N} l_i(\theta)$ for some large N. One optimization consists in only approximating the gradient of l by using the gradient of $k \ll N$ of the l_i's, and use this in an SGD algorithm. Part of the correctness criterion, in this case, is that the approximate gradient should be *unbiased*: the average value of the approximation should be the correct gradient, even though it is fine for it to be 'wrong' at every step in the SGD algorithm. We will come back to this point in Section 5.2 below. Other practical optimizations include reducing the size of the data type used for storing the gradient: e.g. changing from Float64 to FLoat32, which will add errors in general. In other words, an *intensional* derivative can be thought of as a way to replace an actual derivative (if it even exists) that is program specific, tailored to be unbiased, low variance or cheap, depending on its use in another algorithm.

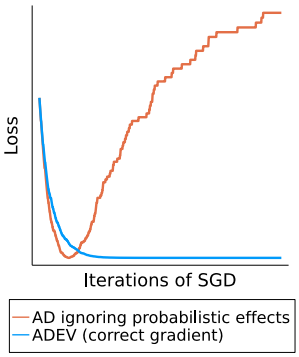
Guarantees on AD. We have seen that the guarantee that AD computes an intensional derivative is somewhat subtle in applications: it is sometimes somewhat unexpectedly sufficient, and can sometimes subtly derail algorithms such as gradient descent. On top of this, this work does not account for the usage of approximate reals such as floats, which introduces several further problems in practice. On top of the propagation of errors being hard to track, because they have a finite range, people in machine learning have to be careful of vanishing and exploding gradients. Indeed, as the chain rule multiplies derivatives, when differentiating a long program as is often the case in deep learning for instance, the real number can quickly explode beyond the maximum representable float or underflow and return 0, both cases breaking the optimization process. An alternative to these problems is to use exact reals, as in Sherman [2020]. An obvious drawback to this method is that this is prohibitively expensive for most applications. Something we have not looked at in this dissertation in more detail is the interaction of derivatives with integrals. If we integrate a function containing some derivatives w.r.t. a measure μ, we only need the derivatives μ almost everywhere. For instance, if μ is Lebesgue, then any intensional derivative would give an equivalent result.

5.2 Discussion on related work

Efficient implementations and reverse-mode AD. AD is ubiquitous in practice. One main reason for its success is that it is fast: running one pass of AD takes time proportional to running the original program, with usually a small constant overhead. In situations where the function to differentiate has type $\mathbb{R}^n \to \mathbb{R}$, we would normally need n runs of forward-mode AD. Instead, one only needs to run reverse-mode AD once. An alternative option explored by Shaikhha et al. [2019] is to optimize forward-mode AD to get performance matching the one of reverse-mode in several cases. This approach is nice as it combines the conceptual simplicity of forward-mode with a standard optimization scheme using techniques from PL. I co-authored a journal version of their work [Shaikhha et al., 2022a] to help with the semantics of their language and of AD, and show the correctness of their optimizations. Inspired by this idea of combining a simple purely functional transformation with sound optimizations which are mostly standard from PL, I developed a prototype in OCaml for a purely functional efficient reverse-mode AD and jet-based AD. (https://github.com/MathieuHuot/RAD_Compiler). The formalization, proofs, and new ideas developed to do this can be found in Huot and Shaikhha [2022]. One particularity of this work was to aim at compiling code to a second-order array language (such as Futhark [Henriksen et al., 2017]), and then performing AD at this level. This is reasonable as Futhark is purely functional, parallel-friendly, and supports a lot of optimizations. This idea of using a second-order array language was also explored independently by Paszke et al. [2021]. Several other recent independent works came up with similar stories: a typed, functional, efficient reverse-mode AD transform on a purely functional language [Radul et al., 2022, Smeding and Vákár, 2022, Krawiec et al., 2022].

ADEV: AD for expected values of probabilistic programs. Optimizing the expected values of probabilistic processes is a recurrent and important problem in various fields ranging from AI to operations research to statistical computing. Unfortunately, using the simple version of automatic differentiation for deterministic programs we have been looking at does not in general compute the correct gradients. We give an example in Figure 5.1. However, in this work we called ADEV [Lew et al., 2023], we have shown how to extend forward-mode AD to compute gradients in such a setting. The correctness property is that AD should yield *unbiased* estimates of the true gradient. There are several interesting points raised by this line of work that relate to what we discussed in this dissertation. AD on probabilistic primitives is intensional: there are several primitives representing the same distribution, for which AD returns a different result. They represent different strategies for obtaining an unbiased estimate of the gradient. There is no optimal strategy in general as they have can different validity conditions, or represent different tradeoffs between variance and computational cost. AD on first-order stochastic primitives can be higher-order. Indeed, the evaluation of the gradient depends on the rest of the program, which is handled by careful use of continuations.

