THE UNIVERSITY *of* EDINBURGH

Automatic Differentiation via Effects and Handlers

Jesse Aaron Sigal

Doctor of Philosophy

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2024

Abstract

Machine learning, artificial intelligence, scientific modelling, information analysis, and other data heavy fields have driven the demand for tools that enable derivative based optimization. Automatic differentiation is a family of algorithms used to calculate the derivatives of programs with only a constant factor slowdown. There are many implementation strategies, some built into a language and some outside of it, and there are many different members of the family. The utility of automatic differentiation makes it worthwhile to implement it in as many languages as possible.

Effects and handlers are a powerful control flow construct in programming languages based upon delimited continuations. They are a structured method of including side effects into programs, and have found many uses including nondeterminism, state management, and concurrency. Effects and handlers excel in facilitating non-local control flow and also provide methods of abstracting and composing effects. Mainstream programming languages are increasingly incorporating effects and handlers, notably OCaml and WebAssembly.

We show that effects and handlers are well-suited for implementing automatic differentiation algorithms while maintaining the desirable asymptotic efficiency. In particular, effects and handlers allow for succinctness in the presence of complex control flow. On a practical level, we implement eight automatic differentiation algorithms in four languages with effects and handlers. The implementations range from standard AD algorithms such as forward mode and continuation-based reverse mode, to more advanced modes such as checkpointed reverse mode. We benchmark the standard modes to empirically show that we can reach the correct asymptotic complexity.

Furthermore, we build up a mathematical framework in which to prove correctness of selected standard modes. To do so, we extend the set-theoretic denotational semantics of a simple effect and handler language to a category-theoretic semantics. We then describe how to perform a generalized proof by logical relations in this setting, and identify sufficient conditions for our proof method to apply. Equipped with our conditions, we show that diffeological spaces (a generalization of Euclidean spaces) admit proof by logical relations. Ultimately, this enables us to prove our implementations of forward mode and continuation reverse mode correct.

Lay summary

"Optimization" is the process of finding the best solutions to a problem based on a numerical quantity capturing how good the solution is. A "derivative" calculates how a quantity changes based on its input, and fields such as machine learning and AI use derivative based optimization to find solutions. An "algorithm" is a step-by-step description of how to solve a problem, and "automatic differentiation" (AD) is a family of algorithms which calculate derivatives efficiently and accurately. There are many ways to implement AD, and given that AD is so useful, we want every programming language to support it.

The "control flow" of a program describes the order in which pieces of the program are run. "Effects and handlers" are a programming language feature which can express complicated control flow. They also help specify how a program can interact with the world, like reading files or going online, which are called "side effects". Effects and handlers help with the "abstraction" (ignoring unnecessary details) and the "composition" (combining simpler things to make a complex thing) of side effects. Effects and handlers are becoming more popular, and have made it into real-world programming languages.

We show by example that effects and handlers are a good match for AD by concisely implementing eight AD algorithms in four different languages. Our implementations range from standard AD algorithms like "forward mode" (which computes from input to output) and "reverse mode" (which computes from output to input) to advanced versions like "checkpointed reverse mode" (similar to reverse mode, but using less space by calculating more). For each standard implementation, we analyze its "asymptotic efficiency" (or how the efficiency changes as the problem we solve gets larger) through test programs.

We also create a mathematical framework to prove the "correctness", i.e. giving the correct answer, of some standard modes. We extend a basic "denotational semantics" (a way of giving each program a mathematical meaning) to a more advanced one using "category theory", which is a theory of mathematical structures. Next, we show how to do a general proof by "logical relations" which describes how programs are related to each other. We then show our method supports "diffeological spaces", an area of math which supports taking derivatives. Finally, we use logical relations and diffeological spaces to prove that our implementations of forward mode and continuation reverse mode are correct.

Acknowledgements

I would not have been able to complete this thesis alone. First and foremost, I would like to thank my lead supervisor Chris Heunen. My PhD would have been impossible without his guidance and knowledge over the years. His advice and collaboration helped me become an independent researcher, while his empathy supported me through all the tough times.

Before I ever arrived at Edinburgh, Ohad Kammar was a mentor and teacher to me. I was saddened when he left Oxford, and delighted when my path brought me to Edinburgh. I think that Ohad is second only to Chris in how much time and effort he spent supporting me.

I am grateful to my examiners Gordon Plotkin and Nicolas Wu, who improved my thesis through their deep reading and engagement with it. Their insights and suggestions have materially enhanced the presentation and technical content of this work.

My other supervisors James Cheney and Ian Stark were integral throughout the PhD process, and helped me chart a path over the years. Thank you.

My friends and colleagues (with plenty of overlap between these categories!) helped make my PhD fulfilling and rewarding. I would like to thank the members of Chris' extended group for their camaraderie: Robert Booth, Carmen Constantin, Matthew Di Meglio, Nuiok Dicaire, Robert Furber, Robin Kaarsgaard, and Nesta van der Schaaf. I learned from each of them. My office mates, including Nick McKenna, Andreas Griveas, Eric Munday, and Asif Khan, made it a joy to come to the office. The members of the School and LFCS provided a wonderful community, I cannot list all of them. Those that have been particularly generous with their time include: Frank Emrich, Daniel Hillerström, Jean-Simon Pacaud Lemay, Sam Lindley, Anton Lorenzen, Cristina Matache, James McKinna, Chad Nester, and Wenhao Tang. Among those outside Informatics, I would like to especially thank the following people: Mario Álvarez Picallo, Swaraj Dash, Bruno Gavranović, Mathieu Huot, Alex Lew, Jesse Michel, Sean Moss, Tom Smeding, and Matthijs Vákár. I am also thankful to my friends who are not contained in the other categories, particularly: Amy Chang, Isabel Cornacchia, Bhargavi Ganesh, Robyn Greene, and Nick Hu.

Finally, my family has always been my foundation, and I would not be where I am today without them.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Jesse Aaron Sigal*)

Table of Contents

1 Introduction **1**

I Background **7**

2 Automatic Differentiation **9**

2.1 Historical background 9

2.2 Deriving Forward and Reverse Modes 10

2.3 Practice 14

2.4 Theory 17

3 Effects and Handlers **19**

3.1 Programming with Effects and Handlers 20

3.2 Design Space 24

3.3 Overview of Languages Used 27

3.3.1 Frank 28

3.3.2 Eff 30

3.3.3 Koka 32

3.3.4 OCaml 34

II Implementation **37**

4 Implementation of Standard AD Modes **39**

4.1 Forward Mode 44

4.2 Stateful Reverse Mode 47

4.3 Continuation Reverse Mode 51

4.4 Taped Reverse Mode 53

4.5 Combined Modes 57

5 Implementation of Advanced AD Modes **63**
5.1 Higher-Order Derivatives 63
5.2 Checkpointing 65
5.3 Higher-Order Functions 69

6 Performance Analysis **75**
6.1 Asymptotic Analysis 75
6.1.1 Frank 78
6.1.2 Eff 80
6.1.3 Koka 80
6.1.4 OCaml 84
6.2 Discussion of results 84
6.3 Real World Benchmarks 87

III Correctness **93**

7 Mathematical Tools **95**
7.1 Initial Algebras and Free Monads 96
7.2 Reasoning About Effects and Handlers 105
7.3 Logical Relations 132

8 Correctness of Selected Standard AD Modes **145**
8.1 Diffeological spaces 146
8.2 Correctness of Forward Mode 156
8.3 Correctness of Continuation Reverse Mode 164

9 Conclusion **169**

A Programs **175**
A.1 Smooth Effect and Helper Functions 175
A.2 Evaluation 181
A.3 Forward Mode 183
A.4 Continuation Reverse Mode 184
A.5 Stateful Reverse Mode 186
A.6 Taped Reverse Mode 188
A.7 Higher Derivatives 194

A.8 Checkpointing 212
A.9 Higher-Order Functions 217
A.10 Real World Benchmarks Code 228
A.11 Frank Patch 240

B Diffeology on subsets **259**

List of Figures

6.1 Frank benchmark results 79
6.2 Eff benchmark results 81
6.3 Koka benchmark results 82
6.4 Koka benchmark results, linear effects 83
6.5 OCaml benchmark results 85
6.6 GMM results, N = 1,000 90
6.7 GMM results, N = 10,000 91

7.1 Diagrams for theorem 7.1.20 103
7.2 Upper right triangle of fig. 7.1c precomposed with $\ulcorner\alpha\urcorner \times \mathrm{id}$ 104
7.3 MAM syntax 108
7.4 MAM operational semantics 109
7.5 MAM kinds and types 110
7.6 MAM kind system 111
7.7 MAM type system 112
7.8 MAM denotational semantics for types 113
7.9 MAM denotational semantics for terms 114
7.10 EFF syntax (extending fig. 7.3) 118
7.11 EFF operational semantics (extending fig. 7.4) 118
7.12 EFF kinds and types (extending fig. 7.5) 119
7.13 EFF kind system (extending fig. 7.6) 119
7.14 EFF type system (extending fig. 7.7) 120
7.15 EFF denotational semantics for types (extending fig. 7.8) 122
7.16 EFF denotational semantics for terms (extending fig. 7.9) 123
7.17 EFF categorical denotational semantics for types 127
7.18 EFF categorical denotational semantics for terms 128
7.19 EFF syntax extension (extending fig. 7.10) 131

7.20 EFF operational semantics extension (extending fig. 7.11) 131
7.21 EFF types extension (extending fig. 7.12) 131
7.22 EFF kind system extension (extending fig. 7.13) 131
7.23 EFF type system extension (extending fig. 7.14) 131
7.24 EFF categorical semantics extension for types (extending fig. 7.17) 132
7.25 EFF categorical semantics extension for terms (extending fig. 7.18) 132

List of Listings

4.1 Smooth effect (Frank) . . . 40
4.2 Helper functions (Frank) . . . 41
4.3 Evaluation (Frank) . . . 42
4.4 Forward mode (Frank) . . . 44
4.5 Stateful reverse mode (Frank) . . . 48
4.6 Reverse pass . . . 50
4.7 Continuation reverse mode (Frank) . . . 51
4.8 Taped reverse mode (Frank) . . . 53
4.9 Lifting (Frank) . . . 58
4.10 Lifting multiple levels (Frank) . . . 59
5.1 Second derivative forward mode (Frank) . . . 64
5.2 Checkpointed reverse mode (Frank) . . . 66
5.3 Higher-order reverse mode (Frank) . . . 70
5.4 Higher-order checkpointed reverse mode (Frank) . . . 71
A.1 Smooth effect and helper functions (Eff) . . . 175
A.2 Smooth effect and helper functions (Koka) . . . 176
A.3 Smooth effect and helper functions (OCaml) . . . 179
A.4 Evaluation (Eff) . . . 181
A.5 Evaluation (Koka) . . . 182
A.6 Evaluation (OCaml) . . . 182
A.7 Forward mode (Eff) . . . 183
A.8 Forward mode (Koka) . . . 183
A.9 Forward mode (OCaml) . . . 184
A.10 Continuation reverse mode (Eff) . . . 184
A.11 Continuation reverse mode (Koka) . . . 185
A.12 Continuation reverse mode (OCaml) . . . 186
A.13 Stateful reverse mode (Eff) . . . 186

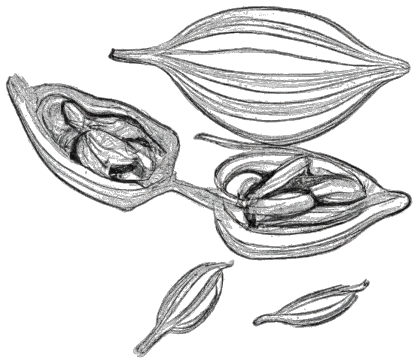
A.14 Stateful reverse mode (Koka) . . . 187
A.15 Stateful reverse mode (OCaml) . . . 188
A.16 Taped reverse mode (Eff) . . . 188
A.17 Taped reverse mode (Koka) . . . 190
A.18 Taped reverse mode (OCaml) . . . 192
A.19 Second derivative forward mode (Eff) . . . 194
A.20 Second derivative forward mode (Koka) . . . 195
A.21 Second derivative forward mode (OCaml) . . . 196
A.22 Hessian reverse mode (Frank) . . . 197
A.23 Hessian reverse mode (Eff) . . . 201
A.24 Hessian reverse mode (Koka) . . . 204
A.25 Hessian reverse mode (OCaml) . . . 208
A.26 Checkpointed reverse mode (Eff) . . . 212
A.27 Checkpointed reverse mode (Koka) . . . 213
A.28 Checkpointed reverse mode (OCaml) . . . 215
A.29 Higher-order reverse mode (Eff) . . . 217
A.30 Higher-order reverse mode (Koka) . . . 218
A.31 Higher-order reverse mode (OCaml) . . . 220
A.32 Higher-order checkpointed reverse mode (Eff) . . . 221
A.33 Higher-order checkpointed reverse mode (Koka) . . . 223
A.34 Higher-order checkpointed reverse mode (OCaml) . . . 226
A.35 Smooth effect and helper functions, tensors (OCaml) . . . 228
A.36 Stateful reverse mode, tensors (OCaml) . . . 235
A.37 GMM objective function (OCaml) . . . 237
A.38 Patch for Frank codebase . . . 240

List of Tables

6.1 Frank, R^2 values for linear and quadratic best fits 78
6.2 Eff, R^2 values for linear and quadratic best fits 80
6.3 Koka, R^2 values for linear and quadratic best fits 84
6.4 OCaml, R^2 values for linear and quadratic best fits 86

Chapter 1

Introduction



Cardamom[1]

Machine learning, artificial intelligence, scientific modelling, information analysis, and other data heavy fields have driven the demand for tools which enable derivative based optimization. The family of algorithms known as automatic differentiation (AD) is the foundation of the tools which allow automated calculation of derivatives. The family can be coarsely divided into *forward mode* and *reverse mode*. Multiple modes exist because their asymptotics depend on different features of the differentiated programs. Forward mode AD was introduced in 1964 by Wengert [Wengert, 1964], and reverse mode AD was created by Speelpenning in his 1980 thesis [Speelpenning, 1980]. It is not surprising that, given its long history, AD has been implemented in many different ways. The commonality between implementations is the preservation of the surprising efficiency of AD. Forward and reverse mode AD are only a constant multiple slower than the program being differentiated, where the optimal factors are two to three and

[1] all chapter heading images are generated by DALL·E 2

three to four times slower respectively. All in all, AD has experienced widespread adoption, either directly or through tools and systems based upon it.

Given the utility of AD, it is desirable to have implementations of it in as many languages as possible. However, the implementation strategy is heavily dependent on the language being used. Furthermore, the problem which AD is being applied to can necessitate the use of a particular mode of AD, and so the strategy employed must be flexible enough for many variations of AD. Identifying a suitable set of features in a programming language that can cope with these varied demands is not straightforward.

Effects and handlers are a structured method of including side-effects into programs, and are themselves a structured form of delimited continuations. Algebraic effects were introduced in 2001 by Plotkin and Power (2001a) and handlers for them were introduced in 2009 by Plotkin and Pretnar (2009). Effects and handlers can be viewed as an extension of the common feature of catchable exceptions. Catching an exception terminates the program delimited by the exception handling code. In contrast, effect handlers can resume the handled code and pass a value to it. Effects and handlers can implement many common side effects such as state, exceptions, non-determinism, logging, and input-output. They also support effect abstraction, composition, and program reuse through the ability of handlers to provide multiple interpretations of an effect. Furthermore, they provide a unified base in which to implement complex control flow constructs such as coroutines, generators, and async/await. In each instance, the control is non-local, an aspect in which effects and handlers excel. These use cases and others have motivated the inclusion of effects and handlers into mainstream projects such as OCaml [K. Sivaramakrishnan et al., 2021] and WebAssembly [Phipps-Costin et al., 2023].

The ability of effects and handlers to capture non-local control flow and manage effects make them an ideal match for implementing AD. An effect can be defined where there is one operation for each primitive mathematical function, and a handler can be defined for each AD algorithm. The power of effect abstraction allows a program to be written once against a specified interface and later executed using any AD algorithm. Compositionality allows AD modes to be combined to create new modes. We can also reuse handlers implementing AD modes in the definition of more advanced modes. Thus, the end-user of our framework has access to a modular and composable framework for performing

AD. Furthermore, effects and handlers can provide the desired asymptotics for AD.

Finally, the rich mathematical formulation of effects and handlers allow us to prove simple AD modes correct in a manner which can be extended later for the more complex algorithms.

Thesis Statement

This thesis provides a recipe for implementing automatic differentiation algorithms using effects and handlers, as well as a general categorical system for performing proofs by logical relations on effects and handlers. Thus, when a language supports effects and handlers, AD can be readily translated from our examples, providing a valuable tool for users. Our implementations are not tied to a specific language as we provide implementations in four different languages. Furthermore, languages with an effect type system can help prevent AD specific mathematical errors. The implementations themselves consist of the fundamental components of each AD mode, and are thus easy to reason about. Beyond this, we also show experimentally that our implementations satisfy the core asymptotic complexity of standard AD algorithms, as well as provide a real world benchmark showing that our implementation is competitive with comparable AD systems.

We then formulate a framework of logical relations for an effect and handler language, which is described in category theoretic terms using fibrations. We identify sufficient conditions in which our framework is applicable and show that the category of diffeological spaces satisfy them. Finally, we substantiate our claim that handler based AD is easy to reason about by proving two implementations correct via denotational semantics

Contributions

We make the following contributions in this thesis:

- We provide implementations of eight different AD modes across Frank, Eff, Koka, and OCaml using effects and handlers (chapters 4 and 5);

- We examine how advanced modes of AD are straightforwardly expressed using effects and handlers (chapter 5);

- We provide experimental evidence that the standard modes, including forward and reverse mode, have the correct asymptotics (section 6.1);

- We provide experimental evidence that stateful reverse mode is competitive with real world tools by adding our implementation to a benchmarking suite and showing we are competitive with comparable tools (section 6.3);

- We describe how to perform proofs by logical relations for a terminating effect and handler language whose semantics is categorical (chapter 7);

- We set out sufficient conditions to apply such methods (chapter 7);

- We prove that the category of diffeological spaces satisfies these sufficient conditions (section 8.1); and

- We prove forward and continuation-based reverse mode correct (sections 8.2 and 8.3).

There are limitations to our work. We have yet to compare our implementations to tools used by practitioners. Furthermore, we do not always achieve the correct asymptotics for each standard AD mode in some languages, although there is always at least one example for each mode except continuation reverse mode. Additionally, we have not examined how our implementations behave in the presence of other effects. Finally, we only prove correctness of two of the most basic modes, and our model language contains no inbuilt state or recursion.

There is previous work in implementing AD with handlers as well as proving implementations correct. The first implementation we are aware of is [K. C. Sivaramakrishnan, 2018], and is of stateful reverse mode AD, which was adapted from an implementation by [F. Wang and Rompf, 2018] which used delimited continuations. F. Wang, X. Wu, et al. also extended their delimited continuation AD approach in [F. Wang, Zheng, et al., 2019]. Finally, [de Vilhena and Pottier, 2023] prove the correctness of an implementation analogous to that of K. C. Sivaramakrishnan. They use their separation logic for effects and handlers to prove stateful reverse mode correct with respect to an operational semantics. In contrast, we define and use a denotational semantics for our proofs.

Outline

Part I presents an overview of AD and effects and handlers. **Chapter 2** provides background for AD. We give a historical overview, derive the main forms of AD, as well as cover the practice and theory of AD. **Chapter 3** covers the basics of effect and handler systems with a short tutorial and a discussion of the design space of such systems. We also introduce each of the four languages we will implement AD algorithms in and note what design choices they make.

Part II presents the implementation of various AD modes and discusses their performance. **Chapter 4** implements the standard AD modes in each language while noting any important differences between programs, while **Chapter 5** implements the more advanced AD modes. **Chapter 6** examines the asymptotic performance of each program and discusses the results, as well showing that stateful reverse mode is competitive with comparable real world tools.

Part III develops our mathematical tools and applies them. **Chapter 7** builds up the mathematical tools needed in order to prove the correctness of selected AD algorithms. **Chapter 8** then applies these tools to prove the correctness of forward mode AD and continuation reverse mode AD.

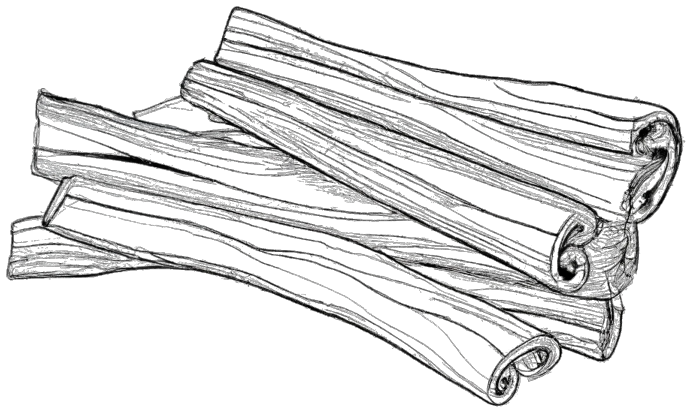
Finally, **Chapter 9** concludes the thesis with a summary of what we have achieved and discusses further research directions.

Part I

Background

Chapter 2

Automatic Differentiation



Cinnamon

2.1 Historical background

The history of differentiation and programming languages stretches back to the 1960's. In 1964, Wengert introduced a method which compositionally calculated the derivative of a program involving real numbers [Wengert, 1964]. Wengert's method was the first in a family of algorithms and methods now known as *automatic differentiation (AD)*[1]. AD can be coarsely split into two main categories, *forward mode* and *reverse mode*. Forward mode is essentially Wengert's method. Reverse mode was created in Speelpenning's 1980 PhD thesis [Speelpenning, 1980]. Each of these modes has many variations, and the field of AD has been productive since its inception.

Differentiable programming is a programming language paradigm in which

[1]Sometimes also known (mostly historically) as *algorithmic differentiation*.

some or all language constructs can be differentiated[2], and it is made possible through AD. The term differentiable programming initially appeared in Christopher Olah's blog in 2015 [Olah, 2015], originally as "differentiable functional programming". The current form, lacking the modifier functional, first appeared in a contribution by David Dalrymple in 2016 [Dalrymple, 2016]. Since then, it has been championed by deep learning practitioner Yann LeCun [LeCun, 2018] among others and has gained traction as a useful research direction.

2.2 Deriving Forward and Reverse Modes

Forward and reverse mode AD can be easily derived for pure, straight-line programs. We will do so by example. We assume that the reader is familiar with partial derivatives of real-valued functions, as well as matrix-matrix and matrix-vector multiplication. Consider the algebraic definition

$$z = h(g(f(a), b), f(a))$$

where $a, b \in \mathbb{R}$, $f \colon \mathbb{R} \to \mathbb{R}$, $g, h \colon \mathbb{R}^2 \to \mathbb{R}$, and all functions are differentiable. We can rewrite this as a sequence of calculations using intermediate variables

$$x = f(a) \quad (1)$$
$$y = g(x, b) \quad (2)$$
$$z = h(y, x) \quad (3)$$

and consider the sequence as a pure, straight-line program where the variables a, b are inputs and the variables x, y, z are initialized to 0. We now regard the state of the program at each line as a five-tuple $(a, b, x, y, z) \in \mathbb{R}^5$ containing the values of our variables. Thus, each line (i) gives a function $F_i \colon \mathbb{R}^5 \to \mathbb{R}^5$, i.e.

$$F_1(v_0, v_1, v_2, v_3, v_4) = (v_0, v_1, f(v_0), v_3, v_4)$$
$$F_2(v_0, v_1, v_2, v_3, v_4) = (v_0, v_1, v_2, g(v_2, v_1), v_4)$$
$$F_3(v_0, v_1, v_2, v_3, v_4) = (v_0, v_1, v_2, v_3, h(v_3, v_2)).$$

Our program can then be rewritten to

$$\vec{x}_0 = (a, b, 0, 0, 0)$$
$$\vec{x}_1 = F_1(\vec{x}_0)$$
$$\vec{x}_2 = F_2(\vec{x}_1)$$
$$\vec{x}_3 = F_3(\vec{x}_2)$$

[2]Here, differentiated is used in the standard mathematical sense, i.e. calculus.

where $\vec{x}_3$ gives the final state. The multivariate version of differentiation is given by the Jacobian, which for a differentiable function $F\colon \mathbb{R}^n \to \mathbb{R}^m$ and a point $\vec{x} \in \mathbb{R}^n$ we denote by $\nabla F(\vec{x})$. The Jacobian $\nabla F(\vec{x})$ is an $m \times n$ matrix containing all the partial derivatives of F at $\vec{x}$. Thus, writing $F(\vec{x})$ as $F(\vec{x}) =: (f_1(\vec{x}), \ldots, f_m(\vec{x}))$ for differentiable functions $f_j\colon \mathbb{R}^n \to \mathbb{R}$, the Jacobian $\nabla F(\vec{x})$ is

$$\nabla F(\vec{x}) \coloneqq \begin{pmatrix} \partial_1 f_1(\vec{x}) & \cdots & \partial_n f_1(\vec{x}) \\ \vdots & \ddots & \vdots \\ \partial_1 f_m(\vec{x}) & \cdots & \partial_n f_m(\vec{x}) \end{pmatrix}$$

where ∂_i is the i^{th} partial derivative operator. The Jacobian satisfies the multivariate chain rule $\nabla(G \circ F)(\vec{x}) = \nabla G(F(\vec{x})) \times \nabla F(\vec{x})$. Therefore, by viewing our program as a composition of state-transforming functions, namely $F_3 \circ F_2 \circ F_1$, we calculate

$$\nabla(F_3 \circ F_2 \circ F_1)(\vec{x}_0) = \nabla F_3(\vec{x}_2) \times \nabla F_2(\vec{x}_1) \times \nabla F_1(\vec{x}_0)$$

where $\times$ is matrix-matrix multiplication, and later matrix-vector multiplication as well. The crux of both forward and reverse mode AD is this calculation, which each mode uses differently.

For forward mode, we observe that the matrix product can be computed from right-to-left by

$$\begin{aligned} X_1 &= \nabla F_1(\vec{x}_0) \\ X_2 &= \nabla F_2(\vec{x}_1) \times X_1 \\ X_3 &= \nabla F_3(\vec{x}_2) \times X_2. \end{aligned}$$

It would be inefficient to materialize entire matrices in practice, and so we can pre-multiply by a vector $\vec{dx}_0$ to obtain

$$\nabla(F_3 \circ F_2 \circ F_1)(\vec{x}_0) \times \vec{dx}_0 = \nabla F_3(\vec{x}_2) \times \nabla F_2(\vec{x}_1) \times \nabla F_1(\vec{x}_0) \times \vec{dx}_0$$

giving the sequence of vectors

$$\begin{aligned} \vec{dx}_1 &= \nabla F_1(\vec{x}_0) \times \vec{dx}_0 \\ \vec{dx}_2 &= \nabla F_2(\vec{x}_1) \times \vec{dx}_1 \\ \vec{dx}_3 &= \nabla F_3(\vec{x}_2) \times \vec{dx}_2. \end{aligned}$$

Calculating the Jacobian of the function $F_1(v_0, v_1, v_2, v_3, v_4) = (v_0, v_1, f(v_0), v_3, v_4)$

at $\vec{x}_0$, we see

$$\nabla F_1(\vec{x}_0) = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ \partial f(a) & & 0 & & \\ & & & 1 & \\ & & & & 1 \end{pmatrix}$$

where ∂f is shorthand for the derivative of $f\colon \mathbb{R} \to \mathbb{R}$ at a and empty entries are 0. Similarly,

$$\nabla F_2(\vec{x}_1) = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ \partial_R g(x,b) & & & \partial_L g(x,b) & 0 \\ & & & & 1 \end{pmatrix}$$

$$\nabla F_3(\vec{x}_2) = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ & \partial_R h(y,x) & & \partial_L h(y,x) & 0 \end{pmatrix}$$

where $\partial_R g$ is the partial derivative of g in the right argument and so on. Observe that the Jacobians are sparse due to each of the F_i's only changing one variable. We now calculate the vectors $\vec{dx}_i$. We use the notation $\vec{d}_i[a]$, $\vec{d}_i[b]$, $\vec{d}_i[x]$, $\vec{d}_i[y]$, and $\vec{d}_i[z]$ for the first, second, third, fourth, and fifth components of $\vec{dx}_i$ respectively. Pairing each vector with the matching line of our original program, we get

$$\begin{aligned}
x = f(a) \qquad & \vec{dx}_1 = (\vec{d}_0[a], \vec{d}_0[b], \partial f(a) \cdot \vec{d}_0[a], \vec{d}_0[y], \vec{d}_0[z]) \\
y = g(x,b) \qquad & \vec{dx}_2 = (\vec{d}_1[a], \vec{d}_1[b], \vec{d}_1[x], \partial_R g(x,b) \cdot \vec{d}_1[b] + \partial_L g(x,b) \cdot \vec{d}_1[x], \vec{d}_1[z]) \\
z = h(y,x) \qquad & \vec{dx}_3 = (\vec{d}_2[a], \vec{d}_2[b], \vec{d}_2[x], \vec{d}_2[y], \partial_R h(y,x) \cdot \vec{d}_2[x] + \partial_L h(y,x) \cdot \vec{d}_2[y]).
\end{aligned}$$

Observe that $\vec{d}_3[x] = \vec{d}_2[x] = \vec{d}_1[x]$ because the x components of the $\vec{dx}_i$'s are only changed when x is assigned to. Thus, we do not need to define a vector $\vec{dx}_i$ at each step, it is sufficient to only define one new scalar variable. We can therefore

rewrite the above as

$$\begin{aligned} x &= f(a) & dx &= \partial f(a) \cdot da \\ y &= g(x,b) & dy &= \partial_R g(x,b) \cdot db + \partial_L g(x,b) \cdot dx \\ z &= h(y,x) & dz &= \partial_R h(y,x) \cdot dx + \partial_L h(y,x) \cdot dy \end{aligned}$$

which exactly captures the forward mode algorithm. Namely, each line is paired with a derivative calculation using the partial derivatives, i.e. $y = f(x_1, x_2, \ldots, x_n)$ is paired with

$$dy = \sum_{i=1}^{n} \partial_i f(x_1, x_2, \ldots, x_n) \cdot dx_i.$$

for a fresh variable dy. Forward mode AD can also be viewed as arithmetic in the ring of truncated Taylor series [Griewank and A. Walther, 2008, Ch. 13].

For reverse mode, we observe that the matrix product can be transformed by transposition

$$\nabla(F_3 \circ F_2 \circ F_1)(\vec{x}_0)^\intercal = \nabla F_1(\vec{x}_0)^\intercal \times \nabla F_2(\vec{x}_1)^\intercal \times \nabla F_3(\vec{x}_2)^\intercal$$

and that this *reverses* the order of matrix multiplication. We can again calculate right-to-left,

$$\begin{aligned} X_3 &= \nabla F_3(\vec{x}_2)^\intercal \\ X_2 &= \nabla F_2(\vec{x}_1)^\intercal \times X_3 \\ X_1 &= \nabla F_1(\vec{x}_0)^\intercal \times X_2 \end{aligned}$$

and similarly opt for pre-multiplying by a vector $\vec{\delta x}_4$

$$\nabla(F_3 \circ F_2 \circ F_1)(\vec{x}_0)^\intercal \times \vec{\delta x}_4 = \nabla F_1(\vec{x}_0)^\intercal \times \nabla F_2(\vec{x}_1)^\intercal \times \nabla F_3(\vec{x}_2)^\intercal \times \vec{\delta x}_4$$

and thus we can define a sequence of intermediate vectors

$$\begin{aligned} \vec{\delta x}_3 &= \nabla F_3(\vec{x}_2)^\intercal \times \vec{\delta x}_4 \\ \vec{\delta x}_2 &= \nabla F_2(\vec{x}_1)^\intercal \times \vec{\delta x}_3 \\ \vec{\delta x}_1 &= \nabla F_1(\vec{x}_0)^\intercal \times \vec{\delta x}_2. \end{aligned}$$

The transposes of the Jacobians

$$\nabla F_1(\vec{x}_0)^\intercal = \begin{pmatrix} 1 & & \partial f(a) & & \\ & 1 & & & \\ & & 0 & & \\ & & & 1 & \\ & & & & 1 \end{pmatrix} \quad \nabla F_2(\vec{x}_1)^\intercal = \begin{pmatrix} 1 & & & & \\ & 1 & & \partial_R g(x,b) & \\ & & 1 & \partial_L g(x,b) & \\ & & & 0 & \\ & & & & 1 \end{pmatrix}$$

$$\nabla F_3(\vec{x}_2)^\intercal = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \partial_R h(y,x) \\ & & & 1 & \partial_L h(y,x) \\ & & & & 0 \end{pmatrix}$$

are also sparse. Let $\vec{\delta}_i[a]$, $\vec{\delta}_i[b]$, $\vec{\delta}_i[x]$, $\vec{\delta}_i[y]$, and $\vec{\delta}_i[z]$ for the first, second, third, fourth, and fifth components of $\vec{\delta x}_i$ respectively. Calculating with components, we see

$$\begin{aligned}
\vec{\delta x}_3 &= (\vec{\delta}_4[a], \vec{\delta}_4[b], \vec{\delta}_4[x] + \partial_R h(y,x) \cdot \vec{\delta}_4[z], \vec{\delta}_4[y] + \partial_L h(y,x) \cdot \vec{\delta}_4[z], 0) \\
\vec{\delta x}_2 &= (\vec{\delta}_3[a], \vec{\delta}_3[b] + \partial_R g(x,b) \cdot \vec{\delta}_3[y], \vec{\delta}_3[x] + \partial_L g(x,b) \cdot \vec{\delta}_3[y], 0, \vec{\delta}_3[z]) \\
\vec{\delta x}_1 &= (\vec{\delta}_2[a] + \partial f(a) \cdot \vec{\delta}_2[x], \vec{\delta}_2[b], 0, \vec{\delta}_2[y], \vec{\delta}_2[z])
\end{aligned}$$

and note that each line accumulates derivatives into the arguments of the function used based on the resulting variable. For example, $x = f(a)$ adds $f(a) \cdot \vec{\delta}_2[x]$ to $\vec{\delta}_2[a]$. We can use mutable variables δa, δb, δx, and δy initialized to 0 to perform the above calculation

$$\begin{aligned}
x &= f(a) \\
y &= g(x,b) \\
z &= h(y,x) \\
\delta y &\mathrel{+}= \partial_L h(y,x) \cdot \delta z, \quad \delta x \mathrel{+}= \partial_R h(y,x) \cdot \delta z \\
\delta x &\mathrel{+}= \partial_L g(x,b) \cdot \delta y, \quad \delta b \mathrel{+}= \partial_R g(x,b) \cdot \delta y \\
\delta a &\mathrel{+}= \partial f(a) \cdot \delta x
\end{aligned}$$

which is exactly reverse mode AD, modulo zeroing out mutable variables. Namely, each line has a corresponding stateful derivative update which accumulates into the mutable derivative associated with its arguments, i.e. $y = f(x_1, x_2, \ldots, x_n)$ is paired with

$$\delta x_1 \mathrel{+}= \partial_i f(x_1, \ldots, x_n) \cdot \delta y, \ \ldots, \ \delta x_n \mathrel{+}= \partial_n f(x_1, \ldots, x_n) \cdot \delta y$$

in the reverse order of the original program.

2.3 Practice

Automatic differentiation can be broadly categorized by mode (i.e. the specific algorithm) and implementation strategy. Some popular systems use a domain-

specific language (DSL) strategy where the user specifies a computation graph which is then the main object from which derivatives are calculated. The computation graph and resulting derivative graph are often optimized after construction. The DSL can either be fine-grained (operator level), or coarse-grained (computational module or model level). The operator level encompasses basic scalar operations such as addition and multiplication and tensor operations such as summing along a dimension and taking slices. On the other hand, the module level includes examples such as fully-connected neural networks and convolutional layers. Examples of fine-grained systems are Theano [Theano Development Team, 2016], CNTK [Seide and Amit Agarwal, 2016], and TensorFlow [Abadi, Ashish Agarwal, et al., 2015] and examples of coarse-grained systems are Torch7 [Collobert and Kavukcuoglu, 2011] and Caffe [Jia et al., 2014]. The computation graph approach, while useful, is usually limited to a subset of the host languages expressiveness. Thus, computation graph DSLs are generally considered to be *algorithmic* differentiation but not automatic differentiation, although this distinction is somewhat artificial.

Forward and reverse mode are the main categories of AD. There are also variations of these main modes; we list some examples.

- Sparse versions of forward and reverse mode take advantage the of structure of the program and requested results to perform less computation [Griewank and A. Walther, 2008, Ch. 7].

- Reverse mode has a memory footprint which is linear in the length of the calculation, and so there exists a checkpointed form which re-runs portions of the original program in exchange for a lower memory footprint [Griewank and A. Walther, 2008, Ch. 12][Hascoët and Araya-Polo, 2006].

- Forward and reverse mode are in fact extreme choices on a spectrum. A given first-order program can be viewed as directed acyclic graph with mathematical operations as nodes and data dependencies as edges. AD can then be defined in terms of edge and face eliminations on this graph, with forward and reverse mode being extremal choices in the order of elimination. [Griewank and A. Walther, 2008, Ch. 9]

- Forward mode can be derived from truncating Taylor series at their linear terms. Thus, truncating at higher-order terms allows for method similar

to forward mode which calculates higher-order derivatives. [Griewank and A. Walther, 2008, Ch. 13] [Barak A Pearlmutter and Jeffrey Mark Siskind, n.d.]

- Forward and reverse mode can also be layered on top of each other in order to compute higher-order derivatives [Barak A Pearlmutter and Jeffrey M Siskind, 2008][Betancourt, 2018].

Beyond the mode used, there are various non-DSL implementation strategies for AD. A useful categorization is into elemental, compiler-based, source transformations, and operator overloading [Baydin et al., 2018]. Elemental methods consist of programming with substitute mathematical functions defined by an AD library. Elemental AD is the simplest method to provide when the language does not support operator overloading. Examples include WCOMP and UCOMP [Lawson, 1971]. Compiler-based AD uses special purpose compilers to generate derivative code during compilation. Examples include Stalingrad [Barak A. Pearlmutter and Jeffrey Mark Siskind, 2008], Tangent [Merriënboer, Wiltschko, and Moldovan, 2017], SLANG [Thames, 1969], and PROSE [Pfeiffer, 1987]. Source transformation methods take program text and generates new program text containing the old code which also computes derivatives. Examples include ADIFOR [C. Bischof et al., 1996], ADIC [C. H. Bischof, Roh, and Mauer-Oats, 1997], and Tapenade [Pascual and Hascoët, 2008; Hascoët and Pascual, 2013]. Finally, operator overloading simply overloads the chosen mathematical functions to effectively perform the elemental method more ergonomically. Examples include ADOL-C [Andrea Walther, 2009], the *ad* package for Python[3], the *ad* package for Haskell[4], and the DiffSharp package for F# and C# [Baydin et al., 2018].

The last distinction we make cuts across our other categorizations. Some AD systems are *define-then-run*, or static, whereby a the program written is statically analyzed and transformed into a new program. Static approaches include DSL and source transformation techniques, and are often the fastest methods due to optimization opportunities. Other AD systems are *define-by-run*, or dynamic, where the derivative is calculated as the defined program runs. Dynamic approaches are usually slower but more flexible and interactive, and includes methods such as elemental and operator overloading techniques.

[3] `https://pypi.org/project/ad/`
[4] `https://hackage.haskell.org/package/ad`

2.4 Theory

The straight-forward derivation shown above for forward and reverse mode does not handle programming language features such as control flow, impurity, and higher-order functions. There exist libraries and languages which support these features, but features such as higher-order functions can cause subtle bugs [Manzyuk et al., 2012]. Hence, there has been a recent rise in interest for denotational semantics for AD.

The *differential lambda-calculus* of Ehrhard and Regnier extends lambda-calculus with differential operators [Ehrhard and Regnier, 2003], and these operators relate to substitution. Namely, for terms t, u and $i \in \mathbb{N}^{>0}$, we have a term $\mathrm{D}_i t \cdot u$ which is the partial derivative of t at argument i evaluated at u, and when $t \equiv \lambda x.s$, $\mathrm{D}_i(\lambda x.s) \cdot u$ reduces to $\lambda x.\left(\frac{\partial s}{\partial x} \cdot u\right)$. The operation $\frac{\partial s}{\partial x} \cdot u$ is a linear version of beta-reduction, which substitutes u for a linear occurrence of x in s. There may be multiple occurrences of x in s, and so differential lambda-calculus also allows finite linear combinations of terms. The subsequently introduced *differential categories* of [R. F. Blute, J. R. B. Cockett, and R. A. G. Seely, 2006] is a valid semantics for the differential lambda-calculus, and differential categories do in fact relate to smooth functions, but it is not clear to this author if a direct connection between the differential lambda-calculus and AD has been made. There have been other categorical models of differentiation which have found use for AD, such as cartesian differential categories [R F Blute, J R B Cockett, and R A G Seely, 2009], change actions [Alvarez-Picallo and Ong, 2019], tangent bundle categories [Rosický, 1984; J. R. B. Cockett and Cruttwell, 2014], and diffeological spaces [Huot, Staton, and Vákár, 2020].

The cartesian category framework used by [Elliott, 2018] was a mix of theory and practice, but was heavily influenced by category theory as an organizational framework. It can be seen as a family of semantics for first-order programs. Another simple extension to a pure base language was introduced by [Barak A. Pearlmutter and Jeffrey Mark Siskind, 2008]. It does not included denotational semantics, but it showed how to implement AD in a lightly augmented lambda-calculus.

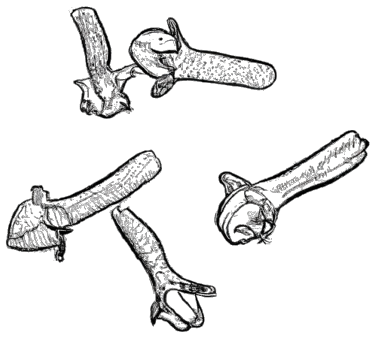
A fully fledged account of operational and denotational semantics for reverse mode AD has been given by [Abadi and Plotkin, 2020]. Their language is first order, but includes conditionals and recursion. They also prove adequacy theo-

rems, and this holistic approach is a first for AD. For the higher-order setting, the pair of works [Huot, Staton, and Vákár, 2020] and [Brunel, Mazza, and Pagani, 2020] provide denotational semantics and correctness in the higher order case under a different set-up. More recently, [Vákár and Smeding, 2022] have described and proved correct both forward and reverse mode AD in the presence of higher-order function. Furthermore, they implement their algorithms as a source-to-source transformation on Haskell.

Finally, [F. Wang, X. Wu, et al., 2018] shows how to implement reverse mode with delimited continuations. The work does not contain a denotational semantics, but their methods inspired our approach to the denotational semantics AD via algebraic effects and handlers.

Chapter 3

Effects and Handlers



Clove

Algebraic effects and handlers are a form of control flow and method of manipulating side-effects. Algebraic effects were introduced by [Plotkin and Power, 2001a] in order to incorporate operational semantics into Moggi's computational lambda-calculus based on monads. They showed their semantics is adequate for a non-recursive PCF[1] and how to use this proof to prove adequacy for specific effects. A second paper in [Plotkin and Power, 2001b] identified the generic effect formulation of algebraic effects, the more common formulation today. This identification strengthened the connection between algebraic effects and Lawvere theories. Plotkin and Power also speculated that the algebras of the monad induced by operations could be used to generalize their results to systems such as call-by-push-value (CBPV), which has indeed been achieved by [Kammar, 2014].

[1] Gordon Plotkin (Dec. 1977). "LCF considered as a programming language". In: *Theoretical Computer Science* 5.3, pp. 223–255. ISSN: 0304-3975. DOI: 10.1016/0304-3975(77)90044-5. URL: https://www.sciencedirect.com/science/article/pii/0304397577900445 (visited on 08/25/2023).

The idea of handlers for effects was introduced by [Plotkin and Pretnar, 2009]. They observe that handlers can be considered as destructors for effects while algebraic operations are constructors. We will give a short introduction on programming with effects and handler systems, an outline of the design space of such systems, and an overview of the languages we will use.

3.1 Programming with Effects and Handlers

For a tutorial on algebraic effects we suggest [Bauer, 2019] and for effect handlers we suggest [Pretnar, 2015]. We will give a short informal introduction to programming with effects and handlers by explaining examples in the Koka programming language [Leijen, 2014; Leijen, 2017].

Our first example will show how effects and handlers generalize exceptions:

```
1  effect ctl exception(info : string) : a
2
3  fun divide(x, y)
4    if y == 0 then exception("Divided by zero") else x / y
5
6  fun main()
7    (
8      handler
9        ctl exception(info) ->
10         print("Exception: " ++ info)
11   ) {
12     val a = divide(4, 2)
13     print("'a' is " ++ show(a) ++ ", ")
14     val b = divide(4, 0)
15     print("'b' is " ++ show(b))
16   }
```

Line 1 defines a new effectful operation `exception` which takes a string and produces a value of any type `a`. The function `divide` on lines 3 and 4 defines a division function which throws an exception when we divide by 0. The `main` function on lines 6 to 16 consists of two main parts. The first part is the handler on lines 7 to 11, which defines how to handle the exception effect in a similar manner to try-catch blocks in other languages. The second part on lines 11 to 16 is a block (or nullary function) containing the main body. We make use of printing throughout, which is a built-in effect.

When the `main` function executes, the main body of the function is passed to the handler, thus delimiting the scope of the handler. The body will then execute, with the call to `divide` on line 12 succeeding, and subsequently `'a' is 2,`

is printed. The second call to `divide` on line 13 tries to divide by 0, and thus the computation `exception("Divided by zero")` is executed. Control then propagates outward from this execution site to the handler, which consists of exactly one case. The argument of `exception` is bound to `info`, and the body of the case is executed, printing `Exception: Divided by zero`. Note that the print statement on line 15 is not executed. In Koka, handlers are first class, and so the following program is equivalent:

```
1  effect ctl exception(info : string) : a
2
3  val print-exception = handler
4    ctl exception(info) ->
5      print("Exception: " ++ info)
6
7  fun divide(x, y)
8    if y == 0 then exception("Divided by zero") else x / y
9
10 fun main()
11   with print-exception
12   val a = divide(4, 2)
13   print("'a' is " ++ show(a) ++ ", ")
14   val b = divide(4, 0)
15   print("'b' is " ++ show(b))
```

The `with` syntax is sugar for passing the following lines as thunk, i.e. `with f; <body>` is equivalent to `f(fn() {<body>})`.

The important difference between exceptions and effect handlers is that the latter can resume execution after an effect is handled. Consider the following program using an effect `default`, parameterized by the type `a` via `<a>`, which we will use to provide a default value:

```
1  effect<a> ctl default() : a
2
3  fun main()
4    with ctl default() ->
5      resume(1)
6    val a = default()
7    val b = default()
8    print(show(a : int) ++ ", " ++ show(b : int))
```

Line 1 declares our effect. The `main` function first installs a handler, and then uses the `default` effect. As `main` is executed, the handler is passed lines 6 to 8, which then begin to execute. The call to `default` on line 6 transfers control to the handler, which subsequently causes the body on line 5 to execute. The `resume` keyword is a function which has bound the rest of the computation, i.e. it is equivalent to

```
fun resume(x)
```

```
  val a = x
  val b = default()
  print(show(a : int) ++ ", " ++ show(b : int))
```

Thus, the program then continues from line 6 where the value of `a` is 1. The same process happens again for `b`, and so the program prints `1, 1`.

Different effects and handlers can be combined, for example our `exception` and `default` effects can be used together:

```
1  effect<a> ctl default() : a
2  effect ctl exception(info : string) : a
3
4  fun divide(x, y)
5    if y == 0 then exception("Divided by zero") else x / y
6
7  fun combined()
8    divide(default(), default())
9
10 fun main() : console () {
11   with ctl exception(info) -> print("Exception: " ++ info)
12   with ctl default() -> resume(0)
13   print(combined())
14 }
```

When the `main` function is run, the handlers on lines 11 and 12 are used. Note that they handle their effects through function call boundaries. The result of calling `main` is that `Exception: Divided by zero` is printed.

An important aspect of handlers is that they have scope. Consider the following program:

```
1  effect<a> ctl default() : a
2
3  fun main()
4    val a =
5      with ctl default() -> resume(1)
6      default()
7    val b =
8      with ctl default() -> resume(2)
9      default()
10   print(show(a : int) ++ ", " ++ show(b : int))
```

There are two handlers, the first on line 5 and the second on line 8. The calls to `default` on lines 6 and 9 are scoped by the first and second handler respectively. Thus, calling main results in `1, 2` being printed. The ability to give an effect or program different interpretations through the use of handlers is one the key features of effect and handlers systems, as it aids in composition, abstraction, and reusability. Likewise, the scoped nature of effects and handlers allows interpretations to be local and more tractable to reason about.

Handlers can model state. The following implements state in the standard state-passing function style:

```
1  effect state<s>
2    ctl get() : s
3    ctl put(s : s) : ()
4
5  val runState = handler
6    return(x) -> fn(s) {(x, s)}
7    ctl get() -> fn(s) {resume(s)(s)}
8    ctl put(s') -> fn(_) {resume(())(s')}
9
10 fun comp()
11   val x = get()
12   put(x + 1 : int)
13   val y = get()
14   put(y + y : int)
15   get()
16
17 fun main()
18   val res =
19     with runState
20     comp()
21   res(1)
```

We begin by defining the `state` effect on lines 1 to 3. The `runState` handler on lines 5 to 8 produces a function which threads the state through the computation. On line 6 we use a `return` clause, which is matched upon when the computation passed to the handler terminates in a value. Values embedded into computations are essentially the base case of computations. The function `comp` on lines 10 to 15 uses our effect, and the function `main` calls the function created by handling `comp` with `runState`, giving a result of `(4, 4)`.

Our last example shows the interesting control flow achievable with effects and handlers. Using the same `state` effect, consider the following handler:

```
val logState = handler
  return(x) -> fn(_) {(x, [])}
  ctl get() -> fn(s) {resume(s)(s)}
  ctl put(s') -> fn(_) {val (x, ss) = resume(())(s'); (x, Cons(s', ss))}
```

We now log the state every time it is changed. In the case where we handle `put`, note that we must run the resumption in order to get a list of past changes before we can place the new state at the head of the list. Running `main` with `logState` instead of `runState` produces `(4, [2, 4])`. Note that the early state of 2 has been added to the list *after* the later state of 4. This behavior is due to executing code after the resume statement, and will be integral to reverse mode AD.

3.2 Design Space

There are many different design choices for effects and handlers. We do not aim to provide a full survey, and so will only cover a few of the most important differences which will be visible across the four languages we have chosen:

Typed vs. Untyped: a type system may have specific features for effects;

Shallow vs. Sheep vs. Deep: shallow handlers do not reinstall any handler, sheep must install a handler, and deep handlers install themselves;

Forwarding vs. Non-forwarding: effect handlers may be responsible for every effect in their scope or only a subset while forwarding the rest;

Static vs. Generative: it may be possible to generate new effects or effect instances dynamically during program execution;

Single-shot vs. Multi-shot: the continuation captured for the handler may have restrictions on the number of times that it is used;

Effect type systems: there are many variations of effect type systems such as different levels of polymorphism and the underlying type theory; and

Manipulation of effects: when multiple instances of one effect exist, there are differing approaches for encapsulation and management.

The languages which we will use cover many of the above considerations, thus showing that implementing AD does not overly depend on any one choice. However, some features benefit some algorithms. Additionally, type systems help ensure correctness and can help avoid a class of AD specific errors.

Typed vs. Untyped

It is not necessary to have an effect type system in an effect and handler system. One advantage of not having an effect type system is that in some cases this allows one to macro translate between effect handlers and other control primitives as shown by [Forster et al., 2019]. In their work, they consider effects and handlers, monadic reflection, and delimited control without answer-type modification. All three can be macro translated in the untyped setting. When given

type systems, they provide proofs of non-existence for some translations, and notably the theorems rely on the specific type systems defined.

Type systems are effective at recording static information. For effect and handler systems, they can be used to indicate effects which may occur or must occur. Furthermore, type systems can provide guarantees for which effects are handled. They can also be used to ensure that there are no unhandled effects at the top level of the program. We will examine different type systems further on.

Shallow vs. Sheep vs. Deep

A continuation is captured when an effect is handled. In analogy to various delimited control operators, the handler may or may not be reinvoked to handle the remaining effects produced when the continuation is called. There are three usual choices, namely shallow, sheep, and deep. Shallow handlers are not reinvoked, sheep handlers must specify a handler for the continuations effects but it need not be the same handler, and finally deep handlers are always reinvoked. Shallow and deep handlers are inter-expressible up to some administrative reductions given an adequate background language, as shown by [Hillerström and Lindley, 2018]. Expressing shallow handlers using deep handlers is not very succinct, but deep handlers have the advantage of an inbuilt inductive structure.

Forwarding vs. Non-forwarding

A valuable feature of user-defined effects and handlers is modularity and multiple effects. Given multiple distinct effects, it is often useful for a handler to only handle a few effects or only one effect. A system with forwarding handlers allows a handler to automatically leave unwanted effects unhandled, delegating them to the enclosing handlers. Non-forwarding handlers intercept all effects. These handlers can of course re-perform the same effect they have handled, thus modelling forwarding handlers, but this can be error prone by requiring the programmer to manually add forwarding code to all handlers.

Single-shot vs. Multi-shot

The body of a handler has access to the delimited continuation created during effect handling. There can be restrictions on the use of this continuation, often

the restriction is that it can be used at most or exactly once. Such a restriction is termed single-shot, in contrast to multi-shot. The single-shot restriction may allow for more efficient implementations, or may be required for correctness in the presence of other language features or compiler optimizations. On the other hand, multi-shot handlers are more powerful, allowing for effects such as backtracking, non-determinism, and memoization. However, they require an overhead due to the delimited continuation being retained.

Static vs. Generative

Effects can be specified statically, i.e. at compile time, or can be generated at runtime. Static effects can aid in reasoning and optimization during compilation. For instance, effects can be declared to be single-shot, allowing for the generation of more efficient code. Generative effects can provide extra flexibility and power. For example, local state with reference cells can be implemented in some systems supporting generative effects.

Effect type systems

There are many different approaches to effect type systems. The essential responsibility of such a system is to record what effects may or must occur. Most systems are based on row or record types, examples of such include systems such as [Rémy, 1994], [Berthomieu and le Moniès de Sagazan, 1995], and [Leijen, 2005]. The system of Rémy uses a type level function from field labels to either a type of said field or a singleton type denoting absence, and this system allows records to have principal types. Berthomieu and le Moniès de Sagazan give a system of tagged types combining aspects of row and record types which allows polymorphism. Finally, the approach of Leijen allows duplicate labels, which is particularly useful for effects. Effect type systems can also support effect polymorphism, allowing type signatures to quantify over effects. For example, a mapping function on lists will only use the effects of its function argument, and so it is desirable to have a type signature recording this. Finally, it is possible in some systems for the effect operations themselves to be polymorphic. We will explore relevant features in more detail later.

Manipulating of effects

Managing effects used in a program is important for compositionality, abstraction, and usability. One method of manipulation is dynamically generating first-class instances, in a similar way to reference cells. The instances can be kept private, and thus limit what code can call the generated effect. In languages with type systems, handling an effect can remove the effect from the effect type. The code using the result of the handler is thus unaware and unable to handle the removed effect, enforcing encapsulation. Such languages can also allow duplicate effects at the type level, which often occurs when nested programs use the same effect. Type and term level masking (or injection) can reorder or combine these duplications, ensuring that the correct handler is invoked. Finally, some languages introduce named handlers to allow the differentiation between handlers of the same effect. Names can be used in either a lexical or dynamic fashion.

Note 3.2.1. One design consideration we have not covered is that of *scoped effects* introduced in [N. Wu, Schrijvers, and Hinze, 2014]. In many effect handler languages, effectful operations can take non-ground values as arguments, and thus can take suspended computations as arguments. When a handler handles these effects, there is no requirement that the handling of computations passed as arguments are handled by the selfsame handler. Scoped effects and handlers of them [Yang et al., 2022] help structure such handlers, aiding reasoning. None of the languages we will use support scoped effects, but some of our advanced AD mode implementations involve operations which have computations as arguments. Thus, scoped effects and handlers may be helpful in better structuring these implementations.

3.3 Overview of Languages Used

We will use four languages to implement a variety of AD algorithms: Frank [Lindley, McBride, and McLaughlin, 2017; Convent et al., 2020], Eff [Bauer and Pretnar, 2014; Bauer and Pretnar, 2015], Koka [Leijen, 2014; Leijen, 2017], and OCaml [K. Sivaramakrishnan et al., 2021; Leroy et al., 2022]. Each language has different design choices for effects and handlers, allowing us to examine their impact by comparing and contrasting.

3.3.1 Frank

The Frank language was created in [Lindley, McBride, and McLaughlin, 2017] and extended by [Convent et al., 2020]. It compiles to an interpreted effect and handler language called Shonky [McBride, 2023]. Frank is a strict functional programming language inspired by CBPV with a bidirectional effect type system. In Frank handlers are shallow, handlers are forwarding, continuations are multi-shot, and effects are static. Uniquely among our considered languages, Frank generalizes handlers to so-called multi-handlers, which handle multiple effectful computations at once. We have included Frank due to its unique type system and inclusion of shallow handlers.

We will examine Frank's type system through various examples taken from [Lindley, McBride, and McLaughlin, 2017]. Let us define a state effect

```
interface State S =
    get : S
  | put : S -> Unit
```

parameterized by a type `S` with operations `get` and `put`. An example program using this effect is:

```
fst : {X -> Y -> X}
fst x y = x

postAcc : {Int -> [State Int] Int}
postAcc x = fst get! (put (get! + x))
```

Curly braces denote computations, which functions are in Frank. The function `fst` evaluates all its arguments left-to-right, and nullary computations like `get` are evaluated with `!`. Thus, the function `postAcc` gets the state and then accumulates its argument into the state. The type signature of `postAcc` contains the row `[State Int]`, meaning that the calling context must provide *at least* the effect `State Int`. The standard state-passing handler for state is defined as:

```
state : {S -> <State S> X -> X}
state _ x             = x
state s <get -> k>    = state s (k s)
state _ <put s -> k> = state s (k unit)
```

Frank combines handlers and functions into one construct. The first argument of `state` is the state being passed, and none of the effects of this argument are handled. The second argument is the computation being handled, specifically only the effect `State S` is handled, recorded in the type as `<State S>`. Each operation is matched on, and the continuation is captured as `k`. As handlers in Frank are

shallow, the handler then reinvokes itself.

Handlers are invoked by application, in Frank there is no distinction between functions and handlers. To wit, given

```
index : {List X -> List (Pair Int X)}
index xs = state 0 (map {x -> pair (postAcc 1) x} xs)
```

the computation evaluates as:

```
index "abc"  ⇒  [( pair 0 'a'), (pair 1 'b'), (pair 2 'c')]
```

Note that the `State S` effect (where `S` is instantiated to `Int`) does not appear in the type signature of `index`. This is because the handler `state` handles it.

The definition of `index` uses the `map` function, which is defined as

```
map : {{X -> Y} -> List X -> List Y}
map f []        = []
map f (x :: xs) = f x :: map f xs
```

The function `f` has no visible effects, even though in `index` the argument is effectful. The above type signature is actually shorthand for

```
map : {{X -> [e|] Y} -> List X -> [e|] List Y}
```

where `e` is an effect variable representing a row. Thus, `map` is effect-polymorphic, and requires exactly the effects of its argument as expected.