Input Loss as a Probabilistic Program	**AD on deterministic parts only (incorrect)**	**ADEV (correct derivative)**
$\mathcal{L} = \lambda\theta : \mathbb{I}.\,\mathbb{E}(\textbf{do}\ \{$ $\quad b \leftarrow \textbf{flip}\,\theta$ $\quad \textbf{if}\ b\ \textbf{then}$ $\quad\quad \textbf{return}\ 0$ $\quad \textbf{else}$ $\quad\quad \textbf{return}\ -(\theta \div 2)$ $\})$	$\mathcal{L}' = \lambda\theta : \mathbb{I}.\,\mathbb{E}((\textbf{do}\ \{$ $\quad b \leftarrow \textbf{flip}\,\theta$ $\quad \textbf{if}\ b\ \textbf{then}$ $\quad\quad \textbf{return}\ 0$ $\quad \textbf{else}$ $\quad\quad \textbf{return}\ -1 \div 2$ $\})$	$\mathcal{L}' = \lambda\theta : \mathbb{I}.\mathbb{E}((\textbf{do}\ \{$ $\quad b \leftarrow \textbf{flip}\,\theta$ $\quad \textbf{if}\ b\ \textbf{then}$ $\quad\quad \textbf{return}\ 0$ $\quad \textbf{else}$ $\quad\quad \texttt{let}\ \delta p = 1 \div (\theta - 1)$ $\quad\quad \texttt{let}\ \delta l = -1 \div 2$ $\quad\quad \texttt{let}\ l = -\theta \div 2$ $\quad\quad \textbf{return}\ \delta l + l \times \delta p\})$
$\mathcal{L}(\theta) = \frac{\theta^2-\theta}{2}$	$\mathcal{L}'_{naive}(\theta) = \frac{\theta-1}{2}$	$\mathcal{L}'_{correct}(\theta) = \theta - \frac{1}{2}$



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

Figure 5.1: Example of AD on a probabilistic program

5.3 Some directions for future work

Covariant derivatives on manifolds. We have mainly looked at differentiation in Euclidean spaces. The generalization we mentioned so far (reverse-mode, non-smooth functions, higher-order functions, probabilistic) all eventually target a Euclidean space. A nice property of any Euclidean space E that we constantly used without even realizing it is that at each point $x \in E$, the tangent space T_x at x is isomorphic to E. In fact, for every choice of a basis of E, we obtain such an isomorphism $T_x E \cong E$ which is independent of x. On a curved space, such as a manifold, this simple assumption may break. The covariant derivative is the generalization of the usual derivative on Euclidean spaces to Riemannian manifolds (smooth manifolds whose tangent space is equipped with an inner product). They have been largely studied in differential geometry and in physics, such as in general relativity, quantum field theory and fluid dynamics. It has also been recently used in machine learning [Batard and Bertalmío, 2014]. It would be nice to extend AD

to such a setting in a unified and principled way. A first step might be to start with embedded manifolds (they are automatically Riemannian) and introduce a simple dependent system to track the dependency of the tangent space of the value of x. It would then be interesting to see if the dependent type system could be optimized away and investigate a certain well-behaved sublanguage with good complexity guarantees. Indeed, in general, the connection between tangent spaces will involve costly matrix operations, which currently seem prohibitively expensive for usage on space of very high dimension such as in deep learning. There are methods for doing optimization in the bigger space, and then projecting onto the constrained space [Hauswirth et al., 2016, Chen and Wainwright, 2015], or generalizations such as proximal gradient descent [Rockafellar, 1970, Parikh et al., 2014]. Several useful manifolds in practice come up as manifolds of matrices. In this situation, there is also a well-developed theory on how to perform efficient optimization on such spaces [Luchnikov et al., 2021, Townsend et al., 2016, Absil et al., 2009, Boumal, 2020].

Unified treatment with other forms of derivatives. I already mentioned some of them, like the calculus of variation or the adjoint method from perturbation theory [Pontryagin, 1987, Kato, 2013]. It is likely to be feasible to extend the theory we developed for several such notions to be added as new primitives in our language which could be proved correct (as we did with the ODE solver in Section 4.1.3). However, it would be nice to extend in a principled way. For instance, it would be nice to differentiate the code of the ODE solver directly via a generalization of AD that would give the correct derivative of the ODE solver directly. Likewise, it would be nice to extend to non-continuous values and combine discrete and continuous differentiation in a unified way. In popular AD framework, there is now a unified treatment of several variations on gradient descent: basic one, with adaptive momentum, stochastic, etc. It would also be interesting to possibly add more levels of granularity to the way we do AD as a program transform (the usual derivative is the limit of $\frac{f(x+\epsilon)-f(x)}{\epsilon}$ as $\epsilon \to 0$, the discrete one just takes $\epsilon = 1$, the second-order derivative gives information about the curvature of the graph of f, etc.). One interesting direction comes from Jacobs [2021], who introduced formal infinitesimal to track extra information about the dimension. Likewise, Arya et al. [2022] introduced an extension of AD in a probabilistic setting to track events of 'infinitesimal probabilities', but which must be taken into account to differentiate properly. In this dissertation, I focused on AD as a program transform which implements ideas from calculus. However, derivative-like operators which sometimes have barely anything to do with smoothness are also ubiquitous. Examples from mathematics include derived functors in homology and cohomology, differentials in differential algebra, exterior derivatives in differential geometry, as well as Radon-Nikodym derivatives in measure theory, or even weak derivatives for Schwartz distributions. There are other examples in PL and type theory [McBride, 2001, Ehrhard and Regnier, 2003]. It would be interesting to extend AD to further grounds, using ideas from some of these theories. For instance, there are already some attempts to incorporate ideas from differentiating Schwartz distributions into AD [Bangaru et al., 2021].

Unified treatment with efficient derivative. I have already discussed how one main reason for the adoption of AD is that it computes accurate derivatives in a fast way. A general question is therefore what level in the compiler stack is better to perform AD on. There are at least three parameters to play with when considering this. The first one is how expressive the language is. Two extremes in this spectrum are differentiating an expressive high-level language such as the one we looked at in this dissertation, and differentiating low-level SSA code such as in Innes [2018]. AD, like any program rewrite, may introduce new code that can be optimized when combined with other rewrites, and it is an open question what is the optimal intermediate representation (IR) to perform AD on. Likewise, a second parameter is how dynamic the IR is. The more information one has, the more optimized the AD transform can be, But then one may have to perform AD on the program each time we run it, as the computation flow may change between runs if it contains branches or recursion (this is, for instance, the case in TensorFlow). Thirdly, one can define AD on an IR that will be "algebra-friendly". To illustrate this point, let me give three examples. First, we can say that for second-order array languages, the algebra at play relates to parallelism, and doing AD on such a language might reveal opportunities for optimizing parallelism preservation. Second, for array-based languages such as NumPy, maybe the algebra would relate to tensor-sparsity and memory optimization. Third, the use of the right calculus seems to play an important role. For instance, generalization of the vector-valued calculus (differentiating e.g. a matrix w.r.t. a vector), such as the Ricci calculus, have shown to lead up to several orders of magnitude improvement in certain cases [Laue et al., 2018, 2020, Bernstein et al., 2020]. To conclude this part, it would be interesting for future work to define AD in a unified way that could take all of these points into account, automatically. For instance, AD could be computed (i.e. remove the syntactic operation $\mathcal{D}$ and replace with code) in several stages, some parts being kept symbolic in certain IR and replaced only in later IR in the compilation pipeline. Maybe something like the Julia multiple-dispatch feature could be used to determine what the AD transform does, based on the information given at compile or run time. This might require some advanced metaprogramming techniques, which could be of independent interest.

AD internal to the language. The defined AD as a transformation acting on the language that is external to the language. Another way would be to add syntax and typing rules for $\mathcal{D}(-)$ directly. This approach is for instance explored in Abadi and Plotkin [2020]. Then, the goal is to define rewrite rules for eliminating $\mathcal{D}(-)$, recursively, on the structure of terms on which it applies. One disadvantage of this method is that one has to deal with two rewrite systems, one for the operational semantics and one for computing $\mathcal{D}(-)$. Another is that it is non-trivial to eliminate $\mathcal{D}(-)$, and one might need to intertwine the rewrites coming from the operational semantics and from the reduction of $\mathcal{D}(-)$. There are also problems such as reducing terms of the form $\mathcal{D}(\mathcal{D}(t)$. One advantage, however, is that it can be more expressive. For instance, if the language allows recursion, $\mathcal{D}(t)$ and conditionals, we can write

the solution to the following linear ODE with an initial condition

$$g(x) = a(x)g'(x) \qquad g(0) = b$$

as the program

$$\mu g : \mathbf{real} \to \mathbf{real}.\lambda x : \mathbf{real}.\mathbf{if}\ x = 0\ \mathbf{then}\ b\ \mathbf{else}\ a(x) * \mathcal{D}(g)$$

And of course, more generally, one could write arbitrary complex PDE. It would be quite interesting to develop program analysis in such a context to recognize that for certain cases, we can simplify the program when we know an analytic solution, or define a program rewrite that would produce the code an ODE solver would give in this situation. There might also be exciting opportunities to develop denotational models for such languages, as the power of the interaction between higher-order recursive functions and differentiation really shines in this ODE example.

A PL account to optimisation. As we have seen in Section 4.2, AD on programs can produce subtle behaviours that can derail other algorithms, such as optimization algorithms. Of course, this is well-known in the numerical computing communities, and e.g. in machine learning: one has to carefully craft its programs to avoid exploding or vanishing gradients, control the approximation errors when discretizing differential equations, beware of burn-in for Metropolis-Hasting based simulations methods, etc. There is some interesting recent attempt at using AD [Menon et al., 2018]. In this work, for instance, they use algorithmic differentiation to provide accurate estimates about the final output error for mixed-precision analysis on HPC workloads. They provide a floating-point precision sensitivity profile, which can be used to make algorithmic choices and to develop mixed-precision configurations for a program. More generally, it would be great to have a more principled PL approach to some of these problems. This could occur in several cases. First, when the analysis is sufficiently compositional, as in Menon et al. [2018]. Second, when one can design a DSL encoding a useful mathematical property, or a variant of it, such as convexity, differentiability and PAP. Third, when one can design a DSL with some complexity or algorithmic guarantees (e.g. if it is a DSL for linear algebra, we can rely on efficient matrix multiplication). There are several recent works trying to make some mathematical theories, e.g. implicit differentiation, more appealing to computer scientists and programmers [Bolte et al., 2021, Bolte and Pauwels, 2021, Blondel et al., 2021]. These seem like good first steps into making the theory fitted for the programs one can actually write, and go beyond usual smoothness. Lastly, there are interesting and hugely popular methods for doing optimization using data structures such as trees for tree boosting algorithms [Ke et al., 2017, Chen and Guestrin, 2016]. This also seems like an exciting and promising area for the PL community to investigate.

AD on declarative languages. Following up on the previous thread, an interesting idea is to perform AD in declarative languages, such as extensions of Datalog. A lot of the data required for learning in machine learning and other

fields is processed via query languages such as SQL or other variants of Datalog. This happens before the data is sent to the learning algorithm, preventing potential cross-optimisations. This point is argued e.g. in Khamis et al. [2020]. Trying to do AD in a declarative language such as Datalog introduces several challenges. One of them is that programs usually denote relations or sets instead of mere functions. These relations can also be seen as higher-order functions, but it becomes non-trivial to define AD at this level. Another aspect is that even if the program denotes a function, it might have relational non-functional subparts, similar to the situation we had with higher-order functions in a first-order term. So one is compelled to reason about a set-valued version of differentiation. Derivatives are also more intrinsically intensional, as there might be several ways to parse a Datalog program as a function. A simple example is the following rule

$$R(x, y, z) := z = x + y, z = x * y, S(x, y)$$

This is to be interpreted as follows: we want the set of triples (x, y, z), which will be stored in R, that satisfy the conjunctions of the three conditions on the RHS, where S is a fixed finite set of pairs (x, y). We can interpret R as a function $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$ in two ways. It can be seen as the function $(x, y) \mapsto x + y$, which we want to evaluate on the inputs coming from S, and then filter to only remember those triples (x, y, z) for which $z = x * y$. Likewise, R can be interpreted as the function $(x, y) \mapsto x * y$, which we evaluate on the same set S of inputs, and then filter according to $z = x + y$. Thus, AD must be defined to take functional dependency information into account and can do so non-uniquely. Datalog-like languages are also challenging to reason about because they allow recursive rules, which should come as an additional challenge when reasoning about the correctness of AD for such languages.