Frank effect operations can be polymorphic, for example the aborting effect is

```
interface Abort = abort X : X
```

where `abort` is a polymorphic nullary effect. Furthermore, ML-style references are possible with the following definition

```
interface RefState =
    new X   : X -> Ref X
  | read X  : Ref X -> X
  | write X : Ref X -> X -> Unit
```

The above effect is a built-in effect and is automatically handled by the interpreter in a top-level computation.

Another feature of Frank is the ability to have multiple instances of the same effect in the effect type and manipulate the handling of such duplicates. Consider the following two functions

```
sqrt: {Int -> [Abort] Int}
parseInt: {String -> [Abort] Int}
```

which computes a square root (and aborts on negative numbers) and parse an integer (and aborts on invalid input) respectively. We can combine these two functions

```
conflatedComp: {String -> [Abort] Int}
conflatedComp s = sqrt (parseInt s)
```

where the computation aborts if the input cannot be parsed or if it is successfully parsed but is negative. The effect type `[Abort]` contains one instance of the effect `Abort`, thus a handler would only see one kind of `abort` command. We can differentiate between the two different types of failure using a *mask* or *adaptor*, resulting in

```
distinctComp: {String -> [Abort, Abort] Int}
distinctComp s = sqrt (<Abort> (parseInt s))
```

The mask `<Abort>` changes the effect type `[Abort, Abort]` to `[Abort]` by removing the inner/right instance. The type system of Frank says that an operation always corresponds to the innermost effect instance, and so we have now distinguished between the two failure modes. Frank has more effect manipulation features which we will explain when they arise.

Finally, we note that Frank does not have floating point operations, the operator `:=` for setting references, or the operator `@` for getting reference values. Thus, we have patched Frank to include these features. The patch can be found in Appendix A, and should be applied to commit `b5585a050221abfe03e5b6824cd1dd2b516a27f2`. Our patch builds upon the patch for adding floating point operations created by Leo Poulson[2].

3.3.2 Eff

The Eff language was created in [Bauer and Pretnar, 2014; Bauer and Pretnar, 2015]. The 3.0 version of the language matches the cited papers, whereas the newest 5.0 version has some differences. Thus, we will use the 3.0 version and shall mean this version when Eff is referred to unqualified.

Eff is an ML-style functional programming language with an interpreter written in OCaml. It is a typed language, and while the specification in [Bauer and Pretnar, 2014] contains effect annotations in types, the implemented language does not, as the effect system can infer such annotations. In, Eff, handlers are

[2] `https://github.com/frank-lang/frank/pull/6`

deep, handlers are forwarding, continuations are multi-shot, and effect instances can be dynamically created. Operations in Eff cannot be polymorphic.

Let us consider some Eff example code. We take the definition of the state effect from the built-in definitions

```
type 'a ref = effect
  operation lookup: unit -> 'a
  operation update: 'a -> unit
end
```

where the effect `ref` is parameterized by the variable `'a`. Effect instances are first-class values in Eff, and so when we define our handlers, a value of type `'a ref` will be an argument. The standard state handler is defined as

```
let state r x = handler
  | val y -> (fun _ -> y)
  | r#lookup () k -> (fun s -> k s s)
  | r#update s' k -> (fun _ -> k () s')
  | finally f -> f x;;
```

were `r` is the effect instance and `x` is the initial state. Note that an operation, e.g. `lookup`, is associated to a specific instance, e.g. `r#lookup`. The `finally` clause is a post-processing step that applies the function created during handling to the initial state. Handlers are then invoked using a `with` clause, for example

```
let r = new ref in
  with state r x handle
    <computation>
```

where the `new` keyword is used to create a new `ref` instance. A unique feature of Eff is resources based on handlers. The function which creates a fresh mutable state references is defined as

```
let ref x =
  new ref @ x with
    operation lookup _ @ x -> (x, x)
    operation update y @ _ -> ((), y)
  end
```

with the special `@` syntax. When an operation corresponding to a `ref` instance created using this function is invoked, the above handler is invoked. The handler has a state, initialized to the argument `x`, attached to it, which is the variable after `@` in each case. A pair consisting of the operations result and the new state is then returned.

The masking construct of Frank is not necessary in Eff because Eff has no effect type system in the implementation. The use case for masking can easily be achieved by using the operation corresponding to the desired instance, which

would need to be passed along to the call site. We will later see that coding discipline and helper functions can aid composability by closely matching the creation and use of instances.

Eff has small and big step operational semantics, as well as denotational semantics which is sound and adequate with respect to the operational semantics. Furthermore, the denotational semantics is similar to the approach we will use for correctness proofs. Additionally, Bauer and Pretnar provide an induction principle for reasoning about effects which is again similar to our proofs. It is for these reasons that we include Eff as it most closely matches our mathematical considerations.

3.3.3 Koka

The Koka language was created in [Leijen, 2014; Leijen, 2017]. It is a strongly-typed functional language with an effect type system which has an interpreter and can compile to C, C#, Javascript, and Wasm. In Koka, handlers are deep by default, handlers can be forwarding or non-forwarding, continuations can be multi-shot, and effects are static. There is undocumented support for shallow handlers and named handlers (giving dynamic effects), but we will make use of neither. Furthermore, Koka lets the programmer annotate effect definitions with continuation multiplicity, which can force continuations to be used exactly once. Operations in Koka can be polymorphic. Koka is included to exhibit different effect type system choices and to show the effect of tracking continuation re-use, as well as being quite performant for a research language.

Let us define a state effect

```
effect state<s>
  ctl get() : s
  ctl put(s : s) : ()
```

which is parameterized by the state type `s`. The `ctl` keyword denotes a "control" effect, meaning that the continuations available when handling this effect are multi-shot. The standard state handler is then defined as

```
val stateHandler = handler
  return(x) -> fn(s) {(x, s)}
  ctl get() -> fn(s) {resume(s)(s)}
  ctl put(s') -> fn(_) {resume(())(s')}
```

where we have defined `stateHandler` as a value using the `handler` keyword. The `resume` keyword gives the captured continuation as a first-class function. Note that the

type of this value is `forall<a,b>. (() -> (state<a>) b) -> ((s : a) -> (b, a))`, i.e. it is a function which takes in a thunk. Thus, handling is done via application. It is preferred in Koka to define handlers using the function syntax, and so we define

```
fun state(action : () -> <state<s>> a) : (s -> (a, s))
  with handler
    return(x) -> fn(s) {(x, s)}
    ctl get() -> fn(s) {resume(s)(s)}
    ctl put(s') -> fn(_) {resume(())(s')}
  action()
```

where we use the `with` syntax. This syntax is sugar for passing the following lines as thunk, i.e. `with f; <body>` is equivalent to `f(fn() {<body>})`. Furthermore, it is preferred in Koka not to use parameter passing, but to instead use lexically scoped local mutable state, resulting in

```
fun state'(init : s, action : () -> <state<s>> a) : <div> (a, s)
  var s := init
  with handler
    return(x) -> (x, s)
    ctl get() -> resume(s)
    ctl put(s') -> {s := s'; resume(())}
  action()
```

where `var` declares a locally mutable variable. The return signature now contains the `div` effect, which witnesses that local state can cause divergence. Programs without the `div` effect are terminating. Observe that there is no effect annotation for local state in the type signature. This is because Koka automatically erases universally quantified local state, justified using the rank-2 polymorphism approach of [Launchbury and Peyton Jones, 1994]. Koka also has a built-in global mutable state with ML-style references.

The handlers we have written so far are all non-forwarding handlers by virtue of the types we have given them. We can use effect polymorphism to turn them into forwarding handlers. Changing the type signature of `state'`, we get

```
fun state''(init : s, action : () -> <div,state<s>|e> a) : <div|e> (a, s)
  var s := init
  with handler
    return(x) -> (x, s)
    ctl get() -> resume(s)
    ctl put(s') -> {s := s'; resume(())}
  action()
```

which is parametric in the effect row `e`. The effect type system is based on the row type system of [Leijen, 2005]. This system allows for duplicate effects, and the leftmost instance corresponds to the innermost handler. Note that Koka can

also infer effects, and the above is the inferred one. Thus, in non-annotated Koka where handlers are written in the function style, handlers are forwarding.

Koka also has effect manipulation features, although the basic operations are less general than those of Frank. For any effect label `l`, where is an operation `mask<l> : (action: () -> e a) -> <l|e> a` which removes the effect `l` from the calling environment of `action`. For completeness, we note that there is another effect manipulation operation `mask behind<l> : (() -> <l|e> a) -> <l,l|e> a`, although we will not make use of it.

3.3.4 OCaml

The OCaml language was created in 1996 by Xavier Leroy, Jérôme Vouillon, Damien Doligez, and Didier Rémy[3], and is currently worked on by many others including [Leroy et al., 2022]. Effect handlers were added to OCaml as of version 5.0 by [K. Sivaramakrishnan et al., 2021] and others, and we mean this version when talking about OCaml unqualified.

OCaml is a strongly-typed programming language with multiple paradigms including functional, imperative, and object-oriented. It is the most mature and widely used out of the languages we consider. In OCaml, handlers can be sheep or deep, handlers are forwarding by convention, effects are generative via the module system, and continuations are single-shot. Note that what we refer to as sheep, OCaml refers to as shallow. It is possible to intercept all effects with a handler, but then one must provide a value of type $\forall a.a$, i.e. loop or throw an exception, or discontinue the continuation. There is currently no effect type system in OCaml 5.0, but various OCaml contributors wish to add one in the future[4]. Effects can be manipulated with the module system, giving a similar result as in Eff. Operations can be polymorphic. We have chosen to include OCaml due to its performance, maturity, and scale of use.

We will assume that the reader is familiar with OCaml sans effects and handlers, and so we will only cover these features. Let us consider again the state effect, where our code will be taken from the OCaml effects example repository[5]. We first define a some modules and a functor:

[3] `https://ocaml.org/about`, retrieved 03-08-2023

[4] Private communication.

[5] `https://github.com/ocaml-multicore/effects-examples/blob/b7f805b0b0708d4253a9c75d2d0dd29637cc6033/state.ml`, retrieved 03-08-2023

```
module type TYPE = sig
  type t
end

module type STATE = sig
  type t
  val get : unit -> t
  val set : t -> unit
  val run : init:t -> (unit -> 'a) -> t * 'a
end

module type CELL = functor (T : TYPE) -> STATE with type t = T.t
```

Modules of type TYPE contain the type of the state we wish to use, modules of type STATE provide the operations and an associated handler, and finally functors of type CELL let us generate new instances of our state effect. We then define an implementation of CELL using the state-passing style handler:

```
open Effect
open Effect.Deep

module StPassing : CELL = functor (T : TYPE) -> struct
  type t = T.t
  type _ Effect.t += Get : t Effect.t
  type _ Effect.t += Set : t -> unit Effect.t

  let get () = perform Get
  let set y  = perform (Set y)

  let run (type a) ~init (main : unit -> a) : t * a =
    match_with main () {
      retc = (fun res x -> (x, res));
      exnc = raise;
      effc = fun (type b) (e : b Effect.t) ->
        match e with
        | Get -> Some (fun (k : (b, t -> (t * a)) continuation) ->
            fun (x : t) -> continue k x x)
        | Set y -> Some (fun k ->
            fun (_x : t) -> continue k () y)
        | _ -> None
    } init
end
```

The definition begins by opening the standard library modules for effects. We then add our operations to the effect type `Effect.t`, an extensible variant type, and define the operations. Effects are handled using the `match_with` function, the first argument is the function to be called which produces effects, the second argument is the input to said function, and the third argument is the handler. A handler is a record type with three fields, `retc` gives the return clause for values, `exnc` says how

to handle exceptions, and the `effc` says how to handle each operation. We pattern match on the `Effect.t` type, where the final wildcard case is needed as `Effect.t` is an extensible variant. When `Some c` is returned, the effect is handled with `c`, and when `None` is returned, the effect is propagated onward.

Defining effects and handlers as above allow effects to be generative:

```
module IntCell1 = StPassing(struct type t = int end)
module IntCell2 = StPassing(struct type t = int end)

let main () =
  let x = IntCell1.get () + 1 in
  IntCell2.set x

let res = (* gives (0, (1, ())) *)
  IntCell1.run ~init:0 (fun () ->
    IntCell2.run ~init:0 main
  )
```

The above snippet creates two instances of the state effect and shows that they can be used together without interference. Thus, because modules are first-class in OCaml, so are effect instances.

Part II

Implementation

Chapter 4

Implementation of Standard AD Modes

Cumin

Automatic differentiation is best seen as a family of algorithms. We will make a distinction between the most well known and used variants and the lesser known and used ones. We shall refer to the former as *standard* and the later as *advanced*, but this dichotomy is informal. Each AD algorithm has been implemented in Frank, Koka, Eff, and OCaml. We will chiefly explain the algorithms in Frank, while highlighting features of each algorithm that are meaningfully different between implementations. Most such differences are related to the presence of type systems. The remaining implementations can be found in Appendix A.

Common among all the algorithms is an effect for *smooth functions*. A function $f : \mathbb{R}^n \to \mathbb{R}^m$ is called *smooth* when it has all partial derivative of all orders, meaning that the partial derivatives and Jacobian of such an f are also smooth. Thus, smooth functions are closed under differentiation, and are the natural class

to consider when creating compositional AD algorithms. Of course, we cannot express all smooth functions, and so we choose a subset of smooth functions closed under differentiation as a family. The minimal collection of smooth functions is addition, multiplication, and constant functions taking values in the non-negative integers. For simplicity, we choose only this minimal collection extended with unary negation, but as we will see sections 6.1 and 6.3, we can support many more smooth functions such as exponentials, logarithms and trigonometric functions. Note that if we would include, for example, the function $\sin(x)$, we would also need to include the function $\cos(x)$ because $\partial/\partial x(\sin(x)) = \cos(x)$. Alternatively, because $\cos(x) = \sin(x + \pi/2)$, we could leave $\cos(x)$ out, but in practice redundant functions are included for clarity and numerical considerations.

Thus, we begin by defining data types enumerating what family of functions we wish to use, split by the number of arguments each function takes. Next, we define our effect and helper functions. Note that Frank does not allow user defined infix operators, and so we use one letter functions instead.

Listing 4.1: Smooth effect (Frank)

```
1  data Nullary = constE Float   -- Nullary functions
2  data Unary   = negateE        -- Unary functions
3  data Binary  = plusE | timesE -- Binary functions
4
5  interface Smooth X =
6        ap0 : Nullary -> X           -- Apply a nullary function
7      | ap1 : Unary   -> X -> X      -- Apply a unary function
8      | ap2 : Binary  -> X -> X -> X -- Apply a binary function
9
10 c : Float -> [Smooth X] X
11 c i = ap0 (constE i)
12
13 n : X -> [Smooth X] X
14 n x = ap1 negateE x
15
16 p : X -> X -> [Smooth X] X
17 p x y = ap2 plusE x y
18
19 t : X -> X -> [Smooth X] X
20 t x y = ap2 timesE x y
```

The `Smooth` effect is parameterized by a type `x` which will be instantiated with a different type depending on the AD algorithm. This parameterization is one of the key aspects of the compositionality of our system. Each AD mode with instantiate `x` to some new type, but often this new type has a further free type variable, which in turn can be instantiated by another AD mode, thus allowing

the combination of modes. We also define the functions `c`, `n`, `p`, and `t` as shorthand for the smooth effects we can use.

Next, we define helper functions for calculating derivatives, used in defining handlers, as well as an example term. We also define a helper data type `Arg` to specify which argument of a binary function we are differentiating with respect to.

Listing 4.2: Helper functions (Frank)

```
22 der1 : Unary -> X -> [Smooth X] X -- ∂/∂x (u(x))
23 der1 negateE x = n (c 1.0)        -- ∂(−x)/∂x = −1
24
25 data Arg = L | R -- Derivative w.r.t. the left or right argument
26
27 der2 : Arg -> Binary -> X -> X -> [Smooth X] X -- ∂/∂x_arg (b(x_L, x_R)), for x_L = x, x_R = y
28 der2 L plusE  x y = c 1.0                      -- ∂(x + y)/∂x = 1
29 der2 R plusE  x y = c 1.0                      -- ∂(x + y)/∂y = 1
30 der2 L timesE x y = y                          -- ∂(x · y)/∂x = y
31 der2 R timesE x y = x                          -- ∂(x · y)/∂x = y
32
33 dder1 : Unary -> X -> [Smooth X] X -- ∂²/∂x² (u(x))
34 dder1 negateE x = c 0.0            -- ∂²(−x)/∂x² = 0
35
36 -- ∂²/(∂x_arg1 ∂x_arg2) (b(x_L, x_R)), for x_L = x, x_R = y, check if x_L ≡ x_R
37 dder2 : Bool -> Arg -> Arg -> Binary -> X -> X -> [Smooth X] X
38 dder2 true  L _ plusE  x y = c 0.0 -- ∂²(x + x)/∂x² = 0
39 dder2 true  R _ plusE  x y = c 0.0 -- ∂²(y + y)/∂y² = 0
40 dder2 false L L plusE  x y = c 0.0 -- ∂²(x + y)/∂x² = 0
41 dder2 false L R plusE  x y = c 0.0 -- ∂²(x + y)/∂x∂y = 0
42 dder2 false R L plusE  x y = c 0.0 -- ∂²(x + y)/∂x∂y = 0
43 dder2 false R R plusE  x y = c 0.0 -- ∂²(x + y)/∂y² = 0
44 dder2 true  L _ timesE x y = c 2.0 -- ∂²(x · x)/∂x² = 2
45 dder2 true  R _ timesE x y = c 2.0 -- ∂²(y · y)/∂y² = 2
46 dder2 false L L timesE x y = c 0.0 -- ∂²(x · y)/∂x² = 0
47 dder2 false L R timesE x y = c 1.0 -- ∂²(x · y)/∂x∂y = 1
48 dder2 false R L timesE x y = c 1.0 -- ∂²(x · y)/∂x∂y = 1
49 dder2 false R R timesE x y = c 0.0 -- ∂²(x · y)/∂y² = 0
50
51 -- 1 + x³ + (−y²)
52 term : X -> X -> [Smooth X] X
53 term x y = p (c 1.0) (p (t (t x x) x) (n (t y y)))
```

Lines 22 to 31 record information about the first derivatives of our functions, where we use the `Arg` data type to specify which argument of a binary function we are taking a partial derivative with respect to. Next, lines 33 to 49 do the same for second partial derivatives. The boolean argument to `dder2` records if the two arguments are definitionally equivalent, which is used to distinguish between cases such as $x \cdot x = x^2$ versus $x \cdot y$. Note that we have no derivative functions for

our nullary operation, it is constant and thus always has a derivative of 0. Finally, line 52 define our example term, namely $f(x, y) = 1 + x^3 + (-y^2)$. We specifically choose a simple term in order to aid with the explanation of our algorithms later through Frank's operational semantics. However, any arbitrary program with the same type could be used for `term`. For example, we could have a loop using mutable state (as we have in section 6.1) or we could define and call auxillary functions (as we do in section 6.3, see Appendix A).

When handling our smooth effect, we will ultimately need a numeric result. Thus, we define an evaluation handler that interprets our effectful operations as their built-in floating point counterparts. Note that we have instantiated the type variable in the `Smooth` effect with the `Float` type, allowing us to define `evluate` in terms of `Float` operations.

Listing 4.3: Evaluation (Frank)

```
1 include prelude
2 include smooth
3
4 evaluate : <Smooth Float> X -> X
5 evaluate x = x
6 evaluate <ap0 (constE i) -> k> = evaluate (k i)
7 evaluate <ap1 negateE x -> k>  = evaluate (k (-. x))
8 evaluate <ap2 plusE x y -> k>  = evaluate (k (x +. y))
9 evaluate <ap2 timesE x y -> k> = evaluate (k (x *. y))
```

The type of `evaluate` states that it handles exactly the effect `Smooth Float`, and performs no effects to do so. The value case on line 5 does nothing to the final value. Each of the matching cases on lines 6 to 9 takes the arguments of the effect and calculates the corresponding float result, passes it to the continuation, and then re-invokes the evaluate handler. Therefore, `evaluate` is implementable as a deep handler as well.

The `evaluate` handler will always be our top-level handler, and it is the only way to remove all `Smooth` interfaces. We shall evaluate an example program where `evaluate` is the top-level handler to illustrate how Frank executes and thus how effects and handlers behave. We will be paying special attention to how delimited continuations are captured. We will use underlining to show what term is currently at the focus of evaluation. Our initial program is below, and represents the term $1 + x^3 + -y^2$ evaluated at $x = 2$ and $y = 4$, which equals -7.

```
evaluate (p (c 1.0) (p (t (t 2.0 2.0) 2.0) (n (t 4.0 4.0))))
```

The current focus of evaluation is the command `c 1.0`.

```
evaluate (p (c 1.0) (p (t (t 2.0 2.0) 2.0) (n (t 4.0 4.0))))
```

The argument `1.0` is in normal form (i.e. fully evaluated). Therefore, we can handle the command `c 1.0`. The handling process begins by capturing the proper delimited continuation by incrementally capturing the stack of evaluation frames. We represent capturing by a wave underline.

```
evaluate (p (c 1.0) (p (t (t 2.0 2.0) 2.0) (n (t 4.0 4.0))))
evaluate (p (c 1.0) (p (t (t 2.0 2.0) 2.0) (n (t 4.0 4.0))))
```

We have now reached a handler, `evaluate`, for the command in focus. The command and enclosing evaluation frames (wave underlined) are the captured delimited continuation. The `ap0` case of `evaluate` is then matched to the command `c 1.0`, where `k` is bound to the continuation with `c 1.0` removed and `i` is bound to `1.0`. The bound variables `k` and `i` are then substituted into the corresponding body of the matching case of `evaluate`.

```
evaluate ({x -> (p x (p (t (t 2.0 2.0) 2.0) (n (t 4.0 4.0))))} 1.0)
```

The next step applies the continuation to `1.0`.

```
evaluate (p 1.0 (p (t (t 2.0 2.0) 2.0) (n (t 4.0 4.0))))
```

The focus of evaluation now moves to `t 2.0 2.0`, and a new delimited continuation is dynamically captured.

```
evaluate (p 1.0 (p (t (t 2.0 2.0) 2.0) (n (t 4.0 4.0))))
evaluate (p 1.0 (p (t (t 2.0 2.0) 2.0) (n (t 4.0 4.0))))
evaluate (p 1.0 (p (t (t 2.0 2.0) 2.0) (n (t 4.0 4.0))))
evaluate (p 1.0 (p (t (t 2.0 2.0) 2.0) (n (t 4.0 4.0))))
evaluate (p 1.0 (p (t (t 2.0 2.0) 2.0) (n (t 4.0 4.0))))
```

We have now again reached the `evaluate` handler, and this time match the `ap2` case, resulting in the following.

```
evaluate ({x -> (p 1.0 (p (t x 2.0) (n (t 4.0 4.0))))} (2.0 * 2.0))
evaluate ({x -> (p 1.0 (p (t x 2.0) (n (t 4.0 4.0))))} 4.0)
evaluate (p 1.0 (p (t 4.0 2.0) (n (t 4.0 4.0))))
```

Evaluation will continue as such until the final answer of -7 is calculated. We have now seen how the `evaluate` handler interprets `Smooth` commands with the builtin arithmetic operations. Even though `evaluate` is simple, it allows us to write our other handlers in a polymorphic fashion independent of `Float`.

Finally, there is no appreciable difference between the Frank implementation and any other.

4.1 Forward Mode

The derivation of forward mode in section 2.2 shows that each smooth function will now operate on a pair of values, namely the original value and its derivative. Thus, we define a data type of paired numbers, and make it parameterized to allow nesting of AD. The implementation of the forward mode handler is then a straightforward transcription of the algorithm, which calls for replacing the original operations with operations on paired numbers. Operator overloading is a common method for implementing forward mode AD because it is a local transformation, and thus well suited for overloading. Tail recursive deep handlers are similar to dynamic operator overloading in the sense that when the operation is used, the implementation is dynamically sought out, executed, and then control immediately returns to the call site. This analogy can be seen concretely in the example execution we performed for the `evaluate` handler. Thus, our forward mode handler is also a tail recursive deep handler. We will also define a helper function to differentiate a function of type The type

```
{(Paired X) -> [Smooth X, Smooth (Paired X)] (Paired X)} -> X -> [Smooth X] X
```

which shows that we remove one instance of the `Smooth` effect when differentiating.

Listing 4.4: Forward mode (Frank)

```
1  include prelude
2  include smooth
3
4  data Paired X = paired X X
5
6  v : Paired X -> X
7  v (paired x _) = x
8
9  dv : Paired X -> X
10 dv (paired _ dx) = dx
11
12 diff : <Smooth (Paired X)> Y -> [Smooth X] Y
13 diff x = x
14 diff <ap0 n -> k> =
15     -- v = n,   dv = 0
16     let r = paired (ap0 n)
17                    (c 0.0) in
18     diff (k r)
19 diff <ap1 u (paired x dx) -> k> =
20     -- v = u(x),   dv = ∂u(x) · dx
21     let r = paired (ap1 u x)
22                    (t (der1 u x) dx) in
23     diff (k r)
24 diff <ap2 b (paired x dx) (paired y dy) -> k> =
```

```
25      -- v = b(x,y),   dv = ∂_L b(x,y)·dx + ∂_R b(x,y)·dy
26      let r = paired (ap2 b x y)
27                     (p (t (der2 L b x y) dx)
28                        (t (der2 R b x y) dy)) in
29      diff (k r)
30
31  -- d f x = ∂f(z)/∂z (x)
32  d : {(Paired X) -> [Smooth X, Smooth (Paired X)] (Paired X)}
33   -> X -> [Smooth X] X
34  d f x = dv (diff (f (paired x (<Smooth> (c 1.0)))))
```

Lines 4 to 10 define the type of paired numbers over some base type `X` and projections for the components. Lines 12 to 29 define the `diff` handler. The value case on line 13 does no postprocessing. Each of the operation matching cases implements the operations calculated in section 2.2. Note that we use the `der` families of helper functions. Finally, each result is passed to the continuation `k` with subsequent effects being handled by `diff` as well. The function `d` on lines 32 to 34 takes a function `f` to be differentiated at `x` and uses `diff` to do so. Note the use of *adaptor* `<Smooth>`, which specifies that the effects of `c 1.0` correspond to `Smooth X` and not the innermost `Smooth (Paired X)`. Of course, this function could be evaluated outside the scope of the `diff` handler, rendering `<Smooth>` unnecessary. We will prove `diff` and `d` correct in section 8.2.

Let us analyze the types of `diff` and `d`. The type `<Smooth (Paired X)> Y -> [Smooth X] Y` of `diff` says that `diff` handles exactly the `Smooth (Paired X)` effect while requiring the calling environment to provide at least the `Smooth X` effect. By being parametric on `X` and using another `Smooth` effect while handling, the `diff` handler can be combined compositionally with other handlers. For example, we can instantiate `X` with `Paired Y`, allowing us to use `d` nested in itself as we do in section 4.5. The type

```
{(Paired X) -> [Smooth X, Smooth (Paired X)] (Paired X)} -> X -> [Smooth X] X
```

expresses that the function we are differentiating requires the `Smooth (Paired X)` and `Smooth X` effects, and removes the `Smooth (Paired X)`. It is also possible to give `d` the type

```
{(Paired X) -> [Smooth (Paired X)] (Paired X)} -> X -> [Smooth X] X
```

but we will see that this type poses issues for compositionality. Finally, we expect the user to always define functions of the form

```
X -> [Smooth X] X
```

i.e. those which are parametric in `X`, similar to `term`, which can be passed to `d`.

We will evaluate an example program similar to our previous one. The program will represent the same mathematical term $1 + x^3 + -y^2$ evaluated at $x = 2$

and $y = 4$, but additionally we shall be calculating the derivative with respect to x at this point, which is 12. This is achieved by setting x to `paired 2.0 1.0` and y to `paired 4.0 0.0`, where x has its second component set to 1 to treat it as the differentiated variable and y has its second component set to 0 to treat it as a constant.

```
evaluate (diff (
  p (c 1.0) (p (t (t (paired 2.0 1.0) (paired 2.0 1.0)) (paired 2.0 1.0))
               (n (t (paired 4.0 0.0) (paired 4.0 0.0))))
```

Evaluation begins as before, with the `c 1.0` command being in focus and a delimited continuation being captured.

```
evaluate (diff (
  p (c 1.0) (p (t (t (paired 2.0 1.0) (paired 2.0 1.0)) (paired 2.0 1.0))
               (n (t (paired 4.0 0.0) (paired 4.0 0.0))))
```

Note how the continuation captured is delimited by `diff` and not `evaluate`. This behavior is due to the effect typing system of Frank. There are two different instances of the `Smooth` interface available to the portion of the program being handled. By default, the innermost handler provides the instance for the operation being handled. As we shall see later, Frank provides constructs allowing us to select handlers other than the innermost one. The top case of `diff` is matched by `c 1.0` with the following result.

```
evaluate (
  let r = paired (ap0 (constE 1.0)) (c 0.0) in
  diff (
    {x -> (p x (p (t (t (paired 2.0 1.0) (paired 2.0 1.0)) (paired 2.0 1.0))
                  (n (t (paired 4.0 0.0) (paired 4.0 0.0))))))} r)


evaluate (
  let r = paired (c 1.0) (c 0.0) in
  diff (
    {x -> (p x (p (t (t (paired 2.0 1.0) (paired 2.0 1.0)) (paired 2.0 1.0))
                  (n (t (paired 4.0 0.0) (paired 4.0 0.0))))))} r)
```

We now have two `c` commands which will be be handled by `evaluate`, producing `paired 1.0 0` for `r`'s value. After handling, `r` will be substituted and the continuation applied.

```
evaluate (diff (
  p (paired 1.0 0.0) (p (t (t (paired 2.0 1.0) (paired 2.0 1.0)) (paired 2.0 1.0))
                        (n (t (paired 4.0 0.0) (paired 4.0 0.0))))
```

Evaluation will then continue in a similar manner for all remaining commands. Each command will first be handled by `diff`, and the commands in the body of each `diff` case handled by `evaluate`, eventually producing `paired -7.0 12.0`.

Lastly, the only appreciable difference between the Frank implementation and any other is that in Eff and OCaml we must pass the correct effect instance value as opposed being type system directed.

4.2 Stateful Reverse Mode

Recall our example straight-line program from section 2.2:

$$\begin{aligned} x &= f(a) && (1)\\ y &= g(x,b) && (2)\\ z &= h(y,x) && (3) \end{aligned}$$

The reverse mode algorithm applied to this program can be viewed as being applied recursively from the first line onwards, where the lines responsible for derivative accumulation are prepended

$$\begin{aligned} &x = f(a) && (1a)\\ &\cdots\\ &\delta a \mathrel{+}= \partial f(a) \cdot \delta x && (1b) \end{aligned}$$

$$\downarrow$$

$$\begin{aligned} &x = f(a) && (1a)\\ &y = g(x,b) && (2a)\\ &\cdots\\ &\delta x \mathrel{+}= \partial_L g(x,b) \cdot \delta y && (2b)\\ &\delta b \mathrel{+}= \partial_R g(x,b) \cdot \delta y && (2b)\\ &\delta a \mathrel{+}= \partial f(a) \cdot \delta x && (1b) \end{aligned}$$

$$\rightarrow$$

$$\begin{aligned} &x = f(a) && (1a)\\ &y = g(x,b) && (2a)\\ &z = h(y,x) && (3a)\\ &\delta y \mathrel{+}= \partial_L h(y,x) \cdot \delta z && (3b)\\ &\delta x \mathrel{+}= \partial_R h(y,x) \cdot \delta z && (3b)\\ &\delta x \mathrel{+}= \partial_L g(x,b) \cdot \delta y && (2b)\\ &\delta b \mathrel{+}= \partial_R g(x,b) \cdot \delta y && (2b)\\ &\delta a \mathrel{+}= \partial f(a) \cdot \delta x && (1b) \end{aligned}$$

where the ellipsis represents the program yet to be consumed. Stateful reverse mode, when implemented as a source-to-source program transformation, is very

non-local as the dataflow is reversed. The powerful control flow of effect handlers allows us dynamically create this reversal of dependencies. Stateful reverse mode works by creating a mutable cell for each value of the original program, and this cell accumulates contributions to the value's derivative. The method of accumulation is a generalized version of the *backpropagation* algorithm made prominent by machine learning. We define the datatype `Prop` for backpropagation where `Ref X` is a reference to a mutable cell containing a value of type `X`. The `reverse` handler handles the `Smooth` effect where the type variable `X` ist instantiated to `Prop`. We believe the handler version of this approach was first produced by K. C. Sivaramakrishnan, 2018, which itself was inspired by the approach of F. Wang, Zheng, et al., 2019 based on delimited control operators. Our implementation is essentially that of K. C. Sivaramakrishnan.

Listing 4.5: Stateful reverse mode (Frank)

```
1  include prelude
2  include smooth
3
4  data Prop X = prop X (Ref X) -- Value with mutable derivative
5
6  fwd : Prop X -> X
7  fwd (prop x _) = x
8
9  deriv : Prop X -> Ref X
10 deriv (prop _ r) = r
11
12 reverse : <Smooth (Prop X)> Unit -> [RefState, Smooth X] Unit
13 reverse x = x
14 reverse <ap0 n -> k> =
15     let r = prop (ap0 n) (new (c 0.0)) in -- r = n,   δr = 0
16     reverse (k r)                         -- Rest of program
17 reverse <ap1 u (prop x dx) -> k> =
18     let r = prop (ap1 u x) (new (c 0.0)) in -- r = u(x),   δr = 0
19     reverse (k r);                          -- Rest of program
20     dx := p @dx (t (der1 u x) @(deriv r))   -- δx += ∂u(x) · δr
21 reverse <ap2 b (prop x dx) (prop y dy) -> k> =
22     let r = prop (ap2 b x y) (new (c 0.0)) in  -- r = b(x, y),   δr = 0
23     reverse (k r);                             -- Rest of program
24     dx := p @dx (t (der2 L b x y) @(deriv r)); -- δx += ∂_L b(x, y) · δr
25     dy := p @dy (t (der2 R b x y) @(deriv r))  -- δy += ∂_R b(x, y) · δr
26
27 -- grad f x = ∂f(z)/∂z (x)
28 grad : {(Prop X) -> [RefState, Smooth X, Smooth (Prop X)] (Prop X)}
29     -> X -> [RefState, Smooth X] X
30 grad f x =
31     let z = prop x (new (c 0.0)) in
32     -- Set the output derivative to 1 to get derivative of f
33     reverse (deriv (f z) := <Smooth> (c 1.0));
```

```
34      @(deriv z)
```

Lines 4 to 10 define the data type for values and mutable derivatives. Lines 12 to 25 define the `reverse` handler, which makes use of the `der` functions, but is different from `evaluate` and `diff` from continuation reverse mode in two important ways. Firstly, the type of `reverse` shows that it requires access to the `RefState` interface of mutable state (a builtin effect of Frank that can be handled by the language implementation). Secondly, the body of the `ap1` and `ap2` cases contains code after the use of the captured delimited continuation `k`. We shall see these writes will occur in reverse order compared to the original dataflow. Finally, we define a helper function `grad` on lines 28 to 34. Importantly, on line 32 the output of the function being differentiated has its derivative set to 1, which allows the backpropagation to begin to accumulate.

To see how the handler works, we will evaluate the same term as before with a small change.

```
evaluate (grad ({x ->
  let y = c 4.0 in p (c 1.0) (p (t (t x x) x) (n (t y y)))
```

```
evaluate (
  let z = prop 2.0 (new (c 0.0)) in
  reverse ((deriv ({x ->
    let y = c 4.0 in p (c 1.0) (p (t (t x x) x) (n (t y y)))
  } z)) := (<Smooth> (c 1.0)));
  @(deriv z)
```

The term `new (c 0.0)` is handled first by `evaluate` for `c 0.0` (returning `0.0`), and the command `new 0.0` is handled by the Frank implementation and returns a new reference `<z>` whose cell contains `0.0`. The result is then substituted for `z`.

```
evaluate (
  reverse ((deriv ({x ->
    let y = c 4.0 in p (c 1.0) (p (t (t x x) x) (n (t y y)))
  } (prop 2.0 <z>))) := (<Smooth> (c 1.0)));
  @(deriv (prop 2.0 <z>))
```

Next, the anonymous function is applied to `prop 2.0 <z>`.

```
evaluate (
  reverse ((deriv (
    let y = c 4.0 in
    p (c 1.0) (p (t (t (prop 2.0 <z>) (prop 2.0 <z>)) (prop 2.0 <z>))
              (n (t y y)))
  )) := (<Smooth> (c 1.0)));
  @(deriv (prop 2.0 <z>))
```

The command `c 4.0` is handled by the `ap0` case of `reverse`, which as before creates a new reference `<r1>`, and thus `y` is substituted by `prop 4.0 <r1>`. The command `c 1.0` will create `<r2>`.

```
evaluate (
  reverse ((deriv (
    p (prop 1.0 <r2>)
      (p (t (t (prop 2.0 <z>) (prop 2.0 <z>)) (prop 2.0 <z>))
         (n (t (prop 4.0 <r1>) (prop 4.0 <r1>))))
  )) := (<Smooth> (c 1.0)));
  @(deriv (prop 2.0 <z>))
```

We have now reached the first interesting effect, which matches the `ap2` case of `reverse`. The captured delimited continuation is now explicitly underlined with waves.

```
evaluate (
  reverse ((deriv (
    p (prop 1.0 <r2>)
      (p (t (t (prop 2.0 <z>) (prop 2.0 <z>)) (prop 2.0 <z>))
         (n (t (prop 4.0 <r1>) (prop 4.0 <r1>))))
  )) := (<Smooth> (c 1.0)));
```

The result of `reverse` handling the command produces a new reference `<r3>`.

```
evaluate (
  reverse ((deriv (
    p (prop 1.0 <r2>)
      (p (t (prop 4.0 <r3>) (prop 2.0 <z>))
         (n (t (prop 4.0 <r1>) (prop 4.0 <r1>))))
  )) := (<Smooth> (c 1.0)));
  <z> := (p @<z> (t (der2L timesE 2.0 2.0) @(deriv (prop 4.0 <r3>))));
  <z> := (p @<z> (t (der2R timesE 2.0 2.0) @(deriv (prop 4.0 <r3>))));
  @(deriv (prop 2.0 <z>))
```

We see that the evaluation of the initial program has produced new expressions to be evaluated after the initial program finishes. The handling by `reverse` will eventually handle all effects meant for it, producing the following.

Listing 4.6: Reverse pass

```
evaluate (
  reverse (<r8> := (<Smooth> (c 1.0)));
  <r2> := (p @<r2> (t (der2L plusE 1.0 -8.0) @(deriv (prop -7.0 <r8>))));
  <r7> := (p @<r7> (t (der2R plusE 1.0 -8.0) @(deriv (prop -7.0 <r8>))));
  <r4> := (p @<r4> (t (der2L plusE 8.0 -16.0) @(deriv (prop -8.0 <r7>))));
  <r6> := (p @<r6> (t (der2R plusE 8.0 -16.0) @(deriv (prop -8.0 <r7>))));
  <r5> := (p @<r5> (t (der1 negateE 16.0) @(deriv (prop -16.0 <r6>))));
  <r1> := (p @<r1> (t (der2L timesE 4.0 4.0) @(deriv (prop 16.0 <r5>))));
  <r1> := (p @<r1> (t (der2R timesE 4.0 4.0) @(deriv (prop 16.0 <r5>))));
```

```
  <r3> := (p @<r3> (t (der2L timesE 4.0 2.0) @(deriv (prop 8.0 <r4>))));
  <z>  := (p @<z>  (t (der2R timesE 4.0 2.0) @(deriv (prop 8.0 <r4>))));
  <z>  := (p @<z>  (t (der2L timesE 2.0 2.0) @(deriv (prop 4.0 <r3>))));
  <z>  := (p @<z>  (t (der2R timesE 2.0 2.0) @(deriv (prop 4.0 <r3>))));
  @(deriv (prop 2.0 <z>))
)
```

The above is essentially a secondary program created by `reverse`, which performs backpropagation, also known as the *reverse pass* in contrast to the initial *forward pass*. This program is exactly what would be produced by a source-to-source stateful reverse mode transformation. Note that the definition of `reverse` could be changed to return a suspended computation, and thus capture this secondary program. In fact, this will be required for more advanced modes later. It could also be possible to use multi-stage programming by reifying the initial and secondary programs as a computation graph in the style of [F. Wang, Zheng, et al., 2019]. Their approach uses delimited continuations and combines normal execution with building an intermediate representation. As effects and handlers are essentially a structured use of delimited continuations, a similar story for other languages may be possible.

For differences between different languages, there is the previously seen type versus value dispatch of effects. The one new difference is that in Frank, Eff, and Koka we use global ML-style references, whereas in OCaml we use mutable record fields. Of course, OCaml has ML-style references, but these are essentially just one-member member mutable records.

4.3 Continuation Reverse Mode

Continuation based reverse mode is structurally very similar to forward mode, but performs the mathematical operations of stateful reverse mode. The pair of a value and its mutable derivative is replaced with a value and a *continuation* backpropagator. This continuation is defined such that it computes gradients of the value its paired with. We will show that this implementation is correct in section 8.3.

Listing 4.7: Continuation reverse mode (Frank)

```
1 include prelude
2 include smooth
3
4 data Prop X [e] = prop X {X -> [e|Smooth X] X} -- Value with backpropagator
```

```
5
6  v : Prop X -> X
7  v (prop x _) = x
8
9  dv : Prop X [e|] -> {X -> [e|Smooth X] X}
10 dv (prop _ dx) = dx
11
12 reverse : <Smooth (Prop X)> Y -> [Smooth X] Y
13 reverse x = x
14 reverse <ap0 n -> k> =
15     let r = prop (ap0 n)
16                  {z -> c 0.0} in
17     reverse (k r)
18 reverse <ap1 u (prop x dx) -> k> =
19     let r = prop (ap1 u x)
20                  -- δx += ∂u(x) · δr
21                  {z -> dx (t (der1 u x) z)} in
22     reverse (k r)
23 reverse <ap2 b (prop x dx) (prop y dy) -> k> =
24     let r = prop (ap2 b x y)
25                  {z ->
26                      -- δx += ∂_L b(x,y) · δr
27                    p (dx (t (der2 L b x y) z))
28                      -- δy += ∂_R b(x,y) · δr
29                      (dy (t (der2 R b x y) z))
30                  } in
31     reverse (k r)
32
33 -- grad f x = ∂f(z)/∂z (x)
34 grad : {(Prop X) -> [RefState, Smooth X, Smooth (Prop X)] (Prop X)}
35 d  : {(Prop X [e|]) -> [e|Smooth X, Smooth (Prop X [e|])] (Prop X [e|])}
36   -> X
37   -> [e|Smooth X] X
38 d f x = let bp = dv (reverse (f (prop x {z -> z}))) in bp (c 1.0)
```

Lines 4 to 10 define our data type for values and continuations. In Frank, data types that contain computations automatically have an effect variable passed in. We have made this explicit by adding the `[e]` to the definition. Thus, the continuation will be able to make use of more than just the `Smooth X` effect needed for propagation. Lines 12 to 31 define the continuation reverse mode handler. Finally, the helper function `d` is defined on Lines 34 to 38, wherein the function `f` to be differentiated at `x` is run under the `reverse` handler producing a propagator, which is then evaluated at 1.

Compared to forward mode, the reverse mode `d` has a type signature that explicitly passes the effect variable `e`. The type signature without this explicit variable would nevertheless be inferred by Frank. We include it for clarity.

Again, the only difference between the Frank implementation and any other is

that in Eff and OCaml we must pass the correct effect instance value as opposed being type system directed.

4.4 Taped Reverse Mode

Taped reverse mode AD is a common implementation strategy to remove the complicated control flow of stateful reverse mode. The crux of the strategy follows from the structure of the reverse pass created in stateful reverse mode. Listing 4.6 is an example of such a reverse pass. Note that each line has one of two structures depending on whether it was generated by a unary or binary operation. For example, the first two lines of the reverse pass of listing 4.6 are

```
<r2> := (p @<r2> (t (der2L plusE 1.0 -8.0) @(deriv (prop -7.0 <r8>))));
<r7> := (p @<r7> (t (der2R plusE 1.0 -8.0) @(deriv (prop -7.0 <r8>))));
```

which record the dependence of a addition operation (producing the derivative stored at `<r8>`) on it's two arguments (which have derivatives stored at `<r2>` and `<r7>`). The corresponding lines in the reverse mode handler in listing 4.5 are

```
dx := p @dx (t (der2 L b x y) @(deriv r));
dy := p @dy (t (der2 R b x y) @(deriv r))
```

We can use the uniformity of the reverse pass to easily defunctionalize it into a data structure called a *tape*.

The taped reverse mode handler is defined below.

Listing 4.8: Taped reverse mode (Frank)

```
1  include prelude
2  include smooth
3
4  data Name = name Int -- Name data type for fresh names
5
6  getValue : Name -> Int
7  getValue (name i) = i
8
9  interface Fresh = fresh : Name -- Generate fresh name
10
11 -- Handle fresh names by incrementing
12 incrementName : Int -> <Fresh> X -> Pair Int X
13 incrementName i x = pair i x
14 incrementName i <fresh -> k> = incrementName (i + 1) (k (name i))
15
16 data Prop X = prop X (Maybe Name) -- Value with named derivative
17
```

```
18 v : Prop X -> X
19 v (prop x _) = x
20
21 dv : Prop X -> Maybe Name
22 dv (prop _ dx) = dx
23
24 -- Defer an accumulation while recording dependency and value
25 data Pointer X = single Name X        -- Single dependency, save derivative
26                | double Name Name X X -- Double dependency, save derivatives
27
28 reverse : List (Pointer X)
29        -> <Smooth (Prop X)> Y
30        -> [Smooth X, Fresh] Pair (List (Pointer X)) Y
31 reverse tape x = pair tape x -- Return updated tape with value
32 reverse tape <ap0 n -> k> =
33     let r = prop (ap0 n) nothing in -- Calculate value, no dependency
34     reverse tape (k r)
35 reverse tape <ap1 u (prop x dx) -> k> =
36     let res = ap1 u x in -- Calculate value
37     let tr = case dx
38         { nothing -> -- No dependency
39             pair tape (prop res nothing)
40         | (just nx) ->
41             -- Do δx += ∂u(x) · δr later
42             pair (single nx (der1 u x) :: tape) (prop res (just fresh!))
43         } in
44     reverse (fst tr) (k (snd tr))
45 reverse tape <ap2 b (prop x dx) (prop y dy) -> k> =
46     let res = ap2 b x y in -- Calculate value
47     let tr = case (pair dx dy)
48         { (pair nothing nothing) -> -- No dependency
49             pair tape (prop res nothing)
50         | (pair (just nx) nothing) ->
51              -- Do  δx += ∂L b(x, y) · δr later
52             pair (single nx (der2 L b x y) :: tape) (prop res (just fresh!))
53         | (pair nothing (just ny)) ->
54             -- Do δy += ∂R b(x, y) · δr later
55             pair (single ny (der2 R b x y) :: tape) (prop res (just fresh!))
56         | (pair (just nx) (just ny)) ->
57             -- Do δy += ∂R b(x, y) · δr and δy += ∂R b(x, y) · δr later
58             pair (double nx ny (der2 L b x y) (der2 R b x y) :: tape)
59                  (prop res (just fresh!))
60         } in
61     reverse (fst tr) (k (snd tr))
62
63 initState : Int -> [Abort, RefState, Smooth X] List (Ref X)
64 initState 0 = []
65 initState n = (new (c 0.0)) :: initState (n - 1)
66 initState _ = abort!
67
68 foreachIndexed : Int -> List X -> {Int -> X -> [e|] Unit} -> [e|] Unit
69 foreachIndexed _ [] _ = unit
70 foreachIndexed i (x :: xs) f = f i x; foreachIndexed (i + 1) xs f
```

```
71
72  -- grad f x = ∂f(z)/∂z (x)
73  d : {(Prop X)
74   -> [Smooth X, Smooth (Prop X)] (Prop X)}
75   -> X
76   -> [Abort, RefState, Smooth X] X
77  d f x =
78      -- Get number of derivatives and deferred operations and result derivative
79      let res = <Abort, RefState> (incrementName 0 ( -- Start fresh names at 0
80          let z = prop x (just fresh!)
81          in reverse [] (<Fresh> (f z))
82      )) in
83      let m = fst res in          -- Number of names generated
84      let tape = fst (snd res) in -- Tape of deferred operations
85      let state = initState m in  -- Initialize collection of mutable derivatives
86      -- If result has no derivative, don't initialize anything to 1, else if it
87      -- has one initialize it to 1 for backpropagation.
88      let start = case (dv (snd (snd res)))
89          { nothing -> pair 0 (c 0.0)
90          | (just n) -> pair (getValue n) (c 1.0)
91          } in
92      (nth (fst start) state) := snd start; -- Do the initialization
93      -- Iterate through the tape with index and perform deferred operations
94      foreachIndexed 0 tape
95          { k pointer -> -- Account for the effect of the k-th derivative
96              case pointer
97              { (single nu vu) -> -- Do δu += v_u · δk
98                  let dk = @(nth (m - (k + 1)) state) in -- Tape is in reverse
99                  let du = @(nth (getValue nu) state) in
100                 (nth (getValue nu) state) := (p du (t vu dk))
101             | (double nl nr vl vr) -> -- Do  δl += v_l · δk and δr += v_r · δk
102                 let dk = @(nth (m - (k + 1)) state) in -- Tape is in reverse
103                 let dl = @(nth (getValue nl) state) in
104                 (nth (getValue nl) state) := (p dl (t vl dk));
105                 let dr = @(nth (getValue nr) state) in
106                 (nth (getValue nr) state) := (p dr (t vr dk))
107             }
108         };
109     @(nth 0 state) -- Derivative of x, was the first ‘fresh‘
```

Lines 4 to 14 define a replacement for global references, which we call `Name`. Due to the linear and incremental nature of references produced during reverse mode, names are merely integers that are incremented when a fresh one is required. The `incrementName` handler produces names in this manner and returns the used counter paired with the final result.

Lines 16 to 22 define the adapted `Prop` type. Instead of storing a reference, we possibly store a name. The change from definitely storing a derivative to optionally storing one is an optimization separate from the tape considerations. When a constant operation is encountered, we do not need to allocate a derivative

for it as its derivative is always 0. This lack of dependence is then propagated transitively through the rest of the computation.

Lines 25 to 26 define the `Pointer` data type. Our tape will consist of a list of `Pointer` values viewed as a stack, where `single` denotes the dependence of an operation on one argument and `double` similarly on two arguments. The `Name` components are pointers to elements recorded earlier in the tape, i.e. indices, and thus taking advantage of our definition of `Name` as a wrapped integer. The `x` components record the values to be accumulated later into the derivatives. For example, the lines

```
<r2> := (p @<r2> (t (der2L plusE 1.0 -8.0) @(deriv (prop -7.0 <r8>))));
```

would be recorded as

```
double (name 2) (name 7) (der2L plusE 1.0 -8.0) (der2R plusE 1.0 -8.0)
```

and would be the 8^{th} element of the tape.

Lines 28 to 61 define the taped reverse mode handler. The `reverse` handler has a `tape` parameter that is updated during the handler execution which records the reverse pass. The value case of the handler returns the final tape to be used in the accumulation of derivatives. The constant case on line 33 does not change the tape as its derivative is 0. When handling the other operations, we check to see if the arguments transitively depend on only constants, and thus have a `nothing` in the derivative component. If the derivative is not trivial, then we record this fact onto the front of the tape and continue execution.

Lines 63 to 70 define utility functions for creating a list of 0 initialized references and to iterate over a list while passing the current index.

Finally, lines 73 to 109 define the differentiation operator `d` which uses taped reverse mode. Lines 79 to 82 run the computation and produce the tape and a count of the number of names generated via the `incrementName` handler. We then initialize all derivatives on lines 85 to 92, with 0's for all intermediate values and 1 for the function's output. Lines 94 to 108 then iterate over the tape, performing the accumulation of derivatives based on the recorded dependencies. Finally, line 109 returns the derivative of the input to the function.

Thus, we have moved the accumulation of derivatives outside the handler. This change has some important implications. Foremost, the type of `reverse` in taped reverse mode no longer contains `RefState`, only the type of `d` contains `RefState`. While Frank does not take advantage of this change, Koka does. The derivative operator `d` can be implemented with local state, meaning the type in Koka is

```
forall<a,e>. (
  f : (prop<a>) -> <div,smooth<prop<a>>,smooth<a>|e> prop<a>,
  x : a
) -> <div,exn,smooth<a>|e> a
```

which does not contain any state effect[1]. Another important implication is that the `reverse` handler is now tail resumptive. In Koka, this can be taken advantage of for a large increase in performance by changing our smooth effect to be *linear*, i.e. the continuation captured during effect handling is used at most once. We will come back to this in our performance benchmarks. Eff does not have such a distinction between local and global mutable state. While vanilla OCaml does not have this distinction either, there are proposals for adding stack allocated values [Dolan and White, 2022][2].

The final difference between implementations stems from the relocation of state. The Koka and OCaml languages have have support for mutable arrays, which allow the accumulation of derivatives to be performed on contiguous memory while iterating over the tape.

4.5 Combined Modes

Each of the modes we have defined calculate the first derivative of the given function. Higher-order derivatives are also desired in practice, and are often achieved by nesting AD algorithms when possible. The modularity and compositionality of effects and handlers allows such nesting.

We will show how to calculate second derivatives using forward mode nested with itself. A basic example is

```
basic : {Float}
basic! = evaluate (d {y -> d {x -> t x (t x x)} y} (c 1.0))
```

which calculates $^{d}/_{dy}(^{d}/_{dx}(x^3)|_{x=y})|_{y=1} = {}^{d^2}/_{dy^2}(y^3)|_{y=1}$ correctly as 6. A more complex example is

$$\frac{d}{dx}\left(x \cdot \frac{d}{dy}\left(x+y\right)\Big|_{y=1}\right)\Bigg|_{x=1}$$

which equals 1. Note that x appears in the inner derivative, in which it is treated as a constant. Some systems do this incorrectly, resulting in the phenomenon

[1] `div` is the divergence effect, `exn` is the exception effect (the analogy to `Abort`)

[2] See `https://github.com/ocaml-flambda/ocaml-jst/blob/e3076d2e7321a8e8ff18e560ed7a55d6ff0ebf04/jane/doc/local-intro.md` for more information.

known as *perturbation confusion* [Jeffrey Mark Siskind and Barak A Pearlmutter, 2005]. Such a system would incorrectly evaluate the example to 2.

The nesting will make use of Frank's ability to dynamically determine which handler handles a command and help us avoid the incorrect answer. Recall that the type of `d` is

```
{(Paired X) -> [Smooth X, Smooth (Paired X)] (Paired X)} -> X -> [Smooth X] X
```

If we transcribe the example into Frank, we get:

```
evaluate (d {x -> t x (d {y -> p x y} (c 1.0))} 1.0)
```

Helpfully, this does not type check due to the type of `x` being `Paired Float` and the type of `y` being `Paired (Paired Float)` in the sub-computation `p x y`. The difference in types stems from using nested paired numbers, and is not a feature unique to effects and handlers. Attempting to fix this type error, we next consider

```
evaluate (d {x -> t x (d {y -> p (paired x (c 0.0)) y} (c 1.0))} 1.0)
```

which generates a new type error. The computation `c 0.0` is typed as returning `Paired (Paired Float)` whereas we require `Paired Float` as we are manually wrapping the result in another `Paired`. The type of `c` is `Float -> [Smooth X] X`. Unifying the effect `[Smooth X]` with the effect context of

```
[Smooth Float, Smooth (Paired Float), Smooth (Paired (Paired Float))]
```

provided by the inner `d` forces `x` to be unified with `Paired (Paired Float)` because the rightmost `Smooth` is `Smooth (Paired (Paired Float))`. Thus, an incorrect result of `Paired (Paired Float)` is produced. To fix this, we change our definition to

```
evaluate (d {x -> t x (d {y -> p (paired x (<Smooth> (c 0.0))) y} (c 1.0))} 1.0)
```

The first argument of `d` has type `{(Paired X) -> [Smooth X, Smooth (Paired X)] (Paired X)}`, and the *adaptor* `<Smooth>` causes the command `c 0.0` to be associated with `Smooth X` instance of `Smooth` and not the rightmost instance `Smooth (Paired X)`. Thus, one layer of `Paired` is removed as required. Finally, if the `d` had the type

```
{(Paired X) -> [Smooth (Paired X)] (Paired X)} -> X -> [Smooth X] X
```

then the program would also be rejected as `c 0.0` could not be associated with `Smooth X`

With these observations, we define an auxiliary function.

Listing 4.9: Lifting (Frank)

```
lift : X -> [Smooth X, Smooth (Paired X)] (Paired X)
lift x = paired x (<Smooth> (c 0.0))
```

This allows our example to be written as

```
evaluate (d {x -> t x (d {y -> p (lift x) y} (c 1.0))} 1.0)
```

which computes the correct value. Attempting to use `lift` in other positions

```
evaluate (d {x -> t x (lift (d {y -> p x y} (c 1.0)))} (c 1.0))
```

```
evaluate (d {x -> t x (d {y -> p (lift (p x x)) y} (c 1.0))} (c 1.0))
```

correctly produces errors due again to the effect system. It is possible to achieve the incorrect answer using `lift`

```
evaluate (d {x -> t x (lift (<Smooth> (d {y -> p x y} (c 1.0))))} (c 1.0))
```

which gives 2. However, it is obvious that the effects are being manipulated by the use of `<Smooth>`. Thus, when we only use the `lift` operation, we mitigate a class of AD errors. Koka's effect type system is similar enough to also benefit, but Eff and OCaml do not benefit as they have no effect system.

One important difference between Frank and Koka can be seen when trying to calculate

$$\frac{d}{dx}\left(x \cdot \frac{d}{dy}\left(y \cdot \frac{d}{dz}\left(z \cdot (y \cdot x)\right)\Big|_{z=1}\right)\Big|_{y=1}\right)\Big|_{x=1}$$

which evaluates to 4. Transcribing the above to Frank produces

```
evaluate (
  d {x -> t x (
      d {y -> t y (
          d {z -> t z (t y x)} (c 1.0)
        )} (c 1.0)
    )} (c 1.0)
)
```

which does not type check for the same reasons as before. However, we now require the innermost `x` to skip over two handlers, meaning `lift` does not suffice (and neither does any combination of `lift` with itself). The innermost effect context is

```
[ Smooth Float,
  Smooth (Paired Float),
  Smooth (Paired (Paired Float)),
  Smooth (Paired (Paired (Paired Float)))
]
```

Thus, we must define a new lift operation

Listing 4.10: Lifting multiple levels (Frank)

```
lift2 : X ->
```

```
  [Smooth X,
   Smooth (Paired X),
   Smooth (Paired (Paired X))
  ] (Paired (Paired X))
lift2 x = paired (paired x (<Smooth(s a b c -> s a)> (c 0.0)))
                 (<Smooth(s a b c -> s b)> (c 0.0))
```

which when used unifies `X` with `Paired Float`. The adaptors `<Smooth(s a b c -> s a)>` and `<Smooth(s a b c -> s b)>` can be understood as finite mappings. We take the three rightmost instances of `Smooth`, and bind them to `a`, `b`, and `c`. The remaining instances are bound as a row to `s`. We then create a new row as specified on the right hand side of the adaptor. Thus, `<Smooth(s a b c -> s a)>` retains `Smooth X` and `<Smooth(s a b c -> s b)>` retains `Smooth (Paired X)`. We can then write the correct Frank program

```
evaluate (
  d {x -> t x (
      d {y -> t y (
          d {z -> t z (t (lift y) (lift2 x))} (c 1.0)
        )} (c 1.0)
    )} (c 1.0)
)
```

which produces the correct answer. Koka does not have such adaptors built in. However, Koka's `mask` operation is the same as Franks basic adaptors of the form `<Smooth>`, and this is sufficient to create any general adaptor. The construction of a general adaptor from a `mask` operation is due to [Biernacki et al., 2017]. Nonetheless, their method requires creating intermediate thunks, and is thus much less desirable compared to having general adaptors.

Another form of perturbation confusion occurs when differentiating higher-order functions. [Manzyuk et al., 2012] define a family of generalized differentiation operators $\mathbb{D}$ based on forward mode AD such that given

$$s \colon \mathbb{R} \to (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R}$$
$$s\, u\, f\, x \coloneqq f\,(x + u)$$

and any $f \colon \mathbb{R} \to \mathbb{R}$ and $y \in \mathbb{R}$,

$$\mathbb{D}\, s\, 0\, f\, y = \frac{\partial}{\partial u}\,(s\, u\, f\, y)\big|_{u=0} = \frac{\partial}{\partial u}\,(f\,(x+u))\Big|_{u=0} = f'(y) = \mathbb{D}\, f\, y$$

and so by eta reduction we should have $\mathbb{D}\, s\, 0 = \mathbb{D}$. However, they examine how naive implementations compute the wrong answer in some cases. Using our forward mode `d` function, we can define an analogous operator

```
s : X -> {X -> [Smooth X] X} -> X -> [Smooth X] X
s u f x = f (p u x)

ds : {(Paired X) -> [Smooth X, Smooth (Paired X)] (Paired X)}
  -> X -> [Smooth X] X
ds f x = d {u -> s u f (lift x)} (c 0.0)
```

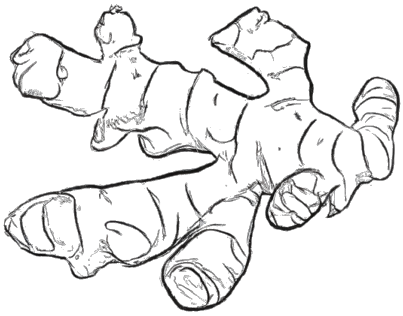
Testing `ds` on the function $f(x) = x^3$, so that $f''(x) = 6x$,

```
test : {Float}
test! = evaluate (ds {x -> ds {y -> t y (t y y)} x} (c 5.0))
```

gives the correct result of 30. Thus, our implementation does not suffer from their identified higher-order perturbation confusion [Manzyuk et al., 2012].

Chapter 5

Implementation of Advanced AD Modes



Ginger

Our advanced modes of AD will cover higher-order derivatives, checkpointed AD, and incorporating higher-order functions. Again, we will illustrate each algorithm in Frank, and when there is an important difference in the other languages we will highlight it. The remaining implementations can be found in Appendix A.

5.1 Higher-Order Derivatives

Higher-order derivatives can be calculated by nesting AD modes. However, this can lead to redundant computation. For example, when using forward mode on forward mode AD to calculate second derivatives of a unary function, the data type used in Frank is `Paired (Paired Float)` which consists of four `Float` values. Two of these values are the first derivatives, and thus are identical and redundant. The

solution is to use triple a of numbers consisting of the original value, the first derivative of the value, and the second derivative of the value. For a comprehensive explanation of this and other similar extensions, see [Betancourt, 2018]. Betancourt's methods involve generalized Taylor series and how they are transformed by smooth maps. These generalized Taylor series can then be truncated to a finite prefix of powers (such as two for forward mode and three for second derivative forward mode) while maintaining mathematical correctness.

The implementation is analogous to forward mode.

Listing 5.1: Second derivative forward mode (Frank)

```
include prelude
include smooth

data Triple X = triple X X X

v : Triple X -> X
v (triple x _ _) = x

dv : Triple X -> X
dv (triple _ dx _) = dx

ddv : Triple X -> X
ddv (triple _ _ ddx) = ddx

diff : <Smooth (Triple X)> Y -> [Smooth X] Y
diff x = x
diff <ap0 n -> k> =
    let r = triple (ap0 n)
                   (c 0.0)
                   (c 0.0) in
    diff (k r)
diff <ap1 u (triple x dx ddx) -> k> =
    let r = triple (ap1 u x)
                   (t (der1 u x) dx)
                   (t (dder1 u x) ddx) in
    diff (k r)
diff <ap2 b (triple x dx ddx) (triple y dy ddy) -> k> =
    let r = triple (ap2 b x y)
                   (p (t (der2 L b x y) dx)
                      (t (der2 R b x y) dy))
                   (p (p (t (dder2 false L R b x y) (t dx dy))
                         (p (t (t (c 0.5) (dder2 false L L b x y)) (t dx dx))
                            (t (t (c 0.5) (dder2 false R R b x y)) (t dy dy))))
                      (p (t (der2 L b x y) ddx)
                         (t (der2 R b x y) ddy))) in
    diff (k r)

dd : {(Triple X) -> [Smooth X, Smooth (Triple X)] (Triple X)}
  -> X
```

```
40    -> [Smooth X] X
41  dd f x = t (c 2.0) (ddv (diff (
42           f (triple x (<Smooth> (c 1.0)) (<Smooth> (c 0.0)))
43      )))
44
45  lift : X -> [Smooth X, Smooth (Triple X)] (Triple X)
46  lift x = triple
47      x (<Smooth(s a b -> s a)> (c 0.0)) (<Smooth(s a b -> s a)> (c 0.0))
```

Lines 4 to 13 define the `Triple` data type which contains a value paired with its first and second derivatives. Lines 15 to 36 define the `diff` handler which implements the proper arithmetic based on truncated Taylor series. Lines 38 to 47 define the `dd` function which performs the differentiation, and finally we include an analogous `lift` function.

It is also possible to calculate higher-order derivatives in reverse mode directly in one reverse pass. As an example, we implement the algorithm of [M. Wang, 2022] which calculates the entire Hessian of the given function. The implementation can be found in listing A.22. The high-level structure is similar to the standard stateful reverse mode, in that there is a forward pass followed by a reverse pass to accumulate the requisite derivatives. However, computing the Hessian requires much more information, especially in the formulation of M. Wang which dynamically minimizes the saved values to only those required. Effects and handlers help manage the state by associating it with the handler, and therefore giving a more local view of the complex computation and data dependencies.

5.2 Checkpointing

Stateful reverse mode AD allocates a mutable cell for each operation, which during the course of execution has the associated derivative accumulated into it. Thus, the maximum memory residency is directly proportional to the number of operations. Large computations may perform enough computations that it is impractical to allocate all required state simultaneously. Checkpointed reverse mode solves this problem by trading space for time via repeated computation of the forward pass, once without allocating memory and an additional time with memory. However, any memory allocated in between these two runs can be safely deallocated, as it corresponds to code after the checkpointed subprogram in the original program, thus reducing maximum memory residency.

We will focus specifically on user specified checkpointing, i.e. the user must

choose what portion of the program should be recomputed in order to save memory. For an in-depth explanation, we recommend [Hascoët and Araya-Polo, 2006]. Checkpointing without user annotation is possible, see [Jeffrey Mark Siskind and Barak A. Pearlmutter, 2017], and we leave it as future work.

Our implementation will begin with defining a new effect to mark out the checkpointed computation. We then define two handlers, one which evaluates the checkpointed program without creating a reverse pass, and one which does while respecting the checkpoint annotations. Each of these handlers are unique compared to our previously defined ones, as they will recursively call themselves on the *argument* of the checkpoint effect, not just on the captured continuation. Finally, note how succinct and clear the handling of the checkpoint effect is. We can read off almost directly that the checkpointed code is run twice, once just for evaluation and a second time for creating the reverse pass. Also note how we are able to reuse the previously defined stateful reverse mode handler.

Listing 5.2: Checkpointed reverse mode (Frank)

```
include prelude
include smooth
include reverse

interface Checkpoint X [e] =
    checkpoint : {[e|Checkpoint X [e|], Smooth (Prop X)] Prop X}
              -> Prop X

evaluatet : Ref X
         -> <Checkpoint X [e|], Smooth (Prop X)> Y
         -> [e|Smooth X] Y
evaluatet _ x = x
evaluatet s <checkpoint p -> k> =
    let res = evaluatet s (<Smooth(s a b -> s b)> p!) in
    evaluatet s (k (prop (fwd res) s))
evaluatet s <ap0 n -> k> =
    evaluatet s (k (prop (<Smooth> (ap0 n)) s))
evaluatet s <ap1 u (prop x dx) -> k> =
    evaluatet s (k (prop (<Smooth> (ap1 u x)) s))
evaluatet s <ap2 b (prop x dx) (prop y dy) -> k> =
    evaluatet s (k (prop (<Smooth> (ap2 b x y)) s))

reversec : <Checkpoint X [e|], Smooth (Prop X)> Unit
        -> [e|RefState, Smooth X] Unit
reversec x = x
reversec <checkpoint p -> k> =
    let s = new (c 0.0) in
    let res = <RefState> (evaluatet s (<Smooth(s a b -> s b)> p!)) in
    let r = prop (fwd res) (new (c 0.0)) in
    reversec (k r);
```

```
31      reversec ((deriv (<Smooth(s a b -> s b), RefState> p!)) := @(deriv r))
32  reversec <m> = reversec (<Smooth(s a -> s)> (reverse m!))
33
34  gradc : {(Prop X)
35        -> [RefState, Checkpoint X, Smooth X, Smooth (Prop X)] (Prop X)
36         }
37      -> X -> [RefState, Smooth X] X
38  gradc f x =
39      let z = prop x (new (c 0.0)) in
40      reversec ((deriv (f z)) := (<Smooth> (c 1.0)));
41      @(deriv z)
```

We begin on line 3 by importing our previous stateful reverse mode definition, allowing us to use the stateful reverse mode handler `reverse`. Lines 5 to 7 define a new effect `Checkpoint` used to annotate the checkpointed subprograms which will be run multiple times. We add an explicit effect row variable `e` for clarity. Lines 9 to 21 define a simple `evaluate`-like handler, `evaluatet`, which only calculates a forward pass, and recursively runs itself on checkpointed subprograms. Lines 23 to 32 define the checkpointed reverse mode handler. Frank also contains a catch-all pattern match `<m>` which matches values and effects not handled above it. We use this feature to extend `reverse` by delegating any `Smooth` commands received to `reverse` and only adding a case for `checkpoint`. Note how the checkpointed subprogram (the suspended computation `p` which is the argument of `checkpoint`) is called twice, once with `evaluatet` as the handler and once with `reversec` as the handler. We require the adaptor `<Smooth(s a b -> s b)>` on lines 14, 28 and 31 to ensure the effects of `p` are handled by the correct handler. Thus, we are relying on the flexibility of effects and handlers to interpret effects in different ways. Additionally, the last case will match every `Smooth` command, and then reinvoke the captured computation with a new `reverse` handler to handle the command. Finally, lines 34 to 41 define the function which computes gradients using checkpointed reverse mode.

Let us begin to execute an example program in order to understand how the `reversec` handler works. Consider the following program which makes use of nested checkpoints.

```
evaluate (gradc ({x ->
  let y = c 2.0 in
  let z = checkpoint {p x y} in
  let a = checkpoint {let w = checkpoint {t x z} in p w y} in
  p a x
```

The first interesting evaluation step is after the underlined `checkpoint` has been handled.

```
evaluate (
  reversec (<Smooth(s a -> s)> (reverse ((deriv (
    let z = prop 4.0 <r2> in
    let a = checkpoint {
      let w = checkpoint {t (prop 2.0 <z>) z} in
      p w (prop 2.0 <r1>)} in
    p a (prop 2.0 <z>)
  )) := (<Smooth> (c 1.0)))));
  reversec ((deriv (<Smooth(s a b -> s b), RefState>
      {p (prop 2.0 <z>) (prop 2.0 <r1>)}!)
    ) := @(deriv (prop 4.0 <r2>)));
  @(deriv (prop 2.0 <z>))
```

Note how on the second line the `reverse` handler has been made the innermost delimiter of the remainder of the initial program, via the catch-all case of `reversec`. Additionally, note how the checkpointed code (underlined) is stored as a thunk to be run after the initial program in the second use of `reversec`. After the initial program has been evaluated away, we obtain the following.

```
evaluate (
  reversec (<Smooth(s a -> s)> (reverse (<r4> := (<Smooth> (c 1.0)))));
  <r3> := (p @<r3> (t (der2L plusE 10.0 2.0) @(deriv (prop 12.0 <r4>))));
  <z> := (p @<z> (t (der2R plusE 10.0 2.0) @(deriv (prop 12.0 <r4>))));
  reversec ((deriv (<Smooth(s a b -> s b), RefState>
      {let w = checkpoint {t (prop 2.0 <z>) (prop 4.0 <r2>)} in
       p w (prop 2.0 <r1>)}!)
    ) := @(deriv (prop 10.0 <r3>)));
  reversec ((deriv (<Smooth(s a b -> s b), RefState>
      {p (prop 2.0 <z>) (prop 2.0 <r1>)}!)
    ) := @(deriv (prop 4.0 <r2>)));
  @(deriv (prop 2.0 <z>))
```

The remaining `checkpoint` command illustrates the recursive nature of `reversec`. It shows how even nested checkpointing in checkpointed code can be properly evaluated.

As observed before, Koka does not have generalized adaptors, but they can be implemented in a less efficient manner. A further difference is that in Frank, one handler can handle multiple effects simultaneously, whereas in Koka one handler can handle exactly one effect. Thus, the structure of the checkpointed reverse handler is slightly different. In Eff and OCaml handlers can handler multiple effects, but for variety we write our checkpointed modes without reusing the previous reverse mode.

5.3 Higher-Order Functions

Higher-order functions are an important feature of functional languages which enables modularity, abstraction, and code reuse. So far, all of our algorithms work in the *presence* of higher-order functions, but they do not interact with them. Intuitively, if we take any fixed program which uses higher-order functions and repeatedly inline or defunctionalize, we can create an equivalent first-order program. The original program and the new program, when executed, will perform the same sequence of effects, and so each handler will handle the same sequence of effects regardless of the program. Therefore, we are ignoring an significant feature of functional languages.

One basic example of the benefits of higher-order functions is the use of map. Consider the following term:

```
map f (map g xs)
```

The two uses of `map` means that we must traverse the list `xs` twice. Many compilers of functional languages can recognize this pattern and perform *map fusion*, transforming the above into

```
map (f . g) xs
```

which only traverses the list once. Note that an effectful language, the two programs may be observationally inequivalent. With a powerful enough effect system or through the knowledge of the programmer, certain applications of the above rule may still be possible. In conclusion, an AD system which is integrated with higher-order functions can benefit from the modularity, abstraction, and code reuse.

An obvious obstruction for an effect handler based implementation is that function application is not an effect, and so we can not handle it. Therefore, the following algorithms are illustrative of the expressiveness of effects and handlers, even if manual manipulation of function abstraction and application is required. Our implementations are based upon [Vákár and Smeding, 2022] which introduces a define-then-run reverse mode in a pure functional language with support for higher-order functions. Their approach essentially applies continuation-based checkpointing transformation to every function. In contrast, our version is define-by-run and makes use of mutable state. However, the primary idea is similar.

Consider the following code, which does *not* implement the core concept of Vákár and Smeding (but was inspired by it). We shall use this implementation

to contrast with the subsequent implementation which *is* based on their transformation.

Listing 5.3: Higher-order reverse mode (Frank)

```
include prelude
include smooth
include evaluate

data Prop X = prop X (Ref X)

fwd : Prop X -> X
fwd (prop x _) = x

deriv : Prop X -> Ref X
deriv (prop _ r) = r

data Func X [e] =
    func {Prop X -> [e|RefState, Smooth X] Pair (Prop X)
                                                {[e|RefState, Smooth X] Unit}
         }

interface High X [e] =
      abs : {Prop X -> [e|High X [e|], RefState, Smooth (Prop X)] Prop X}
         -> Func X [e|]
    | app : Func X [e|] -> Prop X -> Prop X

reverseh : <Smooth (Prop X), High X [e|]> Y
        -> [e|RefState, Smooth X] Pair Y {[e|RefState, Smooth X] Unit}
reverseh x = pair x {unit}
reverseh <ap0 n -> k> =
    let r = prop (ap0 n) (new (c 0.0)) in
    reverseh (k r)
reverseh <ap1 u (prop x dx) -> k> =
    let r = prop (ap1 u x) (new (c 0.0)) in
    let res = reverseh (k r) in
    pair
        (fst res)
        { (snd res)!;
          dx := (p @dx (t (der1 u x) @(deriv r)))
        }
reverseh <ap2 b (prop x dx) (prop y dy) -> k> =
    let r = prop (ap2 b x y) (new (c 0.0)) in
    let res = reverseh (k r) in
    pair
        (fst res)
        { (snd res)!;
          dx := (p @dx (t (der2 L b x y) @(deriv r)));
          dy := (p @dy (t (der2 R b x y) @(deriv r)))
        }
reverseh <abs f -> k> =
    let g = {x -> reverseh (<Smooth(s a b -> s b)> (f x))} in
    reverseh (k (func g))
```

```
49 reverseh <app (func f) x -> k> =
50     let r = f x in
51     let res = reverseh (k (fst r)) in
52     pair (fst res) {(snd res)!; (snd r)!}
53
54 gradh : {(Prop X)
55       -> [e|RefState, Smooth X, Smooth (Prop X), High X [e|]] (Prop X)
56        }
57     -> X -> [e|RefState, Smooth X] X
58 gradh f x =
59     let z = prop x (new (c 0.0)) in
60     let res = reverseh ((deriv (f z)) := (<Smooth> (c 1.0))) in
61     (snd res)!;
62     @(deriv z)
```

On lines 5 to 11 we define the `Prop` data type as before. Lines 13 to 21 define a data type `Func` of instrumented functions and a new effect `High` for instrumenting functions by way of explicit abstraction and application. Note that we require the user to explicitly decide which functions the algorithm is to be made aware of by using this effect. An instrumented function will produce the same result as the original, but in addition it will provide the required portion of the reverse pass. Lines 23 to 52 implement our handler, where the `Smooth` operations are handled similarly to stateful reverse mode, except that we explicitly capture the reverse pass as a thunk to be run later. Lines 46 to 52 handle the `High` operations. For `abs`, we take the original function `f` and create a new function `g` which builds in the reverse pass. Then when handling `app`, we take this reverse pass and ensure it is run in the correct sequence with respect to the rest of the program. Finally, lines 54 to 62 define the differentiation function, which is similar to stateful reverse mode except for the need to explicitly run the reverse pass on line 61. The algorithm we have illustrated works, but is not too different from the original stateful reverse mode. If our instrumented functions remained in their original condition, an analogous execution would happen under stateful reverse mode.

The following uses the core idea of Vákár and Smeding, which is to checkpoint each instrumented function, and produces a truly different execution than stateful reverse mode.

Listing 5.4: Higher-order checkpointed reverse mode (Frank)

```
1 include prelude
2 include smooth
3 include evaluate
4
5 data Prop X = prop X (Ref X)
6
```

```
7  fwd : Prop X -> X
8  fwd (prop x _) = x
9
10 deriv : Prop X -> Ref X
11 deriv (prop _ r) = r
12
13 data Func X [e] =
14     func {Prop X -> [e|RefState, Smooth X] Pair (Prop X)
15                                                 {[e|RefState, Smooth X] Unit}
16          }
17
18 interface High X [e] =
19       abs : {Prop X -> [e|High X [e|], RefState, Smooth (Prop X)] Prop X}
20          -> Func X [e|]
21     | app : Func X [e|] -> Prop X -> Prop X
22
23 evaluatet : Ref X
24          -> <Smooth (Prop X), High X [e|]> Y
25          -> [e|RefState, Smooth X] Pair Y {[e|RefState, Smooth X] Unit}
26 evaluatet _ x = pair x {unit}
27 evaluatet s <abs f -> k> =
28     let g = {x -> evaluatet s (<Smooth(s a b -> s b)> (f x))} in
29     evaluatet s (k (func g))
30 evaluatet s <app (func f) x -> k> =
31     let res = f x in
32     evaluatet s (k (fst res))
33 evaluatet s <ap0 n -> k> =
34     evaluatet s (k (prop (<Smooth> (ap0 n)) s))
35 evaluatet s <ap1 u (prop x dx) -> k> =
36     evaluatet s (k (prop (<Smooth> (ap1 u x)) s))
37 evaluatet s <ap2 b (prop x dx) (prop y dy) -> k> =
38     evaluatet s (k (prop (<Smooth> (ap2 b x y)) s))
39
40 reversehc : <Smooth (Prop X), High X [e|]> Y
41         -> [e|RefState, Smooth X] Pair Y {[e|RefState, Smooth X] Unit}
42 reversehc x = pair x {unit}
43 reversehc <ap0 n -> k> =
44     let r = prop (ap0 n) (new (c 0.0)) in
45     reversehc (k r)
46 reversehc <ap1 u (prop x dx) -> k> =
47     let r = prop (ap1 u x) (new (c 0.0)) in
48     let res = reversehc (k r) in
49     pair
50         (fst res)
51         { (snd res)!;
52           dx := (p @dx (t (der1 u x) @(deriv r)))
53         }
54 reversehc <ap2 b (prop x dx) (prop y dy) -> k> =
55     let r = prop (ap2 b x y) (new (c 0.0)) in
56     let res = reversehc (k r) in
57     pair
58         (fst res)
59         { (snd res)!;
```

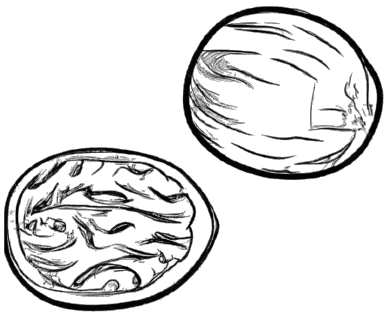
```
60            dx := (p @dx (t (der2 L b x y) @(deriv r)));
61            dy := (p @dy (t (der2 R b x y) @(deriv r)))
62          }
63  reversehc <abs f -> k> =
64      let g =
65          {x ->
66              let s = new (c 0.0) in
67              let res = evaluatet s (<Smooth(s a b -> s b)> (f x)) in
68              let r = prop (fwd (fst res)) (new (c 0.0)) in
69              pair
70                  r
71                  { let q = reversehc (
72                      (deriv (<Smooth(s a b -> s b)> (f x))) := @(deriv r)
73                    ) in
74                    (snd q)!
75                  }
76          } in
77      reversehc (k (func g))
78  reversehc <app (func f) x -> k> =
79      let r = f x in
80      let res = reversehc (k (fst r)) in
81      pair (fst res) {(snd res)!; (snd r)!}
82
83  gradhc : { (Prop X)
84         -> [e|RefState, Smooth X, Smooth (Prop X), High X [e|]] (Prop X)
85          }
86      -> X -> [e|RefState, Smooth X] X
87  gradhc f x =
88      let z = prop x (new (c 0.0)) in
89      let res = reversehc ((deriv (f z)) := (<Smooth> (c 1.0))) in
90      (snd res)!;
91      @(deriv z)
```

Lines 5 to 21 are as before. Lines 23 to 38 define an `evaluatet` handler which is analogous to the one of checkpointed reverse mode. Lines 40 to 81 define the core handler `reversehc`, which is the same as above except for the handling of `abs` on lines 63 to 77, where the function `f` is instrumented. Instead of recursively using `reversehc`, we use a checkpointed approach where `f` is run twice, once without producing the reverse pass and an additional time which does. Note that when handling `app`, the process is the same. Finally, Lines 83 to 91 define the differentiation function as before.

The differences between the Frank implementation and those of the other languages are the same as for checkpointed reverse mode, except for Koka. We could not devise an implementation which allowed the reuse of the smooth effect, and thus defined a new effect which combined `Smooth` and `High` effects.

Chapter 6

Performance Analysis



Nutmeg

6.1 Asymptotic Analysis

An important aspect of AD is the asymptotic behavior. [Griewank and A. Walther, 2008, Sec. 4.4] show that for a composite measure of "work", both forward and reverse mode only need perform bounded constant multiple more work than the original program. Their measure of work accounts for four categories: memory fetches and stores, additions and subtractions, multiplications, and non-linear operations. Paired with reasonable assumptions, they then prove that forward mode applied to a program should be between 2 to 2.5 times slower than the original program, and reverse mode should be between 3 to 4 times slower. We will thus examine forward mode and the reverse modes for correct performance.

We would like to show that our implementations differentiate with only a

constant multiple slowdown, and that this holds across different problem sizes. To do so, we create a simple program with an input n such that the number of smooth operations invoked is directly proportional to n. Thus, graphing time t against n should produce a line, and if another program takes time $c \cdot t$, then it is also a line when graphed against n. Furthermore, in this case the original program has slope t/n and the derived program has slope $(c \cdot t)/n$, and thus the ratio of the slopes is c the slowdown factor. Our simple program will approximate the Taylor series of $\frac{1}{x}$ around 1, i.e. it will approximate the right-hand side of

$$\frac{1}{x} = \sum_{n=0}^{\infty} (-1)^n (x-1)^n$$

which converges when $|x-1| < 1$. Let a_n denote the n^{th} term of the above series. Then we have the recurrence

$$a_0 = 1, \qquad a_n = -(x-1) \cdot a_{n-1}$$

and so we can iteratively generate a_n as shown below in OCaml:

```
open Smooth

module Taylor_Recip_Benchmark (T : SMOOTH) = struct
  let approx_recip iters x = let open T in
    let prev = ref (c 1.0) in (* a_0 *)
    let acc = ref (c 1.0) in  (* sum_{n=0}^{0} a_n *)
    for _i = 1 to iters do
      prev := !prev *. (~. (x -. (c 1.0))); (* a_i = -(x - 1) * a_{i-1} *)
      acc := !prev +. !acc                  (* sum_{n=0}^{i} a_n = a_i + sum_{n=0}^{i-1} a_n *)
    done;
    !acc (* sum_{n=0}^{iters} a_n *)
end
```

Each iteration of the loop executes five smooth operations. Therefore, the number of operations and thus the time to execute should be directly proportional to `iters`, and thus each algorithm applied to `apporx_recip` should be directly proportional if our implementations have the correct behavior. We then create an executable which takes `iters` as a command line argument, e.g. in OCaml:

```
let _ =
  (* Increase the minor heap size to 500MiB to stop quadratic behaviour in
     reverse mode due to deep callstack. 1MiB = 1048576. *)
  Gc.set { (Gc.get ()) with Gc.minor_heap_size = (500 * 1048576)};
  let iters = int_of_string Sys.argv.(1) in
  let module E = Evaluate in
  let module R = Reverse(E) in
  let module T = Taylor_Recip_Benchmark(R) in
  let res = match_with (R.grad (T.approx_recip iters)) 0.5 E.evaluate in
  Printf.printf "%f\n" res
```

The above example is straightforward except for the change in the garbage collector (GC) minor heap size parameter. Stateful reverse mode creates a deep call stack which is long-lived (the length of the entire program), causing stack scans by the GC to add a linear overhead. By increasing the minor heap size, this issue is alleviated. The needed minor heap size increases proportional to `iters`, and we have chosen a suitable value for our tested range. It is also possible to change the behavior of OCaml 5.0's GC, but that is out of our scope here[1]. We also make similar test programs for each of the other languages.

We will now describe our testing methodology. Due to the disparate performance across modes and languages, our testing methodology for a single program depends on the following four variables, which we explain in more detail shortly:

- w: the number of *warmup* runs;
- s: the *starting* value of our parameter n;
- e: the *ending* value of our parameter n; and
- d: the *delta* used to increment n.

For a specific program and a fixed value of each of these variables, we perform the following for each $n \in \{s, s + d, s + 2d, \ldots, e\}$:

1. With input n, we run the program w times as a warmup in an attempt to reach a more steady state.

2. With input n, we run the program at least 10 times, keeping track of how long each execution took.

3. For each input n, calculate the mean and standard deviation of the executions times.

Thus, at the end of each batch of runs we have for each n a mean time and standard deviation. We automate the above process with the `hyperfine` program [Peter, 2023], which performs exactly the described methodology. Note that we do not test the effect of the number of warmup runs nor do we analyze the behavior of the garbage collector in the relevant languages beyond the OCaml fix[2].

We plot the data collected from the above process by showing the means (connecting neighbors by a line) and adding symmetric error bars showing the

[1] Thanks to Stephen Dolan who diagnosed the issue and suggested the used solution.
[2] Frank is executed via an interpreter written in Haskell, Eff via one written in OCaml.

standard deviation at each point. We label each graph with a four-tuple (w, s, e, d) to record the parameters used to generate the data. Furthermore, we perform linear and polynomial (specifically quadratic) regressions using least squares on each result in order to classify each implementation. To provide quantitative evidence for our classifications, we compute an R^2 value to three significant figures for each plot. When the quadratic regression R^2 value is particularly high, we contend that the result is quadratic. When the quadratic regression R^2 value is lower but much better than the linear regression R^2 value, we characterize the behavior as nonlinear, which in our case always means superlinear. For linear algorithms, we also record the the ratio of runtimes compared to evaluation by using the slope of the lines of best fit. We perform the above analyses using the Python library `scikit-learn` [Pedregosa et al., 2011].

Unless otherwise stated, the following benchmarks were run on an HP EliteBook 840 G6 with a eight core (4.60 GHz boost) Intel i7-8565U, 2×16GB = 32GB 2400 MHz DDR4, and 512 GB M.2 PCIe 3.0x4 NVMe SSD. The operating system is Ubuntu 23.10 with Linux kernel 5.15.146.1-microsoft-standard-WSL2 running inside Windows Subsystem for Linux 2 on a Windows 11 host.

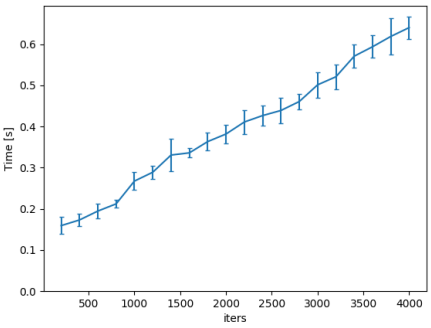
6.1.1 Frank

Figure 6.1 presents the asymptotics for our Frank implementations. Figures 6.1a to 6.1e show the results for each of the standard modes, and fig. 6.1f compares all modes using log-log axes.

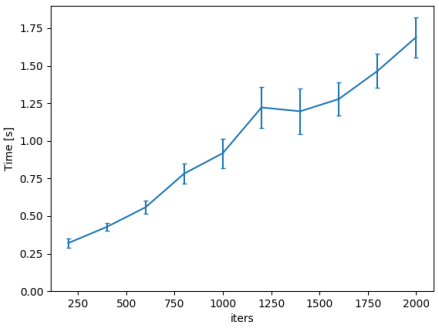
Table 6.1 shows the R^2 values for each mode. We see that evaluate and forward mode are linear, and all reverse modes are nonlinear or quadratic. Additionally, forward mode is 5.91 times slower than evaluation.

Mode	**Linear R^2**	**Quadratic R^2**	**Order**	**Ratio**
Evaluate	0.971	0.971	Linear	1.00
Forward	0.934	0.935	Linear	5.91
Continuation	0.838	0.962	Nonlinear	-
Stateful	0.870	0.942	Nonlinear	-
Taped	0.936	0.995	Quadratic	-

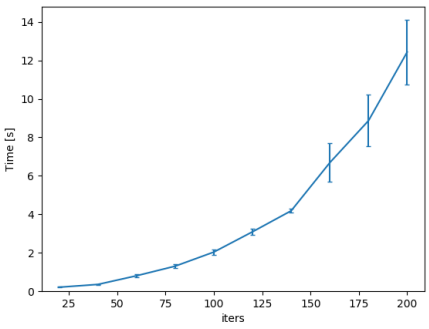
Table 6.1: Frank, R^2 values for linear and quadratic best fits



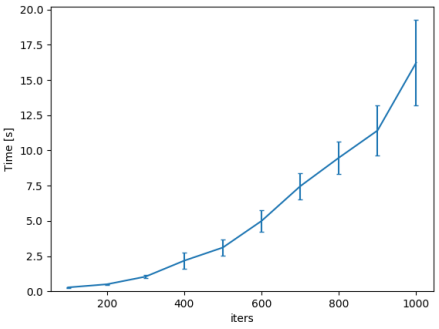
(a) Evaluate
$(10, 200, 4000, 200)$



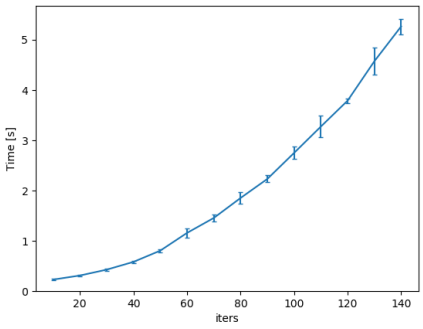
(b) Forward mode
$(5, 200, 2000, 200)$



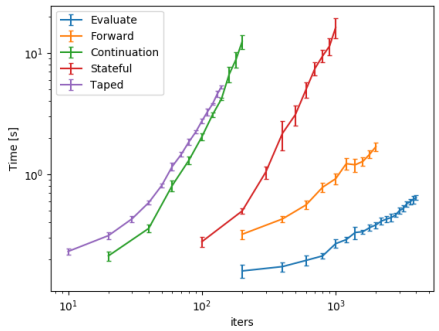
(c) Continuation reverse mode
$(5, 20, 200, 20)$



(d) Stateful reverse mode
$(5, 100, 1000, 100)$



(e) Taped reverse mode
$(5, 10, 140, 10)$



(f) Log-log comparison
of all modes

Figure 6.1: Frank benchmark results

6.1.2 Eff

Figure 6.2 presents the asymptotics for our Eff implementations. Figures 6.2a to 6.2e show the results for each of the standard modes, and fig. 6.2f compares all modes using log-log axes.

Table 6.2 shows the R^2 values for each mode. We see that evaluate and forward mode are linear, and all reverse modes are nonlinear or quadratic. Additionally, forward mode is 4.83 times slower than evaluation.

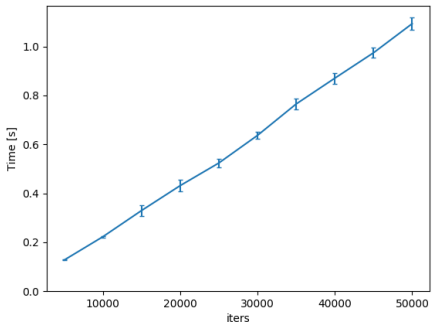
Mode	**Linear** R^2	**Quadratic** R^2	**Order**	**Ratio**
Evaluate	0.997	0.997	Linear	1.00
Forward	0.996	0.997	Linear	4.83
Continuation	0.874	0.997	Quadratic	-
Stateful	0.884	0.998	Quadratic	-
Taped	0.851	0.970	Nonlinear	-

Table 6.2: Eff, R^2 values for linear and quadratic best fits

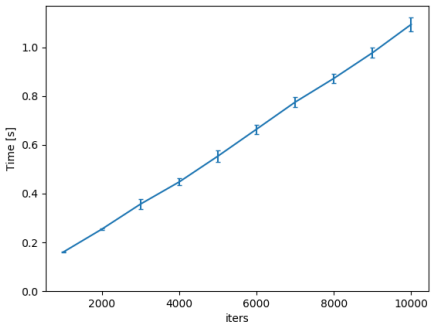
6.1.3 Koka

Koka has a special feature compared to our other languages. An effect can be declared `linear`, which places a requirement on handlers to be tail resumptive. Thus, we have implemented each algorithm twice, once without `linear` and once with, except for stateful reverse mode which is not tail recursive. Figure 6.3 presents the asymptotics for our Koka implementations without `linear`. Figures 6.3a to 6.3e show the results for each of the standard modes, and fig. 6.3f compares all modes using log-log axes. Figure 6.4 presents the asymptotics for our Koka implementations using `linear`. Figures 6.4a to 6.4d show the results for each of the standard modes using linear effects, and fig. 6.4e compares all linear modes using log-log axes.

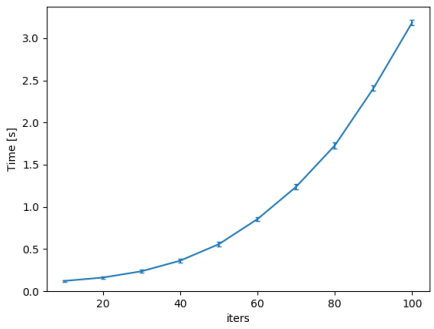
Table 6.3 shows the R^2 values for each mode. We see that of the non-`linear` modes only evaluate is linear, which is due to a bug in Koka where nested handlers of general (non-`linear`) effects can have nonlinear/quadratic behavior. For the `linear` effect, we see that evaluate, forward, and taped reverse are linear while continuation reverse is quadratic. Note also that `linear` evaluate is 20.8 times faster than the non-`linear` version due to Koka's efficient compilation scheme for



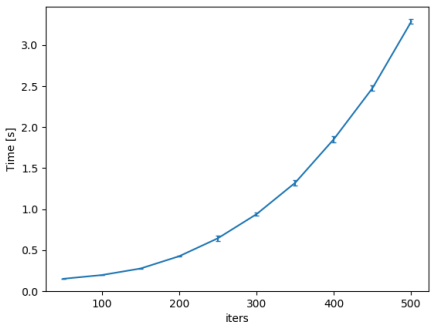
(a) Evaluate
$(10, 5000, 50000, 5000)$



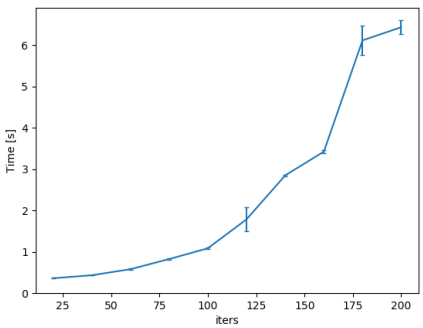
(b) Forward mode
$(10, 1000, 10000, 1000)$



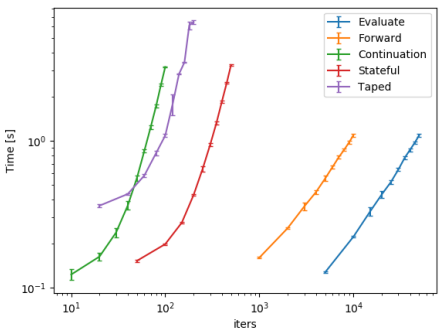
(c) Continuation reverse mode
$(10, 10, 100, 10)$



(d) Stateful reverse mode
$(10, 50, 500, 50)$

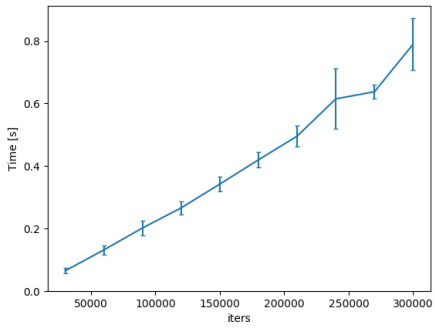


(e) Taped reverse mode
$(10, 20, 200, 20)$

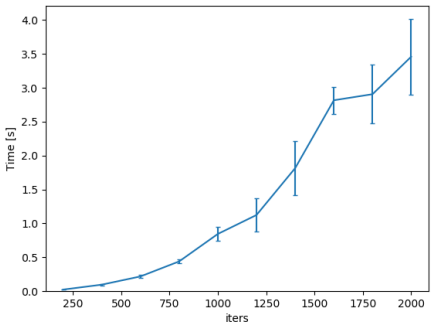


(f) Log-log comparison of all modes

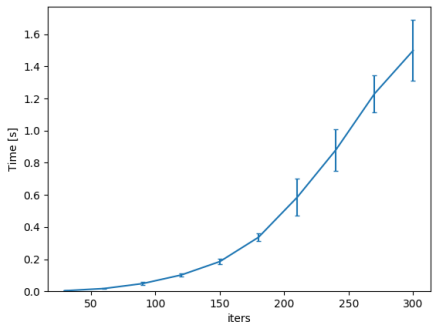
Figure 6.2: Eff benchmark results



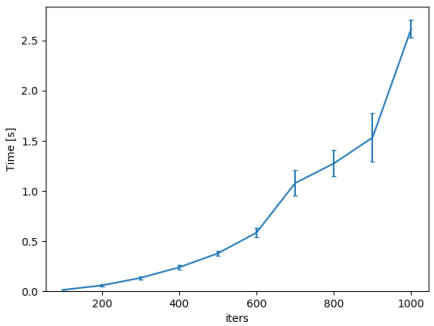
(a) Evaluate
$(10, 30000, 300000, 30000)$



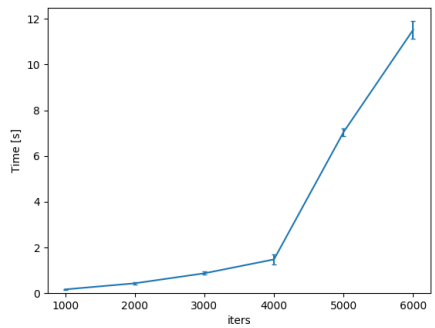
(b) Forward mode
$(10, 200, 2000, 200)$



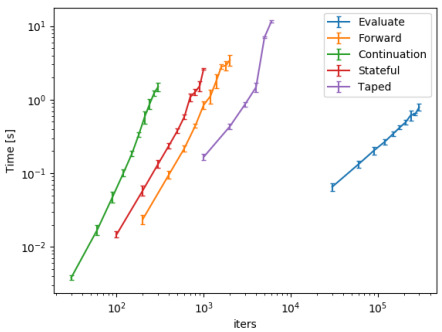
(c) Continuation reverse mode
$(10, 30, 300, 30)$



(d) Stateful reverse mode
$(10, 100, 1000, 100)$

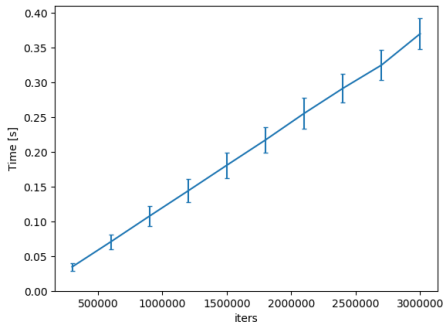


(e) Taped reverse mode
$(10, 1000, 6000, 1000)$

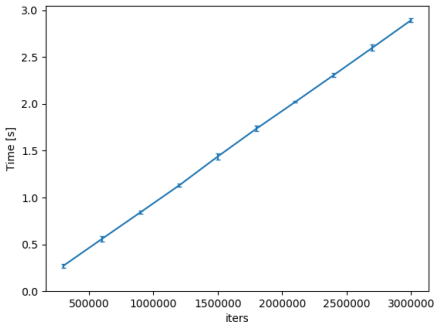


(f) Log-log comparison of all modes

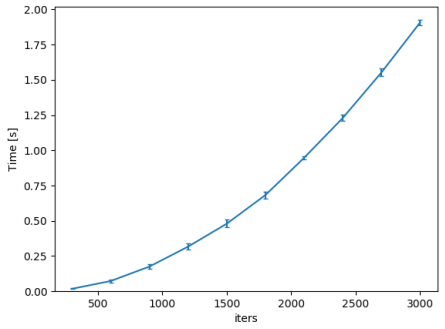
Figure 6.3: Koka benchmark results



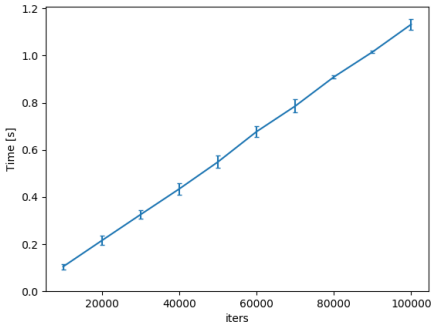
(a) Evaluate, linear
$(10, 30000, 300000, 30000)$



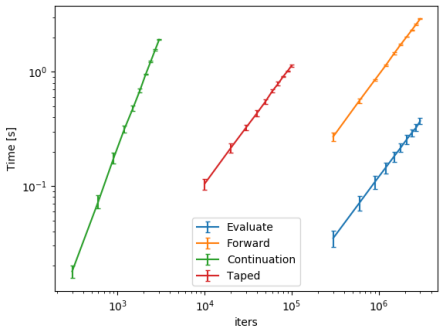
(b) Forward mode, linear
$(10, 300000, 3000000, 300000)$



(c) Continuation reverse mode, linear
$(10, 300, 3000, 300)$



(d) Taped reverse mode, linear
$(10, 10000, 100000, 10000)$



(e) Log-log comparison of
all linear effect modes

Figure 6.4: Koka benchmark results, linear effects

tail resumptive handlers. We see that `linear` forward mode is 7.95 times slower than evaluate and that taped reverse mode is 93.4 times slower than evaluate.

Mode	**Linear R^2**	**Quadratic R^2**	**Order**	**Ratio**
Evaluate	0.972	0.975	Linear	20.8
Forward	0.910	0.960	Nonlinear	-
Continuation	0.787	0.980	Quadratic	-
Stateful	0.859	0.972	Nonlinear	-
Taped	0.779	0.975	Nonlinear	-
Linear				
Evaluate	0.982	0.982	Linear	1.00
Forward	0.999	0.999	Linear	7.95
Continuation	0.951	0.999	Quadratic	-
Taped	0.997	0.997	Linear	93.4

Table 6.3: Koka, R^2 values for linear and quadratic best fits

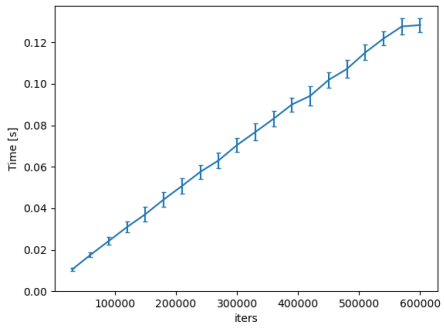
6.1.4 OCaml

These benchmarks have been run on a Dell Precision T3600 with a quad core (3.60 GHz boost) Intel Xeon E5-1620, 4×8GB $= 32$GB 1600 MHz DDR4, and 256 GB SATA 6 Gb/s SSD. The operating system used is headless Debian 12 (bookworm) with Linux kernel release 6.1.0-18-amd64. Figure 6.5 presents the asymptotics for our OCaml implementations. Figures 6.5a to 6.5e show the results for each of the standard modes, and fig. 6.5f compares all modes using log-log axes.

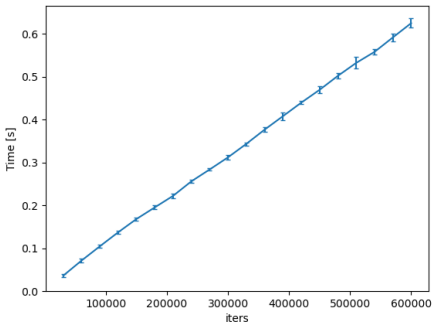
Table 6.4 shows the R^2 values for each mode. We see that evaluate, forward mode, stateful reverse mode, and taped reverse mode are linear while continuation reverse mode is quadratic. Additionally, forward mode is 4.76 times slower than evaluation, stateful reverse mode is 8.54 times slower than evaluation, and taped reverse mode is 8.26 times slower than evaluation.

6.2 Discussion of results

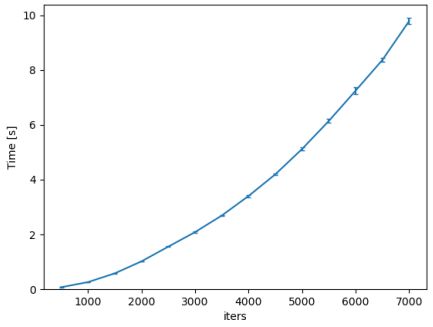
We have shown that each mode, except continuation reverse mode, has an implementation exhibiting the correct AD asymptotics, i.e. they are $O(1)$ with respect



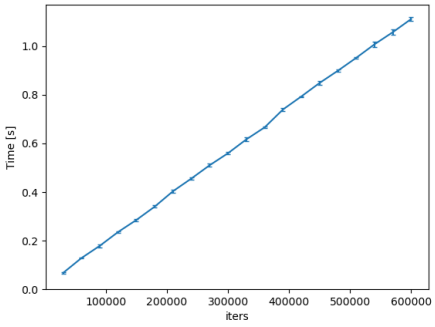
(a) Evaluate
$(10, 30000, 600000, 30000)$



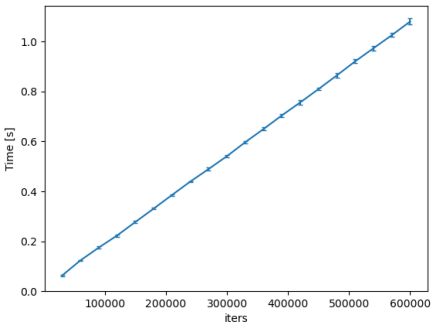
(b) Forward mode
$(10, 30000, 600000, 30000)$



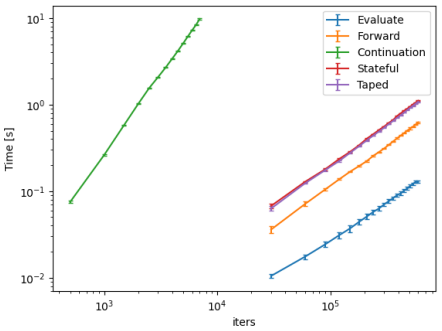
(c) Continuation reverse mode
$(10, 500, 7000, 500)$



(d) Stateful reverse mode
$(10, 30000, 600000, 30000)$



(e) Taped reverse mode
$(10, 30000, 600000, 30000)$



(f) Log-log comparison
of all modes

Figure 6.5: OCaml benchmark results

Mode	Linear R^2	Quadratic R^2	Order	Ratio
Evaluate	0.993	0.993	Linear	1.00
Forward	0.999	0.999	Linear	4.76
Continuation	0.952	1.000	Quadratic	-
Stateful	1.000	1.000	Linear	8.54
Taped	1.000	1.000	Linear	8.26

Table 6.4: OCaml, R^2 values for linear and quadratic best fits

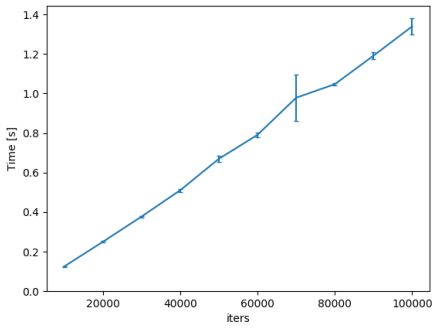
to evaluation. Our best results are from OCaml, where forward mode is approximately 4.8× slower than evaluation, stateful reverse mode is approximately 8.6× slower, and taped reverse mode is approximately 8.3 times slower. We have not reached the theoretical optimal bounds derived by [Griewank and A. Walther, 2008] of 2.5× and 4×, but we are not far off. Therefore, we claim that our AD modes in OCaml are performant enough to be practical, supporting the general claim that effect handlers are suitable for AD.

Furthermore, we have seen in the case of Koka that `linear` effects and handlers are much faster than non-`linear` ones. This decrease in execution time is due to Koka's changes to its compilation strategy based on the guarantee that all handlers resume exactly once and that no data needs to be captured in order to be used after the resumption concludes. We also note for completeness that, due to a bug in Koka, the `linear` taped mode executable was compiled in debug mode.

We believe that the correct asymptotics are possible for reverse mode in Koka as well. Consider the following version of stateful reverse mode in Koka.

```
val reverse = handler {
  ctl ap0(n) -> {
    val r = evaluate{Prop(op0(n), ref(c(0.0)))}
    resume(r)
  }
  ctl ap1(u,x) -> {
    val r = evaluate{Prop(op1(u,x.v), ref(c(0.0)))}
    resume(r)
    evaluate{set(x.dv, !x.dv +. (der1(u,x.v) *. !r.dv))}
  }
  ctl ap2(b,x,y) -> {
    val r = evaluate{Prop(op2(b,x.v,y.v), ref(c(0.0)))}
    resume(r)
    evaluate{set(x.dv, !x.dv +. (der2(b,L,x.v,y.v) *. !r.dv))}
    evaluate{set(y.dv, !y.dv +. (der2(b,R,x.v,y.v) *. !r.dv))}
  }
}
```

We have taken each effectful line in the `reverse` handler and handled them directly with the `evaluate` handler as opposed to using the handler at the call-site of the gradient operator. Running our benchmark with this version of `reverse` produces the following correct behavior.



Thus, the reversal of dependencies by virtue of using captured data after resuming in a handler is not the source of the slowdown.

Unfortunately, at present there is a bug in Koka which causes programs containing nested handlers for general effects to perform incorrectly. For example, consider the following program.

```
effect ctl op() : int

fun work(iters)
  var acc := 0
  repeat(iters) {acc := (acc + op())}
  acc

fun main()
  val iters = try {get-args().head.unjust.parse-int.unjust} fn(_) {100}
  with ctl op() resume(1)
  with ctl op() resume(op() + 1)
  println(work(iters))
```

The result of running the above exhibits quadratic behavior with respect to the number of iterations, when it should be linear. Indeed, running analogous code in other languages, for example Frank, produces the correct linear behavior.

6.3 Real World Benchmarks

We claim that our implementation of AD via effects and handlers is performant with respect to comparable implementations. By comparable, we mean CPU based, as we do not use GPU based computation, and dynamic/define-by-run, as

static/define-then-run approaches are almost always faster through code generation and optimization. The dynamic approach is often referred to as *eager mode*, for example by PyTorch and TensorFlow. To substantiate our claim, we will use the benchmark suite of [Šrajer, Kukelova, and Fitzgibbon, 2018a][3].

The suite of Šrajer, Kukelova, and Fitzgibbon is reproducible, extensible, realistic, and expansive. It is reproducible through the use of containerization, ensuring that the same version of each tool is used across runs and compilations. Extensibility is achieved through a documented test harness and modular design. The four computations which they benchmark are real world functions which are optimized against in machine learning and computer vision. Additionally, the current iteration of their system supports thirteen different implementations across five languages, including a baseline of finite differences[4] and manually implemented derivatives. Finally, the computed derivatives are checked for correctness against a known correct implementation.

The full methodology can be found in their paper and the repository[5], we will highlight the important aspects here. For each implementation and set of parameters, essentially the following is carried out:

- Read the input data and convert it into a consumable format.
- Run any needed preparation code which is not AD.
- For both computation of the objective function and its gradient:
  - Find the number of times r needed to run to reach a prescribed minium time.
  - Run n lots of r computations, find the average time for each lot.
  - Pick the minium average time of from the n lots to alleviate noise.
- Save the times recorded and the numerical results to check correctness.

We have chosen one of their four computations to implement, namely the objective function used for the fitting of Gaussian mixture models. Specifically, let $m, N, K, D \in \mathbb{N}$ and let $1 \leq i \leq N$ and $1 \leq k \leq K$. We use $|| \cdot ||$ for Euclidean

[3] A longer preprint is available [Šrajer, Kukelova, and Fitzgibbon, 2018b] and the base suite is available at `https://github.com/microsoft/ADBench`.

[4] Finite differences approximate the derivative via, for example, $\partial f(x)/\partial x(y) \cong f(y+\varepsilon) - f(y)/\varepsilon$, which holds for small ε.

[5] `https://github.com/microsoft/ADBench/blob/master/docs/Methodology.md`

norm, and use an unspecified function $Q\colon \mathbb{R}^D \times \mathbb{R}^{D(D-1)/2} \to \mathbb{R}^{D\times D}$ which creates a $D \times D$ lower triangular matrix. Then for vectors $\boldsymbol{x}_i \in \mathbb{R}^D$, $\boldsymbol{q}_k \in \mathbb{R}^D$, $\boldsymbol{l}_k \in \mathbb{R}^{D(D-1)/2}$, $\boldsymbol{\mu}_k \in \mathbb{R}^D$, and $\boldsymbol{\alpha} \in \mathbb{R}^K$, we define

$$\begin{aligned} L(\boldsymbol{\alpha}, \mathbf{M}, \mathbf{Q}, \mathbf{L}) \coloneqq & \sum_{i=1}^{N} \operatorname{logsumexp}\left(\left[\alpha_k + \operatorname{sum}(\boldsymbol{q}_k) - \frac{1}{2}||Q(\boldsymbol{q}_k, \boldsymbol{l}_k)(\boldsymbol{x}_i - \boldsymbol{\mu}_k)||^2\right]_{k=1}^{K}\right) \\ & - N \operatorname{logsumexp}\left([\alpha_k]_{k=1}^{K}\right) \\ & + \frac{1}{2}\sum_{k=1}^{K}\left(||\exp(\boldsymbol{q}_k)||^2 + ||\boldsymbol{l}_k||^2\right) - m\operatorname{sum}(\boldsymbol{q}_k) \end{aligned} \tag{6.1}$$

where we have matrices $\mathbf{M} \coloneqq [\boldsymbol{\mu}_k]_{k=1}^{K}$, $\mathbf{Q} \coloneqq [\boldsymbol{q}_k]_{k=1}^{K}$, and $\mathbf{L} \coloneqq [\boldsymbol{l}_k]_{k=1}^{K}$. The derivation of this objective function and the definition of Q can be found in [Šrajer, Kukelova, and Fitzgibbon, 2018a]. The variables $\boldsymbol{\alpha}$, $\mathbf{M}$, $\mathbf{Q}$, and $\mathbf{L}$ are the independent variables which we must find the derivatives of, where the $\boldsymbol{x}_i$'s have a fixed value. The dimensions of the independent variables will change depending on N, K, and D and the total number of independent variables will be the increasing parameter which we measure time against.

We implement the above function using the Owl scientific computing library [L. Wang, Zhao, and Mortier, 2022][6]. Doing so gives us access to primitive operations such as summation and transposition on tensors (n-dimensional arrays). Thus, our family of smooth functions can now include tensor-valued operations. Owl itself can perform AD, but we do not use this feature. The new version of `Smooth` can be found in Appendix A. Consequently, the number of effectful operations greatly decreases, e.g. 999 uses of binary addition for a 1000 element vector versus 1 summation operation, which reduces the overhead of effect handling. Besides the change to operations involving tensors, the structure of reverse mode is the same, which can be seen in Appendix A. Finally, the actual implementation of the function L can also be found in Appendix A.