Beyond usual differentiability In logic programming, as in Datalog, naive evaluation of recursive queries can be quite slow as each iteration in the recursive evaluation can recompute previously computer tuples. As an example, consider the transitive closure relation TC of a graph, whose edges are encoded in a known relation E:

$$TC(x, y) := E(x, y) \text{ or } \exists z : E(x, z) \text{ and } TC(z, y)$$

This is to be understood as saying a pair (x, y) belongs to the transitive closure TC if there is an edge between x and y, or if there is an intermediate node z such that there is an edge between x and z, and that (z, y) are already in the transitive closure. The relation is can be computed starting from the empty set for TC, and iterating until a fixpoint is reached. Looking at the evaluation process, computing the next iteration TC_{i+1} from TC_i will be done as follows, following naive evaluation:

$$\begin{aligned} &newTC_{i+1}(x, y) = E(x, y) \text{ or } \exists z : E(x, z) \text{ and } TC_i(z, y) \\ &TC_{i+1} = TC_i \cup newTC_{i+1} \end{aligned}$$

We can see that at each step, $E(x, y)$ will be added again to TC. Instead, semi-naive evaluation is the strategy that one should only add an over-approximation of the

possibly new facts to the next iteration. This is stored in a new intermediate relation δTC. In our example, the evaluation process would look something like

$$
\begin{aligned}
&TC_0 = \emptyset \\
&\delta TC_0(x, y) = E(x, y) \\
&TC_{i+1} = TC_i \cup \delta TC_i \\
&\delta TC_{i+1}(x, y) = \exists z : E(x, z) \text{ and } \delta TC_i(z, y)
\end{aligned}
$$

This procedure can be automated, and obeys a product rule: in other words, it looks like automatic differentiation! Another related kind of differentiation on Datalog program is incremental computation, see e.g. Cai et al. [2014], Alvarez-Picallo and Ong [2019], Alvarez-Picallo et al. [2019]. For a more detailed introduction to Datalog and these topics, see Green et al. [2013]. As we can see from these examples, AD-like algorithms can also be interpreted as information propagation algorithms. Perhaps this should not come as a surprise, as the usual derivative simply contains rate-of-change information, but this might give a way o think of these algorithms in a more general way. What is also interesting is the role of linearity: it often plays a crucial role in enforcing good performance on these algorithms. For instance, the version of semi-naive evaluation we gave works nicely for linearly recursive queries and is a bit more complex otherwise. A similar process happens when implementing magic sets, another AD-like optimization in Datalog: a recursive query that is non-linearly so will force introducing mutually recursive predicates, which can be avoided otherwise.

Towards a generalized Stokes theorem? Recall the fundamental theorem of calculus, given by $\int_a^b f'(x)dx = f(b) - f(a)$ Let us write df for $f'(x)dx$. We can see the interval $[a, b]$ as an oriented 1-dimensional manifold. It has for positive boundary the 0-dimensional manifold $\{b\}$ and for negative boundary the 0-dimensional manifold $\{a\}$. Write ∂M for the signed boundary of an oriented smooth manifold M with boundary. We can rephrase the fundamental theorem of calculus as

$$\int_{[a,b]} df = \int_{\partial[a,b]} f$$

More generally, we can reinterpret the fundamental theorem of calculus not as a formula between derivative and integrals, but between derivative and boundary. This formula has generalizations to higher dimensions (Green theorem, divergence theorem), all of which can be seen as special cases of exterior derivatives and the generalized Stokes theorem. It allows us to formulate an equivalent compact formula for an arbitrary dimension n. It reads

$$\int_M df = \int_{\partial M} f$$

I will not describe the mathematical details here, but the intuition is sometimes summarized as: "the sum of the little change on the inside (LHS) is equal to the total change on the outside (RHS)". An interesting idea is whether we could extend

this formula to other forms of derivatives, such as the ones we have briefly mentioned above. For instance, we could perhaps see semi-naive evaluation in Datalog as follows. We want to compute the recursive query f on new facts, which are at the boundary of the already explored transitive closure (think of performing Dijkstra's algorithm from a fixed node, at iteration i we have explored all nodes at distance $i \leq i$, and we are interested in those at the current boundary: those at distance $i+1$). This would represent the RHS in Stokes' formula. Then, a correct implementation of semi-naive produces a recursive query (or list of queries) df such that the Stokes formula holds, where $\int$ is now to be understood as a set union.

Bibliography

M. Abadi and G. D. Plotkin. A simple differentiable programming language. In *Proc. POPL 2020*. ACM, 2020.

M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

P. Ablin, G. Peyré, and T. Moreau. Super-efficiency of automatic differentiation for functions defined as a minimum. In *International Conference on Machine Learning*, pages 32–41. PMLR, 2020.