These benchmarks have been run on a Dell Precision T3600 with a quad core (3.60 GHz boost) Intel Xeon E5-1620, 4×8GB = 32GB 1600 MHz DDR4, and 256 GB SATA 6 Gb/s SSD. The operating system used is headless Debian 12 (bookworm) with Linux kernel release 6.1.0-18-amd64. The results of our implementation along with the other systems is summarized in fig. 6.6 and fig. 6.7, where the x-axis is the number of independent variables and the y-axis is the amount of time to compute the Jacobian, with each axis logarithmic scale. The

[6] `https://ocaml.xyz/`

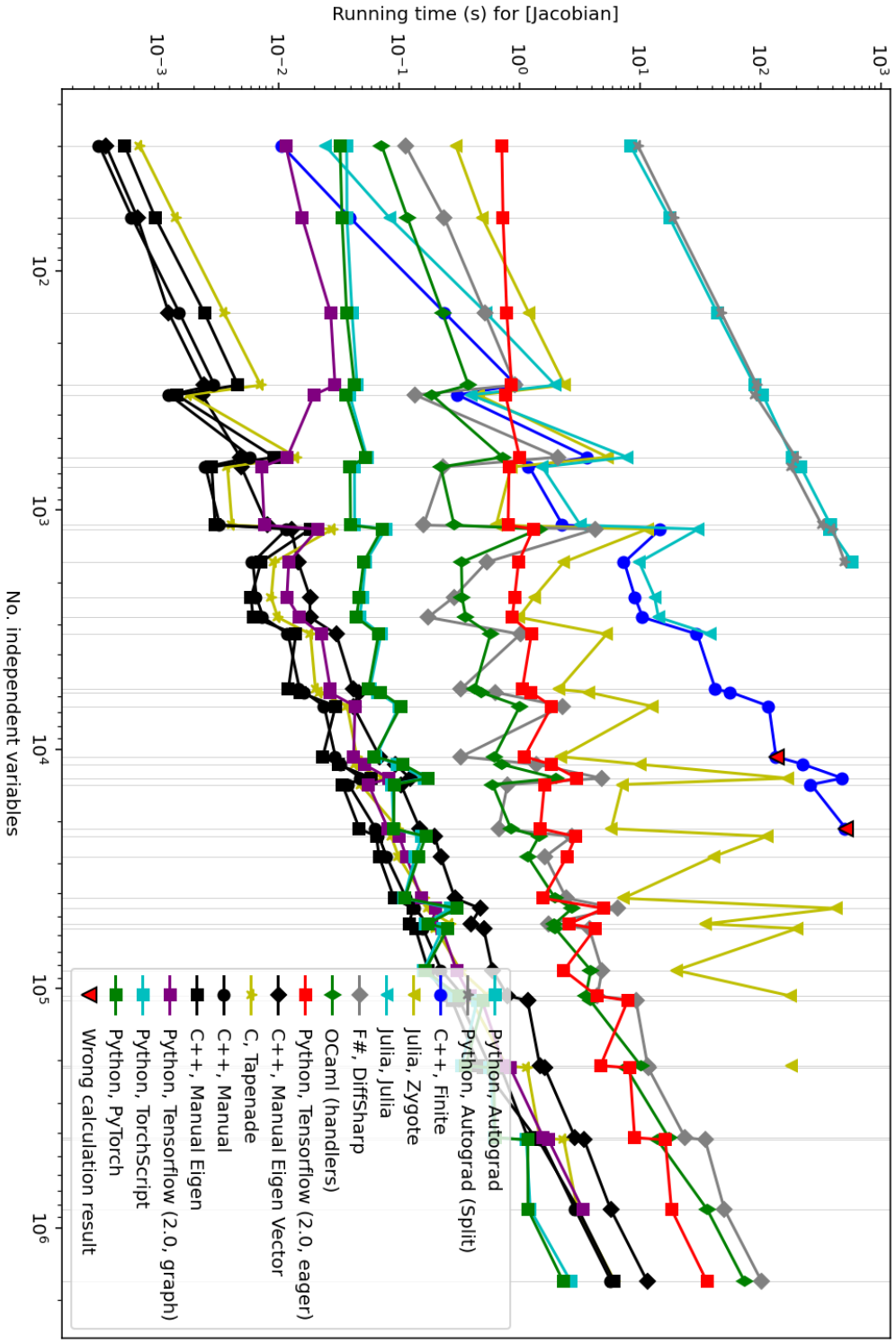


Figure 6.6: GMM results, N = 1,000

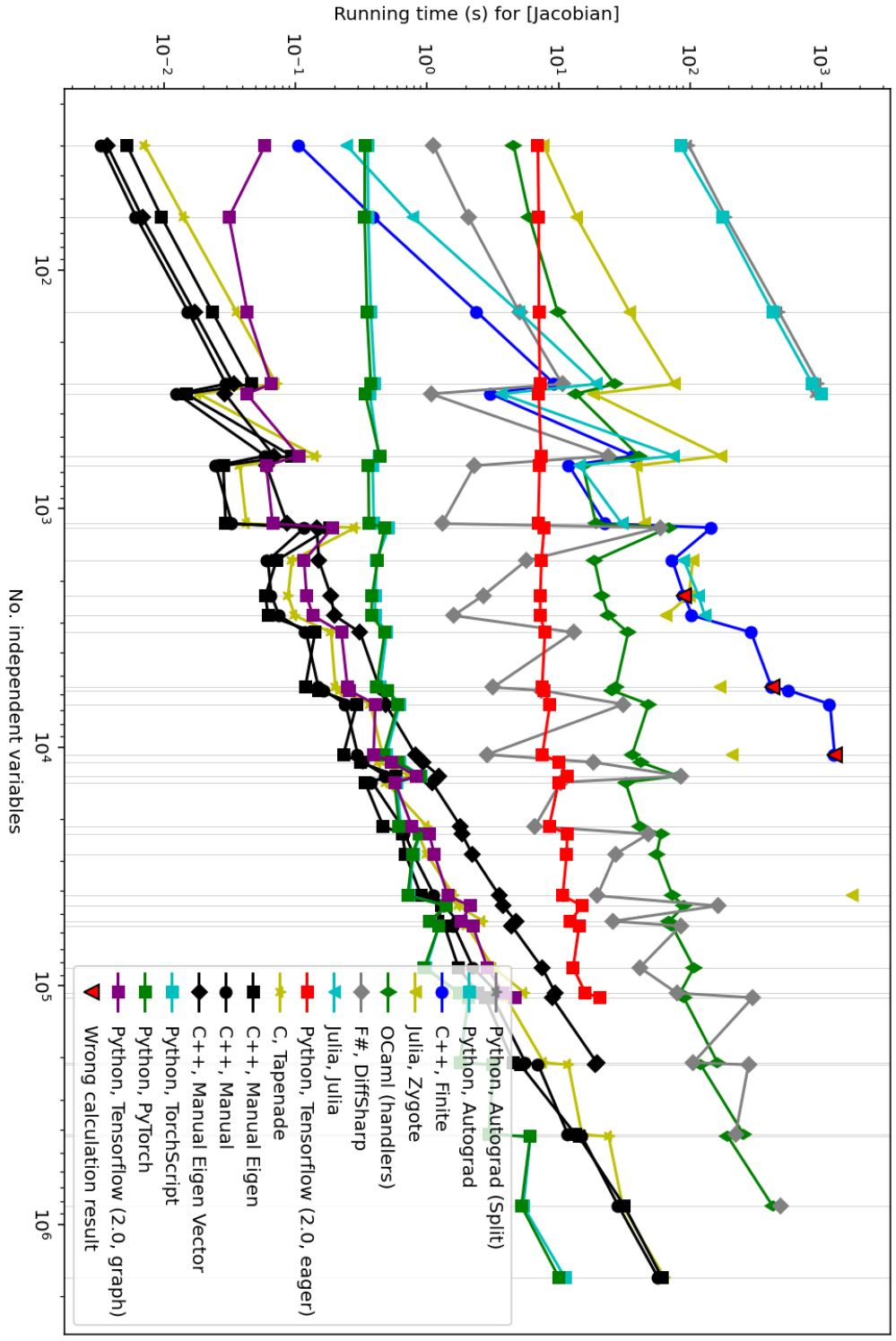


Figure 6.7: GMM results, N = 10,000

input data for fig. 6.6 always has $N = 1,000$, while fig. 6.7 always has $N = 10,000$, and we note that this does not effect the number of independent variables. In both figures, our implementation is more performant in the long run than: finite differences (C++), Autograd (Python), Zygote (Julia), and pure Julia (Julia). For $N = 1,000$ we are competitive with eager TensorFlow 2.0 (Python), although we are not for $N = 10,000$. Finally, we are competitive with DiffSharp (F#) in both instances. Of the previous systems, the only define-by-run system we do not out perform is eager Tensorflow. Furthermore, the remaining seven implementations which outperform us are either handcrafted, source transformations, or define-then-run systems. Therefore, we substantiate our claim that we are a competitive define-by-run system.

Part III

Correctness

Chapter 7

Mathematical Tools

Rose

We have presented AD algorithms implemented with effects and handlers, and have given example programs to illustrate correctness. However, it is clearly preferable to be able to prove correctness. This chapter will recall and develop the necessary tools to do so. Section 7.1 presents the semantic basis for effects and handlers, which are expressed using initial algebras and free monads. Section 7.2 provides a model effect handler language in which to reason about our programs, both operationally and denotationally. Finally, section 7.3 sets out our main proof technique of logical relations for our model language, suitably generalized to account for differentiation, which will allow correctness proofs via the denotational semantics.

We assume that the reader is familiar with basic category theory concepts such as functors, natural transformations, adjunctions, monads, and limits. We also assume the reader is familiar with strong functors and monads. Finally, we assume the reader is acquainted with bi-cartesian closed categories (bi-CCCs),

meaning a cartesian closed category with finite coproducts. Let us fix some notation for bi-CCCs. For a bi-CCC $\mathcal{C}$, we write 1 for the terminal object, 0 for the initial object, $\times$ and $\prod$ for binary and finite products, $+$ and $\coprod$ for binary and finite coproducts, and $\Rightarrow$ for exponentials. We write ! for terminal maps, ? for initial maps, π_i for projections, $\langle -, - \rangle$ for tupling, ι_i for coprojections, $[-, -]$ for cotupling, and eval for evaluation. We often leave the currying and uncurrying operations implicit, but mention when they are used.

7.1 Initial Algebras and Free Monads

Given a functor $F \colon \mathcal{C} \to \mathcal{C}$, an *$F$-algebra* is a pair $\langle A, \alpha \colon FA \to A \rangle$. One can view F as specifying some data, and thus an F-algebra is an object A along with a structure map showing how this data interacts with A. When the object component of an algebra is clear, we may sometimes leave it implicit. A *map of F-algebras* $f \colon \langle A, \alpha \rangle \to \langle B, \beta \rangle$ is a map $f \colon A \to B$ such that $f \cdot \alpha = \beta \cdot Ff$. Thus, such a map preserves how the data interacts with each algebra. The category **Alg** F has F-algebras as objects and maps of F-algebras as morphisms. The category **Alg** F can be used to characterize induction and recursion for structures based on the data that F specifies. An important example of this characterization is the existence of an *initial* F-algebra.

Definition 7.1.1 (Awodey, 2010, pg. 268)**.** An algebra for F is *initial* if it admits a unique homomorphism into every F-algebra, i.e. it is the initial object of **Alg** F.

The initial F-algebra does not necessarily exist. When it does, the object component is often denoted by $\mu X.FX$ or μF. An important property of initial algebras is that they are fixed points, i.e. $\mu F \simeq F(\mu F)$.

Lemma 7.1.2 (Lambek's lemma [Lambek, 1968])**.** *Suppose $F \colon \mathcal{C} \to \mathcal{C}$ has an initial algebra $\iota \colon F(\mu F) \to \mu F$. Then ι is an isomorphism.*

Let us look at a familiar example of a structure supporting induction and recursion, namely the natural numbers, through the lens of initial algebras.

Example 7.1.3. Define the functor $FX \coloneqq X + 1$ on **Set** where $1 \coloneqq \{\star\}$ is a singleton set. An F-algebra is a pair $\langle A, \alpha \colon A + 1 \to A \rangle$. The map α is equivalent to a pair of maps $z \colon 1 \to A$ and $s \colon A \to A$. The natural numbers $\mathbb{N}$ are the initial F-algebra, with the algebra structure $\mathbb{N} + 1 \to \mathbb{N}$ given by $[succ, 0]$ where $succ$

is the successor function. Given another F-algebra $\langle A, [s, z] \colon A + 1 \to A \rangle$, the unique map $f \colon \mathbb{N} \to A$ is given by recursion so that $f(0) = z(\star)$ and $f(succ(n)) = s(f(n))$. The fact that this f is unique is equivalent to induction on the natural numbers.

Another important class of F-algebras are the free F-algebras. Given any object $A \in \mathcal{C}$, we would like to characterize what it means for A to have a "minimal" F-algebra structure.

Definition 7.1.4 (Adámek, Milius, and Moss, 2021, Remark 2.2.19)**.** A *free F-algebra* on an object A in $\mathcal{C}$ is an algebra $\varphi_A \colon FA^\sharp \to A^\sharp$ together with a universal arrow $\eta_A \colon A \to A^\sharp$. Universality here means that for every $\beta \colon FB \to B$ and every morphism $f \colon A \to B$ in $\mathcal{C}$, there exists a unique homomorphism $\bar{f} \colon A^\sharp \to B$ extending f, i.e. a unique morphism of $\mathcal{C}$ for which the diagram below commutes:

$$\begin{array}{ccccc} FA^\sharp & \xrightarrow{\varphi_A} & A^\sharp & \xleftarrow{\eta_A} & A \\ {\scriptstyle F\bar{f}}\downarrow & & {\scriptstyle \bar{f}}\downarrow & \swarrow_{f} & \\ FB & \xrightarrow[\beta]{} & B & & \end{array}$$

In the case where $\mathcal{C}$ has binary coproducts, we can reduce the above diagram to a commutative square as follows:

$$\begin{array}{ccc} FA^\sharp + A & \xrightarrow{[\varphi_A, \eta_A]} & A^\sharp \\ {\scriptstyle F\bar{f} + \mathrm{id}_A}\downarrow & & \downarrow{\scriptstyle \bar{f}} \\ FB + A & \xrightarrow[[\beta, f]]{} & B \end{array}$$

Note that for an endofunctor F and a fixed object A, we get a new functor $F(-) + A$ defined in the obvious way, and the above square shows $\bar{f}$ is an algebra homomorphism.

Let us look at a familiar example of free algebras.

Example 7.1.5. Define the functor $FX \coloneqq X \times X$ on **Set**. An F-algebra $\langle A, \alpha \colon A \times A \to A \rangle$ is a set A and a binary operation. For a set T, the free F-algebra $T^\sharp$ on T is the set of finite binary trees with leaves labelled by elements of T. Thus, $T^\sharp$ is the set inductively defined by $\mathsf{leaf}(t) \in T^\sharp$ for each $t \in T$ and $\mathsf{branch}(t_1, t_2) \in T^\sharp$ for each $t_1, t_2 \in T^\sharp$. Given $\langle A, \alpha \rangle$ and a map $f \colon T \to A$, there is a unique map $\overline{f} \colon T^\sharp \to A$ defined by recursion such that $\overline{f}(\mathsf{leaf}(t)) = f(t)$ and $\overline{f}(\mathsf{branch}(t_1, t_2)) = \alpha(\overline{f}(t_1), \overline{f}(t_2))$.

Initial algebras and free algebras are closely related.

Proposition 7.1.6 (Adámek, Milius, and Moss, 2021, Proposition 2.2.20)**.** *Let $\mathcal{C}$ be a category with finite coproducts. For every endofunctor F, the free F-algebra on A is precisely the initial algebra for $F(-) + A$. That is, if $A^\sharp$ is free, then $\mu X.(FX + A) = A^\sharp$ with the algebra structure $[\varphi_A, \eta_A]$, and vice versa.*

Thus, free algebras for such a $\mathcal{C}$ have a notion of induction and recursion similar to that of initial algebras. However, even when $\mathcal{C}$ does not have coproducts, there are other interesting properties of free algebras, particularly when all free algebras exist.

Definition 7.1.7 (Adámek, Milius, and Moss, 2021, Definition 2.2.22)**.** A functor F is a *varietor* if every object generates a free algebra for F.

It is natural to ask if the construction of free algebras for a varietor F is functorial. This is indeed the case, and in fact the construction forms a monad.

Proposition 7.1.8 (Adámek, Milius, and Moss, 2021, Remark 2.2.23)**.** *A functor F is a varietor if and only if the forgetful functor from $U\colon \mathbf{Alg}\ F \to \mathcal{C}$ has a left adjoint $A \mapsto A^\sharp$. This defines a monad $\mathcal{M} = (M, \eta, \mu)$ on $\mathcal{C}$ by $MA = A^\sharp$.*

Sketch: For a morphism $f\colon A \to B$, we have a unique homomorphism of F-algebras $Mf : A^\sharp \to B^\sharp$ with $Mf \cdot \eta_A = \eta_B \cdot f$. The unit $\eta : \mathrm{Id} \to M$ has components the universal arrows $\eta_A\colon A \to A^\sharp$, and the monad multiplication $\mu\colon MM \to M$ has as its components the unique F-algebra homomorphism $\mu_A\colon (A^\sharp)^\sharp \to A^\sharp$ with $\mu_A \cdot \eta_{A^\sharp} = \mathrm{id}_{A^\sharp}$. □

The monad $\mathcal{M}$ of free F-algebras naturally has a category of algebras $\mathbf{Alg}\ \mathcal{M}$. Recall $U\colon \mathbf{Alg}\ F \to \mathcal{C}$ is *monadic* if it has a left adjoint and $\mathbf{Alg}\ M \simeq \mathbf{Alg}\ F$.

Theorem 7.1.9 (Barr and Wells, 2005, Ch. 9, Proposition 4.1)**.** *If U has a left adjoint, then it is monadic.*

Thus we have three equivalent conditions, $U\colon \mathbf{Alg}\ F \to \mathcal{C}$ has a left adjoint if and only if $\mathbf{Alg}\ \mathcal{M} \simeq \mathbf{Alg}\ F$ if and only if F is a varietor. There is another, weaker, notion of free monad.

Definition 7.1.10 (Barr, 1970, sec. 5)**.** Let $F\colon \mathcal{C} \to \mathcal{C}$ be a functor. We say that $\mathcal{M} = (M, \eta, \mu)$ is the *free monad generated by* F if there exists a natural

transformation $\tau\colon F \to M$ such that for any other monad $\mathcal{M}' = (M', \eta', \mu')$ equipped with a natural transformation $\tau'\colon F \to M'$, there exists a unique monad morphism θ such that $\tau' = \theta \cdot \tau$.

[Barr, 1970] showed that the monad $\mathcal{M}$ induced by the free-forgetful adjunction for **Alg** F is free in the above sense, where the natural transformation $F \to M$ is given by $\varphi_A \cdot F\eta_A$ where $\varphi_A\colon FMA \to MA$ is the free F-algebra map [Barr, 1970]. The converse is not true, a free monad via a natural transformation $F \to M$ does not imply that the forgetful functor $U\colon$ **Alg** $F \to \mathcal{C}$ has a left adjoint. Nonetheless, there is a partial converse.

Theorem 7.1.11 (Adámek, Milius, and Moss, 2021, Theorem 2.2.24, Barr, 1970, Corollary 5.10, Kelly, 1980, Proposition 22.4)**.** *Given a locally small, complete category, an endofunctor F generates a free monad if and only if it is a varietor.*

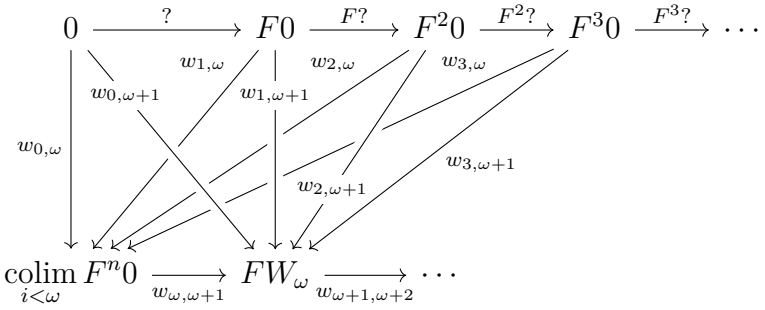
Our denotational semantics will require the free algebra monad for an endofunctor F, and so when we say *free monad*, we mean the free algebra monad.

A useful method of constructing initial algebras exists in categories with colimits of chains, whereby we can apply proposition 7.1.6 to produce free algebras. The construction mirrors that of inductively defined sets. We begin with defining a ordinal indexed collection of objects.

Definition 7.1.12 (Adámek, Milius, and Moss, 2021, Definition 6.1.4)**.** Let $\mathcal{C}$ be a category with colimits of chains. For an endofunctor F we define the *initial-algebra chain* $W\colon$ **Ord** $\to \mathcal{C}$. Its objects are denoted by W_i and its connecting morphisms by $w_{i,j}\colon W_i \to W_j$, $i \leq j \in$ **Ord**. They are defined by transfinite recursion:

$$\begin{aligned} W_0 &\coloneqq 0, \\ W_{j+1} &\coloneqq FW_j && \text{for all ordinals } j, \\ W_j &\coloneqq \operatorname*{colim}_{i<j} W_i && \text{for all limit ordinals } j, \text{ and} \\ w_{0,1} &\coloneqq ?_{F0}, \\ w_{j+1,k+1} &= Fw_{j,k}, \\ w_{i,j} &\coloneqq \text{colimit cocone for limit ordinals } j \text{ for all } i < j. \end{aligned}$$

Note 7.1.13 (Adámek, Milius, and Moss, 2021, Remark 6.1.5)**.** A portion of the initial-algebra chain is shown below:



The above definition does not mention $w_{\omega,\omega+1}$ because $w_{\omega,\omega+1} \colon W_\omega \to FW_\omega$ is uniquely determined by universality of the colimit $\mathrm{colim}_{i<\omega} F^n 0$, and likewise for other limit ordinals.

To see why, leave out 0 in the ω-chain prefix. We get the same colimit, i.e. $\mathrm{colim}_{i+1<\omega} W_{i+1}$ is isomorphic to W_ω, with cocone given by $w_{i+1,\omega} \colon W_{i+1} \to W_\omega$ for $i + 1 < \omega$. We also see that the morphisms $w_{i+1,\omega+1} \coloneqq F w_{i,\omega}$ form a cocone. Hence, we obtain $w_{\omega,\omega+1}$ uniquely such that $w_{\omega,\omega+1} \cdot w_{i+1,\omega} = F w_{i,\omega}$. Furthermore, proceeding in a similar fashion, we obtain $w_{i,j}$ for all pairs of ordinals $i \leq j$.

The initial-algebra chain may or may not contain the initial algebra for F. For example, the powerset functor $\mathcal{P} \colon \mathbf{Set} \to \mathbf{Set}$ does not have any fixed-points, and thus cannot have an initial algebra. When one of the connecting morphisms is an isomorphism, the chain does produce the initial algebra, and we say the chain *converges*.

Definition 7.1.14 (Adámek, Milius, and Moss, 2021, Definition 6.1.10)**.** We say that the initial-algebra chain of a functor F *converges in* λ *steps* if $w_{\lambda,\lambda+1}$ is an isomorphism. We say that the chain converges in *exactly* λ *steps* if λ is the least such ordinal.

The previous definition only considers connecting morphisms from the successor case, but this is in fact sufficient.

Proposition 7.1.15 (Adámek, Milius, and Moss, 2021, Proposition 6.1.11)**.** *If* $i < j$ *and* $w_{i,j}$ *is an isomorphism, then so is* $w_{i,i+1}$*.*

We now record the theorem for which the initial-algebra chain was defined, along with a useful corollary.

Theorem 7.1.16 (Adámek, Milius, and Moss, 2021, Theorem 6.1.12)**.** *Let $\mathcal{C}$ be a category with colimits of chains. If the initial-algebra chain of an endofunctor F converges in j steps, then W_j is the initial algebra with the algebra structure*

$$w_{j,j+1}^{-1} \colon FW_j \to W_j.$$

Corollary 7.1.17 (Adámek, Milius, and Moss, 2021, Corollary 6.1.13)**.** *Let $\mathcal{C}$ be a category with colimits of chains. Let $F \colon \mathcal{C} \to \mathcal{C}$ preserve colimits of λ-chains for some ordinal λ. Then the initial-algebra chain of F converges in λ steps. Therefore, $\mu F = W_\lambda$.*

Note that when when a endofunctor F preserves colimits of λ-chains, then so does $A + F(-)$ for any $A \in \mathcal{C}$. Thus, we derive the following corollaries.

Corollary 7.1.18 (Adámek, Milius, and Moss, 2021, Corollary 6.1.13)**.** *Let $\mathcal{C}$ be a cocomplete category. Every endofunctor preserving colimits of λ-chains for some infinite cardinal λ is a varietor.*

Corollary 7.1.19. *Let $\mathcal{C}$ be a complete and cocomplete category. Every endofunctor preserving colimits of λ-chains for some infinite cardinal λ generates a free monad.*

Suppose the free algebra monad for an endofunctor $F \colon \mathcal{C} \to \mathcal{C}$ exists and denote it by $\mathcal{M} = (M \colon \mathcal{C} \to \mathcal{C}, \mu, \eta)$. By the freeness of $\mathcal{M}$, we have a map of sets $\Phi \colon \mathcal{C}(FB, B) \times \mathcal{C}(A, B) \to \mathcal{C}(MA, B)$ for all $A, B \in \mathcal{C}$. We will later work with a cartesian closed category (CCC) $\mathcal{C}$, and we will need to know when this map can be internalized to $\mathcal{C}$ as a map $\overline{\Phi} \colon (FB \Rightarrow B) \times (A \Rightarrow B) \to (MA \Rightarrow B)$. To be precise, the forgetful functor $\mathcal{C}(1, -) \colon \mathcal{C} \to \mathbf{Set}$ satisfies the natural isomorphisms

$$\mathcal{C}(1, A \Rightarrow B) \simeq \mathcal{C}(A, B) \qquad \mathcal{C}(1, A \times B) \simeq \mathcal{C}(1, A) \times \mathcal{C}(1, B)$$

and thus we want $\mathcal{C}(1, \overline{\Phi}) = \Phi$ up to these isomorphisms.

To do so we will first consider an endofunctor $F \colon \mathcal{C} \to \mathcal{C}$ which has an initial algebra μF and thus a map of sets $\mathrm{rec} \colon \mathcal{C}(FA, A) \to \mathcal{C}(\mu F, A)$ for all $A \in \mathcal{C}$. We will show that when F is strong that rec internalizes to a map $\overline{\mathrm{rec}} \colon (FA \Rightarrow A) \to (\mu F \Rightarrow A)$ such that $\mathcal{C}(1, \overline{\mathrm{rec}}) = \mathrm{rec}$.

We begin by defining an F-algebra on $A^{FA \Rightarrow A}$, making use of the exponential

notation for exponential objects. The map is defined as the curried version of

$$(FA \Rightarrow A) \times F\left(A^{FA\Rightarrow A}\right) \xrightarrow{\Delta\times\mathrm{id}} (FA \Rightarrow A) \times (FA \Rightarrow A) \times F\left(A^{FA\Rightarrow A}\right)$$

$$\xleftarrow{\mathrm{id}\times\mathrm{str}}$$

$$(FA \Rightarrow A) \times F\left((FA \Rightarrow A) \times \left(A^{FA\Rightarrow A}\right)\right) \xrightarrow{\mathrm{id}\times F\mathrm{eval}} (FA \Rightarrow A) \times FA \xrightarrow{\mathrm{eval}} A$$

and denote the result by $\omega\colon F\left(A^{FA\Rightarrow A}\right) \to A^{FA\Rightarrow A}$. The suggestion of this construction is courtesy of Ohad Kammar. The induced map $\mathrm{rec}\,\omega\colon \mu F \to A^{FA\Rightarrow A}$ is a map of algebras and thus we have the following commutative diagram

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{F\,\mathrm{rec}\,\omega} & F\left(A^{FA\Rightarrow A}\right) \\ {\scriptstyle\iota}\downarrow & & \downarrow{\scriptstyle\omega} \\ \mu F & \xrightarrow[\mathrm{rec}\,\omega]{} & A^{FA\Rightarrow A} \end{array}$$

We will show that the map given by swapping the arguments of $\mathrm{rec}\,\omega$ is our required map $\overline{\mathrm{rec}}$.

Theorem 7.1.20. *The map* $\overline{\mathrm{rec}}$ *derived from* $\mathrm{rec}\,\omega$ *is an internalization of* rec.

Proof. Let $\alpha\colon FA \to A$ be an algebra and define $\ulcorner\alpha\urcorner := \mathcal{C}(1,\alpha)\colon 1 \to (FA \Rightarrow A)$. We now define a map $\mu F \to A$ by $\mathrm{eval}\cdot(\ulcorner\alpha\urcorner \times \mathrm{id})\cdot\lambda^{-1}\cdot\mathrm{rec}\,\omega$ (see fig. 7.1a) which we will show is equal to $\mathrm{rec}\,\alpha$. By the initiality of μF, it is sufficient to show that our defined map is a map of F-algebras.

Figure 7.1b shows the commutative diagram which must commute for our map to be a map of F-algebras. The rightmost polygon requires more work, and is expanded to fig. 7.1c, where we can uncurry ω. Finally, the upper right triangle of fig. 7.1c precomposed with $\ulcorner\alpha\urcorner \times \mathrm{id}$ is expanded to fig. 7.2, which commutes.

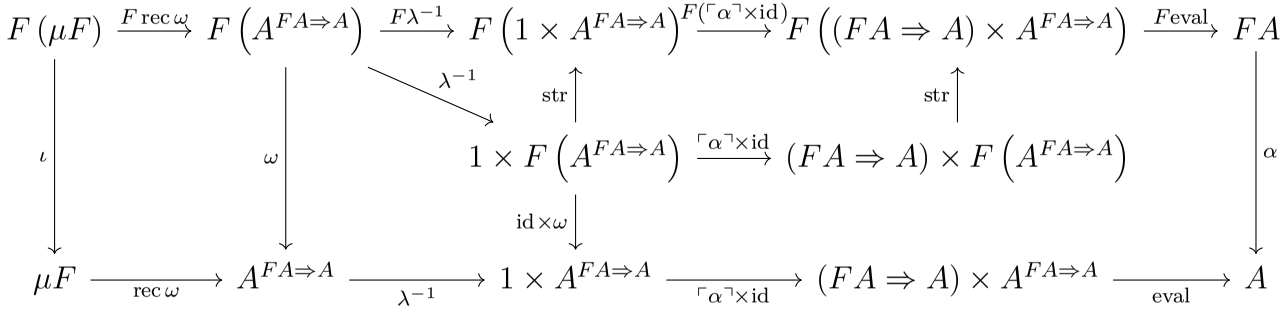
Thus, we have shown $\mathrm{rec}\,\alpha = \mathrm{eval}\cdot(\ulcorner\alpha\urcorner \times \mathrm{id})\cdot\lambda^{-1}\cdot\mathrm{rec}\,\omega$, and the later is equivalent to $\overline{\mathrm{rec}}\cdot\ulcorner\alpha\urcorner$ under the isomorphisms induced by $\mathcal{C}(1,-)$. In conclusion, $\mathcal{C}(1,\overline{\mathrm{rec}}) = \mathrm{rec}$ as desired. □

We can now prove our theorem about free monads.

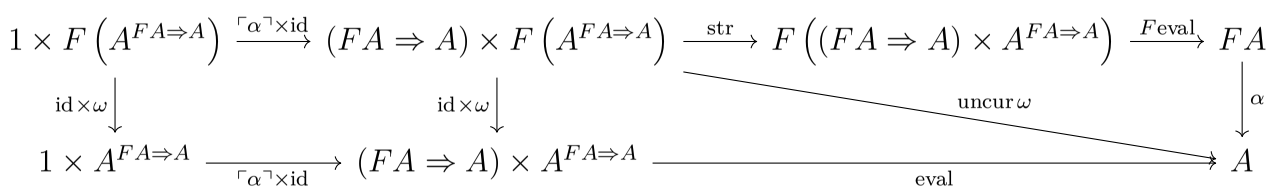
Theorem 7.1.21. *Let* $\mathcal{C}$ *be a bi-CCC and* F *a strong endofunctor such that the free monad* $\mathcal{M}$ *exists. Then the construction of universal maps out of* $\mathcal{M}$, $\varphi\colon \mathcal{C}(FB,B)\times\mathcal{C}(A,B) \to \mathcal{C}(MA,B)$, *can be internalized to* $\mathcal{C}$, *meaning there is a morphism* $\overline{\varphi}\colon (FB \Rightarrow B)\times(A \Rightarrow B) \to (MA \Rightarrow B)$ *for all* $A, B \in \mathcal{C}$ *such that* $\mathcal{C}(1,\overline{\varphi}) = \varphi$.

$$\mu F \xrightarrow{\operatorname{rec}\omega} A^{FA\Rightarrow A} \xrightarrow{\lambda^{-1}} 1 \times A^{FA\Rightarrow A} \xrightarrow{\ulcorner\alpha\urcorner\times\mathrm{id}} (FA \Rightarrow A) \times A^{FA\Rightarrow A} \xrightarrow{\mathrm{eval}} A$$

(a) Morphism to be shown equal to $\operatorname{rec}\alpha$



(b) Algebra map diagram for $\operatorname{rec}\alpha$



(c) Rightmost polygon of above diagram

Figure 7.1: Diagrams for theorem 7.1.20

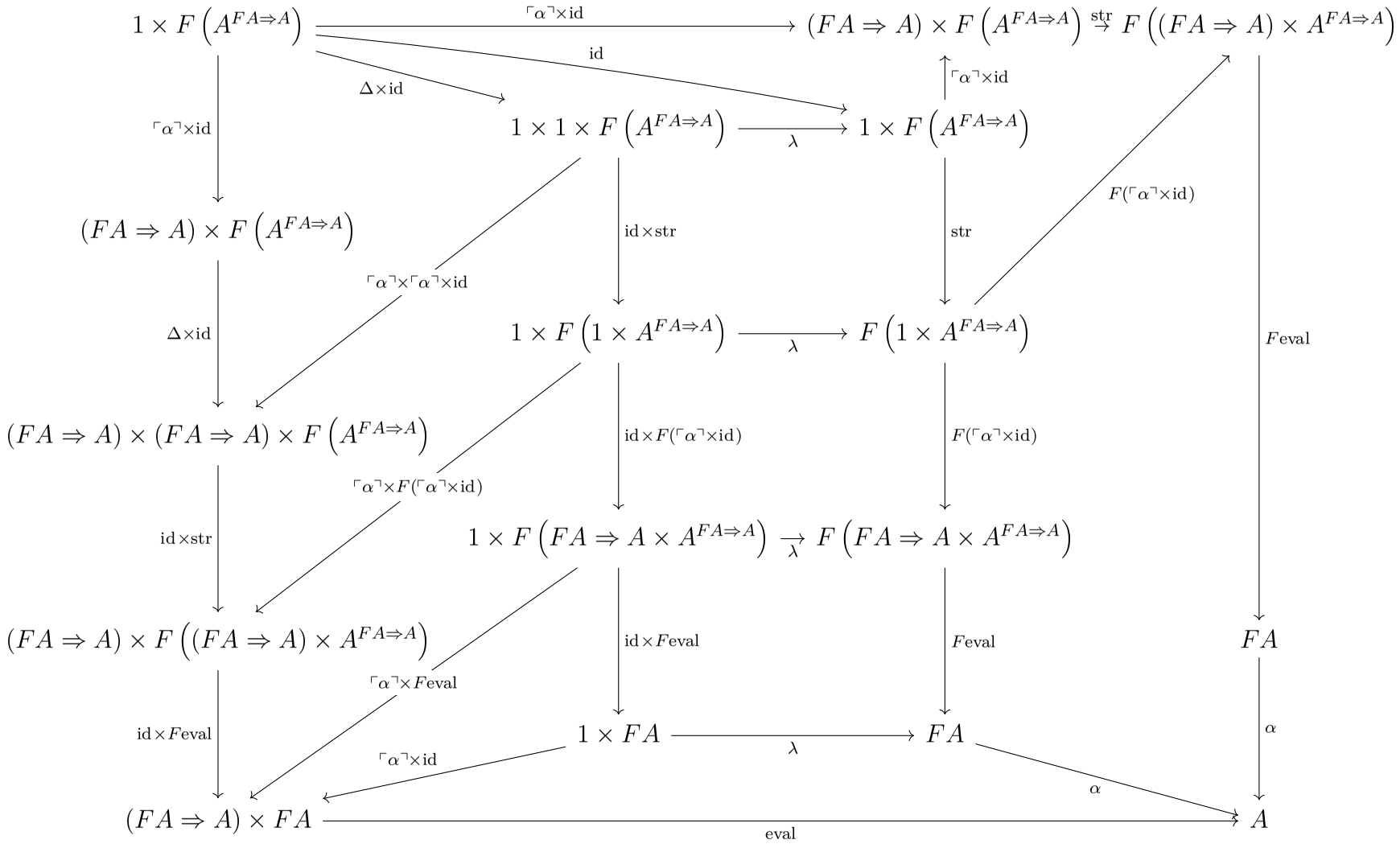


Figure 7.2: Upper right triangle of fig. 7.1c precomposed with $\ulcorner\alpha\urcorner \times \mathrm{id}$

Proof. Proposition 7.1.6 shows that MA is the initial algebra for $A + F(-)$. Furthermore, when F is strong so is $A+F(-)$. Thus, we can apply theorem 7.1.20 and use the isomorphism $(A + FB) \Rightarrow B \simeq (FB \Rightarrow B) \times (A \Rightarrow B)$. □

Another important requirement for our denotational semantics is that the free monad $\mathcal{M}$ is strong. Fortunately, $\mathcal{M}$ is strong when F is.

Proposition 7.1.22 (Fiore and Hur, 2008, Proposition 2.7)**.** *Suppose that the forgetful functor* $U\colon \mathbf{Alg}\ F \to \mathcal{C}$ *has a left adjoint and let* $\mathcal{M}$ *be the induced monad. Then* $\mathcal{M}$ *is a strong monad.*

7.2 Reasoning About Effects and Handlers

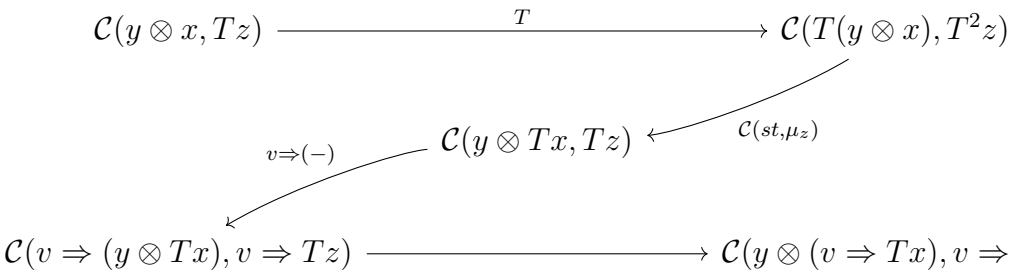
Algebraic effects are a class of effects corresponding to Kleisli morphisms and have an especially nice operational semantics. In particular, they can be characterized as parameterized continuation transformers which respect the strength and multiplication of the monad they derive from. The formalism of [Forster et al., 2019] is suitable for modelling algebraic effects and handlers, except that theirs is set-theoretic. We will need a categorical semantics, and so we begin by recalling the definition of algebraic effects in an enriched setting. We must use basic enriched category theory to give the definition of algebraic effects and state theorems about them. Readers unfamiliar with enriched category theory may assume that the category $\mathcal{C}$ is a CCC, replace $\mathcal{V}$ by $\mathcal{C}$, and replace all instances of $\mathcal{V}$-(concept) by (concept) (e.g. $\mathcal{V}$-functor by functor).

Let $\mathcal{V}$ be a complete and cocomplete symmetric monoidal category, $\mathcal{C}$ a symmetric monoidal $\mathcal{V}$-category with cotensors[1], and $\mathcal{T}\colon \mathcal{C} \to \mathcal{C}$ a strong $\mathcal{V}$-monad with strength $st\colon y \otimes Tx \to T(y \otimes x)$. Let $\mathcal{C}_\mathcal{T}$ denote the Kleisli $\mathcal{V}$-category for $\mathcal{T}$ and let $J\colon \mathcal{C} \to \mathcal{C}_\mathcal{T}$ denote the canonical $\mathcal{V}$-functor. We then say that $\mathcal{C}$ has Kleisli $\mathcal{V}$-exponentials when the $\mathcal{V}$-functor $J(- \otimes x)\colon \mathcal{C} \to \mathcal{C}_\mathcal{T}$ has a right $\mathcal{V}$-adjoint for each object $x \in \mathcal{C}$. We will also assume that $\mathcal{C}$ has Kleisli $\mathcal{V}$-exponentials. To define algebraic operations, we need the following construction.

Definition 7.2.1 ([Plotkin and Power, 2003])**.** Define the parametric lifting op-

[1]For readers unfamiliar with enriched category theory, the cotensor of an object $x \in \mathcal{C}$ by an object $v \in \mathcal{V} = \mathcal{C}$ is just the exponential $v \Rightarrow x$.

eration $(-)^\dagger$ by, for any $v \in \mathcal{V}$



where the unlabelled map is given by composition with the comparison map $y \otimes (v \Rightarrow Tx) \to v \Rightarrow (y \otimes Tx)$ in $\mathcal{C}$ corresponding, via the definition of cotensor, to the map $v \to \mathcal{C}(y \otimes (v \Rightarrow Tx), y \otimes Tx)$ in $\mathcal{V}$ given by the unit for the cotensor $v \to \mathcal{C}(v \Rightarrow Tx, Tx)$ composed with $y \otimes - \colon \mathcal{C} \to \mathcal{C}$.

The $(-)^\dagger$ operation is a form of parameterized lifting which is continuation focused. We now define algebraic operations.

Definition 7.2.2 (Plotkin and Power, 2003, Definition 1)**.** Given a strong $\mathcal{V}$-monad $\mathcal{T} \colon \mathcal{C} \to \mathcal{C}$, an algebraic operation on $\mathcal{T}$ is an Ob $\mathcal{C}$-indexed family of maps:

$$\alpha_x \colon (v \Rightarrow Tx) \to (w \Rightarrow Tx)$$

such that the diagram:

$$\begin{array}{ccc} \mathcal{C}(y \otimes x, Tz) & \xrightarrow{(-)^\dagger} & \mathcal{C}(y \otimes (v \Rightarrow Tx), v \Rightarrow Tz) \\ {\scriptstyle (-)^\dagger}\downarrow & & \downarrow{\scriptstyle \mathcal{C}(y \otimes (w \Rightarrow Tv), \alpha_z)} \\ \mathcal{C}(y \otimes (w \Rightarrow Tx), w \Rightarrow Tz) & \xrightarrow[\mathcal{C}(y \otimes \alpha_x, w \Rightarrow Tz)]{} & \mathcal{C}(y \otimes (v \Rightarrow Tx), w \Rightarrow Tz) \end{array}$$

commutes.

The definition of an algebraic operation can be equivalently characterized by saying that α is natural in $\mathcal{C}$, respects strength, and respects multiplication.

Proposition 7.2.3 (Plotkin and Power, 2003, Proposition 1)**.** *An* Ob $\mathcal{C}$*-indexed family of maps:*

$$\alpha_x \colon (v \Rightarrow Tx) \to (w \Rightarrow Tx)$$

is an algebraic operation if and only if:

1. *α is natural in $\mathcal{C}$;*

2. *α respects st in the sense that:*

$$\begin{array}{ccccc} y \otimes (v \Rightarrow Tx) & \longrightarrow & v \Rightarrow (y \otimes Tx) & \xrightarrow{v \Rightarrow st} & v \Rightarrow T(y \otimes x) \\ \downarrow{\scriptstyle y \otimes \alpha_x} & & & & \downarrow{\scriptstyle \alpha_{y \otimes x}} \\ y \otimes (w \Rightarrow Tx) & \longrightarrow & w \Rightarrow (y \otimes Tx) & \xrightarrow[w \Rightarrow st]{} & w \Rightarrow T(y \otimes x) \end{array}$$

commutes, where the unlabelled maps are comparison maps determined by the universal property of cotensors; and

3. *α respects μ in the sense that:*

$$\begin{array}{ccc} v \Rightarrow T^2 x & \xrightarrow{v \Rightarrow \mu_x} & v \Rightarrow Tx \\ \downarrow{\scriptstyle \alpha_{Tx}} & & \downarrow{\scriptstyle \alpha_x} \\ w \Rightarrow T^2 x & \xrightarrow[w \Rightarrow \mu_x]{} & w \Rightarrow Tx \end{array}$$

commutes.

In the case of $\mathcal{V} = \mathcal{C} = \mathbf{Set}$, it is straightforward to prove that algebraic operations and Kleisli maps are in bijective correspondence. This is also true in the enriched case when $\mathcal{C}$ is $\mathcal{V}$-closed. Let $\mathbf{v}$ be the tensor of $v \in \mathcal{V}$ with $I \in \mathcal{C}$ (which will always exist for us).

Theorem 7.2.4 (Plotkin and Power, 2003, Theorem 2)**.** *If $\mathcal{C}$ is $\mathcal{V}$-closed, the $\mathcal{C}$-enriched Yoneda embedding induces a bijection between maps $\mathbf{w} \to \mathbf{v}$ in $\mathcal{C}_\mathcal{T}$ and algebraic operations $\alpha_x \colon (v \Rightarrow Tx) \to (w \Rightarrow Tx)$.*

We now introduce the formalism of [Forster et al., 2019]. They define a *multi-adjunctive metalanguage*, called MAM, as a base language. MAM extends the call-by-push-value (CBPV) of [Levy, 2003] with a type-and-effect system. Effects will be captured by various adjunctions, and thus the name. After defining MAM and recalling theorems about it, we will extend the set-theoretic semantics to a categorical semantics. Forster et al. extend MAM to a language EFF containing algebraic effects and handlers, which is the language we will use. To do so, we will also extend the set-theoretic semantics of EFF to a categorical semantics.

The raw term syntax of MAM is presented in fig. 7.3. There are two syntactic categories, values and computations. We have a countable number of variables $x, y, z \ldots$ and a countable set of variant constructor labels ℓ. The unit, pairing, and variant constructors are all standard. For a given computation M, we can form a thunk $\{M\}$ which suspends the computation. Product and variant values

$V, W ::=$ values		$M, N ::=$	computations
x	variable	**case** V **of**	product
$\mid ()$	unit value	$\quad (x_1, x_2) \to M$	matching
$\mid (V_1, V_2)$	pairing	$\mid$ **case** V **of** $\{$	variant
$\mid \mathbf{inj}_\ell\, V$	variant	$\quad \mathbf{inj}_{\ell_1}\, x_1 \to M_1$	matching
$\mid \{M\}$	thunk	$\quad \vdots$	
		$\quad \mathbf{inj}_{\ell_n}\, x_n \to M_n\}$	
		$\mid V!$	force
		$\mid$ **return** V	returner
		$\mid x \leftarrow M;\ N$	sequencing
		$\mid \lambda x.M$	abstraction
		$\mid M\ V$	application
		$\mid \langle M_1, M_2 \rangle$	pairing
		$\mid \mathbf{prj}_i\, M$	projection

Figure 7.3: MAM syntax

can be matched upon in the standard manner. Given a thunk V, we can resume a suspended computation by forcing it $V!$. A computation which does nothing but merely returns a value V is given by **return** V. We write sequencing using do-notation. Function computations abstract over values, and thus can only be applied to values. Finally, there are pairing and projection operations for computations.

The operational semantics of MAM is presented in fig. 7.4 in the style of [Felleisen and Friedman, 1987]. We have two kinds of frames, basic and computation frames, and two kinds of contexts, evaluation and hoisting contexts. Frames and contexts along with their versions of substitution are defined in the standard way. In MAM, there is no distinction between basic and computation frames, but there will be in EFF. Additionally, the hoisting frames are not used in MAM's operational semantics, but will be used in EFF's. Note that every context has exactly one hole. The reduction semantics are defined in terms of basic CBPV β-reductions. Note that at most one β-reduction can be applied to any term. Finally, any reducible term uniquely decomposes into an evaluation context and a β-reducible term, and therefore the semantics is deterministic.

The kinds and types of MAM are presented in fig. 7.5. There are four kinds,

Frames and contexts

$\mathcal{B}$	$::= x \leftarrow [\,];\ N \mid [\,]\ V \mid \mathbf{prj}_i\,[\,]$	basic frames
$\mathcal{F}$	$::= \mathcal{B}$	computation frames
$\mathcal{C}$	$::= [\ ] \mid \mathcal{C}[\mathcal{F}[\ ]]$	evaluation context
$\mathcal{H}$	$::= [\ ] \mid \mathcal{H}[\mathcal{B}[\ ]]$	hoisting context

Reduction $\boxed{M \rightsquigarrow M'}$

$$\frac{M \rightsquigarrow_\beta M'}{\mathcal{C}[M] \rightsquigarrow \mathcal{C}[M']}$$

Beta reduction $\boxed{M \rightsquigarrow_\beta M'}$

$$\begin{array}{lr}
(\times) & \mathbf{case}\ (V_1, V_2)\ \mathbf{of}\ (x_1, x_2) \to M \\
& \qquad \rightsquigarrow_\beta M[V_1/x_1, V_2/x_2] \\
(+) & \mathbf{case}\ \mathbf{inj}_\ell\, V\ \mathbf{of}\ \{\ldots \mathbf{inj}_\ell\, x \to M \ldots\} \\
& \qquad \rightsquigarrow_\beta M[V/x] \\
(F) & x \leftarrow \mathbf{return}\ V;\ M \rightsquigarrow_\beta M[V/x] \\
(U) & \{M\}! \rightsquigarrow_\beta M \\
(\to) & (\lambda x.M)\ V \rightsquigarrow_\beta M[V/x] \\
(\&) & \mathbf{prj}_i\,\langle M_1, M_2\rangle \rightsquigarrow_\beta M_i
\end{array}$$

Figure 7.4: MAM operational semantics

$K ::=$	kinds	$A, B ::=$	value types
$\mid \mathbf{Eff}$	effects	α	type variable
$\mid \mathbf{Val}$	values	$\mid 1$	unit
$\mid \mathbf{Comp}_E$	computations	$\mid A_1 \times A_2$	products
$\mid \mathbf{Ctxt}$	environments	$\mid \{\mathbf{inj}_{\ell_1} A_1$	variants
		$\mid \ldots \mid \mathbf{inj}_{\ell_n} A_n\}$	
$E ::=$	effects	$\mid U_E C$	thunks
$\emptyset$	pure effect		
		$C, D ::=$	computation types
Environments:		FA	returners
$\Theta ::= \alpha_1, \ldots, \alpha_n$		$\mid A \to C$	functions
$\Gamma, \Delta ::= x_1 : A_1, \ldots, x_n : A_n$		$\mid C_1 \,\&\, C_2$	products

Figure 7.5: MAM kinds and types

namely effects, value types, computation types, and environments. CBPV does not need a kind system, but the introduction of an effect system necessitates it. MAM is pure, and so there is only the empty effect. The value types are standard for CBPV with the addition of type variables and an effect label for thunks. Computation types are standard as well, except now they will be stratified by effects via kinding. Note that none of the computation types need to be labelled by an effect E; the effect E will be part of the computation kinding and computation typing judgments and so is not needed in the computation type formers. Finally, we have two sorts of environments written in list notation, one for value-type variables and one for variables with value types.

The kind system of MAM is presented in fig. 7.6. Due to the existence of type variables, all kinding judgments are with respect to a value-type variable environment. The singular effect, the empty effect, is well-kinded. Kinding contexts requires the value types involved are well-kinded. The rest of the kinding judgments are standard, with the exception of thunks. As thunk types carry their effect type, we must have a well-kinded computation type, which implies that the effect is well-kinded as a side-condition of computation type kinding.

The type system of MAM is presented in fig. 7.7. We make a slight extension of from Forster et al. by adding type annotations to variant injections and to the variable bound in function abstraction. These changes ensure that each

Effect kinding $\boxed{\Theta \vdash_{\mathrm{k}} E : \mathbf{Eff}}$ **Context kinding** $\boxed{\Theta \vdash_{\mathrm{k}} \Gamma : \mathbf{Ctxt}}$

$$\frac{}{\Theta \vdash_{\mathrm{k}} \emptyset : \mathbf{Eff}} \qquad \frac{\text{for all } x \in \mathrm{Dom}(\Gamma)\text{: } \Theta \vdash_{\mathrm{k}} \Gamma(x) : \mathbf{Val}}{\Theta \vdash_{\mathrm{k}} \Gamma : \mathbf{Ctxt}}$$

Value kinding $\boxed{\Theta \vdash_{\mathrm{k}} A : \mathbf{Val}}$

$$\frac{\alpha \in \Theta}{\Theta \vdash_{\mathrm{k}} \alpha : \mathbf{Val}} \qquad \frac{}{\Theta \vdash_{\mathrm{k}} 1 : \mathbf{Val}} \qquad \frac{\Theta \vdash_{\mathrm{k}} A_1 : \mathbf{Val} \quad \Theta \vdash_{\mathrm{k}} A_1 : \mathbf{Val}}{\Theta \vdash_{\mathrm{k}} A_1 \times A_2 : \mathbf{Val}}$$

$$\frac{\text{for every } 1 \leq i \leq n\text{: } \Theta \vdash_{\mathrm{k}} A_i : \mathbf{Val}}{\Theta \vdash_{\mathrm{k}} \{\mathbf{inj}_{\ell_1} A_1 \mid \ldots \mid \mathbf{inj}_{\ell_n} A_n\} : \mathbf{Val}} \qquad \frac{\Theta \vdash_{\mathrm{k}} C : \mathbf{Comp}_E}{\Theta \vdash_{\mathrm{k}} U_E C : \mathbf{Val}}$$

Computation kinding $\boxed{\Theta \vdash_{\mathrm{k}} C : \mathbf{Comp}_E}$ $(\Theta \vdash_{\mathrm{k}} E : \mathbf{Eff})$

$$\frac{\Theta \vdash_{\mathrm{k}} A : \mathbf{Val}}{\Theta \vdash_{\mathrm{k}} FA : \mathbf{Comp}_E} \qquad \frac{\Theta \vdash_{\mathrm{k}} A : \mathbf{Val} \qquad \Theta \vdash_{\mathrm{k}} C : \mathbf{Comp}_E}{\Theta \vdash_{\mathrm{k}} A \to C : \mathbf{Comp}_E}$$

$$\frac{\Theta \vdash_{\mathrm{k}} C_1 : \mathbf{Comp}_E \qquad \Theta \vdash_{\mathrm{k}} C_2 : \mathbf{Comp}_E}{\Theta \vdash_{\mathrm{k}} C_1 \,\&\, C_2 : \mathbf{Comp}_E}$$

Figure 7.6: MAM kind system

Value typing $\boxed{\Theta; \Gamma \vdash V : A}$ $(\Theta \vdash_{\mathrm{k}} \Gamma : \mathbf{Ctxt}, A : \mathbf{Val})$

$$\frac{(x : A) \in \Gamma}{\Theta; \Gamma \vdash x : A} \qquad \frac{}{\Theta; \Gamma \vdash () : 1} \qquad \frac{\Theta; \Gamma \vdash V_1 : A_1 \qquad \Theta; \Gamma \vdash V_2 : A_2}{\Theta; \Gamma \vdash (V_1, V_2) : A_1 \times A_2}$$

$$\frac{\Theta; \Gamma \vdash V : A_i}{\Theta; \Gamma \vdash \mathbf{inj}_{\ell_i}^{\{\ell_i, A_i\}_{i \in I}} V : \{\mathbf{inj}_{\ell_1} A_1 \mid \ldots \mid \mathbf{inj}_{\ell_n} A_n\}} \qquad \frac{\Theta; \Gamma \vdash_E M : C}{\Theta; \Gamma \vdash \{M\} : U_E C}$$

Computation typing $\boxed{\Theta; \Gamma \vdash_E M : C}$ $(\Theta \vdash_{\mathrm{k}} \Gamma : \mathbf{Ctxt}, E : \mathbf{Eff}, C : \mathbf{Comp}_E)$

$$\frac{\Theta; \Gamma \vdash V : A_1 \times A_2 \qquad \Theta; \Gamma, x_1 : A_1, x_2 : A_2 \vdash_E M : C}{\Theta; \Gamma \vdash_E \mathbf{case}\ V\ \mathbf{of}\ (x_1, x_2) \to M : C} \qquad \frac{\Theta; \Gamma \vdash V : U_E C}{\Theta; \Gamma \vdash_E V! : C}$$

$$\frac{\begin{array}{c}\Theta; \Gamma \vdash V : \{\mathbf{inj}_{\ell_1} A_1 \mid \cdots \mid \mathbf{inj}_{\ell_n} A_n\} \\ \text{for every } 1 \le i \le n\text{: } \Theta; \Gamma, x_i : A_i \vdash_E M_i : C\end{array}}{\Theta; \Gamma \vdash_E \mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inj}_{\ell_1} x_1 \to M_1; \cdots; \mathbf{inj}_{\ell_n} x_n \to M_n\} : C}$$

$$\frac{\Theta; \Gamma \vdash V : A}{\Theta; \Gamma \vdash_E \mathbf{return}\ V : FA} \qquad \frac{\Theta; \Gamma \vdash_E M : C_1 \,\&\, C_2}{\Theta; \Gamma \vdash_E \mathbf{prj}_i\, M : C_i}$$

$$\frac{\Theta; \Gamma \vdash_E M : FA \qquad \Theta; \Gamma, x : A \vdash_E N : C}{\Theta; \Gamma \vdash_E x \leftarrow M;\ N : C} \qquad \frac{\Theta; \Gamma, x : A \vdash_E M : C}{\Theta; \Gamma \vdash_E \lambda x^A.M : A \to C}$$

$$\frac{\Theta; \Gamma \vdash_E M : A \to C \qquad \Theta; \Gamma \vdash V : A}{\Theta; \Gamma \vdash_E M\ V : C} \qquad \frac{\Theta; \Gamma \vdash_E M_1 : C_1 \qquad \Theta; \Gamma \vdash_E M_2 : C_2}{\Theta; \Gamma \vdash_E \langle M_1, M_2 \rangle : C_1 \,\&\, C_2}$$

Figure 7.7: MAM type system

judgement has a unique typing derivation, and so our denotational semantics will be well defined on judgments. However, we will not write these annotations in practice as they have no important effect for us. All typing judgments are made with respect to a value-type variable context and a value context. Furthermore, computation typing judgments must be indexed by the effect at which the typing occurs, a difference from CBPV. The value typing judgments are standard, except for the thunk typing rule which now records the effect from the computation judgment at the type level. The computation typing judgments are also standard, barring the addition of the effect in scope, with the exception of forcing, which recovers the effect index from the type of the thunk.

The denotational semantics of MAM is set-theoretic. Forster et al. use the Kleisli presentation of monads, where a monad is a triple $\mathcal{T} = \langle T, \mathbf{return}, \gg\!= \rangle$ with $T \colon \mathbf{Set} \to \mathbf{Set}$ a functor, $\mathbf{return} \colon \mathrm{Id} \to T$ a natural transformation, and $\gg\!=$ a map from functions $f \colon X \to TY$ to functions $\gg\!= f \colon TX \to TY$, subject to well-known algebraic equations. The functorial action of T can be recovered as an internal map $\mathbf{fmap}\ f\ xs \coloneqq xs \gg\!= (\mathbf{return} \circ f)$. A $\mathcal{T}$-algebra $C = \langle |C|, c \rangle$ consists of a carrier $|C|$ and a algebra structure c as normal. The free $\mathcal{T}$-algebra $\langle TX, \gg\!= \mathrm{id} \rangle$ will be denoted as FX.

Effects

$$[\![\emptyset]\!]_\theta \coloneqq \langle \mathrm{Id} \colon \mathbf{Set} \to \mathbf{Set}, \mathrm{id} \colon \mathrm{Id} \to \mathrm{Id}, \mathrm{id} \colon \mathbf{Set}(X, \mathrm{Id}Y) \to \mathbf{Set}(\mathrm{Id}X, \mathrm{Id}Y) \rangle$$

Value types

$$[\![\alpha]\!]_\theta \coloneqq \theta(\alpha) \quad [\![1]\!]_\theta \coloneqq \{\star\} \quad [\![A_1 \times A_2]\!]_\theta \coloneqq [\![A_1]\!]_\theta \times [\![A_2]\!]_\theta \quad [\![U_E C]\!]_\theta \coloneqq \left|[\![C]\!]_\theta\right|$$

$$\left[\!\left[\{\mathbf{inj}_{\ell_1}\ A_1 \mid \ldots \mid \mathbf{inj}_{\ell_n}\ A_n\}\right]\!\right]_\theta \coloneqq (\{\ell_1\} \times [\![A_1]\!]_\theta) \cup \cdots \cup (\{\ell_n\} \times [\![A_n]\!]_\theta)$$

Computation types

$$[\![FA]\!]_\theta \coloneqq F[\![A]\!]_\theta \quad [\![A \to C]\!]_\theta \coloneqq \langle \left|[\![C]\!]_\theta\right|^{[\![A]\!]_\theta}, \lambda f_s . \lambda x . c(\mathbf{fmap}\ (\lambda f . f(x))\ f_s) \rangle$$

$$[\![C_1\ \&\ C_2]\!]_\theta \coloneqq \left\langle \left|[\![C_1]\!]_\theta\right| \times \left|[\![C_2]\!]_\theta\right|, \lambda c_s . \langle c_1(\mathbf{fmap}\ \pi_1\ c_s), c_2(\mathbf{fmap}\ \pi_2\ c_s) \rangle \right\rangle$$

Figure 7.8: MAM denotational semantics for types

The denotational semantics for types in MAM is presented in fig. 7.8. All semantic functions in MAM are parameterized by an assignment θ mapping type variables α in Θ to a set $\theta(\alpha)$. For a fixed θ, we associate to each

- effect: a monad $[\![\Theta \vdash_{\mathrm{k}} E : \mathbf{Eff}]\!]_\theta$, denoted by $\left\langle T_{[\![E]\!]_\theta}, \mathbf{return}^{[\![E]\!]_\theta}, \gg\!=^{[\![E]\!]_\theta} \right\rangle$ or $\mathcal{T}_{[\![E]\!]_\theta}$;

- value: a set $[\![\Theta \vdash_{\mathrm{k}} A : \mathbf{Val}]\!]_\theta$;
- E-computation: a $\mathcal{T}_{[\![E]\!]_\theta}$-algebra $[\![\Theta \vdash_{\mathrm{k}} C : \mathbf{Comp}_E]\!]_\theta$; and
- context: the set $[\![\Theta \vdash_{\mathrm{k}} \Gamma : \mathbf{Ctxt}]\!]_\theta \coloneqq \prod_{x \in \mathrm{Dom}(\Gamma)} \left[\!\left[\Gamma(x)\right]\!\right]_\theta$.

The pure effect is given the semantics of the identity monad. Value types have standard semantics, and type variables have their semantics determined by θ. Variant types corresponds to disjoint sums, and the empty variant type is the empty set. Computation types also have the standard semantics.

Value terms

$[\![x]\!]_\theta(\gamma) \coloneqq \pi_x(\gamma) \quad [\![(V_1, V_2)]\!]_\theta(\gamma) \coloneqq \left\langle [\![V_1]\!]_\theta(\gamma), [\![V_2]\!]_\theta(\gamma) \right\rangle \quad [\![()]\!]_\theta(\gamma) \coloneqq \star$

$[\![\mathbf{inj}_\ell\, V]\!]_\theta(\gamma) \coloneqq \left\langle \ell, [\![V]\!]_\theta(\gamma) \right\rangle \quad \left[\!\left[\{M\}\right]\!\right]_\theta(\gamma) \coloneqq [\![M]\!]_\theta(\gamma)$

Computation terms

$[\![\mathbf{case}\ V\ \mathbf{of}\ (x_1, x_2) \to M]\!]_\theta(\gamma) \coloneqq [\![M]\!]_\theta(\gamma[x_1 \mapsto a_1, x_2 \mapsto a_2])$
where $[\![V]\!]_\theta(\gamma) = \langle a_1, a_2 \rangle$

$\left[\!\left[\mathbf{case}\ V\ \mathbf{of}\ \{\mathbf{inj}_{\ell_1}\, x_1 \to M_1 \cdots \mathbf{inj}_{\ell_n}\, x_n \to M_n\}\right]\!\right]_\theta \coloneqq [\![M_i]\!]_\theta(\gamma[x_i \mapsto a_i])$
where $[\![V]\!]_\theta(\gamma) = \langle \ell_i, a_i \rangle$

$[\![x \leftarrow M;\ N]\!]_\theta(\gamma) \coloneqq [\![M]\!]_\theta(\gamma) \gg\!=_C \lambda a.\, [\![N]\!]_\theta(\gamma[x \mapsto a])$

$[\![V!]\!]_\theta(\gamma) \coloneqq [\![V]\!]_\theta(\gamma) \quad [\![\mathbf{return}\ V]\!]_\theta(\gamma) \coloneqq \mathbf{return}\ ([\![V]\!]_\theta(\gamma))$

$[\![\lambda x.M]\!]_\theta(\gamma) \coloneqq \lambda a.\, [\![M]\!]_\theta(\gamma[x \mapsto a]) \quad [\![M\ V]\!]_\theta(\gamma) \coloneqq ([\![M]\!]_\theta(\gamma))([\![V]\!]_\theta(\gamma))$

$\left[\!\left[\langle M_1, M_2 \rangle\right]\!\right]_\theta(\gamma) \coloneqq \left\langle [\![M_1]\!]_\theta(\gamma), [\![M_2]\!]_\theta(\gamma) \right\rangle \quad [\![\mathbf{prj}_i\, M]\!]_\theta(\gamma) \coloneqq \pi_i([\![M]\!]_\theta(\gamma))$

Figure 7.9: MAM denotational semantics for terms

Recall that we implicitly have type annotations on variant injections and variables in function abstractions, which guarantees that typing judgments have unique derivations. Therefore, the semantics of MAM can be based on typing judgments alone, but for readability we often only write terms.

The denotational semantics for derivations in MAM is presented in fig. 7.9. Again, all semantic constructs are parameterized by an assignment θ. To each well-typed derivation, we assign to each

- value term: a function $[\![\Theta; \Gamma \vdash V : A]\!]_\theta : [\![\Gamma]\!]_\theta \to [\![A]\!]_\theta$; and
- E-computation term: a function $[\![\Theta; \Gamma \vdash_E M : C]\!]_\theta : [\![\Gamma]\!]_\theta \to \left|[\![C]\!]_\theta\right|$.

The semantics are standard. Note that in the semantics of sequencing, we use the Kleisli extension $\gg\!=_C$ induced by an algebra $C = \langle |C|, c\rangle$ which takes a function $f\colon X \to |C|$ and gives $(\gg\!=_C f) \coloneqq c \circ (\mathbf{fmap}\ f)\colon TX \to |C|$.

We will now record some important properties of MAM.

Theorem 7.2.5 (MAM Safety, Forster et al., 2019, Theorem 2.6)**.** *Well-typed programs do not go wrong: for all closed MAM returners* $\Theta;\ \vdash_\emptyset M : FA$*, either* $M \rightsquigarrow N$ *for some* $\Theta;\ \vdash_\emptyset N : FA$ *or else* $M = \mathbf{return}\ V$ *for some* $\Theta;\ \vdash V : A$*.*

The following theorem extends an existing proof of termination for CBPV by [Doczkal, 2007]. We say that a term M *diverges*, and write $M \rightsquigarrow^\infty$ if for every $n \in \mathbb{N}$ there exists some N such that $M \rightsquigarrow^n N$. We say that M *does not diverge* when $M \not\rightsquigarrow^\infty$.

Theorem 7.2.6 (MAM Termination, Forster et al., 2019, Theorem 2.7)**.** *There are no infinite reduction sequences: for all MAM terms* $;\ \vdash_\emptyset M : FA$*, we have* $M \not\rightsquigarrow^\infty$ *and there exists some unique* $;\ \vdash V : A$ *such that* $M \rightsquigarrow^* \mathbf{return}\ V$*.*

The following exposition is also from [Forster et al., 2019], which we will need to state compositionality, soundness, and adequacy. Define *ground types* as the subclass of value types given by:

$$\text{(ground values)}\quad G ::= 1 \mid G_1 \times G_2 \mid \{\mathbf{inj}_{\ell_1} G_1 \mid \ldots \mid \mathbf{inj}_{\ell_n} G_n\}$$

We will need them in order to define contextual equivalence.

Program contexts $\mathcal{X}[\ ]$ and their type judgments can be defined, but they are verbose and not strictly required. For two computation terms M_1 and M_2, contextual equivalence essentially requires defining and then quantifying over the set $\Xi[M_1, M_2] \coloneqq \left\{\langle \mathcal{X}[M_1], \mathcal{X}[M_2]\rangle \middle| \mathcal{X}[\ ] \text{ is a well-typed context}\right\}$. This set can be defined directly without first defining contexts.

Definition 7.2.7 ([Forster et al., 2019])**.** We say that an environment Γ' *extends* an environment Γ, and write $\Gamma' \geq \Gamma$ if Γ' extends Γ as a partial function from identifiers to value types. Given two well-typed computations $\Theta_0; \Gamma_0 \vdash_{E_0} M_1 : C_0$ and $\Theta_0; \Gamma_0 \vdash_{E_0} M_2 : C_0$, let $\Xi[\Theta_0; \Gamma_0 \vdash_{E_0} M_1, M_2 : C_0]$ be the smallest set of tuples $\langle \Theta', \Gamma', V_1, V_2, A\rangle$ and $\langle \Theta', \Gamma', E', N_1, N_2, C\rangle$ that is compatible with the typing rules and contains all the tuples $\langle \Theta, \Gamma, E_0, M_1, M_2, C_0\rangle$, where $\Theta \supseteq \Theta_0$ and $\Gamma \geq \Gamma_0$. Call this set the *pairs of contexts plugged with* M_1 *and* M_2. Define the set $\Xi[\Theta_0; \Gamma_0 \vdash V_1, V_2 : A]$ for contexts plugged with values analogously.

In the above definition, $\Theta'; \Gamma' \vdash V_1, V_2 : A$ and $\Theta'; \Gamma' \vdash_{E'} N_1, N_2 : C$ are modeled by the tuples $\langle \Theta', \Gamma', V_1, V_2, A \rangle$ and $\langle \Theta', \Gamma', E', N_1, N_2, C \rangle$ respectively. If the tuple $\langle \Theta', \Gamma', V_1, V_2, A \rangle$ is in $\Xi[\Theta_0; \Gamma_0 \vdash_{E_0} M_1, M_2 : C_0]$, then so is the tuple $\langle \Theta', \Gamma', \emptyset, \mathbf{return}\ V_1, \mathbf{return}\ V_2, FA \rangle$ by the compatibility requirement.

Let X range over both value and E-computation types, and phrases P range over both value and computation terms. Judgments of the form $\Theta; \Gamma \vdash_E P : X$ are *meta-judgments*, ranging over value judgments $\Theta; \Gamma \vdash P : X$ and E-computation judgment $\Theta; \Gamma \vdash_E P : X$.

Definition 7.2.8 ([Forster et al., 2019])**.** Let $\Theta; \Gamma \vdash_E P, Q : X$ be two MAM phrases. We say that P and Q are *contextually equivalent* and write $\Theta; \Gamma \vdash_E P \simeq Q : X$ when, for all pairs of plugged *closed ground-returner pure* contexts $\left\langle \emptyset, \emptyset, \emptyset, M_P, M_Q, FG \right\rangle$ in $\Xi[\Theta; \Gamma \vdash_E P, Q : X]$ and for all closed ground value terms $; \vdash V : G$, we have:

$$M_P \rightsquigarrow^* \mathbf{return}\ V \quad \iff \quad M_Q \rightsquigarrow^* \mathbf{return}\ V$$

We can now state the final theorems about MAM.

Theorem 7.2.9 (MAM Compositionality, Forster et al., 2019, Theorem 2.8)**.** *The meaning of a term depends only on the meaning of its sub-terms: for all pairs of well-typed plugged MAM contexts M_P, M_Q in $\Xi[\Theta; \Gamma \vdash_E P, Q : X]$, if $[\![P]\!] = [\![Q]\!]$ then $[\![M_P]\!] = [\![M_Q]\!]$.*

Let $\rightsquigarrow_{\text{cong}}$ be the smallest relation containing $\rightsquigarrow_\beta$ that is closed under the term formation constructs, and so contains $\rightsquigarrow$ as well, and let $\simeq_{\text{cong}}$ be the smallest congruence relation containing $\rightsquigarrow_\beta$.

Theorem 7.2.10 (MAM Soundness, Forster et al., 2019, Theorem 2.9)**.** *Reduction preserves the semantics: for every pair of well-typed MAM terms $\Theta; \Gamma \vdash_E P, Q : X$, if $P \simeq_{\text{cong}} Q$ then $[\![P]\!] = [\![Q]\!]$. In particular, for every well-typed closed term of ground type $; \vdash_\emptyset P : FG$, if $P \rightsquigarrow^* \mathbf{return}\ V$ then $[\![P]\!] = [\![V]\!]$.*

Theorem 7.2.11 (MAM Adequacy, Forster et al., 2019, Theorem 2.10)**.** *Denotational equivalence implies contextual equivalence: for all well-typed MAM terms $\Theta; \Gamma \vdash_E P, Q : X$, if $[\![P]\!] = [\![Q]\!]$ then $P \simeq Q$.*

Thus, we see that the set-theoretic semantics is quite well-behaved. In fact, the above theorems imply that for all well-typed computations $\Theta; \Gamma \vdash_E M, M' : C$, if

$M \rightsquigarrow_{\text{cong}} M'$ then $M \simeq M'$. However, we will need to move to a richer setting in order to prove results about AD. Note that the semantics of MAM is essentially an indexed version of the Eilenberg-Moore adjunction models of CBPV with type variables.

We now recall what is required for an Eilenberg-Moore CBPV model.

Definition 7.2.12 (Kammar, 2014, Definition 2.11)**.** An Eilenberg-Moore CBPV model is a pair $\langle \mathcal{C}, \mathcal{T} \rangle$ where:

- $\mathcal{C}$ is a distributive category;
- $\mathcal{T}$ is a strong monad over $\mathcal{C}$; and
- $\mathcal{C}$ has all exponentials $A \Rightarrow |B|$ of all $\mathcal{T}$-algebra carriers $|B|$ by all objects $A \in \mathcal{C}$.

The definition of an EM CBPV model does not account for effect indices or type variables. MAM only has one effect index, the pure effect $\emptyset$, but it will be useful to generalize to many effects. Adapting the above definition for MAM, we define the following.

Definition 7.2.13. A MAM model $\mathcal{M}$ is a quintuple $\langle \mathcal{C}, \Theta, \theta, \mathbb{E}, \{\mathcal{T}_E\}_{E \in \mathbb{E}} \rangle$ where:

- $\mathcal{C}$ is a distributive category;
- Θ is a list of value-type variables;
- θ is an assignment from Θ to $\mathcal{C}$-objects;
- $\mathbb{E}$ is a set of well-kinded effects $\Theta \vdash_{\text{k}} E : \mathbf{Eff}$ including $\emptyset$;
- $\{\mathcal{T}_E\}_{E \in \mathbb{E}}$ is a collection of strong monads over $\mathcal{C}$ such that $\mathcal{T}_\emptyset = \text{Id}$; and
- for each $E \in \mathbb{E}$, $\mathcal{C}$ has all exponentials $A \Rightarrow |B|$ of all $\mathcal{T}_E$-algebra carriers $|B|$ by all objects $A \in \mathcal{C}$.

The category $\mathcal{C}$ and its distributive structure models values, and the EM categories model computation. Note that each pair $\langle \mathcal{C}, \mathcal{T}_E \rangle$ is an EM CBPV model and that no relation between distinct pairs need hold. Importantly, theorems 7.2.9 and 7.2.10 still hold for the categorical semantics, their proofs are merely induction over judgments and plugged contexts as in the set-theoretic case.

$M, N ::= \ldots$	computations
$\mid \mathsf{op}\ V$	operation call
$\mid \mathbf{handle}\ M\ \mathbf{with}\ H$	handling construct
$H ::=$	handlers
$\{\mathbf{return}\ x \mapsto M\}$	return clause
$\mid H \uplus \{\mathsf{op}\ p\ k \mapsto N\}$	operation clause

Figure 7.10: EFF syntax (extending fig. 7.3)

We can now extend MAM to the EFF language of [Forster et al., 2019] which contains algebraic effects and handlers. The raw term syntax of EFF is defined in fig. 7.10, extending fig. 7.3. We add two new forms of computations, operation calls for effects and handling constructs for handling said effects. We also add the new syntactic class of handlers which contains the syntax for defining handlers. For a handler $H \coloneqq \{\mathbf{return}\ x \mapsto M\} \uplus \left\{\mathsf{op}\ p\ k \mapsto N_{\mathsf{op}}\right\}_{\mathsf{op} \in E}$, define the projections $H^{\mathbf{return}} \coloneqq M$ and $H^{\mathsf{op}} \coloneqq N_{\mathsf{op}}$. The type system will ensure that the operation projection is well-defined.

Frames and contexts

$\cdots\ \mathcal{F} ::= \ldots \mid \mathbf{handle}\ [\ ]\ \mathbf{with}\ H$ computation frame

Beta reduction

(ret) $\mathbf{handle}\ (\mathbf{return}\ V)\ \mathbf{with}\ H \rightsquigarrow_\beta H^{\mathbf{return}}[V/x]$

(op) $\mathbf{handle}\ \ \mathcal{H}[\mathsf{op}\ V]\ \ \mathbf{with}\ H \rightsquigarrow_\beta$
$H^{\mathsf{op}}[V/p, \{\lambda x.\mathbf{handle}\ \mathcal{H}[\mathbf{return}\ x]\ \mathbf{with}\ H\}/k]$

Figure 7.11: EFF operational semantics (extending fig. 7.4)

The operational semantics of EFF is defined in fig. 7.11, extending fig. 7.4. We add a new computation frame consisting of a handler with a hole for the executing computation. The operational semantics is extended with β-reductions for handlers and operations. The (ret)-reduction says that a computation which has finished computing is handled by the **return** clause of a handler. The (op)-reduction is more involved. In the left-hand side of the rule, we use a hoisting frame filled with an operation. Note that a hoisting frame consists only of basic frames and thus cannot contain a handler. Therefore, we always match on the

$$
\begin{array}{lll}
E ::= \ldots & & \text{effects} \\
\mid \{\mathsf{op} : A \to B\} \uplus E & & \text{arity assignment} \\
K ::= \ldots & & \text{kinds} \\
\mid \mathbf{Hndlr} & & \text{handlers} \\
R ::= A \,{}^{E}\!\Rightarrow^{E'} C & & \text{handler types}
\end{array}
$$

Figure 7.12: EFF kinds and types (extending fig. 7.5)

Effect kinding $\cdots$

$$\frac{\Theta \vdash_{\mathrm{k}} A : \mathbf{Val} \qquad \Theta \vdash_{\mathrm{k}} B : \mathbf{Val} \qquad \mathsf{op} \notin E \qquad \Theta \vdash_{\mathrm{k}} E : \mathbf{Eff}}{\Theta \vdash_{\mathrm{k}} \{\mathsf{op} : A \to B\} \uplus E : \mathbf{Eff}}$$

Handler kinding $\boxed{\Theta \vdash_{\mathrm{k}} R : \mathbf{Hndlr}}$

$$\frac{\Theta \vdash_{\mathrm{k}} A : \mathbf{Val} \qquad \Theta \vdash_{\mathrm{k}} E, E' : \mathbf{Eff} \qquad \Theta \vdash_{\mathrm{k}} C : \mathbf{Comp}_{E'}}{\Theta \vdash_{\mathrm{k}} A \,{}^{E}\!\Rightarrow^{E'} C : \mathbf{Hndlr}}$$

Figure 7.13: EFF kind system (extending fig. 7.6)

innermost handler. The right-hand side of the rule captures the the hoisting frame and creates a continuation with it while rewrapping it with the same handler, and thus corresponds to a deep handler. The continuation and the operations parameter are then handled by the appropriate clause of the handler.

The kinds and types of EFF are defined in fig. 7.12, extending fig. 7.5. Effects now consist of a disjoint union of operations $\mathsf{op} : A \to B$, each annotated with its parameter A and arity B. A new kind for handlers is introduced, as well as a corresponding type. A handler of type $A \,{}^{E}\!\Rightarrow^{E'} C$ handles a computation of type FA which may use the effect E and produces a computation of type C which may use the effect E'.

The kind system of EFF is presented in fig. 7.13, extending fig. 7.6. Effect types may only contain each operation at most once. Kinding for handler types ensures that the effect annotation for the output effect E' matches the effect label of the computation type C of the handler. Note that operations and handler types can contain value-type variables.

There is an important subtlety hiding in the kinding rules for EFF. The

kinding rule for an effect $\{\mathsf{op} : A \to B\} \uplus E$ combines a well-kinded effect E with two well-kinded value types A and B. Value types include thunks U_E, which themselves are labelled with effects. Thus, what is preventing us from sneaking in the effect $\{\mathsf{op} : A \to B\} \uplus E$ into A or B and creating a recursive type? If we try to construct a kinding derivation for such a situation, then must first well-kind A and B. We then perform some additional number of kinding steps until we must again well-kind $\{\mathsf{op} : A \to B\} \uplus E$. However, derivations themselves have no recursive structure, and so we must repeat the same sequence of step, thus requiring us to yet again well-kind $\{\mathsf{op} : A \to B\} \uplus E$. Kind derivations are finite, and thus we have a contradiction. Therefore, recursive effect types are not well-kinded. It is possible, however, for A or B to contain just E, and in fact we will take advantage of this fact in our proof of continuation reverse mode to let our operations contain function types.

Computation typing $\cdots$

$$\frac{(\mathsf{op} : A \to B) \in E \qquad \Theta; \Gamma \vdash V : A}{\Theta; \Gamma \vdash_E \mathsf{op}\ V : FB} \qquad \frac{\Theta; \Gamma \vdash_E M : FA \qquad \Theta; \Gamma \vdash H : A\ {}^{E}\!\Rightarrow^{E'} C}{\Theta; \Gamma \vdash_{E'} \mathbf{handle}\ M\ \mathbf{with}\ H : C}$$

Handler typing $\boxed{\Theta; \Gamma \vdash H : R}$ $(\Theta \vdash_{\mathrm{k}} \Gamma : \mathbf{Ctxt}, R : \mathbf{Hndlr})$

$$\frac{\begin{array}{c}\Theta; \Gamma, x : A \vdash_E M : C \\ \text{for all } 1 \le i \le n: \quad \Theta; \Gamma, p : A_i, k : U_E(B_i \to C) \vdash_E N_i : C\end{array}}{\Theta; \Gamma \vdash \left\{\mathbf{return}\ x^A \mapsto M\right\} \uplus \{\mathsf{op}_i\ p\ k \mapsto N_i | 1 \le i \le n\} : A\ {}^{\left\{\mathsf{op}_i : A_i \to B_i | 1 \le i \le n\right\}}\!\Rightarrow^{E} C}$$

Figure 7.14: EFF type system (extending fig. 7.7)

The type system of EFF is presented in fig. 7.14, extending fig. 7.7. We again make a slight extension of Forster et al. by adding type annotations to the return clause of handler definitions to ensure typing judgements have unique derivations. An operation $\mathsf{op} : A \to B \in E$ is parameterized by a value of type A and produces a computation of type FB with effect index E. Note that this rule gives a *may* interpretation to effect indices, any operation in the effect may occur, but it is not required to. The handling construct takes a handler $H : A\ {}^{E}\!\Rightarrow^{E'} C$ and $M : FA$ which may use the effect E and creates a computation of type C which may use the effect E'. The handling construct is the sole method of changing effect indices.

Handler typing ensures if a computation with effect $\{\mathsf{op}_i : A_i \to B_i | 1 \leq i \leq n\}$ has a corresponding clause for each operation op_i. It also ensures that all cases are computations of the same type and effect.

As with MAM, we have safety and termination. Note that the choice to include effect annotations is necessary to ensure termination.

Theorem 7.2.14 (EFF Safety, Forster et al., 2019, Theorem 3.4)**.** *Well-typed programs don't go wrong: for all closed EFF returners* $\Theta; \vdash_\emptyset M : FA$*, either* $M \rightsquigarrow N$ *for some* $\Theta; \vdash_\emptyset N : FA$ *or else* $M = \mathbf{return}\ V$ *for some* $\Theta; \vdash V : A$*.*

Theorem 7.2.15 (EFF Termination, Forster et al., 2019, Theorem 3.5)**.** *There are no infinite reduction sequences: for all EFF terms* $; \vdash_\emptyset M : FA$*, we have* $M \not\rightsquigarrow^\infty$*, and there exists some unique* $; \vdash V : A$ *such that* $M \rightsquigarrow^* \mathbf{return}\ V$*.*

The termination proof of EFF is essentially that of [Kammar, Lindley, and Oury, 2013, Theorem 1]. It observes that during effect handling, the handler is reinvoked, possibly more than once, but always on a subterm. The proof sketch notes that the termination result depends crucially on the effect type system, as is the case for delimited continuations, shown in [Ariola, Herbelin, and Sabry, 2009, Proposition 26]. The strong normalization of the systems described by Ariola, Herbelin, and Sabry holds only in the typed cases in which recursive effect types are disallowed. Likewise, the restriction to non-recursive effect types in EFF is integral to the termination proof.

We now introduce the family of monads used in the denotational semantics of EFF, each of which is a tree-style monad.

Definition 7.2.16 ([Forster et al., 2019])**.** A *signature* Σ is a pair consisting of a set $|\Sigma|$ whose elements we call *operation symbols*, and a function $arity_\Sigma$ from $|\Sigma|$ assigning to each operation symbol $f \in |\Sigma|$ a (possibly infinite) set $arity_\Sigma(f)$. We write $(f : A) \in \Sigma$ when $f \in |\Sigma|$ and $arity_\Sigma(f) = A$. Given a signature Σ and a set X, we inductively form the set $T_\Sigma X$ of Σ*-terms over* X by:

$$t ::= x \mid f\,\langle t_a \rangle_{a \in A} \qquad\qquad (x \in X, (f : A) \in \Sigma)$$

The assignment T_Σ together with the following assignments

$$\mathbf{return}\ x \coloneqq x \qquad\qquad t \gg\!= f \coloneqq t[f(x)/x]_{x \in X} \quad (f : X \to T_\Sigma Y)$$

form a monad $\mathcal{T}_\Sigma$. The $\mathcal{T}_\Sigma$-algebras $\langle C, c \rangle$ are in bijective correspondence with Σ*-algebras* on the same carrier. These are pairs $\langle C, [\![-]\!] \rangle$ where $[\![-]\!]$ assigns to

each $(f : A) \in \Sigma$ a function $[\![-]\!] : C^A \to C$ from A-ary tuples of C elements to C. The bijection is given by setting $[\![f]\!] \langle \xi_a \rangle_{a \in A}$ to be $c(f \langle \xi_a \rangle_{a \in A})$.

We will refer to the aforementioned monad as the Σ-*term* monad. To extend MAM's denotational semantics to EFF, take a fixed variable assignment θ and assign to each

- handler type: a pair $[\![\Theta \vdash_{\mathrm{k}} X : \mathbf{Hndlr}]\!]_\theta = \langle C, f \rangle$ consisting of an algebra C and a function f into the carrier of the algebra $|C|$.

Effects

$$[\![E]\!]_\theta \coloneqq \mathcal{T}_{\left\{\mathsf{op}_p : [\![B]\!]_\theta \,\middle|\, (\mathsf{op} : A \to B) \in E, p \in [\![A]\!]_\theta\right\}}$$

Handler types

$$\left[\!\left[A \,{}^{E}\!\Rightarrow^{E'} C\right]\!\right]_\theta \coloneqq \left\{[\![E]\!]_\theta\text{-algebras with carrier } \left|[\![C]\!]_\theta\right|\right\} \times \left|[\![C]\!]_\theta\right|^{[\![A]\!]_\theta}$$

Figure 7.15: EFF denotational semantics for types (extending fig. 7.8)

The denotational semantics for types in EFF is presented in fig. 7.15, extending fig. 7.8. An effect E induces a signature, for each operation $\mathsf{op} : A \to B \in E$ and $p \in [\![A]\!]_\theta$ we define an operation symbol op_p with arity $[\![B]\!]_\theta$. We define the monad $[\![E]\!]_\theta$ to be the tree monad for this signature. Note that when E is the empty set the induced monad is the identity monad. The semantics for the handler type $A \,{}^{E}\!\Rightarrow^{E'} C$ is a pair of a $[\![E]\!]_\theta$-algebra with carrier $\left|[\![C]\!]_\theta\right|$ and a function $[\![A]\!]_\theta \Rightarrow \left|[\![C]\!]_\theta\right|$. The $[\![E']\!]_\theta$-algebra structure of $[\![C]\!]_\theta$ is not required to relate in any way to the $[\![E]\!]_\theta$-algebra component of the handler.

The denotational semantics for terms in EFF is presented in fig. 7.16, extending fig. 7.9. The interpretation of an effectful operation $\mathsf{op}\ V$ use the fact that the a Σ-term monad where $\mathsf{op} \in \Sigma$ has an operation symbol $\mathsf{op}_{[\![V]\!]_\theta(\gamma)}$, and is given the denotation $\mathsf{op}_{[\![V]\!]_\theta(\gamma)} \langle \mathbf{return}\ b \rangle_{b \in [\![B]\!]_\theta}$. Note that $\lambda k.\lambda p.\mathsf{op}_p \langle k(b) \rangle_{b \in [\![B]\!]_\theta}$ is an algebraic operation. The semantics of the handling construct used the Kleisli extension of the algebra induced by the handler and the function induced by the return clause of the handler. For the handler, the operations induce an algebra for the relevant tree monad by giving an interpretation to all the operation symbols and the return clause induces a function in the obvious way.

EFF shares MAM's ground types, and so plugged contexts and the equivalences $\simeq$ and $\simeq_{\mathrm{cong}}$ are defined the same as in MAM. Again, as with MAM, we have the following theorems.

Computation terms $\cdots$

$\llbracket \mathsf{op}\ V \rrbracket_\theta(\gamma) \coloneqq \mathsf{op}_{\llbracket V \rrbracket_\theta(\gamma)} \langle \mathbf{return}\ b \rangle_{b \in \llbracket B \rrbracket_\theta}$

$\llbracket \mathbf{handle}\ M\ \mathbf{with}\ H \rrbracket_\theta(\gamma) \coloneqq \llbracket M \rrbracket_\theta(\gamma) \gg\!=_D f$

where $\llbracket H \rrbracket_\theta(\gamma) = \left\langle D, f : \llbracket A \rrbracket_\theta \to \left| \llbracket C \rrbracket_\theta \right| \right\rangle$

Handler terms

$\left\llbracket \{\mathbf{return}\ x \mapsto M\} \uplus \{\mathsf{op}\ p\ k \mapsto N_{\mathsf{op}}\}_{\mathsf{op}} \right\rrbracket_\theta(\gamma) \coloneqq \langle D, f \rangle$

where D's algebra structure and f given by:

$\left\llbracket \mathsf{op}_q \right\rrbracket_D \langle \xi_a \rangle_a \coloneqq \left\llbracket N_{\mathsf{op}} \right\rrbracket_\theta (\gamma[q/p, \langle \xi_a \rangle_a / k]) \quad f(a) \coloneqq \llbracket M \rrbracket_\theta (\gamma[a/x])$

Figure 7.16: EFF denotational semantics for terms (extending fig. 7.9)

Theorem 7.2.17 (EFF Compositionality, Forster et al., 2019, Theorem 3.6)**.** *The meaning of a term depends only on the meaning of its sub-terms: for all pairs of well-typed plugged EFF contexts M_P, M_Q in $\Xi[\Theta; \Gamma \vdash_E P, Q : X]$, if $\llbracket P \rrbracket = \llbracket Q \rrbracket$ then $\llbracket M_P \rrbracket = \left\llbracket M_Q \right\rrbracket$.*

Theorem 7.2.18 (EFF Soundness, Forster et al., 2019, Theorem 3.7)**.** *Reduction preserves the semantics: for every pair of well-typed EFF terms $\Theta; \Gamma \vdash_E P, Q : X$, if $P \simeq_{\text{cong}} Q$ then $\llbracket P \rrbracket = \llbracket Q \rrbracket$. In particular, for every well-typed closed term of ground type $; \vdash_\emptyset P : FG$, if $P \rightsquigarrow^* \mathbf{return}\ V$ then $\llbracket P \rrbracket = \llbracket V \rrbracket$.*

Theorem 7.2.19 (EFF Adequacy, Forster et al., 2019, Theorem 3.8)**.** *Denotational equivalence implies contextual equivalence: for all well-typed EFF terms $\Theta; \Gamma \vdash_E P, Q : X$, if $\llbracket P \rrbracket = \llbracket Q \rrbracket$ then $P \simeq Q$.*

All together, the above theorems imply that for all well-typed computations $\Theta; \Gamma \vdash_E M, M' : C$, if $M \rightsquigarrow_{\text{cong}} M'$ then $M \simeq M'$ as they did with MAM. Thus, the set-theoretic operational semantics are quite well-behaved.

We will now show how to extend EFF's semantics to categorical semantics. Let $\mathcal{C}$ be a bi-cartesian closed category. We first need to change our characterization the Σ-term monad. In **Set**, every object is a sum of singleton sets of its elements, which means

$$A \Rightarrow B \cong \left(\coprod_{a \in A} 1 \right) \Rightarrow B \cong \prod_{a \in A} (1 \Rightarrow B) \cong \prod_{a \in A} B.$$

Let Σ be a signature. Define the functor F^1_Σ as

$$F^1_\Sigma X \coloneqq \coprod_{f \in |\Sigma|} \prod_{a \in arity_\Sigma(f)} X.$$

Note that, in **Set**, the functor $\prod_{a \in A} -$ preserves colimits of λ-chains for large enough λ. Then the Σ-term monad $\mathcal{T}_\Sigma$ is the free monad for F^1_Σ which exists by consequence of corollary 7.1.18 as **Set** is cocomplete. Furthermore, $\mathcal{T}_\Sigma$ is the free monad for

$$F^2_\Sigma X \coloneqq \coprod_{f \in |\Sigma|} (arity_\Sigma(f) \Rightarrow X).$$

The natural isomorphism between F^1_Σ and F^2_Σ is used implicitly in the semantics of EFF. In other bi-CCCs, the functor F^1_Σ is not defined because $arity_\Sigma(f)$ is not a set, and thus we focus on F^2_Σ. In the semantics of EFF, the signature Σ is of a special form, i.e. for an effect E and semantic interpretation $[\![-]\!]_\theta$,

$$\Sigma = \left\{ \mathsf{op}_p : [\![B]\!]_\theta \middle| (\mathsf{op} : A \to B) \in E, p \in [\![A]\!]_\theta \right\}$$

and so in **Set** we have

$$\begin{aligned} F^2_\Sigma X &= \coprod_{\mathsf{op}_p \in |\Sigma|} ([\![B]\!]_\theta \Rightarrow X) \\ &= \coprod_{\mathsf{op}:A \to B \in E,\, p \in [\![A]\!]_\theta} ([\![B]\!]_\theta \Rightarrow X) \\ &\cong \coprod_{\mathsf{op}:A \to B \in E} \coprod_{p \in [\![A]\!]_\theta} ([\![B]\!]_\theta \Rightarrow X) \\ &\cong \coprod_{\mathsf{op}:A \to B \in E} [\![A]\!]_\theta \times ([\![B]\!]_\theta \Rightarrow X) =: F^3_\Sigma. \end{aligned}$$

We then have that $\mathcal{T}_\Sigma$ is the free monad on **Set** for F^3_Σ when Σ is induced by an effect. This formulation of $\mathcal{T}_\Sigma$ is what we require to generalize the effect monad to arbitrary bi-CCCs.

Definition 7.2.20. Let $\mathcal{C}$ be a bi-CCC. An *effect signature* Σ is a triple consisting of a finite set $|\Sigma|$ whose elements we call *operation symbols*, an assignment $arity_\Sigma$ assigning each element $\mathsf{op} \in |\Sigma|$ an object $arity_\Sigma(\mathsf{op}) \in \mathcal{C}$ to its *arity*, and an assignment assigning each element $\mathsf{op} \in |\Sigma|$ an object $param_\Sigma(\mathsf{op}) \in \mathcal{C}$ to its *parameters*. We write $(\mathsf{op} : A \to B) \in \Sigma$ when $\mathsf{op} \in |\Sigma|$, $param_\Sigma(\mathsf{op}) = A$ and $arity_\Sigma(\mathsf{op}) = B$. Define the *effect functor* $F_\Sigma \colon \mathcal{C} \to \mathcal{C}$ induced by an effect signature Σ to be

$$F_\Sigma X \coloneqq \coprod_{\mathsf{op}:A \to B \in \Sigma} A \times (B \Rightarrow X).$$

If the free monad for F_Σ exists, we call it the *effect monad* induced by the effect signature Σ and denote it $\mathcal{T}_\Sigma$ with the underlying functor denoted by T_Σ.

Our semantics, and indeed the very definition of an algebraic operation, requires that $\mathcal{T}_\Sigma$ is strong. Proposition 7.1.22 tells us that $\mathcal{T}_\Sigma$ is strong when F_Σ is, and this is always the case.

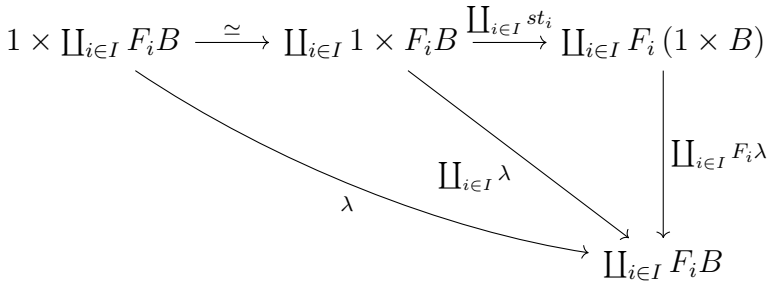
Proposition 7.2.21. *For any effect signature Σ, the effect functor F_Σ is strong.*

Proof. For any objects $A, B \in \mathcal{C}$, the functors $A \times -$ and $B \Rightarrow -$ are strong, and the composition of strong functors is strong, and so $A \times (B \Rightarrow -)$ is strong. F_Σ is a coproduct of functors of this form, and so it suffices to show that strong functors are closed under coproduct.

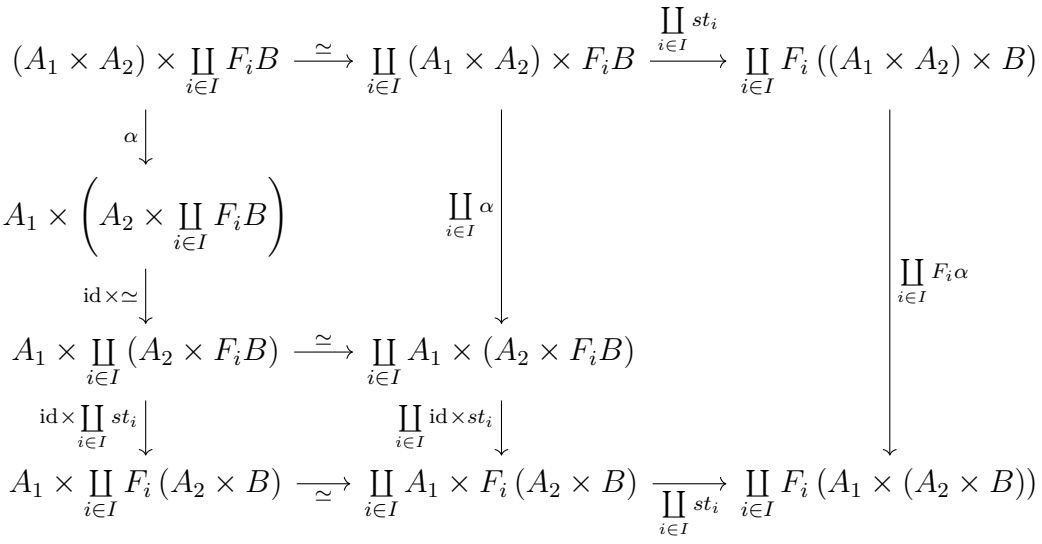
Suppose that for a finite set I we have strong functors F_i for $i \in I$ with strengths $st_i \colon A \times F_i B \to F_i\,(A \times B)$. Define the strength of $\coprod_{i\in I} F_i$ as

$$A \times \coprod_{i\in I} F_i B \xrightarrow{\simeq} \coprod_{i\in I} A \times F_i B \xrightarrow{\coprod_{i\in I} st_i} \coprod_{i\in I} F_i\,(A \times B)$$

where the isomorphism $\simeq$ is distribution of products over coproducts, which holds in any bi-CCC. The above map is clearly natural, and so it remains to show it respects the unitors and associators of the product. For unitors, we have the following diagram



which commutes. For the associator, we have the following diagram



which also commutes. Thus, our proposed strength is indeed one and so F_Σ is strong. □

Note that effect monads are not guaranteed to exist in an arbitrary $\mathcal{C}$. Thus, we will need to assume they do for our semantics. When $\mathcal{T}_\Sigma$ exists, each operation symbol $\mathsf{op} \in |\Sigma|$ induces an algebraic operation of $\mathcal{T}_\Sigma$ in an analogous manner to the construction in **Set**.

Proposition 7.2.22. *Let $\mathcal{C}$ be a bi-CCC and Σ an effect signature. If the effect monad $\mathcal{T}_\Sigma$ exists, then for $\mathsf{op} : A \to B$ the family of maps*

$$\alpha_X^{\mathsf{op}} \colon (B \Rightarrow T_\Sigma X) \to (A \Rightarrow T_\Sigma X)$$
$$k \mapsto \lambda a.\varphi_X\left(\iota_{\mathsf{op}}\left(\langle a, k\rangle\right)\right)$$

defined using the internal language of a bi-CCC forms an algebraic operation.

Proof. The proof is the analogous to the set theoretic case. □

Clearly each effect E and semantic interpretation $[\![-]\!]_\theta$ induce an effect signature $\left\{\mathsf{op} : [\![A]\!]_\theta \to [\![B]\!]_\theta \middle| (\mathsf{op} : A \to B) \in E\right\}$ in any bi-CCC, and furthermore in **Set** that the effect monad induced by this effect signature is isomorphic to the term monad induced by the signature corresponding to E and $[\![-]\!]_\theta$. Thus, we will use effect signatures and effect monads in our extension of EFF's semantics.

We now turn our attention to handlers. Recall the typing rule for handlers where $E \coloneqq \{\mathsf{op}_i : A_i \to B_i | 1 \le i \le n\}$:

$$\frac{\begin{array}{c}\Theta; \Gamma, x : A \vdash_{E'} M : C \\ \text{for all } 1 \le i \le n: \quad \Theta; \Gamma, p : A_i, k : U_{E'}(B_i \to C) \vdash_{E'} N_i : C\end{array}}{\Theta; \Gamma \vdash \{\mathbf{return}\ x \mapsto M\} \uplus \{\mathsf{op}_i\ p\ k \mapsto N_i | 1 \le i \le n\} : A\ {}^{E}\!\Rightarrow^{E'} C}$$

For a fixed value-type assignment θ and semantic interpretation $[\![-]\!]_\theta$, each operation clause gives a morphism

$$[\![N_i]\!]_\theta : [\![\Gamma]\!]_\theta \times [\![A_i]\!]_\theta \times \left([\![B_i]\!]_\theta \Rightarrow \left|[\![C]\!]_\theta\right|\right) \to \left|[\![C]\!]_\theta\right|$$

which we then curry to get

$$\overline{[\![N_i]\!]_\theta} \colon [\![\Gamma]\!]_\theta \to \left([\![A_i]\!]_\theta \times \left([\![B_i]\!]_\theta \Rightarrow \left|[\![C]\!]_\theta\right|\right)\right) \Rightarrow \left|[\![C]\!]_\theta\right|$$

and tuple all clauses together to get

$$\left\langle \overline{[\![N_i]\!]_\theta} \right\rangle_{1 \le i \le n} : [\![\Gamma]\!]_\theta \to \prod_{1 \le i \le n} \left(\left([\![A_i]\!]_\theta \times \left([\![B_i]\!]_\theta \Rightarrow \left|[\![C]\!]_\theta\right|\right)\right) \Rightarrow \left|[\![C]\!]_\theta\right|\right).$$

The codomain is naturally isomorphic to

$$\coprod_{1\leq i\leq n}\left(\llbracket A_i\rrbracket_\theta \times \left(\llbracket B_i\rrbracket_\theta \Rightarrow \left|\llbracket C\rrbracket_\theta\right|\right)\right) \Rightarrow \left|\llbracket C\rrbracket_\theta\right|$$

which is exactly an (internal) F_Σ-algebra $F_\Sigma\left|\llbracket C\rrbracket_\theta\right| \Rightarrow \left|\llbracket C\rrbracket_\theta\right|$ for the effect signature $\Sigma \coloneqq \left\{\mathsf{op} : \llbracket A\rrbracket_\theta \to \llbracket B\rrbracket_\theta \middle| (\mathsf{op} : A \to B) \in E\right\}$ induced by E and $\llbracket -\rrbracket_\theta$. Furthermore, the return clause gives a morphism $\llbracket\Gamma\rrbracket_\theta \to \left(\llbracket A\rrbracket_\theta \Rightarrow \left|\llbracket C\rrbracket_\theta\right|\right)$. Therefore, we define handler types as

$$\left\llbracket A \,{}^{E}\!\Rightarrow^{E'} C\right\rrbracket_\theta \coloneqq \left(F_\Sigma\left|\llbracket C\rrbracket_\theta\right| \Rightarrow \left|\llbracket C\rrbracket_\theta\right|\right) \times \left(\left|\llbracket C\rrbracket_\theta\right| \Rightarrow \llbracket A\rrbracket_\theta\right).$$

The categorical denotational semantics for types in EFF is summarized in fig. 7.17.

Effects

$\llbracket E\rrbracket_\theta \coloneqq \mathcal{T}_{\left\{\mathsf{op}:\llbracket A\rrbracket_\theta\to\llbracket B\rrbracket_\theta \middle| (\mathsf{op}:A\to B)\in E\right\}}$

Handler types

$\left\llbracket A \,{}^{E}\!\Rightarrow^{E'} C\right\rrbracket_\theta \coloneqq \left(F_\Sigma\left|\llbracket C\rrbracket_\theta\right| \Rightarrow \left|\llbracket C\rrbracket_\theta\right|\right) \times \left(\left|\llbracket C\rrbracket_\theta\right| \Rightarrow \llbracket A\rrbracket_\theta\right)$
where $\Sigma \coloneqq \left\{\mathsf{op} : \llbracket A\rrbracket_\theta \to \llbracket B\rrbracket_\theta \middle| (\mathsf{op} : A \to B) \in E\right\}$

Figure 7.17: EFF categorical denotational semantics for types

The categorical denotational semantics for terms in EFF is similar to **Set** semantics. For operations, the typing rule is

$$\frac{(\mathsf{op} : A \to B) \in E \qquad \Theta;\Gamma \vdash V : A}{\Theta;\Gamma \vdash_E \mathsf{op}\ V : FB}$$

so when defining an interpretation $\llbracket -\rrbracket_\theta$ we must define a map $\llbracket\Gamma\rrbracket_\theta \to \llbracket E\rrbracket_\theta\, \llbracket B\rrbracket_\theta$ from a map $\llbracket V\rrbracket_\theta : \llbracket\Gamma\rrbracket_\theta \to \llbracket A\rrbracket_\theta$. Let $\mathcal{T}$ be the effect monad $\llbracket E\rrbracket_\theta$ (which we have assumed exists) and F the associated effect functor. As $\mathcal{T}$ is the free monad for F, there is an algebra $\varphi_{\llbracket B\rrbracket_\theta} : FT\,\llbracket B\rrbracket_\theta \to T\,\llbracket B\rrbracket_\theta$. In the internal language, we define

$$\llbracket \mathsf{op}\ V\rrbracket_\theta\,(\gamma) \coloneqq \varphi_{\llbracket B\rrbracket_\theta}\left(\iota_{\mathsf{op}}\left(\left\langle \llbracket V\rrbracket_\theta\,(\gamma), \mathbf{return}\right\rangle\right)\right)$$

where ι_{op} maps into the op-component of F. This definition is essentially the same as the **Set** semantics. Importantly, note that the denotation is derived from the algebraic operation α^{op} of proposition 7.2.22, i.e. it is $\alpha^{\mathsf{op}}(\mathbf{return})$. For handler terms, we define a handler as described above where $\chi : \prod_i (A_i \Rightarrow B) \cong$

$(\coprod_i A_i) \Rightarrow B$. For handling computations, we use the internal free algebra map $\gg\!=\colon (F_\Sigma B \Rightarrow B) \times (A \Rightarrow B) \to (T_\Sigma A \Rightarrow B)$ which by theorem 7.1.21 exists when $\mathcal{T}_\Sigma$ does as F_Σ is strong. The categorical denotational semantics for terms in EFF is summarized in fig. 7.18.

Computation terms $\cdots$

$$[\![\mathsf{op}\ V]\!]_\theta(\gamma) \coloneqq \varphi_{[\![B]\!]_\theta}\left(\iota_{\mathsf{op}}\left(\left\langle [\![V]\!]_\theta(\gamma), \mathbf{return}\right\rangle\right)\right)$$

$$[\![\mathbf{handle}\ M\ \mathbf{with}\ H]\!]_\theta(\gamma) \coloneqq [\![M]\!]_\theta(\gamma) \gg\!= [\![H]\!]_\theta(\gamma)$$

Handler terms

$$\left[\!\left[\{\mathbf{return}\ x \mapsto M\} \uplus \{\mathsf{op}\ p\ k \mapsto N_{\mathsf{op}}\}_{\mathsf{op}}\right]\!\right]_\theta(\gamma) \coloneqq \left\langle \chi\left(\left\langle \overline{[\![N_{\mathsf{op}}]\!]_\theta}\right\rangle_{\mathsf{op}}\right), \overline{[\![M]\!]_\theta}\right\rangle$$

Figure 7.18: EFF categorical denotational semantics for terms

We can now define an EFF model, which collects our assumptions.

Definition 7.2.23. An EFF model $\mathcal{M}$ is a triple $\langle \mathcal{C}, \Theta, \theta\rangle$ where:

- $\mathcal{C}$ is a bi-CCC;
- Θ is a list of value-type variables;
- θ is an assignment from Θ to $\mathcal{C}$-objects;
- For any effect signature Σ, the effect monad $\mathcal{T}_\Sigma$ exists.

Observe that, unlike a MAM model defined in definition 7.2.13, we do not require a set of effects, nor a collection of monads, nor do we include a condition of exponentiating algebras. These requirements would be redundant. We now have a way to generate well-kinded effects and associate to each a strong monad. Furthermore, because $\mathcal{C}$ is cartesian closed, we can always form exponential algebras. Thus, every EFF model induces a MAM model.

Proposition 7.2.24. *Let $\mathcal{M} = \langle \mathcal{C}, \Theta, \theta\rangle$ be a EFF model. Define the set $\mathbb{E} \coloneqq \{E : \Theta \vdash_{\mathrm{k}} E : \mathbf{Eff}\}$ and for $E \in \mathbb{E}$ define $\mathcal{T}_E \coloneqq \mathcal{T}_{[\![E]\!]_\theta}$. Then $\langle \mathcal{C}, \Theta, \theta, \mathbb{E}, \{\mathcal{T}_E\}_{E\in\mathbb{E}}\rangle$ is a MAM model.*

We believe the categorical denotational semantics still satisfy theorems 7.2.17 and 7.2.18 of compositionality and soundness respectively. Compositionality is

still an induction on the plugged context, and the behavior of the categorical semantics lets this go through. Likewise, soundness requires checks that β-reduction preserves semantics, and by our constructions this holds. We do not provide proofs, but see no obstruction to the aforementioned sketches.

It will be useful to identify some requirements on the category $\mathcal{C}$ in the definition of an EFF model to ensure that the required free monads exist. Suppose that $\mathcal{C}$ is complete and cocomplete, as well as that all functors of the form $B \Rightarrow -$ preserve λ_B-chains for some ordinal λ_B dependent on B. Then any effect functor F_Σ preserves $\bigcup_{\mathsf{op}:A\to B\in\Sigma} \lambda_B$ chains. Therefore, the effect monad $\mathcal{T}_\Sigma$ exists in $\mathcal{C}$ by corollary 7.1.18 and theorem 7.1.11.

Proposition 7.2.25. *Suppose that $\mathcal{C}$ is complete and cocomplete, as well as that all functors of the form $B \Rightarrow -$ preserve λ_B-chains for some ordinal λ_B dependent on B. Then any triple $\langle \mathcal{C}, \Theta, \theta \rangle$ is an EFF model.*

The final extension we shall make to EFF is the addition of base types, base constants, and built-in functions on base types.

Definition 7.2.26. An EFF *signature* is a quintuple $\langle B, C, F, \mathrm{type}_{\mathrm{const}}, \mathrm{type}_{\mathrm{func}} \rangle$ where:

- B is a set of base types;
- C is a set of base constants;
- F is a set of built-in functions;
- $\mathrm{type}_{\mathrm{const}} \colon C \to B$ is a function assigning types to constants; and
- $\mathrm{type}_{\mathrm{func}} \colon F \to \coprod_{n,m\in\mathbb{N}} B^n \times B^m$ is a function assigning types to functions.

Define $C_b \coloneqq \mathrm{type}^{-1}_{\mathrm{const}}(b)$ for $b \in B$ and $C_{(b_1,\ldots,b_n)} \coloneqq \prod_{1\le i\le n} C_{b_i}$ for $(b_1, \ldots, b_n) \in B^n$, $n \in \mathbb{N}$. Next, define $\mathrm{dom}(f) \coloneqq C_{\pi_1(\mathrm{type}_{\mathrm{func}}(f))}$ and $\mathrm{cod}(f) \coloneqq C_{\pi_2(\mathrm{type}_{\mathrm{func}}(f))}$. An EFF *operational signature* consists of a signature as above and additionally

- a dependent function $\mathrm{eval} \colon \prod_{f\in F} \mathrm{dom}(f) \Rightarrow \mathrm{cod}(f)$ specifying the behavior of each built-in function.

Fix an operational signature $\Pi = \langle B, C, F, \mathrm{type}_{\mathrm{const}}, \mathrm{type}_{\mathrm{func}}, \mathrm{eval} \rangle$. The syntax extension for EFF is presented in fig. 7.19. We denote constants by $c \in C$, and built-in functions by $f \in F$, and distinguish their use in syntax with an overline.

The operational semantics extension for EFF is presented in fig. 7.20. We need only add one family of β-reductions that lifts the eval function to syntax. The types extension of EFF is presented in fig. 7.21. We add a base type $\overline{b}$ for each $b \in B$. We will use the preexisting function type for built-in functions. The kind system extension of EFF is presented in fig. 7.22. Every base type $b \in B$ is a well-kinded value type. The type system extension of EFF is presented in fig. 7.23. Each constant and built-in function is given a type based on the signature. We believe that the operational semantics still satisfy theorems 7.2.14 and 7.2.15 of safety and termination respectively. Safety should still hold as we only have one extra case of application and we provide β-reductions when apply a built-in function in an empty typing context. Furthermore, termination should still hold because we eliminate built-in functions during reduction, producing only constants.

An operational signature specifies how a built-in function acts on constants. Thus, we need to ensure that models respect Π.

Definition 7.2.27. Let $\Pi = \langle B, C, F, \mathrm{type}_{\mathrm{const}}, \mathrm{type}_{\mathrm{func}}, \mathrm{eval}\rangle$ be an operational signature. An EFF Π-*model* $\mathcal{M}$ is a sextuple $\langle \mathcal{C}, \Theta, \theta, [\![-]\!]^B, [\![-]\!]^C, [\![-]\!]^F\rangle$ where:

- $\langle \mathcal{C}, \Theta, \theta\rangle$ is an EFF model;
- $[\![-]\!]^B : B \to \mathrm{Ob}\,\mathcal{C}$ is an assignment of base types to objects;
- $[\![-]\!]^C : \prod_{c\in C} \mathcal{C}\left(1, [\![\mathrm{type}_{\mathrm{const}}(c)]\!]^B\right)$ is a dependent function assigning constants to generalized points;
- $[\![-]\!]^F : \prod_{f\in F} \mathcal{C}\left([\![\mathrm{dom}(f)]\!]^B, [\![\mathrm{cod}(f)]\!]^B\right)$ is a dependent function assigning built-in functions to morphisms;

such that $\mathrm{eval}(f)(c_1, \ldots, c_n) = (c'_1, \ldots, c'_m)$ implies

$$[\![f]\!]^F \cdot \left\langle [\![c_1]\!]^C, \ldots, [\![c_n]\!]^C \right\rangle = \left\langle [\![c'_1]\!]^C, \ldots, [\![c'_m]\!]^C \right\rangle$$

for all $f \in F$ and $(c_1, \ldots, c_n) \in \mathrm{dom}(f)$.

We can now describe the semantic extension of EFF with respect to a fixed operation signature Π and Π-model $\mathcal{M}$. The categorical denotational semantics extension of types for EFF is presented in fig. 7.24. The only new class of types are base types, and we use the interpretation provided by $\mathcal{M}$. The categorical denotational semantics extension of terms for EFF is presented in fig. 7.25. We again

$$
\begin{array}{ll}
V, W ::= \ldots & \text{values} \\
\mid \overline{c} & \quad \text{constants} \\
M, N ::= \ldots & \text{computations} \\
\mid \overline{f}\ V & \quad \text{built-in functions}
\end{array}
$$

Figure 7.19: EFF syntax extension (extending fig. 7.10)

Beta reduction

$$
(const) \quad \overline{f}\ \langle \overline{c_1}, \ldots, \overline{c_n} \rangle \rightsquigarrow_\beta \textbf{return}\ \langle \overline{c'_1}, \ldots, \overline{c'_m} \rangle \quad \text{where } \mathrm{eval}(f)(c_1, \ldots, c_n) = (c'_1, \ldots, c'_m)
$$

Figure 7.20: EFF operational semantics extension (extending fig. 7.11)

$$
\begin{array}{ll}
A, B ::= \ldots & \text{value types} \\
\mid \overline{b} & \quad \text{base type for } b \in B
\end{array}
$$

Figure 7.21: EFF types extension (extending fig. 7.12)

Value kinding $\cdots$

$$
\frac{b \in B}{\Theta \vdash_{\mathrm{k}} \overline{b} : \textbf{Val}}
$$

Figure 7.22: EFF kind system extension (extending fig. 7.13)

Value typing $\cdots$

$$
\frac{\mathrm{type}_{\mathrm{const}}(c) = b}{\Theta; \Gamma \vdash \overline{c} : \overline{b}}
$$

Computation typing $\cdots$

$$
\frac{\mathrm{type}_{\mathrm{func}}(f) = \Big((b_1, \ldots, b_n), (b'_1, \ldots, b'_m)\Big)}{\Theta; \Gamma \vdash_E \overline{f} : \left(\prod_{1 \le i \le n} \overline{b_i}\right) \to F\left(\prod_{1 \le j \le m} \overline{b'_j}\right)}
$$

Figure 7.23: EFF type system extension (extending fig. 7.14)

use $\mathcal{M}$ to provide interpretations for constants and built-in functions. Note that the built-in functions are computations, but $\mathcal{M}$ merely specifies $\mathcal{C}$ morphisms, and so we post-compose with **return**. Finally, we again believe the categorical denotational semantics still satisfy theorems 7.2.17 and 7.2.18 of compositionality and soundness respectively as our addition of constants and built-in functions is straightforward.

$$\begin{array}{l}\textbf{Value types} \quad \cdots \\ \left[\!\left[\overline{b}\right]\!\right]_\theta \coloneqq [\![b]\!]^B\end{array}$$

Figure 7.24: EFF categorical semantics extension for types (extending fig. 7.17)

$$\begin{array}{l}\textbf{Value terms} \quad \cdots \\ [\![\overline{c}]\!]_\theta\,(\gamma) \coloneqq [\![c]\!]^C \\ \textbf{Computation terms} \quad \cdots \\ \left[\!\left[\overline{f}\right]\!\right]_\theta\,(\gamma) \coloneqq \mathbf{return} \circ_{\mathcal{C}} [\![f]\!]^F\end{array}$$

Figure 7.25: EFF categorical semantics extension for terms (extending fig. 7.18)

7.3 Logical Relations

Logical relations are a proof method by which one creates a relation-valued denotation. In EFF, we would assign each type a relation, and require all constructs in the language to preserve the relations. However, because we will need to retain more information at each type than mere sets capture, we require a generalization of logical relations. We will achieve this through the use of fibrations.