P.-A. Absil, R. Mahony, and R. Sepulchre. Optimization algorithms on matrix manifolds. In *Optimization Algorithms on Matrix Manifolds*. Princeton University Press, 2009.

J. Adámek and J. Rosicky. *Locally presentable and accessible categories*, volume 189. Cambridge University Press, 1994.

M. Alvarez-Picallo and C.-H. L. Ong. Change actions: models of generalised differentiation. In *International Conference on Foundations of Software Science and Computation Structures*, pages 45–61. Springer, 2019.

M. Alvarez-Picallo, A. Eyers-Taylor, M. P. Jones, C.-H. L. Ong, et al. Fixing incremental computation. In *European Symposium on Programming*, pages 525–552. Springer, Cham, 2019.

G. Arya, M. Schauer, F. Schäfer, and C. Rackauckas. Automatic differentiation of programs with discrete randomness. *arXiv preprint arXiv:2210.08572*, 2022.

A. Astorino and A. Fuduli. Nonsmooth optimization techniques for semisupervised classification. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(12):2135–2142, 2007.

J. Baez and A. Hoffnung. Convenient categories of smooth spaces. *Transactions of the American Mathematical Society*, 363(11):5789–5825, 2011.

S. P. Bangaru, J. Michel, K. Mu, G. Bernstein, T.-M. Li, and J. Ragan-Kelley. Systematically differentiating parametric discontinuities. *ACM Transactions on Graphics (TOG)*, 40(4):1–18, 2021.

G. Barthe, R. Crubillé, U. D. Lago, and F. Gavazzo. On the versatility of open logical relations: Continuity, automatic differentiation, and a containment theorem. In P. Müller, editor, *Programming Languages and Systems*, pages 56–83, Cham, 2020. Springer, Springer International Publishing.

T. Batard and M. Bertalmío. On covariant derivatives and their applications to image regularization. *SIAM Journal on Imaging Sciences*, 7(4):2393–2422, 2014.

T. Beck and H. Fischer. The if-problem in automatic differentiation. *Journal of Computational and Applied Mathematics*, 50(1-3):119–131, 1994.

G. Bernstein, M. Mara, T.-M. Li, D. Maclaurin, and J. Ragan-Kelley. Differentiating a tensor language. *arXiv preprint arXiv:2008.11256*, 2020.

E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research*, 20(1): 973–978, 2019.

M. Blondel, Q. Berthet, M. Cuturi, R. Frostig, S. Hoyer, F. Llinares-López, F. Pedregosa, and J.-P. Vert. Efficient and modular implicit differentiation. *arXiv preprint arXiv:2105.15183*, 2021.

J. Bolte and E. Pauwels. A mathematical model for automatic differentiation in machine learning. *Advances in Neural Information Processing Systems*, 33: 10809–10819, 2020.

J. Bolte and E. Pauwels. Conservative set valued fields, automatic differentiation, stochastic gradient methods and deep learning. *Mathematical Programming*, 188 (1):19–51, 2021.

J. Bolte, T. Le, E. Pauwels, and T. Silveti-Falls. Nonsmooth implicit differentiation for machine-learning and optimization. *Advances in neural information processing systems*, 34:13537–13549, 2021.

N. Boumal. An introduction to optimization on smooth manifolds. *Available online, May*, 3, 2020.

A. Brunel, D. Mazza, and M. Pagani. Backpropagation in the simply typed lambda-calculus with linear negation. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–27, 2019.

Y. Cai, P. G. Giarrusso, T. Rendel, and K. Ostermann. A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 145–155, 2014.

A. Carboni and P. Johnstone. Connected limits, familial representability and artin glueing. *Mathematical Structures in Computer Science*, 5(4):441–459, 1995.

B. Carpenter, M. D. Hoffman, M. Brubaker, D. Lee, P. Li, and M. Betancourt. The stan math library: Reverse-mode automatic differentiation in c++. *arXiv preprint arXiv:1509.07164*, 2015.

B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, and A. Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017.

R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

Y. Chen and M. J. Wainwright. Fast low-rank estimation by projected gradient descent: General statistical and algorithmic guarantees. *arXiv preprint arXiv:1509.03025*, 2015.

K. Cho and B. Jacobs. Disintegration and bayesian inversion via string diagrams. *Mathematical Structures in Computer Science*, 29(7):938–971, 2019.

J. D. Christensen and E. Wu. Tangent spaces and tangent bundles for diffeological spaces. *arXiv preprint arXiv:1411.5425*, 2014.

J. R. B. Cockett, G. S. H. Cruttwell, J. Gallagher, J.-S. P. Lemay, B. MacAdam, G. D. Plotkin, and D. Pronk. Reverse derivative categories. In *Proc. CSL 2020*, 2020.

G. Constantine and T. Savits. A multivariate Faa di Bruno formula with applications. *Transactions of the American Mathematical Society*, 348(2):503–520, 1996.

G. Cruttwell, J. Gallagher, and B. MacAdam. Towards formalizing and extending differential programming using tangent categories. In *Proc. ACT 2019*, 2019.

M. F. Cusumano-Towner, F. A. Saad, A. K. Lew, and V. K. Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th acm sigplan conference on programming language design and implementation*, pages 221–236, 2019.

S. Dash, Y. Kaddar, H. Paquet, and S. Staton. Affine monads and lazy structures for bayesian programming. *arXiv preprint arXiv:2212.07250*, 2022.

P. H. A. de Amorim and C. Lam. Distribution theoretic semantics for non-smooth differentiable programming. *arXiv preprint arXiv:2207.05946*, 2022.