We recall some terminology used for fibrations from [Jacobs, 1999]. Let $p\colon \mathcal{E} \to \mathcal{C}$ be a functor. We will call the domain $\mathcal{E}$ the *total category* and the codomain $\mathcal{C}$ the *base category*. An object $X \in \mathcal{E}$ such that $pX = I \in \mathcal{C}$ is said to be *above* I; similarly, a morphism f of $\mathcal{E}$ with $pf = u$ of $\mathcal{C}$ is said to be *above* u. The subcategory $\mathcal{E}_I$ of $\mathcal{E}$ consisting of the objects above I and morphisms above id_I is called the *fibre category*, or simply *fibre*, over I.

Definition 7.3.1 (Jacobs, 1999, Definition 1.1.3)**.** Let $p\colon \mathcal{E} \to \mathcal{C}$ be a functor.

1. A morphism $f\colon X \to Y$ in $\mathcal{E}$ is *Cartesian over* $u\colon I \to J$ in $\mathcal{C}$ if $pf = u$ and every $g\colon Z \to Y$ in $\mathcal{E}$ for which one has $pg = u \cdot w$ for some $w\colon pZ \to I$, uniquely determines an $h\colon Z \to X$ in $\mathcal{E}$ above w with $f \cdot h = g$. We call $f\colon X \to Y$ in the total category $\mathcal{E}$ *Cartesian* if it is Cartesian over its underlying map pf in $\mathcal{C}$.

2. The functor $p\colon \mathcal{E} \to \mathcal{C}$ is a *fibration* if for every $Y \in \mathcal{E}$ and $u\colon I \to pY$ in $\mathcal{C}$, there is a Cartesian morphism $f\colon X \to Y$ in $\mathcal{E}$ above u.

Our approach to logical relations is based on [Katsumata, 2013]. We recall some of his observations in the case that p is faithful. In this case, each fibre category $\mathcal{E}_I$ is a preorder. The objects above I are like predicates on I, and $\mathcal{E}_I$ like the preorder of predicates on I. We also borrow the following notation for when p is faithful. For $X, Y \in \mathcal{E}$ and $f\colon pX \to pY$, define $f\colon X \dot{\to} Y$ to be the proposition $\exists \dot{f}\colon X \to Y.\, p(\dot{f}) = f$. When $f\colon X \dot{\to} Y$ holds, the morphism $\dot{f}$ existing above f is unique, and is called the *witness* of $f\colon X \dot{\to} Y$.

Definition 7.3.2 (Partial order bifibration with fibrewise small products Katsumata, 2013, Definition 3)**.** A *partial order bifibration with fibrewise small products* is a faithful functor $p\colon \mathcal{E} \to \mathcal{C}$ such that:

Partial order: Each fibre is a partial order.

Fibration: p is a fibration. The property of being a fibration simplifies when p is faithful. For any $I \in \mathcal{C}, Y \in \mathcal{E}$ and $f\colon I \to pY$, there exists $X \in \mathcal{E}$ above I such that $f\colon X \dot{\to} Y$ and the following property holds: for any $Z \in \mathcal{E}$ and $g\colon pZ \to I$, $f \cdot g\colon Z \dot{\to} Y$ implies $g : Z \dot{\to} X$. This property and $\mathcal{E}_I$ being a partial order imply that X is unique; hence we write f^*Y for X. Furthermore, for any $f\colon I \to J$ in $\mathcal{C}$, the mapping $Y \in \mathcal{E}_J \mapsto f^*Y \in \mathcal{E}_I$ extends to a functor $f^*\colon \mathcal{E}_J \to \mathcal{E}_I$. We call it the *inverse image functor* (along f).

Bi-: Each inverse image functor f^* has a left adjoint called the *direct image functor* (along f), denoted by f_*.

Fibrewise small products: Each fibre category has small products and the inverse image functor (necessarily) preserves them.

We now define the core construction for generalized logical relations.

Definition 7.3.3 (Fibration for logical relations Katsumata, 2013, Definition 4)**.** A *fibration for logical relations (FFLR)* over a bi-CCC is a partial order bifibration $p\colon \mathcal{E} \to \mathcal{C}$ with fibrewise small products such that $\mathcal{E}$ is a bi-CCC and p strictly preserves the bi-cartesian closed (bi-CC) structure.

A simple example of a FFLR provides the basis for logical predicates.

Example 7.3.4 (Jacobs, 1999, Exercise 9.2.1, Katsumata, 2013, Example 2)**.** Define the category **Pred** as follows: the objects are pairs of sets (A, X) such that $A \subseteq X$, and a map $f\colon (A, X) \to (B, Y)$ is a function $f\colon X \to Y$ such that $f(A) \subseteq B$. **Pred** is equivalent to the category of subobjects of **Set**. Define the functor $\pi\colon \mathbf{Pred} \to \mathbf{Set}$ by $\pi(A, X) \coloneqq X$ and $\pi(f) \coloneqq f$, clearly π is faithful. Furthermore, the fibre $\mathbf{Pred}_X$ is isomorphic to the poset of subsets of X with fibrewise products given by intersection and π is a bifibration via inverse and direct image of sets. Thus, π is a partial order bifibration with fibrewise small products.

By [Jacobs, 1999, Exercise 9.2.1], **Pred** has a bi-CC structure which is strictly preserved by π, where the object structure is

$$\dot{1} = (1, 1) \qquad (A, X) \mathbin{\dot{\times}} (B, Y) = (A \times B, X \times Y)$$
$$\dot{0} = (0, 0) \qquad (A, X) \mathbin{\dot{+}} (B, Y) = (A + B, X + Y)$$
$$(A, X) \mathbin{\dot{\Rightarrow}} (B, Y) = (\{f : f(A) \subseteq B\}, X \Rightarrow Y)$$

and the structure maps are the lifts from **Set**. In conclusion, π is a FFLR.

We will follow [Katsumata, 2013] in specifying a chosen representation of a *pullback* of a fibration along a functor. Let $p\colon \mathcal{E} \to \mathcal{C}$ be a faithful functor and $F\colon \mathcal{B} \to \mathcal{C}$, then the pullback

$$\begin{array}{ccc} F^*\mathcal{E} & \xrightarrow{q} & \mathcal{E} \\ {\scriptstyle F^*p}\downarrow & & \downarrow{\scriptstyle p} \\ \mathcal{B} & \xrightarrow[F]{} & \mathcal{C} \end{array}$$

consists of a category $F^*\mathcal{E}$ and functors F^*p and q described as follows. The category $F^*\mathcal{E}$ has objects which are pairs (X, I) where $X \in \mathcal{E}, I \in \mathcal{B}$ such that X is above FI, and a morphism from (X, I) to (Y, J) is a morphism $f\colon I \to J$ such that $Ff\colon X \dot{\to} Y$. The functor F^*p sends (X, I) to I and sends f to itself. Finally, the functor q sends (X, I) to I and sends f to its witness $\dot{f}$. We note

that F^*p is again faithful, and the fibre category of $(F^*\mathcal{E})_I$ is isomorphic to $\mathcal{E}_{FI}$. The pullback of a fibration for logical relations p gives a new fibration for logical relations when F is a finite-product preserving functor.

Proposition 7.3.5 (Katsumata, 2013, Proposition 6)**.** *Let $\mathcal{B},\mathcal{C}$ be bi-CCCs, $p\colon \mathcal{E} \to \mathcal{C}$ be a fibration for logical relations and $F\colon \mathcal{B} \to \mathcal{C}$ be a finite-product preserving functor. Then $F^*p\colon F^*\mathcal{E} \to \mathcal{B}$ is a fibration for logical relations.*

Sketch proof. For a map $f\colon I \to J$ in $\mathcal{B}$, the functors $f^*\colon (F^*\mathcal{E})_J \to (F^*\mathcal{E})_I$ and $f_*\colon (F^*\mathcal{E})_I \to (F^*\mathcal{E})_J$ are given on objects by

$$f^*(Y,J) \coloneqq \left((Ff)^*Y, J\right) \qquad f_*(X,I) \coloneqq \left((Ff)_*X, I\right).$$

Each fibre $(F^*\mathcal{E})_I$ is isomorphic to $\mathcal{E}_{FI}$ and so is a partial order with small products. For any functor $F\colon \mathcal{B} \to \mathcal{C}$ between bi-CCCs, we have the following structure maps:

$$!_{F1}\colon F1 \to 1 \qquad l_{I,J} \coloneqq \langle F\pi_1, F\pi_2\rangle\colon F(I\times J) \to FI \times FJ$$
$$?_{F0}\colon 0 \to F0 \qquad m_{I,J} \coloneqq [F\iota_1, F\iota_2]\colon FI + FJ \to F(I+J)$$

and when F is product preserving, we also have

$$n_{I,J} \coloneqq \lambda\left(F(ev)\cdot (l_{I\Rightarrow J,J})^{-1}\right) : F(I\Rightarrow J) \to FI \Rightarrow FJ.$$

The object bi-CC structure is given by

$$\bar{1} = ((!_{F1})^*\dot{1}, 1) \qquad (X,I)\mathbin{\bar{\times}}(Y,J) = (l^*(X\mathbin{\dot{\times}}Y), I\times J)$$
$$\bar{0} = ((?_{F0})_*\dot{0}, 0) \qquad (X,I)\mathbin{\bar{+}}(Y,J) = (m_*(X\mathbin{\dot{+}}Y), I+J)$$
$$(X,I)\mathbin{\bar{\Rightarrow}}(Y,J) = (n^*(X\mathbin{\dot{\Rightarrow}}Y), I\Rightarrow J).$$

□

Example 7.3.6 (Katsumata, 2013, Example 3(2))**.** The binary product functor $F\colon \mathbf{Set}\times\mathbf{Set} \to \mathbf{Set}$, $(X,Y)\mapsto X\times Y$ is a product preserving functor. Let us apply proposition 7.3.5 to F and $\pi\colon \mathbf{Pred} \to \mathbf{Set}$, resulting in the following pullback:

$$\begin{array}{ccc} F^*\mathbf{Pred} & \xrightarrow{\ q\ } & \mathbf{Pred} \\ {\scriptstyle F^*\pi}\downarrow & & \downarrow{\scriptstyle \pi} \\ \mathbf{Set}\times\mathbf{Set} & \xrightarrow[\ F\ ]{} & \mathbf{Set} \end{array}$$

and define $\mathbf{BRel} \coloneqq F^*\mathbf{Pred}$, for binary relations, and define $p \coloneqq F^*\pi$. Then $p\colon \mathbf{BRel} \to \mathbf{Set}\times\mathbf{Set}$ is a FFLR. An object in $\mathbf{BRel}$ is equivalent to a pair

$(R,(X,Y))$ where $X,Y \in \mathbf{Set}$ and R is a subset of $X \times Y$, i.e. a relation, and a map $(f_1,f_2)\colon (R,(X_1,X_2)) \to (S,(Y_1,Y_2))$ is a pair of functions $f_i\colon X_i \to Y_i$ such that $(f_1 \times f_2)(R) \subseteq S$. We will use this representation.

We can use to proposition 7.3.5 to calculate the structure of $p\colon \mathbf{BRel} \to \mathbf{Set} \times \mathbf{Set}$. The calculations are straightforward, and so we only include the product of objects as an example. For objects $(R,(X_1,X_2))$ and $(S,(Y_1,Y_2))$, their product is

$$\left(\left\{((x_1,y_1),(x_2,y_2)) : (x_1,x_2) \in R, (y_1,y_2) \in S\right\}, (X_1 \times Y_1, X_2 \times Y_2)\right).$$

The FFLR $\pi\colon \mathbf{Pred} \to \mathbf{Set}$ is an instance of a more general construction. For any category $\mathcal{C}$, there is a category of arrows $\mathcal{C}^{\to}$ which has morphisms of $\mathcal{C}$ as objects and commutative squares as morphisms. There exists a functor $\mathrm{cod}\colon \mathcal{C}^{\to} \to \mathcal{C}$ which maps a morphism $f\colon A \to B$ to its codomain B, and when $\mathcal{C}$ has pullbacks cod is a fibration [Jacobs, 1999, Proposition 1.1.6]. Given a set X and a subset A, there is a monomorphism $i\colon A \to X$ given by the inclusion $A \subseteq X$. The class of monomorphisms $\mathcal{M}$ of $\mathbf{Set}$ defines a subcategory of $\mathbf{Set}^{\to}$ and the codomain fibration restricts to $\mathcal{M}$. Note that the fibres $\mathcal{M}_I$ are not partial orders, merely large preorders, but they are fibrewise equivalent to the partial order $(\mathcal{P}(I),\subseteq)$. To generalize the above observations, we require the notion of factorization systems.

Definition 7.3.7 (Borceux, 1994a, Definition 5.4.1)**.** Consider two arrows $f\colon A \to B$ and $g\colon C \to D$ in a category $\mathcal{C}$. We say that f *is orthogonal to* g and we write $f \perp g$ when, given arbitrary morphisms u,v such that $v \cdot f = g \cdot u$ there exists a unique morphism w such that $w \cdot f = u$ and $g \cdot w = v$.

Definition 7.3.8 (Borceux, 1994a, Definition 5.5.1)**.** By *factorization system* on a category $\mathcal{C}$ we mean a pair $(\mathcal{E},\mathcal{M})$ where both $\mathcal{E}$ and $\mathcal{M}$ are classes of morphisms of $\mathcal{C}$ and

1. every isomorphism belongs to both $\mathcal{E}$ and $\mathcal{M}$,
2. both $\mathcal{E}$ and $\mathcal{M}$ are closed under composition,
3. for all $e \in \mathcal{E}$ and $m \in \mathcal{M}$, $e \perp m$,
4. every morphism $f \in \mathcal{C}$ can be factored as $f = m \cdot e$ with $e \in \mathcal{E}$ and $m \in \mathcal{M}$.

Not every factorization system induces an FFLR, and so we recall the following definition from [Kammar and McDermott, 2018]. In the following, we also consider the classes $\mathcal{E}$ and $\mathcal{M}$ as full subcategories of $\mathcal{C}^{\to}$, the arrow category with arrows of $\mathcal{C}$ as objects and commutative squares as morphisms. Recall that there is a functor $\mathrm{cod}\colon \mathcal{C}^{\to} \to \mathcal{C}$ which maps each arrow to its codomain, and that it restricts to subcategories of $\mathcal{C}^{\to}$.

Definition 7.3.9 (Kammar and McDermott, 2018, Definition 4.4)**.** Let $\mathcal{C}$ be a bi-CCC. A factorization system $(\mathcal{E}, \mathcal{M})$ over $\mathcal{C}$ is a *factorization system for logical relations* when:

- $\mathcal{C}$ has pullbacks of $\mathcal{M}$-morphisms;
- every morphism in $\mathcal{M}$ is a monomorphism;
- for every $I \in \mathcal{C}$ the cod-fibre $\mathcal{M}_I$ has small products;
- $\mathcal{M}$ is closed under binary coproducts; and
- $\mathcal{E}$ is closed under binary products.

Lemma 7.3.10 (Kammar and McDermott, 2018, Lemma 4.5)**.** *Let $(\mathcal{E}, \mathcal{M})$ be a factorization system over a bi-CCC $\mathcal{C}$. The codomain functor $\mathrm{cod}\colon \mathcal{M} \to \mathcal{C}$ is a FFLR if and only if $(\mathcal{E}, \mathcal{M})$ is a factorization system for logical relations.*

Thus, we now have an additional way to create FFLRs. The above lemma requires a small amount of finessing due to the same issue of the fibres of **Pred** being large preorders. However, for our uses later we will only take $\mathcal{M}$'s such that the fibres $\mathcal{M}_I$ are equivalent to partial orders.

A fibration for logical relations ensures that the bi-cartesian closed category structure of of $\mathcal{C}$ is respected by when lifting to $\mathcal{E}$. To create an EFF model in $\mathcal{E}$, definition 7.2.23 also requires the existence of effect monads and enriched universal construction of maps out of free algebras. We also must ensure that the effect monads in $\mathcal{E}$ lie above the effect monads of $\mathcal{C}$. The following definition formalizes this requirement.

Definition 7.3.11 ([Katsumata, 2013])**.** Let $\mathcal{C}$ be a bi-CCC, $\mathcal{T}$ be a strong monad over $\mathcal{C}$ and $p\colon \mathcal{E} \to \mathcal{C}$ be a fibration for logical relations. We formulate the concept of a logical relation for $\mathcal{T}$ as a strong monad $\dot{\mathcal{T}} = (\dot{T}, \dot{\eta}, \dot{\mu}, \dot{\theta})$ over $\mathcal{E}$ such that

$$p(\dot{T}X) = T(pX), \quad p\dot{\eta}_X = \eta_{pX}, \quad p\dot{\mu}_X = \mu_{pX}, \quad p\dot{\theta}_{X,Y} = \theta_{pX,pY}.$$

We call such $\dot{\mathcal{T}}$ a *lifting* of $\mathcal{T}$.

The denotational semantics of EFF uses the objects of the category of algebras $\mathbf{Alg}\ \mathcal{T}$ of a strong monad $\mathcal{T}$. Thus, given a lift $\dot{\mathcal{T}}$ of $\mathcal{T}$ along an FFLR $p\colon \mathcal{E} \to \mathcal{C}$, we will need to consider Eilenberg-Moore category $\mathbf{Alg}\ \dot{\mathcal{T}}$ of $\dot{\mathcal{T}}$ to define logical relations for EFF. Let $\dot{F} \dashv \dot{U}$ be the standard adjunction for the category of algebras $\mathbf{Alg}\ \dot{\mathcal{T}}$. Consider the (non-commuting) diagram:

$$\begin{array}{ccc} \mathcal{E} & \underset{\dot{U}}{\overset{\dot{F}}{\rightleftarrows}} & \mathbf{Alg}\ \dot{\mathcal{T}} \\ {\scriptstyle p}\downarrow & & \\ \mathcal{C} & \underset{U}{\overset{F}{\rightleftarrows}} & \mathbf{Alg}\ \mathcal{T} \end{array}$$

We want to define a functor $\mathbf{Alg}\ p\colon \mathbf{Alg}\ \dot{\mathcal{T}} \to \mathbf{Alg}\ \mathcal{T}$ such that p and $\mathbf{Alg}\ p$ are a map of adjunctions. For an algebra $\left\langle X, \alpha\colon \dot{T}X \to X \right\rangle$, define $(\mathbf{Alg}\ p)(\langle X, \alpha\rangle) \coloneqq \langle pX, p\alpha\rangle$, and for an algebra map $f\colon \langle X, \alpha\rangle \to \langle Y, \beta\rangle$, define $(\mathbf{Alg}\ p)(f) \coloneqq pf$.

Proposition 7.3.12. $\mathbf{Alg}\ p\colon \mathbf{Alg}\ \dot{\mathcal{T}} \to \mathbf{Alg}\ \mathcal{T}$ *is a functor and the pair* p *and* $\mathbf{Alg}\ p$ *map* $\dot{F} \dashv \dot{U}$ *to* $F \dashv U$*. Furthermore,* p *strictly preserves the cartesian structure of* $\mathbf{Alg}\ \dot{\mathcal{T}}$ *and strictly maps the functor* $X \mathrel{\dot{\Rightarrow}} -\colon \mathbf{Alg}\ \dot{\mathcal{T}} \to \mathbf{Alg}\ \dot{\mathcal{T}}$ *to* $pX \Rightarrow -\colon \mathbf{Alg}\ \mathcal{T} \to \mathbf{Alg}\ \mathcal{T}$*.*

Proof. Straightforward calculation from the fact that p strictly preserves the cartesian closed structure and $\dot{\mathcal{T}}$ is a lift of $\mathcal{T}$. □

Even though we have proved that $\mathbf{Alg}\ p$ is a functor and not merely an object assignment, we will not need this fact as the semantics of EFF only uses morphisms in $\mathcal{C}$. Nonetheless, the denotational semantics of stacks in CBPV uses EM algebras and so we record functorality.

An important aspect of the denotational semantics of EFF is the existence and use of effect monads. Let $p\colon \mathcal{E} \to \mathcal{C}$ be a FFLR and $\Sigma = \{\mathsf{op}_i : K_i \to J_i | 1 \leq i \leq n\}$ be an effect signature in $\mathcal{C}$. Let $Y_i, Z_i \in \mathcal{E}$ be above $J_i, K_i \in \mathcal{C}$ respectively and define $\dot{\Sigma} \coloneqq \{\mathsf{op}_i : Z_i \to Y_i | 1 \leq i \leq n\}$. The effect functor $F_\Sigma = \coprod_{\mathsf{op}_i : K_i \to J_i \in \Sigma} K_i \times (J_i \Rightarrow -)$ has a counterpart

$$F_{\dot{\Sigma}} \coloneqq \dot{\coprod}_{\mathsf{op}_i : Z_i \to Y_i \in \Sigma} Z_i \mathbin{\dot{\times}} (Y_i \mathrel{\dot{\Rightarrow}} -)$$

with $pF_{\dot{\Sigma}} = F_\Sigma$. The effect functor $F_{\dot{\Sigma}}$ appears in the denotation of handlers for $\mathcal{E}$-valued models of EFF, and such a model requires the existence of the effect

monad $\mathcal{T}_{\dot\Sigma}$. It is also required $\mathcal{T}_{\dot\Sigma}$ that be a lifting of $\mathcal{T}_\Sigma$. We will achieve this using the following lifting construction due of [Kammar and McDermott, 2018].

Definition 7.3.13 ([Kammar and McDermott, 2018])**.** Let $p\colon \mathcal{E} \to \mathcal{C}$ be a FFLR and $\mathcal{T}$ a strong monad on $\mathcal{C}$. Let $\{\alpha_i\colon (J_i \Rightarrow T-) \to (K_i \Rightarrow T-)\}_{1\le i\le n}$ be a set of algebraic operations of $\mathcal{T}$ and $Y_i, Z_i \in \mathcal{E}$ above $J_i, K_i \in \mathcal{C}$ respectively. For each object $X \in \mathcal{E}$, define $\mathcal{R}X$ as the set of all $X' \in \mathcal{E}_{T(pX)}$ such that:

- The unit respects X', i.e. $\eta\colon X \dot\to X'$.
- Each algebraic operation respects X' for the given lift, i.e. $\alpha_i\colon (Y_i \dot\Rightarrow X') \dot\to (Z_i \dot\Rightarrow X')$

Define $\dot TX \coloneqq \bigwedge \mathcal{R}X$, i.e. $\dot TX$ is the least element of $\mathcal{R}X$. This uniquely defines a lifting $\dot{\mathcal{T}}$ of $\mathcal{T}$, which we call the *free lifting* with respect to $\{\alpha_i\}_{1\le i\le n}$ and $Y_i, Z_i \in \mathcal{E}$. Furthermore, each algebraic operation α_i lifts to an algebraic operation $\dot\alpha_i$.

The fact that the free lifting is a monad lifting is folklore, and an explicit proof can be found in Kammar's thesis [Kammar, 2014] for the case of $p\colon \mathbf{BRel} \to \mathbf{Set} \times \mathbf{Set}$. His proofs are straightforwardly generalized to an arbitrary FFLR as they make use of only properties and constructions common to all FFLRs.

Suppose that $\mathcal{T}_\Sigma$ exists. We can then consider the free lifting of $\mathcal{T}_\Sigma$ with respect to the algebraic operations $\{\alpha^{\mathsf{op}_i}\}_{1\le i\le n}$ (defined in proposition 7.2.22) and $Y_i, Z_i \in \mathcal{E}$. Denote this lift by $\dot{\mathcal{T}}_\Sigma$. We will show that $\dot{\mathcal{T}}_\Sigma$ is actually the free algebra monad for $F_{\dot\Sigma}$.

Proposition 7.3.14. *$\dot{\mathcal{T}}_\Sigma$ is the free algebra monad for $F_{\dot\Sigma}$ and so $\dot{\mathcal{T}}_\Sigma = \mathcal{T}_{\dot\Sigma}$.*

Proof. We need to show that for any $X \in \mathcal{E}$ there is an $F_{\dot\Sigma}$-algebra structure on $\dot{\mathcal{T}}_\Sigma$ and that it is the free such one. Fix an $X \in \mathcal{E}$. As $\dot{\mathcal{T}}_\Sigma$ is the free lifting with respect to $\{\alpha^{\mathsf{op}_i}\}_{1\le i\le n}$, we have $\alpha^{\mathsf{op}_i}\colon (Y_i \dot\Rightarrow \dot T_\Sigma X) \dot\to (Z_i \dot\Rightarrow \dot T_\Sigma X)$ and thus, after uncurrying, we hav $\overline{\alpha^{\mathsf{op}_i}}\colon Z_i \mathbin{\dot\times} (Y_i \dot\Rightarrow \dot T_\Sigma X) \dot\to \dot T_\Sigma X$. Define $\dot\varphi_X$ to be the lift of

$$[\,\overline{\alpha^{\mathsf{op}_i}}\,]_{\mathsf{op}_i\in\Sigma} : \dot\coprod_{\mathsf{op}_i\in\Sigma} Z_i \mathbin{\dot\times} (Y_i \dot\Rightarrow \dot T_\Sigma X) \dot\to \dot T_\Sigma X$$

which equivalently has type $F_{\dot\Sigma}\dot T_\Sigma X \dot\to \dot T_\Sigma X$. Observe that by proposition 7.2.22 $p\dot\varphi_X = \varphi_{pX}$ where $\varphi_{pX}\colon F_\Sigma T_\Sigma(pX) \to T_\Sigma(pX)$ is the free F_Σ-algebra on pX. We will show that $\dot\varphi_X$ is the free $F_{\dot\Sigma}$-algebra.

Let $\beta\colon F_{\dot\Sigma}Y \to Y$ be an $F_{\dot\Sigma}$-algebra and $f\colon X \to Y$. Then $p\beta\colon F_\Sigma(pY) \to pY$ is an F_Σ-algebra as p strictly preserves the bi-CCC structure and $pf\colon pX \to pY$. Thus, by freeness of φ_{pX}, we get a unique map $g\colon T_\Sigma(pX) \to pY$ such that

$$\begin{array}{ccccc} F_\Sigma T_\Sigma(pX) & \xrightarrow{\varphi_{pX}} & T_\Sigma(pX) & \xleftarrow{\eta_{pX}} & pX \\ {\scriptstyle F_\Sigma g}\downarrow & & {\scriptstyle g}\downarrow & \swarrow_{pf} & \\ F_\Sigma(pY) & \xrightarrow[p\beta]{} & pY & & \end{array}$$

commutes where η is the unit of T_Σ. We will show that g lifts to $g\colon \dot T_\Sigma X \dot\to Y$. Note that if g has such a lift, then the lift $\dot g$ is such that

$$\begin{array}{ccccc} F_{\dot\Sigma}\dot T_\Sigma X & \xrightarrow{\dot\varphi_X} & \dot T_\Sigma X & \xleftarrow{\dot\eta_X} & X \\ {\scriptstyle F_{\dot\Sigma}\dot g}\downarrow & & {\scriptstyle \dot g}\downarrow & \swarrow_{f} & \\ F_{\dot\Sigma}Y & \xrightarrow[\beta]{} & Y & & \end{array}$$

commutes by faithfulness of p. Furthermore, if another map $h\colon \dot T_\Sigma X \to Y$ makes the diagram commute, then $ph = g$ by uniqueness of g and thus $h = \dot g$ by faithfulness of p again. Therefore it suffices to show g lifts.

Define $R \coloneqq g^*(Y)$. We will show the unit of $\mathcal{T}_\Sigma$ respects R and each algebraic operation α^{op_i} respects R. We begin by observing that $g \cdot \eta_{pX} = pf$, and because $f\colon X \to Y$, $(pf)_*(X) \le Y$. Thus, ${\eta_{pX}}_*(X) \le g^*(Y) = R$ and so the unit respects R. The equation $p\beta \cdot F_\Sigma = g \cdot \varphi_{pX}$ can be precomposed on each side with ι_{op_i} for any $\mathsf{op}_i \in \Sigma$ given the structure of F_Σ. This gives the equation

$$p\beta \cdot \iota_{\mathsf{op}_i} \cdot \Big(\mathrm{id}_{K_i} \times (J_i \Rightarrow g)\Big) = g \cdot \overline{\alpha^{\mathsf{op}_i}}$$

and so

$$\begin{aligned} (g \cdot \overline{\alpha^{\mathsf{op}_i}})_*(Z_i \mathbin{\dot\times} (Y_i \Rightarrow R)) &= \Big(p\beta \cdot \iota_{\mathsf{op}_i} \cdot \Big(\mathrm{id}_{K_i} \times (J_i \Rightarrow g)\Big)\Big)_*(Z_i \mathbin{\dot\times} (Y_i \Rightarrow R)) \\ &= \Big(p\beta \cdot \iota_{\mathsf{op}_i}\Big)_*(Z_i \mathbin{\dot\times} (Y_i \Rightarrow g_*(R))) \\ &\le \Big(\beta \cdot \iota_{\mathsf{op}_i}\Big)_*(Z_i \mathbin{\dot\times} (Y_i \Rightarrow Y)) \\ &\le Y \end{aligned}$$

meaning $\overline{\alpha^{\mathsf{op}_i}}_*(Z_i \mathbin{\dot\times} (Y_i \Rightarrow R)) \le g^*(Y) = R$. Therefore α^{op_i} respects R. We have shown $R \in \mathcal{R}X$ and so $\dot T_\Sigma X = \bigwedge \mathcal{R}X \le R = g^*(Y)$ which means $g\colon \dot T_\Sigma X \dot\to Y$ as needed. □

Note that in the above proof we have $p\dot\varphi_X = \varphi_{pX}$. This means, given $I, J \in \mathcal{C}$ and $X, Y \in \mathcal{E}$ with $pX = I$ and $pY = J$, that the construction of free F_Σ-algebras

in $\mathcal{C}$ given by

$$\Phi \colon \mathcal{C}(F_\Sigma J, J) \times \mathcal{C}(I, J) \to \mathcal{C}(T_\Sigma I, J)$$

and of free $F_{\dot\Sigma}$-algebras $\mathcal{E}$ given by

$$\dot\Phi \colon \mathcal{E}(F_{\dot\Sigma} Y, Y) \times \mathcal{E}(X, Y) \to \mathcal{E}(T_{\dot\Sigma} X, Y)$$

satisfy $p\dot\Phi = \Phi$. Furthermore, theorems 7.1.20 and 7.1.21 which show that Φ and $\dot\Phi$ internalize to $\gg\!=$ and $\dot{\gg}\!=$ respectively, only made use of the bi-CCC structure and the construction of free algebras. Therefore, we also have $p(\dot{\gg}\!=) = \gg\!=$.

We can now prove that EFF models can be lifted with respect to an FFLR $p \colon \mathcal{E} \to \mathcal{C}$.

Theorem 7.3.15. *Let $\mathcal{M} = \langle \mathcal{C}, \Theta, \theta \rangle$ be an EFF model, $p \colon \mathcal{E} \to \mathcal{C}$ be an FFLR, and $\dot\theta$ an assignment from Θ to $\mathcal{E}$ objects such that $p\dot\theta(\alpha) = \theta(\alpha)$ for all $\alpha \in \Theta$. Then $\dot{\mathcal{M}} \coloneqq \langle \mathcal{E}, \Theta, \dot\theta \rangle$ is an EFF model. We say that such a model $\dot{\mathcal{M}}$ is a* lifting *of $\mathcal{M}$ along p.*

Proof. By definition of an FFLR, $\mathcal{E}$ is a bi-CCC. Proposition 7.3.14 show that all effect monads exist in $\mathcal{E}$ and that the universal construction of maps out of effect monads enriches. □

Next, we define what it means to lift an EFF Π-model with respect to an FFLR $p \colon \mathcal{E} \to \mathcal{C}$.

Definition 7.3.16. Let $\Pi = \langle B, C, F, \mathrm{type}_{\mathrm{const}}, \mathrm{type}_{\mathrm{func}}, \mathrm{eval} \rangle$ be an operational signature and $\mathcal{M} = \langle \mathcal{C}, \Theta, \theta, [\![-]\!]^B, [\![-]\!]^C, [\![-]\!]^F \rangle$ an EFF Π-model. A *lifting* of $\mathcal{M}$ along p is a Π-model $\dot{\mathcal{M}} = \langle \mathcal{E}, \Theta, \dot\theta, [\![-]\!]^{\dot B}, [\![-]\!]^{\dot C}, [\![-]\!]^{\dot F} \rangle$ such that $\langle \mathcal{E}, \Theta, \dot\theta \rangle$ is a lifting of $\langle \mathcal{C}, \Theta, \theta \rangle$, $p\,[\![b]\!]^{\dot B} = [\![b]\!]^B$ for all $b \in B$, $p\,[\![c]\!]^{\dot C} = [\![c]\!]^C$ for all $c \in C$, and $p\,[\![f]\!]^{\dot F} = [\![f]\!]^F$ for all $f \in F$.

We can now prove the the basic lemmas for logical relations, our main proof method. We first show that the semantics of types lift.

Theorem 7.3.17 (Basic lemma for types)**.** *Let $\Pi = \langle B, C, F, \mathrm{type}_{\mathrm{const}}, \mathrm{type}_{\mathrm{func}}, \mathrm{eval} \rangle$ be an operational signature, $\mathcal{M} = \langle \mathcal{C}, \Theta, \theta, [\![-]\!]^B, [\![-]\!]^C, [\![-]\!]^F \rangle$ an EFF Π-model, $p \colon \mathcal{E} \to \mathcal{C}$ be an FFLR, and $\dot{\mathcal{M}} = \langle \mathcal{E}, \Theta, \dot\theta, [\![-]\!]^{\dot B}, [\![-]\!]^{\dot C}, [\![-]\!]^{\dot F} \rangle$ a lifting of $\mathcal{M}$ along p. Furthermore, let $[\![-]\!]_\theta$ and $[\![-]\!]_{\dot\theta}$ be the semantics induced by $\mathcal{M}$ and $\dot{\mathcal{M}}$ respectively, then for the syntactic class of:*

- *effect types $\Theta \vdash_{\mathrm{k}} E : \mathbf{Eff}$: the monad $[\![E]\!]_{\dot\theta}$ is a lifting of $[\![E]\!]_\theta$ along p;*

- *value types* $\Theta \vdash_{\mathrm{k}} A : \mathbf{Val}$*: the object* $[\![A]\!]_{\dot\theta}$ *is a lifting of* $[\![A]\!]_\theta$ *along* p*;*
- *E-computation types* $\Theta \vdash_{\mathrm{k}} C : \mathbf{Comp}_E$*: the* $[\![E]\!]_{\dot\theta}$*-algebra* $[\![C]\!]_{\dot\theta}$ *is a lifting of the* $[\![E]\!]_\theta$*-algebra* $[\![C]\!]_\theta$ *along* **Alg** p*; and*
- *contexts* $\Theta \vdash_{\mathrm{k}} \Gamma : \mathbf{Ctxt}$*: the object* $[\![\Gamma]\!]_{\dot\theta}$ *is a lifting of* $[\![\Gamma]\!]_\theta$ *along* p*.*

Proof. For effect types, proposition 7.3.14 proves that the effect monads induced by $\dot{\mathcal{M}}$ are lifts of those induced by $\mathcal{M}$. Value types lift because p strictly preserves the distributive structure as an FFLR and each base type is lifted. The definition of monad lifts and proposition 7.3.14 shows that **Alg** p strictly preserves forming products, exponential algebras, and free algebras, so computation types lift. Finally, contexts are preserved for the same reason value types are. □

Finally, we show that the semantics of value and computation terms lift.

Theorem 7.3.18 (Basic lemma for terms)**.** *Let* $\Pi = \langle B, C, F, \mathrm{type}_{\mathrm{const}}, \mathrm{type}_{\mathrm{func}}, \mathrm{eval}\rangle$ *be an operational signature,* $\mathcal{M} = \langle \mathcal{C}, \Theta, \theta, [\![-]\!]^B, [\![-]\!]^C, [\![-]\!]^F\rangle$ *an* EFF Π*-model,* $p\colon \mathcal{E} \to \mathcal{C}$ *be an FFLR, and* $\dot{\mathcal{M}} = \langle \mathcal{E}, \Theta, \dot\theta, [\![-]\!]^{\dot B}, [\![-]\!]^{\dot C}, [\![-]\!]^{\dot F}\rangle$ *a lifting of* $\mathcal{M}$ *along* p*. Furthermore, let* $[\![-]\!]_\theta$ *and* $[\![-]\!]_{\dot\theta}$ *be the semantics induced by* $\mathcal{M}$ *and* $\dot{\mathcal{M}}$ *respectively, then:*

- *if* $\Theta; \Gamma \vdash V : A$ *then* $[\![V]\!]_\theta : [\![\Gamma]\!]_{\dot\theta} \dot\to [\![A]\!]_{\dot\theta}$*; and*
- *if* $\Theta; \Gamma \vdash_E M : C$ *then* $[\![M]\!]_\theta : [\![\Gamma]\!]_{\dot\theta} \dot\to \left|[\![C]\!]_{\dot\theta}\right|$.

Proof. By induction on judgments, the semantics induced by $\dot{\mathcal{M}}$ lie over those induced by $\mathcal{M}$, i.e. $p\,[\![V]\!]_{\dot\theta} = [\![V]\!]_\theta$ and $p\,[\![M]\!]_{\dot\theta} = [\![M]\!]_\theta$. We will prove the cases for operations, handler definitions, and handling as examples.

Suppose we have the an operation judgement $\Theta; \Gamma \vdash_E \mathsf{op}\ V : FB$ for $(\mathsf{op} : A \to B) \in E$. The typing judgment for operations is

$$\frac{(\mathsf{op} : A \to B) \in E \qquad \Theta; \Gamma \vdash V : A}{\Theta; \Gamma \vdash_E \mathsf{op}\ V : FB}$$

By theorem 7.3.17 we have $p\,[\![E]\!]_{\dot\theta} = [\![E]\!]_\theta$, $p\,[\![A]\!]_{\dot\theta} = [\![A]\!]_\theta$, $p\,[\![B]\!]_{\dot\theta} = [\![B]\!]_\theta$, and $p\,[\![\Gamma]\!]_{\dot\theta} = [\![\Gamma]\!]_\theta$. Likewise, we have $p\,[\![FB]\!]_{\dot\theta} = p\left(F_{\dot\Sigma}\,[\![B]\!]_{\dot\theta}\right) = F_\Sigma\,[\![B]\!]_\theta = [\![FB]\!]_\theta$ where $\dot\Sigma$ and Σ are the effect signatures induced by $\dot{\mathcal{M}}$ and $\mathcal{M}$ respectively. By induction we have $[\![V]\!]_\theta : [\![\Gamma]\!]_{\dot\theta} \dot\to [\![A]\!]_{\dot\theta}$, meaning that $p\,[\![V]\!]_{\dot\theta} = [\![V]\!]_\theta$. The denotation of an operation for $\mathcal{M}$ is given by

$$[\![\mathsf{op}\ V]\!]_\theta\,(\gamma) \coloneqq \varphi_{[\![B]\!]_\theta}\left(\iota_{\mathsf{op}}\left(\left\langle [\![V]\!]_\theta\,(\gamma), \mathbf{return}\right\rangle\right)\right)$$

and likewise for $\dot{\mathcal{M}}$. The functor p strictly preserves the bi-CCC structure morphisms as well as free algebras, and the denotation of $\mathsf{op}\ V$ contains only these. Thus, $p\,[\![\mathsf{op}\ V]\!]_{\dot\theta} = [\![\mathsf{op}\ V]\!]_\theta$.

Suppose we have a handler definition

$$\Theta; \Gamma \vdash \{\mathbf{return}\ x \mapsto M\} \uplus \{\mathsf{op}_i\ p\ k \mapsto N_i | 1 \leq i \leq n\} : A\ ^E\!\Rightarrow^{E'} C$$

and denote the term by H. The typing judgement for a handler definition is

$$\frac{\begin{array}{c}\Theta; \Gamma, x : A \vdash_{E'} M : C \\ \text{for all } 1 \leq i \leq n: \quad \Theta; \Gamma, p : A_i, k : U_{E'}(B_i \to C) \vdash_{E'} N_i : C\end{array}}{\Theta; \Gamma \vdash \{\mathbf{return}\ x \mapsto M\} \uplus \{\mathsf{op}_i\ p\ k \mapsto N_i | 1 \leq i \leq n\} : A\ ^E\!\Rightarrow^{E'} C}$$

As before, by theorem 7.3.17 all types involved lift. By induction we have $p\,[\![M]\!]_{\dot\theta} = [\![M]\!]_\theta$ and $p\,[\![N_i]\!]_{\dot\theta} = [\![N_i]\!]_\theta$ for all $1 \leq i \leq n$. The denotation of a handler definition for $\mathcal{M}$ is given by

$$\left[\!\left[\{\mathbf{return}\ x \mapsto M\} \uplus \{\mathsf{op}\ p\ k \mapsto N_{\mathsf{op}}\}_{\mathsf{op}}\right]\!\right]_\theta(\gamma) \coloneqq \left\langle \chi\left(\left\langle \overline{[\![N_{\mathsf{op}}]\!]_\theta} \right\rangle_{\mathsf{op}}\right), \overline{[\![M]\!]_\theta} \right\rangle$$

where $\chi\colon \prod_i (A_i \Rightarrow B) \to ((\coprod_i A_i) \Rightarrow B)$ and likewise for $\dot{\mathcal{M}}$. The map χ is constructed from the bi-CCC, and so are the currying operations $\overline{(-)}$. The functor p strictly preserves the bi-CCC structure morphisms, and the denotation contains only these. Thus, $p\,[\![H]\!]_{\dot\theta} = [\![H]\!]_\theta$.

Suppose we have a handler use $\Theta; \Gamma \vdash_{E'} \mathbf{handle}\ M\ \mathbf{with}\ H : C$. The typing judgement for a handler use is

$$\frac{\Theta; \Gamma \vdash_E M : FA \qquad \Theta; \Gamma \vdash H : A\ ^E\!\Rightarrow^{E'} C}{\Theta; \Gamma \vdash_{E'} \mathbf{handle}\ M\ \mathbf{with}\ H : C}$$

As before, by theorem 7.3.17 all types involved lift. By induction we have $p\,[\![M]\!]_{\dot\theta} = [\![M]\!]_\theta$ and $p\,[\![H]\!]_{\dot\theta} = [\![H]\!]_\theta$. The denotation of a handler use for $\mathcal{M}$ is given by

$$[\![\mathbf{handle}\ M\ \mathbf{with}\ H]\!]_\theta(\gamma) \coloneqq [\![M]\!]_\theta(\gamma) \gg\!= [\![H]\!]_\theta(\gamma)$$

and likewise for $\dot{\mathcal{M}}$. The functor p strictly preserves the bi-CCC structure morphisms and the internal free algebra construction maps, and the denotation contains only these. Thus, $p\,[\![\mathbf{handle}\ M\ \mathbf{with}\ H]\!]_{\dot\theta} = [\![\mathbf{handle}\ M\ \mathbf{with}\ H]\!]_\theta$. □

Chapter 8

Correctness of Selected Standard AD Modes

Saffron

We can now use the mathematical tools we developed to prove correctness of selected AD algorithms. Section 8.1 introduces the category of diffeological spaces. Diffeological spaces are a generalization of Euclidean spaces and smooth manifolds such that the result is suitable for being the denotational semantics of our model effect and handler language. Furthermore, it is sufficiently rich to allow for our generalized logical relations. We will provide two fibrations for logical relations (FFLRs), a simpler one which we use in our proofs and richer construction for possible future use. We then apply the diffeological semantics in sections 8.2 and 8.3 to prove forward mode and continuation reverse mode correct respectively.

8.1 Diffeological spaces

The simplest form of differentiation is defined for real-valued functions $f\colon \mathbb{R} \to \mathbb{R}$. A function $f\colon \mathbb{R} \to \mathbb{R}$ which has derivatives of all orders is known as *smooth*. Differentiation can be extended to functions $f\colon \mathbb{R}^n \to \mathbb{R}^m$ between *Euclidean spaces* $\mathbb{R}^n$ for $n \in \mathbb{N}$. Again, $f\colon \mathbb{R}^n \to \mathbb{R}^m$ which has partial derivatives of all orders is also known as smooth. Furthermore, note that it makes sense to consider smooth maps $f\colon U \to V$ between open subsets $U \in \mathbb{R}^n$ and $V \in \mathbb{R}^m$. Thus, we define the following.

Definition 8.1.1 (Baez and Hoffnung, 2009, Definition 1)**.** An *open set* is an open subset of $\mathbb{R}^n$ for any $n \in \mathbb{N}$. A function $f\colon U \to V$ between open sets is called smooth if it has continuous derivatives of all orders.

We can then form a category with open subsets and smooth maps.

Definition 8.1.2. The category **Eucl** of *Euclidean domains and smooth maps* has open subsets as objects and smooth maps as morphisms.

The category **Eucl** is cartesian with the standard set-theoretic cartesian structure. However, it does not have coproducts nor is it closed, and thus it is insufficient for our purposes. To remedy this, we will use *diffeological spaces*.

Definition 8.1.3 (Diffeological space)**.** A *diffeological space* is a set X equipped with, for each open set U, a collection of maps $\varphi\colon U \to X$ called *plots* denoted $\mathcal{P}_U^X$, such that

- Every constant map $U \to X$ belongs to $\mathcal{P}_U^X$.
- If $\varphi \in \mathcal{P}_U^X$ and $f\colon V \to U$ is a smooth map, then $\varphi \cdot f \in \mathcal{P}_V^X$.
- Let $\varphi\colon U \to X$ and $\{U_i\}_{i\in I}$ be an open cover of U. If $\varphi|_{U_i} \in \mathcal{P}_{U_i}^X$ for all $i \in I$, then $\varphi \in \mathcal{P}_U^X$.

The collection of maps $\mathcal{P}^X \coloneqq \bigcup_U \mathcal{P}_U^X$ is a *diffeology* on X.

Definition 8.1.4. Let X and Y be diffeological spaces. A function $f\colon X \to Y$ is *smooth* if, for every plot $\varphi\colon U \to X$ of X , $f \cdot \varphi\colon U \to Y$ is a plot of Y.

Note that the identity function is smooth and that the composition of smooth maps is again smooth.

Definition 8.1.5. The category **Diff** has diffeological spaces as objects and smooth maps as morphisms.

We will draw facts about diffeological spaces from [Iglesias-Zemmour, 2013], [Baez and Hoffnung, 2009], and [van der Schaaf, 2020]. There are two canonical ways to assign an arbitrary set a diffeology.

Example 8.1.6 (Iglesias-Zemmour, 2013, Section 1.20)**.** Let X be a set. We will define the smallest diffeology possible for X, the *discrete diffeology*, and we denote the resulting diffeological space as $X_\circ$. Define $\mathcal{P}_U^{X_\circ}$ to be maps $\varphi\colon U \to X$ satisfying the following condition: for all $u \in U$ there exists an open neighborhood V of u such that $\varphi|_V$ is constant. Thus, the plots are the locally constant functions.

Example 8.1.7 (Iglesias-Zemmour, 2013, Section 1.21)**.** Let X be a set. The largest diffeology possible for X has plots $\varphi\colon U \to X$ given by all functions. We denote this diffeological space by $X_\bullet$ and it is called the *coarse diffeology*.

There is a forgetful functor $U\colon \mathbf{Diff} \to \mathbf{Set}$ given by forgetting the diffeology. The constructions $(-)_\circ$ and $(-)_\bullet$ each form fully faithful functors from **Set** to **Diff**. Each functor is adjoint to U, specifically we have $(-)_\circ \dashv U \dashv (-)_\bullet$. Thus, U preserves limits and colimits.

Diffeological spaces generalize Euclidean spaces and likewise have products, but they also have coproducts and exponentials.

Example 8.1.8 (Iglesias-Zemmour, 2013, Sections 1.9 and 1.16)**.** Every open set W is canonically a diffeological space by $\mathcal{P}_U^W \coloneqq \mathbf{Eucl}(U, W)$, i.e. all smooth maps from U to W. Clearly constant maps are smooth, the composition of smooth maps is smooth, and compatible smooth maps for an open cover can be glued together. Furthermore, given two open sets W_1, W_2, we have $\mathbf{Diff}(W_1, W_2) = \mathbf{Eucl}(W_1, W_2)$. Thus, **Eucl** embeds fully and faithfully into **Diff**.

Example 8.1.9 (Iglesias-Zemmour, 2013, Section 1.39)**.** Let $\{X_i\}_{i\in I}$ be a family of diffeological spaces for some set I, we will define their coproduct. Define $X \coloneqq \coprod_{i\in I} X_i = \{(i, x) : i \in I, x \in X_i\}$ to be the coproduct of the underlying sets. We will define the smallest diffeology on X such that each inclusion $\iota\colon X_i \to X$ is smooth. Define $\mathcal{P}_U^X$ to be maps $\varphi\colon U \to X$ satisfying the following condition: for every $u \in U$ there exists $i \in I$ and an open neighborhood V of u such that

$\varphi(V) \subseteq \{i\} \times X_i$ and $\varphi|_V \in \mathcal{P}_V^{X_i}$. Namely, the plots of X are maps which are locally plots of some X_i. With this diffeology, X is the coproduct of $\{X_i\}_{i\in I}$.

Example 8.1.10 (Iglesias-Zemmour, 2013, Section 1.55)**.** Let $\{X_i\}_{i\in I}$ be a family of diffeological spaces for some set I, we will define their product. Define $X \coloneqq \prod_{i\in I} X_i$ to be the product of the underlying sets. We will define the largest diffeology on X such that each in projection $\pi_i \colon X \to X_i$ is smooth. Define $\mathcal{P}_U^X$ to be maps $\varphi \colon U \to X$ such that for each $i \in I$, $\pi_i \cdot \varphi \in \mathcal{P}_U^{X_i}$. With this diffeology, X is the product of $\{X_i\}_{i\in I}$.

Example 8.1.11. Let X and Y be diffeological spaces, we will define their exponential object. Define $X \Rightarrow Y \coloneqq \mathbf{Diff}(X, Y)$. We will define a diffeology on $X \Rightarrow Y$ such that evaluation is smooth. Define $\mathcal{P}_U^{X\Rightarrow Y}$ to be maps $\varphi \colon U \to (X \Rightarrow Y)$ such that $\lambda(u, x).\varphi(u)(x) \colon U \times X \to Y$ is smooth, where we take the product in **Diff** of U and X. With this diffeology, $X \Rightarrow Y$ is an exponential object.

With the structures defined, **Diff** is a bi-cartesian closed category. In fact, **Diff** has a very rich structure which we will take advantage of to prove that effect monads exist. One such aspect is that **Diff** is *locally cartesian closed*. A category $\mathcal{C}$ is locally cartesian closed if for all objects $X \in \mathcal{C}$ the slice category $\mathcal{C}/X$ is cartesian closed. Being locally cartesian closed is part of the definition of being a *quasitopos*, see [Johnstone, 2002a, A2.6] for a textbook account. We will not require any specific knowledge of quasitoposes besides it implying local cartesian closedness, we will merely use **Diff** being a quasitopos as part of a sufficient condition in a later theorem.

Theorem 8.1.12 (Baez and Hoffnung, 2009, Theorem 52)**.** *The category* **Diff** *is a quasitopos and has all small limits and small colimits.*

For the interested and knowledgeable reader, the above follows from **Diff** being a category of concrete sheaves. Specifically, **Eucl** has a Grothendieck topology induced by the coverage defined by open covers in the standard topological sense. With this coverage, **Eucl** is a concrete site and **Diff** is the category of concrete sheaves on **Eucl**.

We will now construct the FFLR to be used in our correctness proofs. To do so, we generate a sequence of three FFLRs: the first generalizing predicates to **Diff**, the second generalizing binary relations to **Diff**, and the third capturing

binary relations between smooth curves in **Diff**. The third will be the FFLR used in our proofs.

Example 8.1.13. Let $U\colon \mathbf{Diff} \to \mathbf{Set}$ be the forgetful functor sending a diffeological space to its underlying set; U is product preserving. Recall from example 7.3.4 that $\pi\colon \mathbf{Pred} \to \mathbf{Set}$ is a FFLR. Thus, we can apply proposition 7.3.5 to the pullback

$$\begin{array}{ccc} U^*\mathbf{Pred} & \xrightarrow{q} & \mathbf{Pred} \\ {\scriptstyle U^*\pi}\downarrow & & \downarrow{\scriptstyle \pi} \\ \mathbf{Diff} & \xrightarrow[U]{} & \mathbf{Set} \end{array}$$

and define $\mathbf{PredDiff} \coloneqq U^*\mathbf{Pred}$, meaning predicates in diffeological spaces, and $p \coloneqq U^*\pi$. Then $p\colon \mathbf{PredDiff} \to \mathbf{Diff}$ is a FFLR. An object in **PredDiff** is equivalent to a pair (A, X) where $X \in \mathbf{Diff}$ and A is a mere subset of X [1], and a map $f\colon (A, X) \to (B, Y)$ is a smooth function $f\colon X \to Y$ such that $f(A) \subseteq B$. We will use this representation.

We can use to proposition 7.3.5 to calculate the structure of $p\colon \mathbf{PredDiff} \to \mathbf{Diff}$. The maps $!_{U1}$, $?_{U0}$, l, m, and n are all identity morphisms, and so the bi-CC structure of **PredDiff** is essentially identical to that of **Pred**, and likewise for the inverse and direct image functors.

The category **PredDiff** not seem to retain any diffeological information in the predicate. In fact, each subset of a diffeological space inherits a diffeology, the interested reader may see appendix B for more information.

Example 8.1.14. Let $F\colon \mathbf{Diff} \times \mathbf{Diff} \to \mathbf{Diff}$ be defined as $(X, Y) \mapsto X \times Y$. Then F is product preserving as $(X_1 \times Y_1) \times (X_2 \times Y_2) \cong (X_1 \times X_2) \times (Y_1 \times Y_2)$. Thus, we can apply proposition 7.3.5 to the pullback

$$\begin{array}{ccc} F^*\mathbf{PredDiff} & \xrightarrow{q} & \mathbf{PredDiff} \\ {\scriptstyle F^*p}\downarrow & & \downarrow{\scriptstyle p} \\ \mathbf{Diff} \times \mathbf{Diff} & \xrightarrow[F]{} & \mathbf{Diff} \end{array}$$

and define $\mathbf{BRelDiff} \coloneqq F^*\mathbf{PredDiff}$, for binary relations in diffeological spaces, and $p \coloneqq F^*p$ where which p will be clear by context. Then $p\colon \mathbf{BRelDiff} \to \mathbf{Diff} \times \mathbf{Diff}$ is a FFLR. An object in **BRelDiff** is equivalent to a pair $(R, (X, Y))$ where $X, Y \in \mathbf{Diff}$ and R is a subset of $X \times Y$, and a map $(f_1, f_2)\colon (R, (X_1, X_2)) \to$

[1] Technically, the objects are $((A, UX), X)$ where $X \in \mathbf{Diff}$ and $(A, UX) \in \mathbf{Pred}$ above UX.

$(S, (Y_1, Y_2))$ is a pair of smooth functions $f_i \colon X_i \to Y_i$ such that $(f_1 \times f_2)(R) \subseteq S$. We will use this representation.

We can use to proposition 7.3.5 to calculate the structure of $p \colon \mathbf{BRelDiff} \to \mathbf{Diff} \times \mathbf{Diff}$. The calculations are straightforward, and so we only include the product of objects as an example. For objects $(R, (X_1, X_2))$ and $(S, (Y_1, Y_2))$, their product is

$$\left(\left\{((x_1, y_1), (x_2, y_2)) : (x_1, x_2) \in R, (y_1, y_2) \in S\right\}, (X_1 \times Y_1, X_2 \times Y_2)\right).$$

Example 8.1.15. Consider the functor $F \coloneqq (\mathbb{R} \Rightarrow -) \times (\mathbb{R} \Rightarrow -) \colon \mathbf{Diff} \times \mathbf{Diff} \to \mathbf{Diff} \times \mathbf{Diff}$. Then F is product preserving as $(\mathbb{R} \Rightarrow X_1 \times X_2, \mathbb{R} \Rightarrow Y_1 \times Y_2) \cong ((\mathbb{R} \Rightarrow X_1) \times (\mathbb{R} \Rightarrow X_2), (\mathbb{R} \Rightarrow Y_1) \times (\mathbb{R} \Rightarrow Y_2))$. Thus, we can again apply proposition 7.3.5 to the pullback

$$\begin{array}{ccc} F^*\mathbf{BRelDiff} & \xrightarrow{q} & \mathbf{BRelDiff} \\ {\scriptstyle F^*p}\downarrow & & \downarrow{\scriptstyle p} \\ \mathbf{Diff} \times \mathbf{Diff} & \xrightarrow[F]{} & \mathbf{Diff} \times \mathbf{Diff} \end{array}$$

and define $\mathbf{BRelDiff}^{\mathbb{R}} \coloneqq F^*\mathbf{BRelDiff}$, for binary relations on curves in diffeological spaces, and $p \coloneqq F^*p$ where which p will be clear by context. Then $p \colon \mathbf{BRelDiff}^{\mathbb{R}} \to \mathbf{Diff} \times \mathbf{Diff}$ is a FFLR. An object in $\mathbf{BRelDiff}^{\mathbb{R}}$ is equivalent to a pair $(R, (X, Y))$ where $X, Y \in \mathbf{Diff}$ and R is a subset of $(\mathbb{R} \Rightarrow X) \times (\mathbb{R} \Rightarrow Y)$, and a map $(f_1, f_2) \colon (R, (X_1, X_2)) \to (S, (Y_1, Y_2))$ is a pair of smooth functions $f_i \colon X_i \to Y_i$ such that $((f_1 \cdot -) \times (f_2 \cdot -))(R) \subseteq S$. We will use this representation.

We will calculate the object structure of $p \colon \mathbf{BRelDiff}^{\mathbb{R}} \to \mathbf{Diff} \times \mathbf{Diff}$ using proposition 7.3.5. The fibre $\mathbf{BRelDiff}^{\mathbb{R}}{}_{(X_1, X_2)}$ consists of relations on $(\mathbb{R} \Rightarrow X_1) \times (\mathbb{R} \Rightarrow X_2)$ with the partial order given by set inclusion and fibered products by intersection. Let $(f_1, f_2) \colon (X_1, X_2) \to (Y_1, Y_2)$, then the inverse and direct image functors are

$$\begin{aligned} (f_1, f_2)^* (S, (Y_1, Y_2)) &\coloneqq \left(\{(\gamma_1, \gamma_2) : (f_1 \cdot \gamma_1, f_2 \cdot \gamma_2) \in S\}, (X_1, X_2)\right) \\ (f_1, f_2)_* (R, (X_1, X_2)) &\coloneqq \left(\{(f_1 \cdot \gamma_1, f_2 \cdot \gamma_2) : (\gamma_1, \gamma_2) \in R\}, (Y_1, Y_2)\right) \end{aligned}$$

on objects. The terminal object is $((\mathbb{R} \Rightarrow 1) \times (\mathbb{R} \Rightarrow 1), (1, 1))$ and the initial is $((\mathbb{R} \Rightarrow 0) \times (\mathbb{R} \Rightarrow 0) = \emptyset, (0, 0))$. The product of $(R, (X_1, X_2))$ and $(S, (Y_1, Y_2))$ is

$$\left(\left\{(\gamma, \gamma') : (\pi_1 \cdot \gamma, \pi_1 \cdot \gamma') \in R, (\pi_2 \cdot \gamma, \pi_2 \cdot \gamma') \in S\right\}, (X_1 \times Y_1, X_2 \times Y_2)\right),$$

their coproduct is

$$\left(\left\{(\iota_1 \cdot \gamma, \iota_1 \cdot \gamma') : (\gamma, \gamma') \in R\right\} \cup \left\{(\iota_2 \cdot \gamma, \iota_2 \cdot \gamma') : (\gamma, \gamma') \in S\right\}, (X_1 + Y_1, X_2 + Y_2)\right).$$

For $f \in \mathbb{R} \Rightarrow (X \Rightarrow Y), \gamma \in \mathbb{R} \Rightarrow X$, define $f \circledast \gamma \in \mathbb{R} \Rightarrow Y$ by $(f \circledast \gamma)(x) \coloneqq f(x)(\gamma(x))$. Then the exponential $(R, (X_1, X_2)) \dot{\Rightarrow} (S, (Y_1, Y_2))$ is

$$\left(\{(f_1, f_2) : \forall (\gamma_1, \gamma_2) \in R,\, (f_1 \circledast \gamma_1, f_2 \circledast \gamma_2) \in S\}, (X_1 \Rightarrow Y_1, X_2 \Rightarrow Y_2)\right).$$

The FFLR $p\colon \mathbf{BRelDiff}^{\mathbb{R}} \to \mathbf{Diff} \times \mathbf{Diff}$ is the fibration we will use in our correctness proofs based on logical relations. Recall that our formulation of logical relations requires an EFF model $\langle \mathbf{Diff} \times \mathbf{Diff}, \Theta, \theta \rangle$. We will use proposition 7.2.25, and so must prove that all functors of the form $B \Rightarrow -$ preserve λ_B-chains for some ordinal λ_B dependent on B. In fact, we will prove something stronger, that $B \Rightarrow -$ has *rank*.

Definition 8.1.16 (Borceux, 1994b, Definition 5.5.1)**.** A functor $F\colon \mathcal{C} \to \mathcal{D}$ has *rank* λ, for some regular cardinal λ, when F preserves λ-filtered colimits. It has *rank* when it has rank λ for some regular cardinal λ.

We recall the definition of a *generating set* and a proposition about them.

Definition 8.1.17 (Borceux, 1994a, Definition 4.5.1)**.** Let $\mathcal{C}$ be a category. A family $\{G_i\}_{i \in I}$ of objects of $\mathcal{C}$ is called a *family of generators* when, given any two parallel morphisms $u, v\colon A \to B$ in $\mathcal{C}$, if for all $i \in I$ and for all $g\colon G_i \to A$ we have $u \cdot g = v \cdot g$, then $u = v$. When the family consists of a single element G, then we say that G is a *generator* of $\mathcal{C}$.

Proposition 8.1.18 (Borceux, 1994a, Corollary 4.5.9)**.** *Let $\mathcal{C}$ be a category and $G \in \mathcal{C}$. The following conditions are equivalent:*

- *G is a generator;*
- *the functor $\mathcal{C}(G, -)\colon \mathcal{C} \to \mathbf{Set}$ is faithful.*

We now take advantage of the following theorem.

Theorem 8.1.19 (Johnstone, 2002b, Theorem C2.2.13)**.** *For a category $\mathcal{C}$ the following are equivalent:*

- *$\mathcal{C}$ is a locally small, cocomplete quasitopos with a generating set.*

- *$\mathcal{C}$ is locally presentable, locally cartesian closed, and quasi-effective.*

The category **Diff** is generated by the singleton space 1 as $\mathbf{Diff}(1, -)$ is faithful (because maps in **Diff** are just **Set** maps with extra properties). Furthermore, it is locally small, and by theorem 8.1.12 it is a cocomplete quasitopos. Thus, we conclude by the above that **Diff** is locally presentable. We will only use local presentability as a premise in the application of the next theorem, and so we do not expand on it here.

Theorem 8.1.20 (Borceux, 1994b, Theorem 5.5.7)**.** *Let $F \colon \mathcal{C} \to \mathcal{D}$ be a functor between two locally presentable categories. The following conditions are equivalent:*

- *F has a left adjoint;*
- *F has rank and preserves small limits.*

For any diffeological space B, the endofunctor $B \Rightarrow -$ has rank because it has a left adjoint $B \times -$. We can now conclude the following.

Proposition 8.1.21. *For any $B \in \mathbf{Diff} \times \mathbf{Diff}$, the functor $B \Rightarrow -$ preserves λ_B-chains for some ordinal λ_B dependent on B.*

Proof. By definition of exponentials in $\mathbf{Diff} \times \mathbf{Diff}$, $(B \Rightarrow -) = (B_1 \Rightarrow -) \times (B_2 \Rightarrow -)$ where $B \coloneqq (B_1, B_2)$. Each functor $B_i \Rightarrow -$ has rank, and so preserves λ_{B_i}-filtered colimits for some regular cardinal λ_{B_i}. Define $\lambda_B \coloneqq \max(\lambda_{B_1}, \lambda_{B_2})$, and so each $B_i \Rightarrow -$ preserve λ_B-filtered colimits. Colimits are computed componentwise in product categories, so $B \Rightarrow -$ preserves λ_B-filtered colimits. Clearly λ_B-chains are λ_B-filtered colimits, and so we are done. □

Corollary 8.1.22. *All triples of the form $\langle \mathbf{Diff}, \Theta, \theta \rangle$ and $\langle \mathbf{Diff} \times \mathbf{Diff}, \Theta, \theta \rangle$ are* EFF *models.*

We know have all we need for correctness, and readers may continue to section 8.2 if desired. The rest of this chapter focusses on the more general FFLR. For our second factorization system, we must introduce the notion of a *strong* epimorphism.

Definition 8.1.23 (Baez and Hoffnung, 2009, Definition 35)**.** In any category, an epimorphism $p \colon E \to B$ is *strong* if given any monomorphism $i \colon A \to X$ and

morphisms f, g making the outer square here commute:

$$\begin{array}{ccc} E & \xrightarrow{f} & A \\ {\scriptstyle p}\downarrow & \overset{t}{\nearrow} & \downarrow{\scriptstyle i} \\ B & \xrightarrow[g]{} & X \end{array}$$

then there exists a unique $t\colon B \to A$ making the whole diagram commute.

In **Diff**, there is an alternate characterization of strong epimorphisms in terms of *quotient spaces* which is convenient to work with.

Definition 8.1.24. We say a smooth map $p\colon E \to B$ makes B a *quotient space* of E if for every plot $\varphi \in \mathcal{P}^B_U$, there exists an open cover $\{U_i\}_{i\in I}$ of U and a collection of plots $\left\{\varphi_i : \varphi_i \in \mathcal{P}^E_{U_i}\right\}$ such that $\varphi|_{U_i} = p \cdot \varphi_i$ for all $i \in I$.

Proposition 8.1.25 (Baez and Hoffnung, 2009, Proposition 37)**.** *A smooth map $p\colon E \to B$ is a strong epimorphism if and only if p makes B a quotient space of E.*

We will see that strong epimorphisms in **Diff** have the additional property of being *regular*.

Definition 8.1.26 (Borceux, 1994a, Definition 4.3.1)**.** An epimorphism is called *regular* when it is the coequalizer of a pair of arrows.

Regular epimorphisms are required to define *regular categories*. Regular categories are a setting in which the calculus of relations can be carried out. However, we shall only use them to help generate a factorization system involving strong epimorphisms. Furthermore, the concept of *kernel pair* in the definition below will not be used, so we do not recall their definition.

Definition 8.1.27 (Borceux, 1994b, Definition 2.1.1)**.** A category $\mathcal{C}$ is *regular* when it satisfies the following conditions:

1. every arrow has a kernel pair;
2. every kernel pair has a coequalizer; and
3. the pullback of a regular epimorphism along any morphism exists and is again a regular epimorphism.

We then note the following three facts.

Proposition 8.1.28 (Borceux, 1994b, Proposition 2.1.4)**.** *In a regular category, a morphism is a regular epimorphism if and only if it is a strong epimorphism.*

Lemma 8.1.29 (Jacobs, 1999, Lemma 4.4.6)**.** *In a regular category, (regular epimorphisms, monomorphisms) is a factorization system.*[2]

Lemma 8.1.30 (Johnstone, 2002a, Lemma 1.5.13)**.** *Let $\mathcal{C}$ be a locally cartesian closed category. If $\mathcal{C}$ has coequalizers, then it is regular.*

Thus, we can can conclude the subsequent corollary.

Corollary 8.1.31. Diff *is a regular category, and so has (strong epimorphisms, monomorphisms) as a factorization system.*

We now prove the additional properties required to be a factorization system for logical relations.

Theorem 8.1.32. *The factorization system (strong epimorphisms, monomorphisms) on* **Diff** *is a factorization system for logical relations.*

Proof. We must prove the five properties of definition 7.3.9. Let **Mono** denote the class of monomorphisms in **Diff** and **SEpi** denote the class of strong epimorphisms in **Diff**, both of which can be viewed as a full subcategory of $\mathbf{Diff}^{\to}$ the arrow category of **Diff**.

Diff *has pullbacks monomorphisms:* This is trivial as **Diff** is complete.

Every monomorphism is a monomorphism: Trivially true.

*For every $Y \in$ **Diff** the fibre $\mathbf{Mono}_Y$ has small products:* Fix an object $Y \in \mathbf{Diff}$. The objects in $\mathbf{Mono}_Y$ are the objects of **Mono** over Y via cod, i.e. monomorphisms $m\colon X \to Y$. For monomorphisms $m_1, m_2\colon X_i \to Y$, a map $f\colon m_1 \to m_2$ in **Mono** is a morphism $f\colon X_1 \to X_2$ such that $m_1 = m_2 \cdot f$. Furthermore, in a full subcategory of an arrow category, which **Mono** is, the product of objects is given by their pullback if it belongs to the subcategory. Thus the pullback of a set of monomorphisms exists in **Diff** as it is complete, and the pullback of monomorphisms is a monomorphism, and so we are done.

Mono *is closed under binary coproducts:* Let $m_i\colon X_i \to Y_i$ for $i = 1, 2$ be monomorphisms. As objects in $\mathbf{Diff}^{\to}$, their coproduct is $m_1 + m_2\colon X_1 + X_2 \to$

[2]This lemma has been reduced to the relevant statements.

$Y_1 + Y_2$ and the commutative square

$$\begin{array}{ccc} X_i & \xrightarrow{\iota_i} & X_1 + X_2 \\ {\scriptstyle m_i}\downarrow & & \downarrow{\scriptstyle m_1+m_2} \\ Y_i & \xrightarrow[\iota_i]{} & Y_1 + Y_2 \end{array}$$

gives the coprojections $m_i \to m_1 + m_2$. Thus, we need to show that $m_1 + m_2$ is a monomorphism. Indeed, the map $m_1 + m_2$ is smooth and is injective on the underlying sets, and thus is a monomorphism in **Diff**.

SEpi *is closed under binary products:* Let $e_i \colon X_i \to Y_i$ be strong epimorphisms. As objects in $\mathbf{Diff}^{\to}$, their product is $e_1 \times e_2$ with projections analogously to the coproduct case. We must show $e_1 \times e_2$ is an strong epimorphism. We already know $e_1 \times e_2$ is smooth, and it is clearly surjective on the underlying sets, and so it is an epimorphism.

We will show that $e_1 \times e_2$ makes $Y_1 \times Y_2$ into a quotient space of $X_1 \times X_2$. Let $\varphi \in \mathcal{P}_U^{Y_1 \times Y_2}$. Then there exist $\varphi_1 \in \mathcal{P}_U^{Y_1}$ and $\varphi_2 \in \mathcal{P}_U^{Y_2}$ such that $\varphi = \varphi_1 \times \varphi_2$. Furthermore, because p_1 is strong, there exists an open cover $\left\{U_{i,1}\right\}_{i \in I}$ and collections of plots $\left\{\varphi_{i,1} : \varphi_{1,i} \in \mathcal{P}_{U_{1,i}}^{X_1}\right\}$ such that $\varphi_1|_{U_{1,i}} = p_1 \cdot \varphi_{1,i}$ for all $i \in I$. Likewise, there exists an open cover $\left\{U_{2,j}\right\}_{j \in J}$ and collections of plots $\left\{\varphi_{2,j} : \varphi_{2,j} \in \mathcal{P}_{U_{2,j}}^{E_2}\right\}$ such that $\varphi_2|_{U_{2,j}} = p_2 \cdot \varphi_{2,j}$ for all $j \in J$. Define $V_{i,j} \coloneqq U_{1,i} \cap U_{2,j}$, then $\left\{V_{i,j}\right\}_{i \in I, j \in J}$ is an open cover of U. Next, define $\varphi_{i,j} \coloneqq {\varphi^1}_i|_{V_{i,j}} \times {\varphi^2}_j|_{V_{i,j}} \in \mathcal{P}_{V_{i,j}}^{X_1 \times X_2}$. Then we see that

$$\begin{aligned} (p_1 \times p_2) \cdot \varphi_{i,j} &= \left(p_1 \cdot \left(\varphi_{1,i}|_{V_{i,j}}\right)\right) \times \left(p_2 \cdot \left(\varphi_{2,j}|_{V_{i,j}}\right)\right) \\ &= \left(\left(p_1 \cdot \varphi_{1,i}\right)|_{V_{i,j}}\right) \times \left(\left(p_2 \cdot \varphi_{2,j}\right)|_{V_{i,j}}\right) \\ &= \left(\left(\varphi_1|_{U_{1,i}}\right)|_{V_{i,j}}\right) \times \left(\left(\varphi_2|_{U_{2,j}}\right)|_{V_{i,j}}\right) = \varphi|_{V_{i,j}} \end{aligned}$$

and so $e_1 \times e_2$ makes $Y_1 \times Y_2$ into a quotient space of $X_1 \times X_2$. □

We can now combine lemma 7.3.10 and theorem 8.1.32.

Corollary 8.1.33. *Let* **Mono** *be the full subcategory of the arrow category of* **Diff**. *The restricted codomain fibration* $\mathrm{cod} \colon \mathbf{Mono} \to \mathbf{Diff}$ *is a FFLR.*

It is important to note that the fibres of $\mathrm{cod} \colon \mathbf{Mono} \to \mathbf{Diff}$ are not posets, but are in fact large preorders. However, **Diff** is *well-powered*, meaning that it has a small poset of subobjects, i.e. equivalence classes of monomorphisms. Furthermore, one can create a fiberwise equivalence between $\mathrm{cod} \colon \mathbf{Mono} \to \mathbf{Diff}$

and $\mathrm{cod}'\colon \mathbf{Sub} \to \mathbf{Diff}$ where $\mathbf{Sub}$ is a fiberwise replacement of the large preorder of monomorphisms with the induced poset of subobjects. We will not delve into this construction because this FFLR is not used in the construction of the FFLR used in our correctness proofs.

We can perform the same sequences of steps on $p\colon \mathbf{Sub} \to \mathbf{Diff}$ as we did to $p\colon \mathbf{PredDiff} \to \mathbf{Diff}$ to obtain an analogous FFLR to $p\colon \mathbf{BRelDiff}^{\mathbb{R}} \to \mathbf{Diff} \times \mathbf{Diff}$. The new FFLR would be similar, but would allow an arbitrary diffeology on the relation between curves.

8.2 Correctness of Forward Mode

We can now prove forward mode correct. We begin by defining the data needed for a few operational signatures. Fix $N \in \mathbb{N}$, $N > 2$, and a collection of subspaces $\mathbf{Op}_n \subseteq \left(\mathbb{R}^{\mathbb{R}^n}\right)_\circ$ for $0 \leq n < N$ such that $0 \in \mathbf{Op}_0$ and $+, * \in \mathbf{Op}_2$ which are closed under differentiation as a family. The family $\{\mathbf{Op}_n\}_{0 \leq n < N}$ specifies what basic mathematical operations are allowed. We will also use $\mathbf{Op}_n$ to denote a base type in EFF. Furthermore, we will require the real numbers $\mathbb{R}$ as a diffeological space and will also use $\mathbb{R}$ to denote a base type in EFF. Thus, define our set of base types as $B \coloneqq \{\mathbf{Op}_n\}_{0 \leq n \leq N} \cup \{\mathbb{R}\}$. We define our constants as $C \coloneqq \mathbb{R} \cup \bigcup_n \mathbf{Op}_n$. For built-in functions, we require a way to evaluate smooth functions and determine their derivatives. We need N functions $eval_n\colon \mathbf{Op}_n \times \mathbb{R}^n \to \mathbb{R}$ to evaluate functions for $0 \leq n < N$ and $(N-1)N/2$ functions $\partial^n_m\colon \mathbf{Op}_n \to \mathbf{Op}_n$ for $1 \leq m \leq n < N$ to calculate the partial derivative of the m^{th} argument. Define $F \coloneqq \{eval_n\}_{0 \leq n \leq N} \cup \{\partial^n_m\}_{1 \leq m \leq n < N}$. The functions $\mathrm{types}_{\mathrm{const}}$ and $\mathrm{types}_{\mathrm{func}}$ are given the evident definitions. Finally, the function eval is defined such that each $eval_n$ and ∂^n_m computes the correct mathematical result. We then define the following operational signatures:

Operations only: $\Pi_{\mathbf{Op}} \coloneqq \langle B \setminus \{\mathbb{R}\}, C \setminus \mathbb{R}, \emptyset, \mathrm{types}_{\mathrm{const}}|_{C \setminus \mathbb{R}}, \emptyset, \emptyset \rangle$ where the only base types and constants are operations and there are no built-in functions;

Base types, no real constants: $\Pi_B \coloneqq \langle B, C \setminus \mathbb{R}, \emptyset, \mathrm{types}_{\mathrm{const}}|_{C \setminus \mathbb{R}}, \emptyset, \emptyset \rangle$ where there are all base types, but only constants for operations, and there are no built-in functions; and

Full: $\Pi_F \coloneqq \langle B, C, F, \mathrm{types}_{\mathrm{const}}, \mathrm{types}_{\mathrm{func}}, \mathrm{eval} \rangle$ of all data.

Let $\mathbf{R}$ be a type variable, we define the effect

$$E^{\mathbf{R}} \coloneqq \left\{ \mathsf{ap}_n^{\mathbf{R}} : \mathbf{Op}_n \times \mathbf{R}^n \to \mathbf{R} : 0 \leq n \leq N \right\}$$

for smooth functions, where $\mathsf{ap}_n^{\mathbf{R}}$ represents applying an n-ary function to n arguments. We will focus on differentiating terms of the form $\mathbf{R}; x : \mathbf{R} \vdash_{E^{\mathbf{R}}} M : F\mathbf{R}$ which use $\Pi_{\mathbf{Op}}$. The type variable $\mathbf{R}$ corresponds to the type variables found in our definition of smooth effects in chapter 4. For evaluation, we instantiated it to floats to model the real numbers. For forward mode in section 4.1, we instantiated the type variable with a pair of numbers, one for the original value an one for the derivative. Thus, we will do the same here. By corollary 8.1.22, the tuple $\langle \mathbf{Diff} \times \mathbf{Diff}, \{\mathbf{R}\}, \left\{ \mathbf{R} \mapsto (\mathbb{R}, \mathbb{R}^2) \right\} \rangle$ is an EFF model. The mapping of $\mathbf{R}$ to $(\mathbb{R}, \mathbb{R}^2)$ models the type variable instantiation for evaluation in the first component and for forward mode in the second component. Moreover, by interpreting the base types $\mathbf{Op}_n$ and its constants using $(\mathbf{Op}_n, \mathbf{Op}_n)$, we obtain a $\Pi_{\mathbf{Op}}$-model, which we denote by $[\![-]\!]$.

We require an FFLR to apply logical relations to $[\![-]\!]$, for which we use $p \colon \mathbf{BRelDiff}^{\mathbb{R}} \to \mathbf{Diff} \times \mathbf{Diff}$. Additionally, we must choose lifts of $[\![\mathbf{Op}_n]\!] = (\mathbf{Op}_n, \mathbf{Op}_n)$ and $[\![\mathbf{R}]\!] = (\mathbb{R}, \mathbb{R}^2)$. For $[\![\mathbf{Op}_n]\!]$, we choose equality and denote it by $\mathbf{Op}_n^{=}$. With this choice, the constants of type $\mathbf{Op}_n$ lift. For $[\![\mathbf{R}]\!]$, we choose the set $\{(f, \langle f, \nabla f \rangle) : f \in \mathbb{R} \Rightarrow \mathbb{R}\} =: R_{\mathrm{Dual}}$. The set R_{Dual} is a relation between smooth curves $f \colon \mathbb{R} \to \mathbb{R}$ and the same curve f paired with its derivative $\langle f, \nabla f \rangle : \mathbb{R} \to \mathbb{R}^2$. We can thus apply the basic lemma of logical relations to $\mathbf{R}; x : \mathbf{R} \vdash_{E^{\mathbf{R}}} M : F\mathbf{R}$.

Proposition 8.2.1. *Given a term* $\mathbf{R}; x : \mathbf{R} \vdash_{E^{\mathbf{R}}} M : F\mathbf{R}$ *using the operational signature* $\Pi_{\mathbf{Op}}$, $[\![M]\!]$ *lifts to*

$$[\![M]\!] : R_{\mathrm{Dual}} \dot{\to} \dot{T}_{[\![E^{\mathbf{R}}]\!]} R_{\mathrm{Dual}}.$$

For our main theorem, we now wish to show that our forward mode calculates derivatives. To do so, we must factorize our interpretation $[\![-]\!]$. We will define two models into $\mathbf{Diff}$ based on different instantiations of $\mathbf{R}$. Define the effect

$$E^{\mathbb{R}} \coloneqq \left\{ \mathsf{ap}_n^{\mathbb{R}} : \mathbf{Op}_n \times \mathbb{R}^n \to \mathbb{R} : 0 \leq n \leq N \right\}$$

and note that $\mathsf{ap}_n^{\mathbf{R}}$ and $\mathsf{ap}_n^{\mathbb{R}}$ are syntactically distinct operations. Next, we define a functorial macro $\mathfrak{M}_{\mathbf{R}}^{\mathbb{R}}$ on programs not containing the operations of $E^{\mathbb{R}}$ in their

effects. By macro, we want to emphasize that the operation $\mathfrak{M}^{\mathbb{R}}_{\mathbf{R}}$ acts on program syntax, and by functorial we want to emphasize that it is structurally recursive. The operation $\mathfrak{M}^{\mathbb{R}}_{\mathbf{R}}$ will essentially replace all occurrences of $\mathbf{R}$ by $\mathbb{R}$. For types, $\mathfrak{M}^{\mathbb{R}}_{\mathbf{R}}(\mathbf{R}) \coloneqq \mathbb{R}$, the identity on other base types, and homomorphically on type formers. Define $\mathfrak{M}^{\mathbb{R}}_{\mathbf{R}}\left(\{\mathsf{op}_i : A \to B\}_i\right) \coloneqq \left\{\mathfrak{M}^{\mathbb{R}}_{\mathbf{R}}(\mathsf{op}_i) : \mathfrak{M}^{\mathbb{R}}_{\mathbf{R}}(A) \to \mathfrak{M}^{\mathbb{R}}_{\mathbf{R}}(B)\right\}_i$ for effects where $\mathfrak{M}^{\mathbb{R}}_{\mathbf{R}}(\mathsf{ap}^{\mathbf{R}}_n) \coloneqq \mathsf{ap}^{\mathbb{R}}_n$ and $\mathfrak{M}^{\mathbb{R}}_{\mathbf{R}}(\mathsf{op}_i) \coloneqq \mathsf{op}_i$ for $\mathsf{op}_i \notin E^{\mathbf{R}}$. For terms, $\mathfrak{M}^{\mathbb{R}}_{\mathbf{R}}(\mathsf{ap}^{\mathbf{R}}_n) \coloneqq \mathsf{ap}^{\mathbb{R}}_n$ on operations and handler cases, the identity on other base terms, and homomorphically on term formers. For contexts, $\left(\mathfrak{M}^{\mathbb{R}}_{\mathbf{R}}(\Gamma)\right)(x) \coloneqq \mathfrak{M}^{\mathbb{R}}_{\mathbf{R}}(\Gamma(x))$ for all $x \in \mathrm{Dom}(\Gamma)$. The above description is sufficient as the only base terms with types involving $\mathbf{R}$ are variables and $\mathsf{ap}^{\mathbf{R}}_n$. We will also define

$$E^{\mathbb{R}^2} \coloneqq \left\{\mathsf{ap}^{\mathbb{R}^2}_n : \mathbf{Op}_n \times \left(\mathbb{R}^2\right)^n \to \mathbb{R}^2 : 0 \leq n \leq N\right\}$$

and let the analogous macro be denoted by $\mathfrak{M}^{\mathbb{R}^2}_{\mathbf{R}}$.

Proposition 8.2.2. *The macros $\mathfrak{M}^{\mathbb{R}}_{\mathbf{R}}$ and $\mathfrak{M}^{\mathbb{R}^2}_{\mathbf{R}}$ respect kinding, typing, substitution, and contextual equivalence for programs not including operations from $E^{\mathbb{R}}$ and $E^{\mathbb{R}^2}$ respectively.*

Proof. By induction on syntax, judgments, and program contexts. Note that $\mathfrak{M}^{\mathbb{R}}_{\mathbf{R}}$ is almost type substitution, except that $\mathsf{ap}^{\mathbf{R}}_n$ is replaced by $\mathsf{ap}^{\mathbb{R}}_n$, which is syntactically distinct. The restriction of not including operations from $E^{\mathbb{R}}$ ensures the well-kindedness of effects. Likewise for $\mathfrak{M}^{\mathbb{R}^2}_{\mathbf{R}}$. □

Next, we define a **Diff**-valued Π_B-model for terms with no type variables. By corollary 8.1.22, the tuple $\langle \mathbf{Diff}, \emptyset, \emptyset \rangle$ is an EFF model. Interpreting $\mathbb{R}$ as a diffeological space and the base types $\mathbf{Op}_n$ and its constants using the corresponding diffeological spaces, we obtain a $\Pi_{\mathbf{Op}}$-model, which we denote by $[\![-]\!]^{\mathbb{R}}$. By proposition 8.2.2, $\left[\!\left[\mathfrak{M}^{\mathbb{R}}_{\mathbf{R}}(-)\right]\!\right]^{\mathbb{R}}$ is a valid $\Pi_{\mathbf{Op}}$-model where we have one type variable $\mathbf{R}$ which is interpreted as $\mathbb{R}$, and likewise for $\left[\!\left[\mathfrak{M}^{\mathbb{R}^2}_{\mathbf{R}}(-)\right]\!\right]^{\mathbb{R}}$ but for $\mathbb{R}^2$. Then on the EFF-fragment which contains operations from neither $E^{\mathbb{R}}$ nor $E^{\mathbb{R}^2}$, we see $[\![-]\!] = \left[\!\left[\mathfrak{M}^{\mathbb{R}}_{\mathbf{R}}(-)\right]\!\right]^{\mathbb{R}} \times \left[\!\left[\mathfrak{M}^{\mathbb{R}^2}_{\mathbf{R}}(-)\right]\!\right]^{\mathbb{R}}$.

We can also extend $[\![-]\!]^{\mathbb{R}}$ to a Π_F-model. Real number constants are interpreted as elements of $\mathbb{R}$. For built-in functions, we have two families. The function $\mathit{eval}_n \colon \mathbf{Op}_n \times \mathbb{R}^n \to \mathbb{R}$ is interpreted using the smooth inclusions $\mathbf{Op}_n \subseteq \left(\mathbb{R}^{\mathbb{R}^n}\right)_\circ \hookrightarrow \mathbb{R}^{\mathbb{R}^n}$ and the evaluation morphism given by the CCC structure. The function $\partial^n_m \colon \mathbf{Op}_n \to \mathbf{Op}_n$ is interpreted as partial differentiation. Clearly these

interpretations satisfy the operational requirements imposed by Π_F. We also denote this Π_F-model by $[\![-]\!]^{\mathbb{R}}$ as the previous is merely a restriction.

Recall that the handler for evaluation is

$$\begin{aligned} H_{\text{eval}} = \ & \{\mathbf{return}\ x \quad \mapsto \mathbf{return}\ x \\ & \ \mathsf{ap}_n^{\mathbb{R}}\ (f, v)\ k \mapsto k!\ (eval_n\ (f, v)) \\ & \} \end{aligned}$$

where we have only shown the n^{th} handling clause for brevity. Additionally, recall that the notation for turning a value k of type $U_E C$ into a computation of type C is $k!$. For forward mode, we again only show the n^{th} clause for brevity, and we use an unspecified but trivial term $unzip$ which unzips tuples. Finally, we use an n-ary sum in place of addition, and all said the final definition is

$$\begin{aligned} H_{\text{forw}} = \ & \{\mathbf{return}\ x \quad \mapsto \mathbf{return}\ x \\ & \ \mathsf{ap}_n^{\mathbb{R}^2}\ (f, v)\ k \mapsto \mathbf{case}\ unzip!\ v\ \mathbf{of} \\ & \quad (w, (dw_1, \ldots, dw_n)) \to k!\ (\mathsf{ap}_n^{\mathbb{R}}\ (f, w), \\ & \qquad r_1 \leftarrow \mathsf{ap}_2^{\mathbb{R}}\ (*, (\mathsf{ap}_n^{\mathbb{R}}\ (\partial_1^n\ f, w), dw_1)); \\ & \qquad \ldots \\ & \qquad r_n \leftarrow \mathsf{ap}_2^{\mathbb{R}}\ (*, (\mathsf{ap}_n^{\mathbb{R}}\ (\partial_n^n\ f, w), dw_n)); \\ & \qquad \mathsf{ap}_n^{\mathbb{R}}\ (sum, (r_1, \ldots, r_n)) \\ & \quad) \\ & \} \end{aligned}$$

where we have used some syntactic sugar in each r_i binding. Specifically we wrote the ill-formed

$$r_i \leftarrow \mathsf{ap}_2^{\mathbb{R}}\ (*, (\mathsf{ap}_n^{\mathbb{R}}\ (\partial_i^n\ f, w), dw_i));\ \ldots$$

as shorthand for

$$x \leftarrow \mathsf{ap}_n^{\mathbb{R}}\ (\partial_i^n\ f, w);\ r_i \leftarrow \mathsf{ap}_2^{\mathbb{R}}\ (*, (x, dw_i));\ \ldots$$

for fresh x, which is well-formed. Let $\mathbf{R}; x : \mathbf{R} \vdash_{E^{\mathbf{R}}} M : F\mathbf{R}$ be such that M is in the domain of our macros. Define the following morphisms

$$\beta_1 \coloneqq [\![; y : \mathbb{R} \vdash_\emptyset \mathbf{handle}\ (\lambda x.\mathfrak{M}_{\mathbf{R}}^{\mathbb{R}}(M))\ y\ \mathbf{with}\ H_{\text{eval}} : F(\mathbb{R})]\!]^{\mathbb{R}}$$

$$\beta_2 \coloneqq \left[\!\!\left[; \mathfrak{y} : \mathbb{R}^2 \vdash_\emptyset \begin{array}{l} \mathbf{handle} \\ \quad \mathbf{handle}\ (\lambda x.\mathfrak{M}_{\mathbf{R}}^{\mathbb{R}^2}(M))\ y\ \mathbf{with}\ H_{\text{forw}} \\ \mathbf{with}\ H_{\text{eval}} \end{array} : F(\mathbb{R}^2) \right]\!\!\right]^{\mathbb{R}}$$

$$\beta \coloneqq \beta_1 \times \beta_2$$

using the Π_F-model $[\![-]\!]^{\mathbb{R}}$. The type of β, as a morphism in $\mathbf{Diff} \times \mathbf{Diff}$, is

$$\beta \colon (\mathbb{R}, \mathbb{R}^2) \to (\mathbb{R}, \mathbb{R}^2)$$

as the interpretation of F at the effect $\emptyset$ is the identity monad. We will show that β lifts along p using the definition of the free algebraic lift, i.e. that

$$\beta \colon R_{\mathrm{Dual}} \dot{\to} R_{\mathrm{Dual}}.$$

Let us begin calculating β_1 in the internal language

$$\beta_1 = \lambda y. \left[\!\left[\mathfrak{M}^{\mathbb{R}}_{\mathbf{R}}(M)\right]\!\right]^{\mathbb{R}} (y) \gg\!= H_{\mathrm{eval}}$$

and β_2

$$\beta_2 = \lambda y. \left(\left[\!\left[\mathfrak{M}^{\mathbb{R}^2}_{\mathbf{R}}(M)\right]\!\right]^{\mathbb{R}} (y) \gg\!= H_{\mathrm{eval}} \right) \gg\!= H_{\mathrm{forw}}.$$

Proposition 8.2.1 showed by the basic lemma for logical relations that $[\![M]\!] = \left[\!\left[\mathfrak{M}^{\mathbb{R}}_{\mathbf{R}}(M)\right]\!\right]^{\mathbb{R}} \times \left[\!\left[\mathfrak{M}^{\mathbb{R}^2}_{\mathbf{R}}(M)\right]\!\right]^{\mathbb{R}}$ lifts. Thus, defining

$$\begin{aligned} \alpha_1 &\coloneqq \lambda x. x \gg\!= H_{\mathrm{eval}} \\ \alpha_2 &\coloneqq \lambda x. (x \gg\!= H_{\mathrm{forw}}) \gg\!= H_{\mathrm{eval}} \\ \alpha &\coloneqq \alpha_1 \times \alpha_2 \end{aligned}$$

it is sufficient to show α lifts. Note that

$$\alpha \colon \left(T_{E^{\mathbb{R}}}\mathbb{R}, T_{E^{\mathbb{R}^2}}\mathbb{R}^2\right) \to \left(\mathbb{R}, \mathbb{R}^2\right)$$

and so, where $\dot{T}$ is the lift of $T_{E^{\mathbb{R}}} \times T_{E^{\mathbb{R}^2}}$, we must prove

$$\alpha \colon \dot{T} R_{\mathrm{Dual}} \dot{\to} R_{\mathrm{Dual}}.$$

Define $X \coloneqq \alpha^* R_{\mathrm{Dual}}$. Then

$$\alpha_* \eta_{T*} R_{\mathrm{Dual}} = (\alpha.\eta_T)_* R_{\mathrm{Dual}} = R_{\mathrm{Dual}}$$

because $\alpha.\eta_T = \eta_{T\emptyset} = \mathrm{id}$ by definition of the return clause definitions of each handler. Therefore

$$\eta_{T*} R_{\mathrm{Dual}} \leq \alpha^* R_{\mathrm{Dual}} = X$$

and so η_T respects X at R_{Dual}.

Next, we want to show

$$\overline{\mathsf{ap}_n^{\mathbb{R}}} \times \overline{\mathsf{ap}_n^{\mathbb{R}^2}} : \left(\mathbf{Op}_n^{=} \dot{\times} {R_{\mathrm{Dual}}}^n\right) \dot{\times} X^{R_{\mathrm{Dual}}} \dot{\to} X.$$

Let us begin by calculating $\alpha.\left(\overline{\mathsf{ap}_n^{\mathbb{R}}} \times \overline{\mathsf{ap}_n^{\mathbb{R}^2}}\right)$. Observe that

$$\left[\!\left[; f : \mathbf{Op}_n, v : \mathbb{R}^n \vdash_{E^{\mathbb{R}}} \mathsf{ap}_n^{\mathbb{R}}\ (f, v) : F\mathbb{R}\right]\!\right]^{\mathbb{R}} = \lambda x.\overline{\mathsf{ap}_n^{\mathbb{R}}}\left(x, \eta_{T_{E^{\mathbb{R}}}}\right)$$

and that

$$\begin{aligned}
&\left[\!\left[; f : \mathbf{Op}_n, v : \mathbb{R}^n, k : U_{E^{\mathbb{R}}}(\mathbb{R} \to F\mathbb{R}) \vdash_{E^{\mathbb{R}}} x \leftarrow \mathsf{ap}_n^{\mathbb{R}}\ (f, v);\ k!\ x : F\mathbb{R}\right]\!\right]^{\mathbb{R}} \\
&= \lambda f v k.\overline{\mathsf{ap}_n^{\mathbb{R}}}\left((f, v), \eta_{T_{E^{\mathbb{R}}}}\right) \gg\!=_{E^{\mathbb{R}}} k \\
&= \lambda f v k.\overline{\mathsf{ap}_n^{\mathbb{R}}}\left((f, v), \eta_{T_{E^{\mathbb{R}}}} \gg\!=_{E^{\mathbb{R}}} k\right) \\
&= \lambda f v k.\overline{\mathsf{ap}_n^{\mathbb{R}}}\left((f, v), k\right) \\
&= \overline{\mathsf{ap}_n^{\mathbb{R}}}
\end{aligned}$$

and similarly for $\overline{\mathsf{ap}_n^{\mathbb{R}^2}}$. Furthermore, for $\Gamma_1 \coloneqq f : \mathbf{Op}_n, v : \mathbb{R}^n, k : U_{E^{\mathbb{R}}}(\mathbb{R} \to F\mathbb{R})$ and $\Gamma_2 \coloneqq f : \mathbf{Op}_n, v : (\mathbb{R}^2)^n, k : U_{E^{\mathbb{R}^2}}(\mathbb{R}^2 \to F(\mathbb{R}^2))$, we see

$$\alpha_1.\overline{\mathsf{ap}_n^{\mathbb{R}}} = \left[\!\left[\begin{array}{ll} & \mathbf{handle} \\ & \quad x \leftarrow \mathsf{ap}_n^{\mathbb{R}}\ (f, v); \\ ; \Gamma_1 \vdash_{\emptyset} & \quad k!\ x \\ & \mathbf{with}\ H_{\mathrm{eval}} \end{array} : F(\mathbb{R})\right]\!\right]^{\mathbb{R}}$$

$$\alpha_2.\overline{\mathsf{ap}_n^{\mathbb{R}^2}} = \left[\!\left[\begin{array}{ll} & \mathbf{handle} \\ & \quad \mathbf{handle} \\ & \quad\quad x \leftarrow \mathsf{ap}_n^{\mathbb{R}^2}\ (f, v); \\ ; \Gamma_2 \vdash_{\emptyset} & \quad\quad k!\ x \\ & \quad \mathbf{with}\ H_{\mathrm{forw}} \\ & \mathbf{with}\ H_{\mathrm{eval}} \end{array} : F(\mathbb{R}^2)\right]\!\right]^{\mathbb{R}}$$

and after a few reductions via the operational semantics, which is valid thanks

to the soundness of our denotational semantics, we get

$$\alpha_1.\overline{\mathsf{ap}_n^{\mathbb{R}}} = \left[\!\!\left[;\Gamma_1 \vdash_\emptyset \begin{array}{l} \textbf{handle} \\ \quad k!\ (eval_n\ (f,v)) \\ \textbf{with}\ H_{\text{eval}} \end{array} : F(\mathbb{R}) \right]\!\!\right]^{\mathbb{R}}$$

$$\alpha_2.\overline{\mathsf{ap}_n^{\mathbb{R}^2}} = \left[\!\!\left[;\Gamma_2 \vdash_\emptyset \begin{array}{l} \textbf{handle} \\ \quad \textbf{handle} \\ \quad\quad \textbf{case}\ unzip!\ v\ \textbf{of} \\ \quad\quad (w, (dw_1, \ldots, dw_n)) \to k!\ (\mathsf{ap}_n^{\mathbb{R}}\ (f,w), \\ \quad\quad\quad r_1 \leftarrow \mathsf{ap}_2^{\mathbb{R}}\ (*, (\mathsf{ap}_n^{\mathbb{R}}\ (\partial_1^n\ f, w), dw_1)); \\ \quad\quad\quad \ldots \\ \quad\quad\quad r_n \leftarrow \mathsf{ap}_2^{\mathbb{R}}\ (*, (\mathsf{ap}_n^{\mathbb{R}}\ (\partial_n^n\ f, w), dw_n)); \\ \quad\quad\quad \mathsf{ap}_n^{\mathbb{R}}\ (sum, (r_1, \ldots, r_n)) \\ \quad\quad) \\ \quad \textbf{with}\ H_{\text{forw}} \\ \textbf{with}\ H_{\text{eval}} \end{array} : F(\mathbb{R}^2) \right]\!\!\right]^{\mathbb{R}}.$$

Thus,

$$\alpha_1.\overline{\mathsf{ap}_n^{\mathbb{R}}} = \alpha_1.\left[\!\!\left[;\Gamma_1 \vdash_{E^{\mathbb{R}}} k!\ (eval_n\ (f,v)) : F(\mathbb{R}) \right]\!\!\right]^{\mathbb{R}}$$

$$\alpha_2.\overline{\mathsf{ap}_n^{\mathbb{R}^2}} = \alpha_2.\left[\!\!\left[;\Gamma_2 \vdash_{E^{\mathbb{R}^2}} \begin{array}{l} \textbf{case}\ unzip!\ v\ \textbf{of} \\ (w, (dw_1, \ldots, dw_n)) \to k!\ (eval_n\ (f,w), \\ \quad eval_n\ (sum, (\\ \quad\quad eval_2\ (*, (eval_n\ (\partial_1^n\ f, w), dw_1)), \\ \quad\quad \ldots, \\ \quad\quad eval_2\ (*, (eval_n\ (\partial_n^n\ f, w), dw_n)) \\ \quad)) \\ ) \end{array} : F(\mathbb{R}^2) \right]\!\!\right]^{\mathbb{R}}.$$

Call the denotation of the two terms γ_1 and γ_2 respectively. By short calculation, we see that

$$\gamma_1 \times \gamma_2 \colon \left(\mathbf{Op}_n^{=} \dot{\times} {R_{\text{Dual}}}^n\right) \dot{\times} X^{R_{\text{Dual}}} \dot{\to} X.$$

By definition, $\alpha \colon X \dot{\to} R_{\text{Dual}}$, and so

$$\begin{aligned} &\left(\alpha.\left(\overline{\mathsf{ap}_n^{\mathbb{R}}} \times \overline{\mathsf{ap}_n^{\mathbb{R}^2}}\right)\right)_* \left(\left(\mathbf{Op}_n^{=} \dot{\times} {R_{\text{Dual}}}^n\right) \dot{\times} X^{R_{\text{Dual}}}\right) \\ &= (\alpha.(\gamma_1 \times \gamma_2))_* \left(\left(\mathbf{Op}_n^{=} \dot{\times} {R_{\text{Dual}}}^n\right) \dot{\times} X^{R_{\text{Dual}}}\right) \\ &\leq R_{\text{Dual}} \end{aligned}$$

meaning

$$\left(\overline{\mathsf{ap}_n^{\mathbb{R}}} \times \overline{\mathsf{ap}_n^{\mathbb{R}^2}}\right)_* \left(\left(\mathbf{Op}_n^{=} \dot{\times} {R_{\mathrm{Dual}}}^n\right) \dot{\times} X^{R_{\mathrm{Dual}}}\right) \leq \alpha^* R_{\mathrm{Dual}} = X$$

and so $\overline{\mathsf{ap}_n^{\mathbb{R}}} \times \overline{\mathsf{ap}_n^{\mathbb{R}^2}}$ respects X at R_{Dual}.

We have proven that $X \in \mathcal{R}(R_{\mathrm{Dual}})$ and so $\dot{T} R_{\mathrm{Dual}} \leq X$ giving

$$\alpha_* \dot{T} R_{\mathrm{Dual}} \leq \alpha_* X = \alpha_* \alpha^* R_{\mathrm{Dual}} \leq R_{\mathrm{Dual}}$$

which means $\alpha \colon \dot{T} R_{\mathrm{Dual}} \dot{\to} R_{\mathrm{Dual}}$ as desired. Thus, we have proven the following theorem.

Theorem 8.2.3. *Let* $\mathbf{R}; x : \mathbf{R} \vdash_{E^{\mathbf{R}}} M : F\mathbf{R}$ *be an* EFF *term not using operations from* $E^{\mathbb{R}}$ *or* $E^{\mathbb{R}^2}$ *which uses the signature* $\Pi_{\mathbf{Op}}$*. Define the following morphisms*

$$\beta_1 \coloneqq \left[\!\left[; y : \mathbb{R} \vdash_{\emptyset} \mathbf{handle}\ (\lambda x.\mathfrak{M}_{\mathbf{R}}^{\mathbb{R}}(M))\ y\ \mathbf{with}\ H_{\mathrm{eval}} : F(\mathbb{R}) \right]\!\right]^{\mathbb{R}}$$

$$\beta_2 \coloneqq \left[\!\left[; \mathfrak{y} : \mathbb{R}^2 \vdash_{\emptyset} \begin{array}{l} \mathbf{handle} \\ \quad \mathbf{handle}\ (\lambda x.\mathfrak{M}_{\mathbf{R}}^{\mathbb{R}^2}(M))\ y\ \mathbf{with}\ H_{\mathrm{forw}} \\ \mathbf{with}\ H_{\mathrm{eval}} \end{array} : F(\mathbb{R}^2) \right]\!\right]^{\mathbb{R}}$$

$$\beta \coloneqq \beta_1 \times \beta_2$$

using the Π_F*-model* $[\![-]\!]^{\mathbb{R}}$*. Then* $\beta \colon R_{\mathrm{Dual}} \dot{\to} R_{\mathrm{Dual}}$*, i.e. for any smooth* $f \colon \mathbb{R} \to \mathbb{R}$ *we have*

$$\beta \cdot (f \times \langle f, \nabla f \rangle) = g \times \langle g, \nabla g \rangle$$

for some smooth $g \colon \mathbb{R} \to \mathbb{R}$*.*

Let us apply theorem 8.2.3 for $f = \mathrm{id} \colon \mathbb{R} \to \mathbb{R}$, noting that $\nabla \mathrm{id} = \lambda x.1$. Thus, we have

$$\beta \cdot (f \times \langle f, \nabla f \rangle) = (\beta_1 \times \beta_2) \cdot (\mathrm{id} \times \langle \mathrm{id}, \lambda x.1 \rangle) = \beta_1 \times (\beta_2 \cdot \langle \mathrm{id}, \lambda x.1 \rangle)$$

and thus we see $g = \beta_1$, meaning

$$\beta_2 \cdot \langle \mathrm{id}, \lambda x.1 \rangle = \langle \beta_1, \nabla \beta_1 \rangle$$

and finally that for any $x \in \mathbb{R}$ we have

$$\beta_2(x, 1) = (\beta_1(x), \nabla \beta_1(x)) \,.$$

Recall that our helper function for forward mode is

```
d : {(Paired X) -> [Smooth X, Smooth (Paired X)] (Paired X)}
 -> X -> [Smooth X] X
d f x = dv (diff (f (paired x (<Smooth> (c 1.0)))))
```

where `diff` is our forward mode handler. Therefore, we are evaluating the function `f` at $(x, 1)$, which shows that `d` calculates the derivative of `f` when computed under the evaluation handler.

8.3 Correctness of Continuation Reverse Mode

We can now also prove correctness of continuation reverse mode, following a similar structure to the proof of forward mode. We will again use the effects $E^{\mathbf{R}}$ and $E^{\mathbb{R}}$, as well as the operational signatures $\Pi_{\mathbf{Op}}$, Π_B, and Π_F. Define the type $BP \coloneqq U_{E^{\mathbb{R}}}(\mathbb{R} \to F\mathbb{R})$, the type of back propagators. We will focus on differentiating terms of the form $\mathbf{R}; x : \mathbf{R} \vdash_{E^{\mathbf{R}}} M : F\mathbf{R}$ which use $\Pi_{\mathbf{Op}}$. As with forward mode, we will instantiate $\mathbf{R}$ based on the types we used in the implementation of continuation reverse mode in section 4.3, namely of a value paired with a backpropagator which computes the derivative. By corollary 8.1.22, the tuple $\langle \mathbf{Diff} \times \mathbf{Diff}, \{\mathbf{R}\}, \{\mathbf{R} \mapsto (\mathbb{R}, \mathbb{R} \times BP)\}\rangle$ is an EFF model. Moreover, by interpreting the base types $\mathbf{Op}_n$ and its constants using $(\mathbf{Op}_n, \mathbf{Op}_n)$, we obtain a $\Pi_{\mathbf{Op}}$-model, which we denote by $[\![-]\!]$.

We require an FFLR to apply logical relations to $[\![-]\!]$, for which we use $p\colon \mathbf{BRelDiff}^{\mathbb{R}} \to \mathbf{Diff} \times \mathbf{Diff}$. Additionally, we must choose lifts of $[\![\mathbf{Op}_n]\!] = (\mathbf{Op}_n, \mathbf{Op}_n)$ and $[\![\mathbf{R}]\!] = (\mathbb{R}, \mathbb{R} \times BP)$. Here, we write BP to mean the object $\mathbb{R} \Rightarrow T_{E^{\mathbb{R}}}\mathbb{R}$, which is the type BP where the type $\mathbb{R}$ is interpreted as its diffeological space. For $[\![\mathbf{Op}_n]\!]$, we choose equality and denote it by $\mathbf{Op}_n^{=}$. With this choice, the constants of type $\mathbf{Op}_n$ lift. For $[\![\mathbf{R}]\!]$, we choose

$$\left\{(f, (f, k)) : f \in \mathbb{R} \Rightarrow \mathbb{R}, k \in \mathbb{R} \Rightarrow (\mathbb{R} \Rightarrow T_{E^{\mathbb{R}}}\mathbb{R}), (k(x)(r) \gg\!= H_{eval}) = r\nabla f(x)\right\}$$

which we denote R_{Rev}. The relation R_{Rev} pairs a smooth function with a continuation which calculates its derivative multiplied by a scalar. We can thus apply the basic lemma of logical relations to $\mathbf{R}; x : \mathbf{R} \vdash_{E^{\mathbf{R}}} M : F\mathbf{R}$.

Proposition 8.3.1. *Given a term* $\mathbf{R}; x : \mathbf{R} \vdash_{E^{\mathbf{R}}} M : F\mathbf{R}$ *using the operational signature* $\Pi_{\mathbf{Op}}$*,* $[\![M]\!]$ *lifts to*

$$[\![M]\!] : R_{\mathrm{Rev}} \dot{\to} \dot{T}_{[\![E^{\mathbf{R}}]\!]} R_{\mathrm{Rev}}.$$

Define the effect

$$E^{\mathbb{R}\times BP} \coloneqq \left\{ \mathsf{ap}_n^{\mathbb{R}\times BP} : \mathbf{Op}_n \times (\mathbb{R}\times BP)^n \to \mathbb{R}\times BP : 0 \leq n \leq N \right\}.$$

and define the macro $\mathfrak{M}_{\mathbf{R}}^{\mathbb{R}\times BP}$ as before. Thus, we have the decomposition

$$[\![-]\!] = \left[\!\left[\mathfrak{M}_{\mathbf{R}}^{\mathbb{R}}(-)\right]\!\right]^{\mathbb{R}} \times \left[\!\left[\mathfrak{M}_{\mathbf{R}}^{\mathbb{R}\times BP}(-)\right]\!\right]^{\mathbb{R}}.$$

where $[\![-]\!]^{\mathbb{R}}$ is the $\Pi_{\mathbf{Op}}/\Pi_B/\Pi_F$-model from the previous section.

For reverse mode, we again only show the n^{th} clause for brevity, and we use an unspecified term *unzip* which unzips tuples. Furthermore, we write some effectful operations as if they were values with the understanding that they represent unwritten intermediate bindings, just as with forward mode. Finally, we use an n-ary sum in place of addition, and all said the final definition is

$$\begin{aligned}
H_{\text{rev}} = \{&\mathbf{return}\ x \qquad\qquad \mapsto \mathbf{return}\ x \\
&\mathsf{ap}_n^{\mathbb{R}\times BP}\ (f, v)\ k \mapsto \mathbf{case}\ \mathit{unzip}!\ v\ \mathbf{of} \\
&\quad (w, (dw_1, \ldots, dw_n)) \to k!\ (\mathsf{ap}_n^{\mathbb{R}}\ (f, w), \lambda z. \\
&\qquad r_1 \leftarrow dw_1!\ (\mathsf{ap}_2^{\mathbb{R}}\ (*, (\mathsf{ap}_n^{\mathbb{R}}\ (\partial_1\ f, w), z))); \\
&\qquad \ldots \\
&\qquad r_n \leftarrow dw_n!\ (\mathsf{ap}_2^{\mathbb{R}}\ (*, (\mathsf{ap}_n^{\mathbb{R}}\ (\partial_n\ f, w), z))); \\
&\qquad \mathsf{ap}_n^{\mathbb{R}}\ (\mathit{sum}, (r_1, \ldots, r_n)) \\
&\quad) \\
\}.&
\end{aligned}$$

Let $\mathbf{R}; x : \mathbf{R} \vdash_{E^{\mathbf{R}}} M : F\mathbf{R}$ such that M is in the domain of our macros. Define the following terms

$$\beta_1 \coloneqq \left[\!\left[; y : \mathbb{R} \vdash_{\emptyset} \mathbf{handle}\ (\lambda x.\mathfrak{M}_{\mathbf{R}}^{\mathbb{R}}(M))\ y\ \mathbf{with}\ H_{\text{eval}} : F(\mathbb{R})\right]\!\right]^{\mathbb{R}}$$

$$\beta_2 \coloneqq \left[\!\left[\begin{array}{ll} & \mathbf{handle} \\ ; \mathfrak{y} : \mathbb{R}\times BP \vdash_{\emptyset} & \quad \mathbf{handle}\ (\lambda x.\mathfrak{M}_{\mathbf{R}}^{\mathbb{R}\times BP}(M))\ y\ \mathbf{with}\ H_{\text{rev}} \ : F(\mathbb{R}\times BP) \\ & \mathbf{with}\ H_{\text{eval}} \end{array}\right]\!\right]^{\mathbb{R}}$$

$$\beta \coloneqq \beta_1 \times \beta_2.$$

using the Π_F-model $[\![-]\!]^{\mathbb{R}}$. We will show that β lifts along p using the definition of the free algebraic lift. Let us begin calculating β_1 in the internal language

$$\beta_1 = \lambda y.\left[\!\left[\mathfrak{M}_{\mathbf{R}}^{\mathbb{R}}(M)\right]\!\right]^{\mathbb{R}}(y) \gg\!= H_{\text{eval}}$$

and β_2

$$\beta_2 = \lambda y.\left(\left[\!\left[\mathfrak{M}_{\mathbf{R}}^{\mathbb{R}\times BP}(M)\right]\!\right]^{\mathbb{R}}(y) \gg\!= H_{\text{rev}}\right) \gg\!= H_{\text{eval}}$$

Proposition 8.3.1 showed by the basic lemma for logical relations that $[\![M]\!] = \left[\!\left[\mathfrak{M}_{\mathbf{R}}^{\mathbb{R}}(M)\right]\!\right]^{\mathbb{R}} \times \left[\!\left[\mathfrak{M}_{\mathbf{R}}^{\mathbb{R}\times BP}(M)\right]\!\right]^{\mathbb{R}}$ lifts. Thus, defining

$$\begin{aligned}\alpha_1 &\coloneqq \lambda x.x \gg\!= H_{\text{eval}}\\ \alpha_2 &\coloneqq \lambda x.(x \gg\!= H_{\text{rev}}) \gg\!= H_{\text{eval}}\\ \alpha &\coloneqq \alpha_1 \times \alpha_2\end{aligned}$$

it is sufficient to show α lifts. Note that

$$\alpha\colon\ (T_{E^{\mathbb{R}}}\mathbb{R}, T_{E^{\mathbb{R}\times BP}}(\mathbb{R}\times BP)) \to (\mathbb{R}, \mathbb{R}\times BP)$$

and so, where $\dot{T}$ is the lift of $T_{E^{\mathbb{R}}} \times T_{E^{\mathbb{R}\times BP}}$, we must prove

$$\alpha\colon \dot{T}R_{\text{Rev}} \dot{\to} R_{\text{Rev}}.$$

Define $X \coloneqq \alpha^* R_{\text{Rev}}$. Then η_T respects X at R_{Rev} as in forward mode. Next, we want to show

$$\overline{\mathsf{ap}_n^{\mathbb{R}}} \times \overline{\mathsf{ap}_n^{\mathbb{R}\times BP}}\colon\ \left(\mathbf{Op}_n^{=} \dot{\times} {R_{\text{Rev}}}^n\right) \dot{\times} X^{R_{\text{Rev}}} \dot{\to} X.$$

Define $\Gamma_1 \coloneqq f : \mathbf{Op}_n, v : \mathbb{R}^n, k : U_{E^{\mathbb{R}}}(\mathbb{R} \to F\mathbb{R})$ and $\Gamma_2 \coloneqq f : \mathbf{Op}_n, v : (\mathbb{R}\times BP)^n, k : U_{E^{\mathbb{R}\times BP}}(\mathbb{R}\times BP \to F(\mathbb{R}\times BP))$, and as before

$$\alpha_1.\overline{\mathsf{ap}_n^{\mathbb{R}}} = \left[\!\!\left[;\Gamma_1 \vdash_{\emptyset} \begin{array}{l}\textbf{handle}\\ \quad x \leftarrow \mathsf{ap}_n^{\mathbb{R}}\ (f, v);\\ \quad k!\ x\\ \textbf{with}\ H_{\text{eval}}\end{array} : F(\mathbb{R}) \right]\!\!\right]^{\mathbb{R}}$$

$$\alpha_2.\overline{\mathsf{ap}_n^{\mathbb{R}\times BP}} = \left[\!\!\left[;\Gamma_2 \vdash_{\emptyset} \begin{array}{l}\textbf{handle}\\ \quad \textbf{handle}\\ \quad\quad x \leftarrow \mathsf{ap}_n^{\mathbb{R}\times BP}\ (f, v);\\ \quad\quad k!\ x\\ \quad \textbf{with}\ H_{\text{rev}}\\ \textbf{with}\ H_{\text{eval}}\end{array} : F(\mathbb{R}\times BP) \right]\!\!\right]^{\mathbb{R}}$$

as well as

$$\alpha_1.\overline{\mathsf{ap}_n^{\mathbb{R}}} = \alpha_1.\left[\!\left[;\Gamma_1 \vdash_{E^{\mathbb{R}}} k!\ (eval_n\ (f,v)) : F(\mathbb{R})\right]\!\right]^{\mathbb{R}}$$

$$\alpha_2.\overline{\mathsf{ap}_n^{\mathbb{R}\times BP}} = \alpha_2.\left[\!\left[\begin{array}{l;\Gamma_2 \vdash_{E^{\mathbb{R}\times BP}}} \\ \end{array}\begin{array}{l}\textbf{case}\ unzip!\ v\ \textbf{of} \\ (w,(dw_1,\ldots,dw_n)) \to \\ \quad k!\ (eval_n\ (f,w), \lambda z. \\ \quad\quad r_1 \leftarrow dw_1!\ (\\ \quad\quad\quad \mathsf{ap}_2^{\mathbb{R}}\ (*,(\mathsf{ap}_n^{\mathbb{R}}\ (\partial_1\ f,w),z))); \\ \quad\quad \ldots \\ \quad\quad r_n \leftarrow dw_n!\ (\\ \quad\quad\quad \mathsf{ap}_2^{\mathbb{R}}\ (*,(\mathsf{ap}_n^{\mathbb{R}}\ (\partial_n\ f,w),z))); \\ \quad\quad \mathsf{ap}_n^{\mathbb{R}}\ (sum,(r_1,\ldots,r_n)) \\ \quad)\end{array} : F(\mathbb{R}\times BP)\right]\!\right]^{\mathbb{R}}$$

noting that there are still $E^{\mathbb{R}}$ operations inside the continuation $\lambda z.[\cdots]$ passed to k in right-hand side of the second equation. Call the denotation of the two terms γ_1 and γ_2 respectively. Said continuation $\lambda z.[\cdots]$, post-composed with $(-)\gg\!=H_{\text{eval}}$, calculates the derivative of f. By short calculation using the linearity of the continuations in their second argument, we see that

$$\gamma_1 \times \gamma_2 \colon \left(\mathbf{Op}_n^{=} \dot{\times} {R_{\text{Rev}}}^n\right) \dot{\times} X^{R_{\text{Rev}}} \dot{\to} X.$$

Thus, $\overline{\mathsf{ap}_n^{\mathbb{R}}} \times \overline{\mathsf{ap}_n^{\mathbb{R}\times BP}}$ respects X at R_{Rev} analogously to forward mode.