J. V. Dillon, I. Langmore, D. Tran, E. Brevdo, S. Vasudevan, D. Moore, B. Patton, A. Alemi, M. Hoffman, and R. A. Saurous. Tensorflow distributions. *arXiv preprint arXiv:1711.10604*, 2017.

J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul): 2121–2159, 2011.

T. Ehrhard and L. Regnier. The differential lambda-calculus. *Theoretical Computer Science*, 309(1-3):1–41, 2003.

T. Ehrhard, M. Pagani, and C. Tasson. Measurable cones and stable, measurable functions: a model for probabilistic higher-order programming. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–28, 2017.

C. Elliott. The simple essence of automatic differentiation. *Proceedings of the ACM on Programming Languages*, 2(ICFP):70, 2018.

L. H. Encinas and J. M. Masque. A short proof of the generalized Faà di Bruno's formula. *Applied Mathematics Letters*, 16(6):975–979, 2003.

B. Fong, D. Spivak, and R. Tuyéras. Backprop as functor: A compositional perspective on supervised learning. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2019.

C. E. Freer and D. M. Roy. Computable de finetti measures. *Annals of Pure and Applied Logic*, 163(5):530–546, 2012.

M. Ghorbani, M. Huot, S. Hashemian, and A. Shaikhha. Compiling structured tensor algebra. *arXiv preprint arXiv:2211.10482*, 2022.

N. Goodman, V. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.

N. D. Goodman and A. Stuhlmüller. The design and implementation of probabilistic programming languages, 2014.

H. Grauert. On levi's problem and the imbedding of real-analytic manifolds. *Annals of Mathematics*, pages 460–472, 1958.

T. J. Green, S. Huang, B. T. Loo, W. Zhou, et al. Datalog and recursive query processing. *Foundations and Trends® in Databases*, 5(2):105–195, 2013.

A. Griewank and A. Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*, volume 105. Siam, 2008.

A. Hauswirth, S. Bolognani, G. Hug, and F. Dörfler. Projected gradient descent on riemannian manifolds with applications to online power system optimization. In *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 225–232. IEEE, 2016.

T. Henriksen, N. G. Serup, M. Elsman, F. Henglein, and C. E. Oancea. Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 556–571, 2017.

C. Heunen, O. Kammar, S. Staton, and H. Yang. A convenient category for higher-order probability theory. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12. IEEE, 2017.

M. D. Hoffman and A. Gelman. The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15 (1):1593–1623, 2014.

M. Huot and A. Shaikhha. Denotationally correct, purely functional, efficient reverse-mode automatic differentiation, 2022. URL `https://arxiv.org/abs/2212.09801`.

M. Huot and S. Staton. Quantum channels as a categorical completion. In *Proceedings - Symposium on Logic in Computer Science*, volume 2019-June, 2019a. ISBN 9781728136080. doi: 10.1109/LICS.2019.8785700.

M. Huot and S. Staton. Universal properties in quantum theory. In *Electronic Proceedings in Theoretical Computer Science, EPTCS*, volume 287, 2019b. doi: 10.4204/EPTCS.287.12.

M. Huot, S. Staton, and M. Vákár. Correctness of automatic differentiation via diffeologies and categorical gluing. In *FoSSaCS*, pages 319–338, 2020.

M. Huot, A. Lew, S. Staton, and V. Mansinghka. What do posterior distributions of probabilistic programs look like? Preprint available at `https://users.ox.ac.uk/~scro3639/LAFI-2023.pdf`, 2023a.

M. Huot, A. K. Lew, V. K. Mansinghka, and S. Staton. ωpap spaces: Reasoning denotationally about higher-order, recursive probabilistic and differentiable programs. *arXiv preprint arXiv:2302.10636*, 2023b.

P. Iglesias-Zemmour. *Diffeology*. American Mathematical Soc., 2013.

M. Innes. Don't unroll adjoint: Differentiating ssa-form programs. *arXiv preprint arXiv:1810.07951*, 2018.

B. Jacobs. *Categorical logic and type theory*. Elsevier, 1999.

J. Jacobs. Paradoxes of probabilistic programming: And how to condition on events of measure zero with infinitesimal probabilities. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–26, 2021.

P. T. Johnstone. *Sketches of an elephant: A topos theory compendium*, volume 2. Oxford University Press, 2002.

P. T. Johnstone, S. Lack, and P. Sobociński. Quasitoposes, quasiadhesive categories and artin glueing. In *International Conference on Algebra and Coalgebra in Computer Science*, pages 312–326. Springer, 2007.

S. M. Kakade and J. D. Lee. Provably correct automatic sub-differentiation for qualified programs. *Advances in neural information processing systems*, 31, 2018.

O. Kammar and D. McDermott. Factorisation systems for logical relations and monadic lifting in type-and-effect system semantics. *Electronic notes in theoretical computer science*, 341:239–260, 2018.

T. Kato. *Perturbation theory for linear operators*, volume 132. Springer Science & Business Media, 2013.

S.-y. Katsumata. A semantic formulation of TT-lifting and logical predicates for computational metalanguage. In *International Workshop on Computer Science Logic*, pages 87–102. Springer, 2005.

S.-y. Katsumata. Relating computational effects by TT-lifting. *Information and Computation*, 222:228–246, 2013.

G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30, 2017.

J. L. Kelley. *General topology*. Courier Dover Publications, 2017.

M. A. Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. Learning models over relational data using sparse tensors and functional dependencies. *ACM Transactions on Database Systems (TODS)*, 45(2):1–66, 2020.

K. A. Khan and P. I. Barton. A vector forward mode of automatic differentiation for generalized derivative evaluation. *Optimization Methods and Software*, 30(6): 1185–1212, 2015.

J. Kiefer, J. Wolfowitz, et al. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3):462–466, 1952.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

I. Kolář, P. Michor, J. Slovák, and J. Jeník. *Natural Operations in Differential Geometry*. Springer, 09 1996. ISBN 978-3-642-08149-1. doi: 10.1007/978-3-662-02950-3.

E. Kowalski. Measure and integral. *Recovered from: https://people. math. ethz. ch/˜ kowalski/measure-integral. pdf*, 59, 2011.

F. Krawiec, S. P. Jones, N. Krishnaswami, T. Ellis, R. A. Eisenberg, and A. W. Fitzgibbon. Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation. *Proc. ACM Program. Lang.*, 6(POPL):1–30, 2022.

A. Kucukelbir, D. Tran, R. Ranganath, A. Gelman, and D. M. Blei. Automatic differentiation variational inference. *Journal of machine learning research*, 2017.

S. Laue, M. Mitterreiter, and J. Giesen. Computing higher order derivatives of matrix and tensor expressions. In *Advances in Neural Information Processing Systems*, pages 2750–2759, 2018.

S. Laue, M. Mitterreiter, and J. Giesen. A simple and efficient tensor calculus. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 4527–4534, 2020.

J. M. Lee. Smooth manifolds. In *Introduction to Smooth Manifolds*, pages 1–31. Springer, 2013.

W. Lee, H. Yu, and H. Yang. Reparameterization gradient for non-differentiable models. In *Advances in Neural Information Processing Systems*, pages 5553–5563, 2018.

W. Lee, H. Yu, X. Rival, and H. Yang. On correctness of automatic differentiation for non-differentiable functions. *Advances in Neural Information Processing Systems*, 33:6719–6730, 2020.

W. Lee, S. Park, and A. Aiken. On the correctness of automatic differentiation for neural networks with machine-representable parameters. *arXiv preprint arXiv:2301.13370*, 2023.

A. K. Lew, M. F. Cusumano-Towner, B. Sherman, M. Carbin, and V. K. Mansinghka. Trace types and denotational semantics for sound programmable inference in probabilistic languages. *Proceedings of the ACM on Programming Languages*, 4 (POPL):1–32, 2019.

A. K. Lew, M. Huot, and V. K. Mansinghka. Towards denotational semantics of ad for higher-order, recursive, probabilistic languages. *arXiv preprint arXiv:2111.15456*, 2021.

A. K. Lew, M. Huot, S. Staton, and V. K. Mansinghka. Adev: Sound automatic differentiation of expected values of probabilistic programs. *Proceedings of the ACM on Programming Languages*, 7(POPL):121–153, 2023.

A. S. Lewis and M. L. Overton. Nonsmooth optimization via quasi-newton methods. *Mathematical Programming*, 141(1):135–163, 2013.

D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

F. Lucatelli Nunes, M. Vákár, et al. Logical relations for partial features and automatic differentiation correctness. 2022.

I. A. Luchnikov, M. E. Krechetov, and S. N. Filippov. Riemannian geometry and automatic differentiation for optimization problems of quantum physics and quantum technologies. *New Journal of Physics*, 23(7):073006, 2021.

S. MacLane and I. Moerdijk. *Sheaves in geometry and logic: A first introduction to topos theory*. Springer Science & Business Media, 2012.

C. Mak and L. Ong. A differential-form pullback programming language for higher-order reverse-mode automatic differentiation. *arXiv preprint arXiv:2002.08241*, 2020.

C. Mak, C.-H. L. Ong, H. Paquet, and D. Wagner. Densities of almost surely terminating probabilistic programs are differentiable almost everywhere. *Programming Languages and Systems*, 12648:432, 2021.

O. Manzyuk. A simply typed λ-calculus of forward automatic differentiation. In *Proc. MFPS 2012*, 2012.

C. Matache, S. Moss, and S. Staton. Concrete categories and higher-order recursion (with applications including probability, differentiability, and full abstraction). *arXiv preprint arXiv:2205.15917*, 2022.

D. Mazza and M. Pagani. Automatic differentiation in pcf. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–27, 2021.

C. McBride. The derivative of a regular type is its type of one-hole contexts. *Unpublished manuscript*, pages 74–88, 2001.

P.-A. Melliès and N. Zeilberger. Functors are type refinement systems. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3–16, 2015.

H. Menon, M. O. Lam, D. Osei-Kuffuor, M. Schordan, S. Lloyd, K. Mohror, and J. Hittinger. Adapt: Algorithmic differentiation applied to floating-point precision tuning. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 614–626. IEEE, 2018.

J. Merker. Four explicit formulas for the prolongations of an infinitesimal lie symmetry and multivariate Faa di Bruno formulas. *arXiv preprint math/0411650*, 2004.

J. C. Mitchell and A. Scedrov. Notes on sconing and relators. In *International Workshop on Computer Science Logic*, pages 352–378. Springer, 1992.

B. Mityagin. The zero set of a real analytic function. *arXiv preprint arXiv:1512.07276*, 2015.

E. Moggi. Notions of computation and monads. *Information and computation*, 93 (1):55–92, 1991.

C. B. Morrey. The analytic embedding of abstract real-analytic manifolds. *Annals of Mathematics*, pages 159–201, 1958.

A. P. Morse. The behavior of a function on its critical set. *Annals of Mathematics*, pages 62–70, 1939.

P. Narayanan and C.-c. Shan. Symbolic disintegration with a variety of base measures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 42(2):1–60, 2020.

R. M. Neal et al. MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2(11):2, 2011.

N. Parikh, S. Boyd, et al. Proximal algorithms. *Foundations and trends® in Optimization*, 1(3):127–239, 2014.

A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

A. Paszke, M. J. Johnson, R. Frostig, and D. Maclaurin. Parallelism-preserving automatic differentiation for second-order array languages. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*, pages 13–23, 2021.

B. A. Pearlmutter and J. M. Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):7, 2008.

A. M. Pitts. Categorical logic. Technical report, University of Cambridge, Computer Laboratory, 1995.

A. M. Pitts. Denotational semantics, 2012.

G. Plotkin and J. Power. Algebraic operations and generic effects. *Applied categorical structures*, 11:69–94, 2003.

G. D. Plotkin. Some principles of differential programming languages. Invited talk, POPL 2018, 2018.

B. T. Polyak. Introduction to optimization. optimization software. *Inc., Publications Division, New York*, 1:32, 1987.

L. S. Pontryagin. *Mathematical theory of optimal processes*. CRC press, 1987.

N. Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

A. Radul and B. Alexeev. The base measure problem and its solution. In *International Conference on Artificial Intelligence and Statistics*, pages 3583–3591. PMLR, 2021.

A. Radul, A. Paszke, R. Frostig, M. Johnson, and D. Maclaurin. You only linearize once: Tangents transpose to gradients. *arXiv preprint arXiv:2204.10923*, 2022.

H. Robbins and S. Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

R. T. Rockafellar. *Convex analysis*, volume 18. Princeton university press, 1970.

A. Sard. The measure of the critical values of differentiable maps. 1942.

A. Sard. Hausdorff measure of critical images on banach manifolds. *American Journal of Mathematics*, 87(1):158–174, 1965.

T. H. Savits. Some statistical applications of Faa di Bruno. *Journal of Multivariate Analysis*, 97(10):2131–2140, 2006.

A. Ścibior, O. Kammar, M. Vákár, S. Staton, H. Yang, Y. Cai, K. Ostermann, S. K. Moss, C. Heunen, and Z. Ghahramani. Denotational validation of higher-order bayesian inference. *Proceedings of the ACM on Programming Languages*, 2 (POPL):60, 2017.

A. Ścibior, O. Kammar, and Z. Ghahramani. Functional programming for modular bayesian inference. *Proceedings of the ACM on Programming Languages*, 2(ICFP): 1–29, 2018.

A. Shaikhha, A. Fitzgibbon, D. Vytiniotis, and S. Peyton Jones. Efficient differentiable programming in a functional array-processing language. *Proceedings of the ACM on Programming Languages*, 3(ICFP):97, 2019.

A. Shaikhha, M. Huot, S. Ghasemirad, A. Fitzgibbon, S. P. Jones, and D. Vytiniotis. Efficient and sound differentiable programming in a functional array-processing language, 2022a. URL `https://arxiv.org/abs/2212.10307`.

A. Shaikhha, M. Huot, J. Smith, and D. Olteanu. Functional collection programming with semi-ring dictionaries. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–33, 2022b.

O. Shamir and T. Zhang. Stochastic gradient descent for non-smooth optimization: Convergence results and optimal averaging schemes. In *International conference on machine learning*, pages 71–79. PMLR, 2013.

C.-c. Shan and N. Ramsey. Exact bayesian inference by symbolic disintegration. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 130–144, 2017.

B. Sherman. *Programming languages for sound computation with continuous values*. PhD thesis, Massachusetts Institute of Technology, 2020.

B. Sherman, J. Michel, and M. Carbin. λ_s: computable semantics for differentiable programming with higher-order functions and datatypes. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–31, 2021.

L. Simon. Lectures on geometric measure theory. In *Proc. Centre for Math. Anal.*, volume 3. Australian Nat. Univ., 1983.

T. Smeding and M. Vákár. Efficient dual-numbers reverse ad via well-known program transformations. *arXiv preprint arXiv:2207.03418*, 2022.

J.-M. Souriau. Groupes différentiels. In *Differential geometrical methods in mathematical physics*, pages 91–128. Springer, 1980.

A. Stacey. Comparative smootheology. *Theory Appl. Categ.*, 25(4):64–117, 2011.

S. Staton. Commutative semantics for probabilistic programming. In *European Symposium on Programming*, pages 855–879. Springer, 2017.

J. Townsend, N. Koep, and S. Weichwald. Pymanopt: A python toolbox for optimization on manifolds using automatic differentiation. *arXiv preprint arXiv:1603.03236*, 2016.

M. Vákár. Denotational correctness of foward-mode automatic differentiation for iteration and recursion. *arXiv preprint arXiv:2007.05282*, 2020a.

M. Vákár. Reverse ad at higher types: Pure, principled and denotationally correct. *arXiv preprint arXiv:2007.05283*, 2020b.

M. Vákár and L. Ong. On s-finite measures and kernels. *arXiv preprint arXiv:1810.01837*, 2018.

M. Vákár and T. Smeding. Chad: Combinatory homomorphic automatic differentiation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44(3):1–49, 2022.

M. Vákár, O. Kammar, and S. Staton. A domain theory for statistical probabilistic programming. *Proceedings of the ACM on Programming Languages*, 3(POPL):36, 2019.

M. Vákár, S. Staton, and M. Huot. Higher order automatic differentiation of higher order functions. *Logical Methods in Computer Science*, 18, 2022.

F. Wang, X. Wu, G. Essertel, J. Decker, and T. Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proceedings of the ACM on Programming Languages*, 3(ICFP), 2019.

H. Whitney. *Geometric integration theory*. Courier Corporation, 2012.