We have proven that $X \in \mathcal{R}(R_{\text{Rev}})$ which means $\alpha \colon \dot{T}R_{\text{Rev}} \dot{\to} R_{\text{Rev}}$ as desired. Thus, we have proven the following theorem.

Theorem 8.3.2. *Let $\mathbf{R}; x : \mathbf{R} \vdash_{E^{\mathbf{R}}} M : F\mathbf{R}$ be an EFF term not using operations from $E^{\mathbb{R}}$ or $E^{\mathbb{R}\times BP}$ which uses the signature $\Pi_{\mathbf{Op}}$. Define the following morphisms*

$$\beta_1 \coloneqq \left[\!\left[; y : \mathbb{R} \vdash_{\emptyset} \textbf{handle}\ (\lambda x.\mathfrak{M}_{\mathbf{R}}^{\mathbb{R}}(M))\ y\ \textbf{with}\ H_{\text{eval}} : F(\mathbb{R})\right]\!\right]^{\mathbb{R}}$$

$$\beta_2 \coloneqq \left[\!\left[\begin{array}{l;\mathfrak{y} : \mathbb{R}\times BP \vdash_{\emptyset}} \\ \\ \end{array}\begin{array}{l}\textbf{handle} \\ \quad \textbf{handle}\ (\lambda x.\mathfrak{M}_{\mathbf{R}}^{\mathbb{R}^2}(M))\ y\ \textbf{with}\ H_{\text{rev}} \\ \textbf{with}\ H_{\text{eval}}\end{array} : F(\mathbb{R}\times BP)\right]\!\right]^{\mathbb{R}}$$

$$\beta \coloneqq \beta_1 \times \beta_2$$

using the Π_F-model $[\![-]\!]^{\mathbb{R}}$. Then $\beta \colon R_{\text{Rev}} \dot{\to} R_{\text{Rev}}$, i.e. for any smooth $f \colon \mathbb{R} \to \mathbb{R}$ and $k_f \colon \mathbb{R} \to (\mathbb{R} \Rightarrow T_{E^{\mathbb{R}}}\mathbb{R})$ such that $\left(k_f(x)(r)\gg\!=H_{eval}\right) = r\nabla f(x)$, we have

$$\beta \cdot \left(f \times \langle f, k_f\rangle\right) = g \times \langle g, k_g\rangle$$

for some smooth $g\colon \mathbb{R} \to \mathbb{R}$ *such that* $\left(k_g(x)(r) \gg\!= H_{\text{eval}}\right) = r\nabla g(x)$*, and so* $\left(k_g(x)(1) \gg\!= H_{\text{eval}}\right) = \nabla g(x)$*.*

Let us apply theorem 8.3.2 for $f = \text{id}\colon \mathbb{R} \to \mathbb{R}$ and $k_f = \lambda x.\lambda r.\eta(r)$ where η is the unit for $T_{E^{\mathbb{R}}}$, noting that $\nabla\text{id} = \lambda x.1$ and so $\left(k_f(x)(r) \gg\!= H_{eval}\right) = \eta(r) \gg\!= H_{eval} = r = r\nabla f(x)$. Thus, we have

$$\beta \cdot \left(f \times \langle f, k_f\rangle\right) = (\beta_1 \times \beta_2) \cdot (\text{id} \times \langle \text{id}, \lambda x.\lambda r.\eta(r)\rangle) = \beta_1 \times (\beta_2 \cdot \langle \text{id}, \lambda x.\lambda r.\eta(r)\rangle)$$

and thus we see $g = \beta_1$, meaning

$$\beta_2 \cdot \langle \text{id}, \lambda x.\lambda r.\eta(r)\rangle = \langle \beta_1, k_{\beta_1}\rangle$$

where $\left(k_{\beta_1}(x)(r) \gg\!= H_{\text{eval}}\right) = r\nabla\beta_1(x)$, and finally that for any $x \in \mathbb{R}$ we have

$$\beta_2(x, \lambda r.\eta(r)) = (\beta_1(x), k)$$

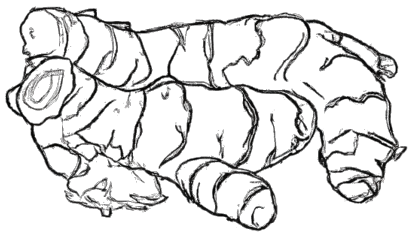
where $(k(1) \gg\!= H_{\text{eval}}) = 1 \cdot \nabla\beta_1(x) = \nabla\beta_1(x)$. Recall that our helper function for continuation reverse mode is

```
d  : {(Prop X [e|]) -> [e|Smooth X, Smooth (Prop X [e|])] (Prop X [e|])}
  -> X -> [e|Smooth X] X
d f x = let bp = dv (reverse (f (prop x {z -> z}))) in bp (c 1.0)
```

where `reverse` is our continuation reverse mode handler. Therefore, we are evaluating the function `f` at $(x, \lambda r.\eta(r))$ and returning $k(1)$, which shows that `d` calculates the derivative of `f` when computed under the evaluation handler.

Chapter 9

Conclusion



Turmeric

This thesis has shown the suitability of effects and handlers for implementing and proving the correctness of automatic differentiation algorithms.

Background

We began this thesis by providing background for our work. We covered the history, practice, and theory of AD, and saw how to derive some of the basic modes. Next, we introduced effects and handlers by example and described important dimensions of their design space. We also examined the four languages we used, namely Frank, Eff, Koka, and OCaml, and analyzed their design choices.

Implementation

The next part of this thesis focused on implementing AD modes in all of the chosen languages, divided into standard and advanced modes, with our main focus

on the Frank implementations. Frank has a formal operational semantics, which we took advantage of by showing how example programs executed. We began by defining an effect for smooth functions which was parameterized by a type variable representing real numbers which would change for each algorithm. We then defined an evaluation handler which interprets our effect. Evaluation mode instantiated the effect's type variable to floats, which allowed each smooth function to be implemented by the corresponding operation on floats. Forward mode and continuation reverse mode were the first standard AD modes we considered. We observed that executing them had the outcome of substituting each effectful operation for a corresponding operation on a generalized numeric type. Stateful reverse mode was the next mode we studied. It took advantage of the complicated control flow that effects and handlers afford, which we examined in detail when we executed an example program. We saw that by executing code after resuming the captured continuation a reverse pass was built up in a manner which reversed the original data dependency. The next standard mode we investigated was taped reverse mode, which captured the reverse pass of stateful reverse mode as a data structure, or tape. This movement of state out of the handler allowed the Koka version of the program to erase the mutable state effect by deducing that the mutability was locally contained. Furthermore, we discussed that the taped reverse mode handler was tail recursive, which Koka took advantage of to generate more efficient code during compilation. The final standard mode we explored was produced by combining previously defined modes. We investigated these combined modes through the lens of perturbation confusion, a class of AD bugs, and showed how effect type systems can help the user avoid such bugs. Finally, we verified that our system did not suffer from a variant of perturbation confusion known as higher-order perturbation confusion.

We continued our implementations by exploring more advanced modes of AD. We described variants of forward mode and taped reverse mode which directly calculated second derivatives. Next, we investigated a particularly interesting AD mode, checkpointed reverse mode, which lowers maximum memory residency at the cost of recomputation. Our implementation was able to reuse the previous definition of stateful reverse mode, and took advantage of the ability of handlers to give multiple effectful interpretations to a single term. Furthermore, the implementation closely matched the high-level description of the algorithm. We also performed an example execution to observe the behavior of the handler. Finally,

we implemented two versions of stateful reverse mode which took into consideration higher-order functions, one of which mirrored the structure of checkpointed reverse mode.

We concluded the implementation part by reporting the asymptotics of each standard AD mode and performing a real world benchmark. Each mode was applied to the same test term, and the resulting data was analyzed for linear and quadratic best fit, where a linear fit meant that the algorithm was a constant multiple slower than performing no AD. We learned that the algorithms can exhibit incorrect asymptotics, but all modes except continuation reverse mode had the correct asymptotics in at least one language. Continuation reverse mode is not used in practice, with either stateful or taped reverse mode being more efficient. We also implemented a version of stateful reverse mode extended with tensor operations and added it to a benchmark suite of practical AD tools applied to a real-world program. Our implementation was competitive with comparable tools (CPU based and define-by-run). Thus, we concluded that effects and handlers provided the correct asymptotics for AD and were competitive on real world examples.

Correctness

The correctness part of this thesis focused on building tools for applying categorical logical relations to effects and handlers. We then applied these tools to diffeological spaces and proved forward mode and continuation reverse mode correct. We began by recalling facts about initial algebras and free monads, focussing on categories $\mathcal{C}$ which were complete, cocomplete, and bi-cartesian closed. We then showed that for an endofunctor $F \colon \mathcal{C} \to \mathcal{C}$ which preserved λ-chains, the construction of maps out of its free monad $\mathcal{M}$ had an enriched version. We then introduced the languages MAM and EFF, described their operational semantics and set theoretic denotational semantics, and recalled relevant theorems. Next, we created a sound categorical denotational semantics for MAM and EFF and defined what a valid model of each was. We then noted that a complete and cocomplete bi-CCC where each functor $B \Rightarrow -$ preserved λ_B-chains for some ordinal λ_B always generated a model. We continued by introducing fibrations for logical relations, a categorical generalization of logical relations, and methods of generating such fibrations. Our linchpin theorem showed that for any FFLR

$p\colon \mathcal{C} \to \mathcal{E}$ and EFF model $\mathcal{M}$, there existed a lifted model $\dot{\mathcal{M}}$. With this theorem, we then proved the basic lemma of logical relations for both types and terms, which completed our framework.

We continued the correctness part by introducing the category **Diff** of diffeological spaces, a generalization of Euclidean spaces and smooth manifolds. We then created two FFLRs for diffeological spaces based on monomorphisms and strong monomorphisms. Next, we concluded that because **Diff** is locally presentable and cartesian closed, the functor $B \Rightarrow -$ had rank, and so **Diff** and **Diff** $\times$ **Diff** each always generated EFF models. Finally, armed with these results, we then proved the correctness of forward mode and continuation reverse mode, where we took advantage of the inductive structure of deep handlers.

Future work

There are several interesting directions in which to continue our work:

- It would be valuable to compare our effect and handler implementations to tools used by practitioners and measure absolute performance. Most effective AD systems allow matrix and tensor level basic operations, and we predict that the relative overhead of effects and handlers extended with these operations will greatly decrease due to the increased computation of each operation.

- The checkpointed reverse mode we implemented is user-driven; only the code explicitly annotated by the user is checkpointed. [Jeffrey Mark Siskind and Barak A. Pearlmutter, 2017] describe a checkpointed reverse mode which does not require user annotation. Their divide-and-conquer algorithm requires "splitting a program in half" with respect to execution cost, and then recursing on each half. Thus, their algorithm is a perfect fit for multi-shot handlers which can run the program once to split it in half with a measurement handler, and then run it again to recurse using a different handler.

- An obvious extension is to add inbuilt state to EFF and then to prove stateful reverse mode as well as the higher-order function modes. We believe FFLRs and our theorems are capable of this by moving to presheaf categories and using a local state monad.

- By working with ωCPOs or similar, we believe we could add recursion and shallow handlers. In fact, [Vákár, 2020] combines diffeological spaces with an ωCPO structure to prove forward mode correct in a language with recursion.

Appendix A

Programs

A.1 Smooth Effect and Helper Functions

Listing A.1: Smooth effect and helper functions (Eff)

```
1  type nullary = Const of float;;
2  type unary = Negate;;
3  type binary = Add | Multiply;;
4
5  type arg = L | R;;
6
7  type 'a smooth = effect
8    operation ap0 : nullary -> 'a
9    operation ap1 : unary * 'a -> 'a
10   operation ap2 : binary * 'a * 'a -> 'a
11 end;;
12
13 let op0 s n = s#ap0 n;;
14 let op1 s u x = s#ap1 (u, x);;
15 let op2 s b x y = s#ap2 (b, x, y);;
16
17 let der1 s u x =
18   let c x = s#ap0 (Const x) in
19   let ( ~- ) x = s#ap1 (Negate, x) in
20   let ( + ) x y = s#ap2 (Add, x, y) in
21   let ( * ) x y = s#ap2 (Multiply, x, y) in
22   match u with
23     | Negate -> ~- (c 1.0);;
24
25 let dder1 s u x =
26   let c x = s#ap0 (Const x) in
27   let ( ~- ) x = s#ap1 (Negate, x) in
28   let ( + ) x y = s#ap2 (Add, x, y) in
29   let ( * ) x y = s#ap2 (Multiply, x, y) in
30   match u with
31     | Negate -> c 0.0;;
```

```
32
33 let der2 s b a x y =
34   let c x = s#ap0 (Const x) in
35   let ( ~- ) x = s#ap1 (Negate, x) in
36   let ( + ) x y = s#ap2 (Add, x, y) in
37   let ( * ) x y = s#ap2 (Multiply, x, y) in
38   match b with
39     | Add -> (match a with L -> c 1.0 | R -> c 1.0)
40     | Multiply -> (match a with L -> y | R -> x);;
41
42 let dder2 s b same a1 a2 x y =
43   let c x = s#ap0 (Const x) in
44   let ( ~- ) x = s#ap1 (Negate, x) in
45   let ( + ) x y = s#ap2 (Add, x, y) in
46   let ( * ) x y = s#ap2 (Multiply, x, y) in
47   if same then
48     match b with
49       | Add -> (match a1 with L -> c 0.0 | R -> c 0.0)
50       | Multiply -> (match a1 with L -> c 2.0 | R -> c 2.0)
51   else
52    match b with
53       | Add -> (
54         match a1 with
55             L -> (match a2 with L -> c 0.0 | R -> c 0.0)
56           | R -> (match a2 with L -> c 0.0 | R -> c 0.0)
57         )
58       | Multiply -> (
59         match a1 with
60             L -> (match a2 with L -> c 0.0 | R -> c 1.0)
61           | R -> (match a2 with L -> c 1.0 | R -> c 0.0)
62         );;
```

Listing A.2: Smooth effect and helper functions (Koka)

```
1  pub module smooth
2
3  import std/num/float64
4
5  pub infixl 6 (+.)
6  pub infixl 7 (*.)
7
8  type nullary {
9      Const(x : float64)
10 }
11
12 type unary {
13     Negate
14     Sin
15     Cos
16     Exp
17 }
18
19 type binary {
```

```
    Plus
    Subtract
    Times
    Divide
}

type arg {
  L
  R
}

effect smooth<a> {
    ctl ap0(n : nullary) : a
    ctl ap1(u : unary, arg : a) : a
    ctl ap2(b : binary, arg1 : a, arg2 : a) : a
}

inline fun c(i : float64) : smooth<a> a {
    ap0(Const(i))
}

inline fun (~.)(x : a) : smooth<a> a {
    ap1(Negate, x)
}

inline fun sin_(x : a) : smooth<a> a {
    ap1(Sin, x)
}

inline fun cos_(x : a) : smooth<a> a {
    ap1(Cos, x)
}

inline fun exp_(x : a) : smooth<a> a {
    ap1(Exp, x)
}

inline fun (+.)(x : a, y : a) : smooth<a> a {
    ap2(Plus, x, y)
}

inline fun (-.)(x : a, y : a) : smooth<a> a {
    ap2(Subtract, x, y)
}

inline fun (*.)(x : a, y : a) : smooth<a> a {
    ap2(Times, x, y)
}

inline fun div_(x : a, y : a) : smooth<a> a {
    ap2(Divide, x, y)
}

```

```
73  inline fun op0(n) {
74    match(n) {
75      Const(x) -> c(x)
76    }
77  }
78
79  inline fun op1(u, x) {
80    match(u) {
81      Negate -> (~.)(x)
82      Sin -> sin_(x)
83      Cos -> cos_(x)
84      Exp -> exp_(x)
85    }
86  }
87
88  inline fun op2(b, x, y) {
89    match(b) {
90      Plus -> x +. y
91      Subtract -> x -. y
92      Times -> x *. y
93      Divide -> div_(x, y)
94    }
95  }
96
97  inline fun der1(u, x) {
98    match(u) {
99      Negate -> (~.)(c(1.0))
100     Sin -> cos_(x)
101     Cos -> (~.)(sin_(x))
102     Exp -> (~.)(c(1.0))
103   }
104 }
105
106 inline fun der2(b, a, x, y) {
107   match(b) {
108     Plus -> match(a) {L -> c(1.0); R -> c(1.0)}
109     Subtract -> match(a) {L -> c(1.0); R -> c(~1.0)}
110     Times -> match(a) {L -> y; R -> x}
111     Divide -> match(a) {L -> div_(c(1.0), y); R -> div_((~.)(x), y *. y)}
112   }
113 }
114
115 inline fun dder1(u, x) {
116   match(u) {
117     Negate -> c(0.0)
118     Sin -> (~.)(sin_(x))
119     Cos -> (~.)(cos_(x))
120     Exp -> exp_(x)
121   }
122 }
123
124 inline fun dder2(b, same, a1, a2, x, y) {
125   if same then
```

```
126     match(b) {
127       Plus -> match(a1) {L -> c(0.0); R -> c(0.0)}
128       Subtract -> match(a1) {L -> c(0.0); R -> c(0.0)}
129       Times -> match(a1) {L -> c(2.0); R -> c(2.0)}
130       Divide -> match(a1) {L -> c(0.0); R -> c(0.0)}
131     }
132   else
133     match(b) {
134       Plus -> match(a1) {
135         L ->  match(a2) {L -> c(0.0); R -> c(0.0)}
136         R ->  match(a2) {L -> c(0.0); R -> c(0.0)}
137       }
138       Subtract -> match(a1) {
139         L ->  match(a2) {L -> c(0.0); R -> c(0.0)}
140         R ->  match(a2) {L -> c(0.0); R -> c(0.0)}
141       }
142       Times -> match(a1) {
143         L ->  match(a2) {L -> c(0.0); R -> c(1.0)}
144         R ->  match(a2) {L -> c(1.0); R -> c(0.0)}
145       }
146       Divide -> match(a1) {
147         L ->  match(a2) {L -> c(0.0); R -> div_(c(~1.0), y *. y)}
148         R ->  match(a2) {
149           L -> div_(c(~1.0), y *. y)
150           R -> div_(c(2.0) *. x, y *. y *. y)
151         }
152       }
153     }
154 }
155
156 fun term(x : a, y : a) : smooth<a> a {
157     c(1.0) +. (x *. x *. x) +. ((~.)(y *. y))
158 }
```

Listing A.3: Smooth effect and helper functions (OCaml)

```
1  open Effect
2
3  type nullary = Const of float
4  type unary = Negate | Sin | Cos | Exp
5  type binary = Plus | Subtract | Times | Divide
6
7  type arg = L | R
8
9  module type SMOOTH = sig
10   type t
11   type _ Effect.t += Ap0 : nullary -> t Effect.t
12                    | Ap1 : unary * t -> t Effect.t
13                    | Ap2 : binary * t * t -> t Effect.t
14
15   val c : float -> t
16   val ( ~. ) : t -> t
17   val sin_ : t -> t
```

```
18    val cos_ : t -> t
19    val exp_ : t -> t
20    val ( +. ) : t -> t -> t
21    val ( -. ) : t -> t -> t
22    val ( *. ) : t -> t -> t
23    val ( /. ) : t -> t -> t
24
25    val op0 : nullary -> t
26    val op1 : unary -> t -> t
27    val op2 : binary -> t -> t -> t
28
29    val der1 : unary -> t -> t
30    val dder1 : unary -> t -> t
31    val der2 : binary -> arg -> t -> t -> t
32    val dder2 : binary -> bool -> arg -> arg -> t -> t -> t
33  end
34
35  module Smooth (T : sig type t end) : SMOOTH with type t = T.t = struct
36    type t = T.t
37    type _ Effect.t += Ap0 : nullary -> t Effect.t
38                     | Ap1 : unary * t -> t Effect.t
39                     | Ap2 : binary * t * t -> t Effect.t
40
41    let c x = perform (Ap0 (Const x))
42    let ( ~. ) a = perform (Ap1 (Negate, a))
43    let sin_ a = perform (Ap1 (Sin, a))
44    let cos_ a = perform (Ap1 (Cos, a))
45    let exp_ a = perform (Ap1 (Exp, a))
46    let ( +. ) a b = perform (Ap2 (Plus, a, b))
47    let ( -. ) a b = perform (Ap2 (Subtract, a, b))
48    let ( *. ) a b = perform (Ap2 (Times, a, b))
49    let ( /. ) a b = perform (Ap2 (Divide, a, b))
50
51    let op0 n = match n with
52      | Const x -> c x
53    let op1 u x = match u with
54      | Negate -> ~. x
55      | Sin -> sin_ x
56      | Cos -> cos_ x
57      | Exp -> exp_ x
58    let op2 b x y = match b with
59      | Plus -> x +. y
60      | Subtract -> x -. y
61      | Times -> x *. y
62      | Divide -> x /. y
63
64    let der1 u x = match u with
65      | Negate -> ~. (c 1.0)
66      | Sin -> cos_ x
67      | Cos -> ~. (sin_ x)
68      | Exp -> exp_ x
69    let dder1 u x = match u with
70      | Negate -> c 0.0
```

```
71      | Sin -> ~. (sin_ x)
72      | Cos -> ~. (cos_ x)
73      | Exp -> exp_ x
74    let der2 b a x y = match b with
75      | Plus -> (match a with L -> c 1.0 | R -> c 1.0)
76      | Subtract -> (match a with L -> c 1.0 | R -> c (-1.0))
77      | Times -> (match a with L -> y | R -> x)
78      | Divide -> (match a with L -> (c 1.0) /. y | R -> (~. x) /. (y *. y))
79    let dder2 b same a1 a2 x y =
80      if same then
81        match b with
82          | Plus -> (match a1 with L -> c 0.0 | R -> c 0.0)
83          | Subtract -> (match a1 with L -> c 0.0 | R -> c 0.0)
84          | Times -> (match a1 with L -> c 2.0 | R -> c 2.0)
85          | Divide -> (match a1 with L -> c 0.0 | R -> c 0.0)
86      else
87        match b with
88          | Plus -> (match a1 with
89            | L ->  (match a2 with L -> c 0.0 | R -> c 0.0)
90            | R ->  (match a2 with L -> c 0.0 | R -> c 0.0)
91            )
92          | Subtract -> (match a1 with
93            | L ->  (match a2 with L -> c 0.0 | R -> c 0.0)
94            | R ->  (match a2 with L -> c 0.0 | R -> c 0.0)
95            )
96          | Times -> (match a1 with
97            | L ->  (match a2 with L -> c 0.0 | R -> c 1.0)
98            | R ->  (match a2 with L -> c 1.0 | R -> c 0.0)
99            )
100         | Divide -> (match a1 with
101           | L ->  (match a2 with L -> c 0.0 | R -> (c (-1.0)) /. (y *. y))
102           | R ->  (match a2 with
103             | L -> (c (-1.0)) /. (y *. y)
104             | R -> (c 2.0 *. x) /. (y *. y *. y)
105             )
106           )
107 end
```

A.2 Evaluation

Listing A.4: Evaluation (Eff)

```
1  let evaluate s = handler
2    | s#ap0 n k -> (match n with Const x -> k x)
3    | s#ap1 (u, x) k -> (match u with Negate -> k (~-. x))
4    | s#ap2 (b, x, y) k ->
5      (match b with
6         | Add -> k (x +. y)
7         | Multiply -> k (x *. y))
8    | val x -> x;;
```

Listing A.5: Evaluation (Koka)

```
pub module evaluate

import std/num/float64
import smooth

val evaluate = handler {
  ctl ap0(n) -> match(n) {Const(i) -> resume(i)}
  ctl ap1(u,x) -> match(u) {
    Negate -> resume(~x : float64)
    Sin -> resume(sin(x) : float64)
    Cos -> resume(cos(x) : float64)
    Exp -> resume(exp(x) : float64)
  }
  ctl ap2(b,x,y) -> match(b) {
    Plus -> resume(x + y : float64)
    Subtract -> resume(x - y : float64)
    Times -> resume(x * y : float64)
    Divide -> resume(x / y : float64)
  }
}
```

Listing A.6: Evaluation (OCaml)

```
open Effect.Deep
open Float
open Smooth

module Evaluate = struct
  include Smooth (struct type t = float end)

  let evaluate = {
    retc = (fun x -> x);
    exnc = raise;
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
      | Ap0 n -> Some (fun (k : (a, _) continuation) ->
          match n with
          | Const x -> continue k x
        )
      | Ap1 (u, x) -> Some (fun k ->
          match u with
          | Negate -> continue k (neg x)
          | Sin -> continue k (sin x)
          | Cos -> continue k (cos x)
          | Exp -> continue k (exp x)
        )
      | Ap2 (b, x, y) -> Some (fun k ->
          match b with
          | Plus -> continue k (add x y)
          | Subtract -> continue k (sub x y)
          | Times -> continue k (mul x y)
          | Divide -> continue k (div x y)
```

```
30          )
31        | _ -> None
32      )
33    }
34  end
```

A.3 Forward Mode

Listing A.7: Forward mode (Eff)

```
1  type 'a dual = Dual of 'a * 'a;;
2
3  let forward i o =
4    let c x = o#ap0 (Const x) in
5    let ( ~- ) x = o#ap1 (Negate, x) in
6    let ( + ) x y = o#ap2 (Add, x, y) in
7    let ( * ) x y = o#ap2 (Multiply, x, y) in
8    handler
9      | i#ap0 n k ->
10       k (Dual (op0 o n, c 0.0))
11     | i#ap1 (u, Dual (x, dx)) k ->
12       k (Dual (op1 o u x, der1 o u x * dx))
13     | i#ap2 (b, Dual (x, dx), Dual (y, dy)) k ->
14       k (Dual (op2 o b x y, (der2 o b L x y * dx) + (der2 o b R x y * dy)))
15     | val x -> x;;
16
17 let diff i o f x =
18   let Dual (r, dr) = with forward i o handle
19     f i (Dual (x, op0 o (Const 1.0))) in
20   dr;;
```

Listing A.8: Forward mode (Koka)

```
1  pub module forward
2
3  import smooth
4
5  value type dual<a> {
6    Dual(v : a, dv : a)
7  }
8
9  val diff = handler {
10   ctl ap0(n) ->
11     resume(Dual(op0(n), c(0.0)))
12   ctl ap1(u,x) ->
13     resume(Dual(op1(u,x.v), der1(u,x.v) *. x.dv))
14   ctl ap2(b,x,y) ->
15     resume(Dual(op2(b,x.v,y.v), (der2(b,L,x.v,y.v) *. x.dv) +.
16                                 (der2(b,R,x.v,y.v) *. y.dv)))
17 }
18
19 fun d(
```

```
20      f : dual<a> -> <smooth<dual<a>>,smooth<a>|e> dual<a>,
21      x : a
22    ) : <smooth<a>|e> a {
23    val res = diff{f(Dual(x,mask<smooth>{c(1.0)}))}
24    res.dv
25  }
26
27  fun lift(x : a) : <smooth<dual<a>>,smooth<a>> dual<a> {
28    mask<smooth>{Dual(x, c(0.0))}
29  }
```

Listing A.9: Forward mode (OCaml)

```
1   open Effect.Deep
2   open Smooth
3
4   type 't dual = {v : 't; dv : 't}
5
6   module Forward (T : SMOOTH) = struct
7     include Smooth (struct type t = T.t dual end)
8
9     let forward = {
10      retc = (fun x -> x);
11      exnc = raise;
12      effc = (fun (type a) (eff : a Effect.t) ->
13        match eff with
14        | Ap0 n -> Some (fun (k : (a, _) continuation) -> let open T in
15            continue k {v = op0 n; dv = c 0.0}
16          )
17        | Ap1 (u, x) -> Some (fun k -> let open T in
18            continue k {v = op1 u x.v; dv = der1 u x.v *. x.dv}
19          )
20        | Ap2 (b, x, y) -> Some (fun k -> let open T in
21            continue k {v = op2 b x.v y.v; dv = (der2 b L x.v y.v *. x.dv) +.
22                                               (der2 b R x.v y.v *. y.dv)}
23          )
24        | _ -> None
25      )
26    }
27
28    let diff f x =
29      let res = match_with f {v = x; dv = T.c 1.0} forward in res.dv
30  end
```

A.4 Continuation Reverse Mode

Listing A.10: Continuation reverse mode (Eff)

```
1   type 'a prop = Prop of 'a * ('a -> 'a);;
2
3   let reverse i o =
4     let c x = o#ap0 (Const x) in
```

```
  let ( ~- ) x = o#ap1 (Negate, x) in
  let ( + ) x y = o#ap2 (Add, x, y) in
  let ( * ) x y = o#ap2 (Multiply, x, y) in
  handler
    | i#ap0 n k ->
      k (Prop (op0 o n, (fun z -> c 0.0)))
    | i#ap1 (u, Prop (x, dx)) k ->
      k (Prop (op1 o u x, (fun z -> dx (der1 o u x * z))))
    | i#ap2 (b, Prop (x, dx), Prop (y, dy)) k ->
      k (Prop (op2 o b x y, (fun z ->
        (dx (der2 o b L x y * z)) + (dy (der2 o b R x y * z))))
      )
    | val x -> x;;

let d i o f x =
  let Prop (r, dr) = with reverse i o handle
    f i (Prop (x, (fun z -> z))) in
  dr (op0 o (Const 1.0));;
```

Listing A.11: Continuation reverse mode (Koka)

```
pub module reverse-continuation

import smooth

value type prop<e,a> {
    Prop(v : a, dv : a -> <smooth<a>|e> a)
}

val reverse = handler {
  ctl ap0(n) -> {
    val r = Prop(op0(n), fn(z) {c(0.0)})
    resume(r)
  }
  ctl ap1(u,x) -> {
    val r = Prop(op1(u,x.v), fn(z) {
      (x.dv)(der1(u,x.v) *. z)
    })
    resume(r)
  }
  ctl ap2(b,x,y) -> {
    val r = Prop(op2(b,x.v,y.v), fn(z) {
      val xv = (x.dv)(der2(b,L,x.v,y.v) *. z)
      val yv = (y.dv)(der2(b,R,x.v,y.v) *. z)
      xv +. yv
    })
    resume(r)
  }
}

fun grad(f, x) {
  val xp = Prop(x, fn(z) {z})
  val r = reverse{f(xp)}
```

```
33   (r.dv)(c(1.0))
34 }
```

Listing A.12: Continuation reverse mode (OCaml)

```
1  open Effect.Deep
2  open Smooth
3
4  type 't prop = {v : 't; dv : 't -> 't}
5
6  module Reverse_continuation (T : SMOOTH) = struct
7    include Smooth (struct type t = T.t prop end)
8
9    let reverse = {
10     retc = (fun x -> x);
11     exnc = raise;
12     effc = (fun (type a) (eff : a Effect.t) ->
13       match eff with
14       | Ap0 n -> Some (fun (k : (a, _) continuation) -> let open T in
15           continue k {v = op0 n; dv = fun _ -> c 0.0}
16         )
17       | Ap1 (u, x) -> Some (fun k -> let open T in
18           continue k {v = op1 u x.v; dv = fun z -> x.dv (der1 u x.v *. z)}
19         )
20       | Ap2 (b, x, y) -> Some (fun k -> let open T in
21           continue k {v = op2 b x.v y.v;
22                       dv = fun z -> x.dv (der2 b L x.v y.v *. z) +.
23                                     y.dv (der2 b R x.v y.v *. z)}
24         )
25       | _ -> None
26     )
27   }
28
29   let d f x =
30     let res = match_with f {v = x; dv = fun z -> z} reverse in res.dv (T.c 1.0)
31 end
```

A.5 Stateful Reverse Mode

Listing A.13: Stateful reverse mode (Eff)

```
1  type 'a prop = Prop of 'a * 'a ref
2
3  let reverse i o =
4    let c x = o#ap0 (Const x) in
5    let ( ~- ) x = o#ap1 (Negate, x) in
6    let ( + ) x y = o#ap2 (Add, x, y) in
7    let ( * ) x y = o#ap2 (Multiply, x, y) in
8    handler
9      | i#ap0 n k ->
10       let r = Prop (op0 o n, ref (c 0.0)) in
11       k r
```

```
12      | i#ap1 (u, Prop (x, dx)) k ->
13        let dr = ref (c 0.0) in
14        let r = Prop (op1 o u x, dr) in
15        ignore (k r);
16        dx := !dx + (der1 o u x * !dr)
17      | i#ap2 (b, Prop (x, dx), Prop (y, dy)) k ->
18        let dr = ref (c 0.0) in
19        let r = Prop (op2 o b x y, dr) in
20        ignore (k r);
21        dx := !dx + (der2 o b L x y * !dr);
22        dy := !dy + (der2 o b R x y * !dr)
23      | val x -> x;;
24
25  let grad i o f x =
26    let dz = ref (op0 o (Const 0.0)) in
27    let z = Prop (x, dz) in
28    ( with reverse i o handle
29        let Prop (r, dr) = f i z in
30        dr := op0 o (Const 1.0)
31    );
32    !dz;;
```

Listing A.14: Stateful reverse mode (Koka)

```
1   pub module reverse
2
3   import smooth
4
5   value type prop<h,a> {
6       Prop(v : a, dv : ref<h, a>)
7   }
8
9   val reverse = handler {
10    ctl ap0(n) -> {
11      val r = Prop(op0(n), ref(c(0.0)))
12      resume(r)
13    }
14    ctl ap1(u,x) -> {
15      val r = Prop(op1(u,x.v), ref(c(0.0)))
16      resume(r)
17      set(x.dv, !x.dv +. (der1(u,x.v) *. !r.dv))
18    }
19    ctl ap2(b,x,y) -> {
20      val r = Prop(op2(b,x.v,y.v), ref(c(0.0)))
21      resume(r)
22      set(x.dv, !x.dv +. (der2(b,L,x.v,y.v) *. !r.dv))
23      set(y.dv, !y.dv +. (der2(b,R,x.v,y.v) *. !r.dv))
24    }
25  }
26
27  fun grad(f, x) {
28    val z = Prop(x, ref(c(0.0)))
29    reverse{set(f(z).dv, mask<smooth>{c(1.0)})}
```

```
30   !z.dv
31 }
```

Listing A.15: Stateful reverse mode (OCaml)

```
1  open Effect.Deep
2  open Smooth
3
4  type 't prop = {v : 't; mutable dv : 't}
5
6  module Reverse (T : SMOOTH) = struct
7    include Smooth (struct type t = T.t prop end)
8
9    let reverse = {
10     retc = (fun x -> x);
11     exnc = raise;
12     effc = (fun (type a) (eff : a Effect.t) ->
13       match eff with
14       | Ap0 n -> Some (fun (k : (a, _) continuation) -> let open T in
15           continue k {v = op0 n; dv = c 0.0}
16         )
17       | Ap1 (u, x) -> Some (fun k -> let open T in
18           let r = {v = op1 u x.v; dv = c 0.0} in
19           continue k r;
20           x.dv <- x.dv +. (der1 u x.v *. r.dv)
21         )
22       | Ap2 (b, x, y) -> Some (fun k -> (let open T in
23           let r = {v = op2 b x.v y.v; dv = c 0.0} in
24           continue k r;
25           x.dv <- x.dv +. (der2 b L x.v y.v *. r.dv);
26           y.dv <- y.dv +. (der2 b R x.v y.v *. r.dv)
27         ))
28       | _ -> None
29     )
30   }
31
32   let grad f x =
33     let r = {v = x; dv = T.c 0.0} in
34     match_with (fun x -> (f x).dv <- T.c 1.0) r reverse;
35     r.dv
36 end
```

A.6 Taped Reverse Mode

Listing A.16: Taped reverse mode (Eff)

```
1  type name = Name of int
2
3  let get_value (Name i) = i
4
5  type fresh = effect
6    operation fresh : unit -> name
```

```
7  end;;
8
9  let increment_name t init = handler
10   | t#fresh () k -> (fun i -> k (Name i) (i + 1))
11   | val x -> (fun i -> (i, x))
12   | finally f -> f init;;
13
14 type 'a prop = Prop of 'a * (name option)
15
16 type 'a pointer = Single of name * 'a
17                  | Double of name * name * 'a * 'a
18
19 let reverse f i o =
20   let c x = o#ap0 (Const x) in
21   let ( ~- ) x = o#ap1 (Negate, x) in
22   let ( + ) x y = o#ap2 (Add, x, y) in
23   let ( * ) x y = o#ap2 (Multiply, x, y) in
24   handler
25     | i#ap0 n k -> (fun tape ->
26       let r = Prop (op0 o n, None) in
27       k r tape
28       )
29     | i#ap1 (u, Prop (x, dx)) k -> (fun tape ->
30       let res = op1 o u x in
31       let tr = match dx with
32         | None ->
33           (tape, Prop (res, None))
34         | Some nx ->
35           (Single (nx, (der1 o u x)) :: tape, Prop (res,  Some (f#fresh ())))
36       in
37       k (snd tr) (fst tr)
38       )
39     | i#ap2 (b, Prop (x, dx), Prop (y, dy)) k -> (fun tape ->
40       let res = op2 o b x y in
41       let tr = match (dx, dy) with
42         | (None, None) ->
43           (tape, Prop (res, None))
44         | (Some nx, None) ->
45           (Single (nx, der2 o b L x y) :: tape, Prop (res, Some (f#fresh ())))
46         | (None, Some ny) ->
47           (Single (ny, der2 o b R x y) :: tape, Prop (res, Some (f#fresh ())))
48         | (Some nx, Some ny) ->
49           ( Double (nx, ny, der2 o b L x y, der2 o b R x y) :: tape,
50             Prop (res, Some (f#fresh ()))
51           )
52       in
53       k (snd tr) (fst tr)
54       )
55     | val x -> (fun tape -> (tape, x))
56     | finally f -> f [];;
57
58 let rec init_state s = function
59   | 0 -> []
```

```
60    | n -> (ref (op0 s (Const 0.0))) :: init_state s (n-1)
61    | _ -> raise invalidArgument "init_state: need positive length";;
62
63  let rec foreach_indexed i xs f = match xs with
64    | [] -> ()
65    | x :: xs -> ignore (f i x); foreach_indexed (i+1) xs f;;
66
67  let rec nth i xs = match (i, xs) with
68    | (0, x :: xs) -> x
69    | (n, x :: xs) -> nth (n-1) xs
70    | _ -> raise invalidArgument "nth: list too short";;
71
72  let d t i o f x =
73    let res =
74      with increment_name t 0 handle
75      let z = Prop (x, Some (t#fresh ())) in
76      with reverse t i o handle (f i z) in
77    let m = fst res in
78    let tape = fst (snd res) in
79    let state = init_state o m in
80    (nth (m - 1) state) := (op0 o (Const 1.0));
81    foreach_indexed 0 tape (fun k p ->
82      let ( +. ) x y = o#ap2 (Add, x, y) in
83      let ( *. ) x y = o#ap2 (Multiply, x, y) in
84      match p with
85        | Single (nu, vu) ->
86          let dk = !(nth (m - (k + 1)) state) in
87          let du = !(nth (get_value nu) state) in
88          (nth (get_value nu) state) := (du +. (vu *. dk))
89        | Double (nl, nr, vl, vr) ->
90          let dk = !(nth (m - (k + 1)) state) in
91          let dl = !(nth (get_value nl) state) in
92          (nth (get_value nl) state) := (dl +. (vl *. dk));
93          let dr = !(nth (get_value nr) state) in
94          (nth (get_value nr) state) := (dr +. (vr *. dk))
95    );
96    !(nth 0 state);;
```

Listing A.17: Taped reverse mode (Koka)

```
1   pub module reverse-tape
2
3   import smooth-tape
4
5   struct name{get-value : int}
6
7   fun show(n : name) : string {
8     "n(" ++ show(n.get-value) ++ ")"
9   }
10
11  effect fresh
12    fun fresh() : name
13
```

```
14 fun increment-name(init, action) {
15   var i := init
16   with handler
17     return(x) -> (i, x)
18     fun fresh() -> {val t = i; i := i + 1; Name(t)}
19   action()
20 }
21
22 value type prop<a> {
23   Prop(v : a, dv : maybe<name>)
24 }
25
26 type pointer<a> {
27   Single(nu : name, vu : a)
28   Double(nl : name, nr : name, vl : a, vr : a)
29 }
30
31 fun reverse(action) {
32   var tape := []
33   with handler
34     return(x) -> (tape, x)
35     fun ap0(n) -> Prop(op0(n), Nothing)
36     fun ap1(u,x) ->
37       val res = op1(u,x.v)
38       match(x.dv)
39         Nothing -> Prop(res, Nothing)
40         Just(nx) ->
41           tape := Cons(Single(nx,der1(u,x.v)), tape)
42           Prop(res, Just(fresh()))
43     fun ap2(b,xy) ->
44       val x = xy.fst
45       val y = xy.snd
46       val res = op2(b,x.v,y.v)
47       with unsafe-no-exn
48       match ((x.dv,y.dv))
49         (Nothing, Nothing) -> Prop(res, Nothing)
50         (Just(nx), Nothing) ->
51           tape := Cons(Single(nx,der2(b,L,x.v,y.v)), tape)
52           Prop(res, Just(fresh()))
53         (Nothing, Just(ny)) ->
54           tape := Cons(Single(ny,der2(b,R,x.v,y.v)), tape)
55           Prop(res, Just(fresh()))
56         (Just(nx), Just(ny)) ->
57           tape := Cons(Double(nx,ny,der2(b,L,x.v,y.v),der2(b,R,x.v,y.v)), tape)
58           Prop(res, Just(fresh()))
59   action()
60 }
61
62 fun grad(
63     f : prop<a> -> <div,smooth<prop<a>>,smooth<a>|e> prop<a>,
64     x : a
65   ) : <exn,div,smooth<a>|e> a {
66   val (m, (tape, _)) =
```

```
67      with mask<exn>
68      with mask<local>
69      with increment-name(0)
70      with reverse
71      val z = Prop(x, Just(fresh()))
72      with mask<fresh>
73      f(z)
74    var state := vector(list(1,m,fn(_) {c(0.0)}))
75    state[m - 1] := c(1.0)
76    tape.foreach-indexed fn(k, p)
77      match(p)
78        Single(nu, vu) ->
79          val dk = state[m - (k + 1)]
80          val du = state[nu.get-value]
81          state[nu.get-value] := (du +. (vu *. dk))
82        Double(nl, nr, vl, vr) ->
83          val dk = state[m - (k + 1)]
84          val dl = state[nl.get-value]
85          state[nl.get-value] := (dl +. (vl *. dk))
86          val dr = state[nr.get-value]
87          state[nr.get-value] := (dr +. (vr *. dk))
88    state[0]
89  }
```

Listing A.18: Taped reverse mode (OCaml)

```
1  open Effect.Deep
2  open Effect
3  open Smooth
4  open Array
5
6  type name = {get_value : int}
7
8  module type FRESH = sig
9    type _ Effect.t += Fresh : unit -> name Effect.t
10
11   val fresh : unit -> name
12 end
13
14 module Fresh : FRESH = struct
15   type _ Effect.t += Fresh : unit -> name Effect.t
16
17   let fresh () = perform (Fresh ())
18 end
19
20 type 't prop = {v : 't; dv : name option}
21
22 type 't pointer
23   = Single of name * 't
24   | Double of name * name * 't * 't
25
26 module Reverse_tape (T : SMOOTH) (F : FRESH) = struct
27   include Smooth (struct type t = T.t prop end)
```

```
28  include Fresh
29
30  let increment_name init =
31    let i = ref init in {
32      retc = (fun x -> (!i, x));
33      exnc = raise;
34      effc = (fun (type a) (eff : a Effect.t) ->
35        match eff with
36        | Fresh () -> Some (fun (k : (a, _) continuation) ->
37            let t = !i in
38            i := !i + 1;
39            continue k {get_value = t}
40          )
41        | _ -> None
42      )
43  }
44  let reverse () =
45    let tape : T.t pointer list ref = ref [] in {
46      retc = (fun x -> (!tape, x));
47      exnc = raise;
48      effc = (fun (type a) (eff : a Effect.t) ->
49        match eff with
50        | Ap0 n -> Some (fun (k : (a, _) continuation) ->
51            let open T in
52            continue k {v = op0 n; dv = None}
53          )
54        | Ap1 (u, x) -> Some (fun k ->
55            let open T in
56            let open F in
57            let res = op1 u x.v in
58            match x.dv with
59            | None -> continue k {v = res; dv = None}
60            | Some nx ->
61              tape := Single (nx, der1 u x.v) :: (!tape);
62              continue k {v = res; dv = Some (fresh ())}
63          )
64        | Ap2 (b, x, y) -> Some (fun k ->
65            let open T in
66            let open F in
67            let res = op2 b x.v y.v in
68            match (x.dv, y.dv) with
69            | (None, None) -> continue k {v = res; dv = None}
70            | (Some nx, None) ->
71              tape := Single (nx, der2 b L x.v y.v) :: (!tape);
72              continue k {v = res; dv = Some (fresh ())}
73            | (None, Some ny) ->
74              tape := Single (ny, der2 b R x.v y.v) :: (!tape);
75              continue k {v = res; dv = Some (fresh ())}
76            | (Some nx, Some ny) ->
77              tape :=
78                Double (nx, ny, der2 b L x.v y.v, der2 b R x.v y.v) :: (!tape);
79              continue k {v = res; dv = Some (fresh ())}
80          )
```

```
81          | _ -> None
82        )
83    }
84
85    let d f x =
86      let (m, (tape, _)) =
87        match_with (fun () ->
88          match_with f {v = x; dv = Some (F.fresh ())} (reverse ())
89        ) () (increment_name 0)
90      in
91      let state = init m (fun _ -> T.c 0.0) in
92      state.(m - 1) <- T.c 1.0;
93      List.iteri (fun k p ->
94        let open T in
95        match p with
96        | Single (nu, vu) ->
97          let dk = state.(m - (k + 1)) in
98          let du = state.(nu.get_value) in
99          state.(nu.get_value) <- (du +. (vu *. dk))
100       | Double (nl, nr, vl, vr) ->
101         let dk = state.(m - (k + 1)) in
102         let dl = state.(nl.get_value) in
103         state.(nl.get_value) <- (dl +. (vl *. dk));
104         let dr = state.(nr.get_value) in
105         state.(nr.get_value) <- (dr +. (vr *. dk))
106     ) tape;
107     state.(0)
108 end
```

A.7 Higher Derivatives

Listing A.19: Second derivative forward mode (Eff)

```
1  type 'a triple = Triple of 'a * 'a * 'a;;
2
3  let forward i o =
4    let c x = o#ap0 (Const x) in
5    let ( ~- ) x = o#ap1 (Negate, x) in
6    let ( + ) x y = o#ap2 (Add, x, y) in
7    let ( * ) x y = o#ap2 (Multiply, x, y) in
8    handler
9      | i#ap0 n k ->
10       k (Triple (op0 o n, c 0.0, c 0.0))
11     | i#ap1 (u, Triple (x, dx, ddx)) k ->
12       k (Triple (op1 o u x, der1 o u x * dx, dder1 o u x * ddx))
13     | i#ap2 (b, Triple (x, dx, ddx), Triple (y, dy, ddy)) k ->
14       k (Triple (
15            op2 o b x y,
16            (der2 o b L x y * dx) + (der2 o b R x y * dy),
17            ((dder2 o b false L R x y * (dx * dy)) +
18             (((c 0.5 * dder2 o b false L L x y) * (dx * dx)) +
19              ((c 0.5 * dder2 o b false R R x y) * (dy * dy)))) +
```

```
20            ((der2 o b L x y * ddx) + (der2 o b R x y * ddy)))
21       )
22     | val x -> x;;
23
24 let diff i o f x =
25   let c x = o#ap0 (Const x) in
26   let ( ~- ) x = o#ap1 (Negate, x) in
27   let ( + ) x y = o#ap2 (Add, x, y) in
28   let ( * ) x y = o#ap2 (Multiply, x, y) in
29   let Triple (r, dr, ddr) = with forward i o handle
30     f i (Triple (x, c 1.0, c 0.0)) in
31   c 2.0 * ddr;;
```

Listing A.20: Second derivative forward mode (Koka)

```
1  pub module forward-second
2
3  import smooth
4
5  type triple<a> {
6    Triple(v : a, dv : a, ddv : a)
7  }
8
9  val diff = handler {
10   ctl ap0(n) ->
11     resume(Triple(
12               op0(n),
13               c(0.0),
14               c(0.0)
15              )
16            )
17   ctl ap1(u,x) ->
18     resume(Triple(
19               op1(u,x.v),
20               der1(u,x.v) *. x.dv,
21               dder1(u,x.v) *. x.ddv
22              )
23            )
24   ctl ap2(b,x,y) ->
25     resume(Triple(
26               op2(b,x.v,y.v),
27               (der2(b,L,x.v,y.v) *. x.dv) +.
28               (der2(b,R,x.v,y.v) *. y.dv),
29               ((dder2(b,False,L,R,x.v,y.v) *. (x.dv *. y.dv)) +.
30                (((c(0.5) *. dder2(b,False,L,L,x.v,y.v)) *. (x.dv *. x.dv)) +.
31                 ((c(0.5) *. dder2(b,False,R,R,x.v,y.v)) *. (y.dv *. y.dv)))) +.
32               ((der2(b,L,x.v,y.v) *. x.ddv) +. (der2(b,R,x.v,y.v) *. y.ddv))
33              )
34            )
35 }
36
37 fun d(
38     f : triple<a> -> <smooth<triple<a>>,smooth<a>|e> triple<a>,
```

```
39      x : a
40    ) : <smooth<a>|e> a {
41    val res = diff{f(Triple(x,mask<smooth>{c(1.0)},mask<smooth>{c(0.0)}))}
42    c(2.0) *. res.ddv
43  }
44
45  fun lift(x : a) : <smooth<triple<a>>,smooth<a>> triple<a> {
46    mask<smooth>{Triple(x, c(0.0), c(0.0))}
47  }
```

Listing A.21: Second derivative forward mode (OCaml)

```
1   open Effect.Deep
2   open Smooth
3
4   type 't triple = {v : 't; dv : 't; ddv : 't}
5
6   module Forward_second (T : SMOOTH) = struct
7     include Smooth (struct type t = T.t triple end)
8
9     let forward = {
10      retc = (fun x -> x);
11      exnc = raise;
12      effc = (fun (type a) (eff : a Effect.t) ->
13        match eff with
14        | Ap0 n -> Some (fun (k : (a, _) continuation) -> let open T in
15            continue k {v = op0 n; dv = c 0.0; ddv = c 0.0}
16          )
17        | Ap1 (u, x) -> Some (fun k -> let open T in
18            continue k {
19              v = op1 u x.v;
20              dv = der1 u x.v *. x.dv;
21              ddv = dder1 u x.v *. x.ddv
22            }
23          )
24        | Ap2 (b, x, y) -> Some (fun k -> let open T in
25            continue k {
26              v = op2 b x.v y.v;
27              dv = (der2 b L x.v y.v *. x.dv) +.
28                   (der2 b R x.v y.v *. y.dv);
29              ddv = ((dder2 b false L R x.v y.v *. (x.dv *. y.dv)) +.
30                     (((c 0.5 *. dder2 b false L L x.v y.v) *. (x.dv *. x.dv)) +.
31                      ((c 0.5 *. dder2 b false R R x.v y.v) *. (y.dv *. y.dv))))
32                    +.
33                    ((der2 b L x.v y.v *. x.ddv) +. (der2 b R x.v y.v *. y.ddv))
34            }
35          )
36        | _ -> None
37      )
38    }
39
40    let diff f x =
41      let res = match_with f {v = x; dv = T.c 1.0; ddv = T.c 0.0} forward in
```

```
    T.( *. ) (T.c 2.0) res.ddv
end
```

Listing A.22: Hessian reverse mode (Frank)

```
include prelude
include map
include set
include smooth

data Name = name Int

getValue : Name -> Int
getValue (name i) = i

interface Fresh = fresh : Name

incrementName : Int -> <Fresh> X -> X
incrementName _ x = x
incrementName i <fresh -> k> = incrementName (i + 1) (k (name i))

data Tagged X = tagged X Name

v : Tagged X -> X
v (tagged x _) = x

tag : Tagged X -> Name
tag (tagged _ t) = t

data NamePair = namePair Name Name

makeNamePair : Name -> Name -> NamePair
makeNamePair n m = if (getValue n < getValue m) {namePair n m} {namePair m n}

ltEq : Int -> Int -> Bool
ltEq x y = or (x < y) (eqInt x y)

allNamePairs : Set Name -> List (Pair Name Name)
allNamePairs s =
    let xs = setToList s in
    filter {(pair x y) -> ltEq (getValue x) (getValue y)}
        (concat (map {x -> map {y -> pair x y} xs} xs))

eqName : Name -> Name -> Bool
eqName (name x) (name y) = eqInt x y

eqNamePair : NamePair -> NamePair -> Bool
eqNamePair (namePair n m) (namePair n' m') = and (eqName n n') (eqName m m')

concatMaybe : List (Maybe X) -> List X
concatMaybe [] = []
concatMaybe (nothing :: xs) = concatMaybe xs
concatMaybe ((just x) :: xs) = x :: concatMaybe xs
```

```
49
50  maybe : X -> Maybe X -> X
51  maybe x nothing = x
52  maybe _ (just v) = v
53
54  fmap : {X -> Y} -> Maybe X -> Maybe Y
55  fmap _ nothing = nothing
56  fmap f (just x) = just (f x)
57
58  foreach : List X -> {X -> Unit} -> Unit
59  foreach [] _ = unit
60  foreach (x :: xs) f = f x; foreach xs f
61
62  saveAndUpdate : Ref (Set Name [Smooth X, RefState])
63                -> Ref (Map Name X [Smooth X, RefState])
64                -> Ref (Map NamePair X [Smooth X, RefState])
65                -> Tagged X
66                -> List (Tagged X)
67                -> [Smooth X, RefState] Pair X (Map Name X [Smooth X, RefState])
68  saveAndUpdate s a h res deps =
69      let w = maybe (c 0.0) (mapLookup (tag res) @a) in
70      a := mapUpdate (pair (tag res) (c 0.0)) @a;
71      let r = makeMap eqName (
72          concatMaybe (map
73              {v ->
74                  let temp = mapLookup (makeNamePair v (tag res)) @h in
75                  h := mapUpdate (pair (makeNamePair v (tag res)) (c 0.0)) @h;
76                  fmap {z -> pair v z} temp
77              } (setToList @s))
78      ) in
79      s := setRemove (tag res) @s;
80      foreach deps {dep -> s := setAdd (tag dep) @s};
81      a := mapRestrict (setToList @s) @a;
82      h := mapRestrict (
83          let ss = setToList @s in
84          concat (map {n -> map {m -> makeNamePair n m} ss} ss)
85      ) @h;
86      pair w r
87
88  reverseHessian : Ref (Set Name [Smooth X, Fresh, RefState])
89                 -> Ref (Map Name X [Smooth X, Fresh, RefState])
90                 -> Ref (Map NamePair X [Smooth X, Fresh, RefState])
91                 -> <Smooth (Tagged X)> Unit
92                 -> [Smooth X, Fresh, RefState] Unit
93  reverseHessian _ _ _ x = x
94  reverseHessian s a h <ap0 n -> k> =
95      reverseHessian s a h (k (tagged (<Smooth> (ap0 n)) fresh!))
96  reverseHessian s a h <ap1 u (tagged x dx) -> k> =
97      let res = tagged (ap1 u x) fresh! in
98      reverseHessian s a h (k res);
99      let wr = saveAndUpdate s a h res [tagged x dx] in
100     let w = fst wr in
101     let r = snd wr in
```

```
    let ax = maybe (c 0.0) (mapLookup dx @a) in
    a := mapUpdate (pair dx (p ax (t (der1 u x) w))) @a;
    foreach (allNamePairs @s)
        {p' ->
            let n1 = fst p' in
            let n2 = snd p' in
            let key = makeNamePair n1 n2 in
            let h12 = maybe (c 0.0) (mapLookup key @h) in
            let r1 = maybe (c 0.0) (mapLookup n1 r) in
            let r2 = maybe (c 0.0) (mapLookup n2 r) in
            let rres = maybe (c 0.0) (mapLookup (tag res) r) in
            let n1Dep = eqName dx n1 in
            let n2Dep = eqName dx n2 in
            case (pair n1Dep n2Dep)
                { (pair true true) ->
                    h := mapUpdate (pair key
                        (p h12
                            (p (t (c 2.0) (t (der1 u x) r1))
                                (p (t (der1 u x) (t (der1 u x) rres))
                                    (t (dder1 u x) w)
                                )
                            )
                        )
                    ) @h
                | (pair true false) ->
                    h := mapUpdate (pair key (p h12 (t (der1 u x) r2))) @h
                | (pair false true) ->
                    h := mapUpdate (pair key (p h12 (t (der1 u x) r1))) @h
                | (pair false false) -> unit
                }
        }
reverseHessian s a h <ap2 b (tagged x dx) (tagged y dy) -> k> =
    let res = tagged (ap2 b x y) fresh! in
    reverseHessian s a h (k res);
    let wr = saveAndUpdate s a h res [tagged x dx, tagged y dy] in
    let w = fst wr in
    let r = snd wr in
    let ax = maybe (c 0.0) (mapLookup dx @a) in
    a := mapUpdate (pair dx (p ax (t (der2 L b x y) w))) @a;
    let ay = maybe (c 0.0) (mapLookup dy @a) in
    a := mapUpdate (pair dy (p ay (t (der2 R b x y) w))) @a;
    foreach (allNamePairs @s)
        {p' ->
            let n1 = fst p' in
            let n2 = snd p' in
            let key = makeNamePair n1 n2 in
            let h12 = maybe (c 0.0) (mapLookup key @h) in
            let r1 = maybe (c 0.0) (mapLookup n1 r) in
            let r2 = maybe (c 0.0) (mapLookup n2 r) in
            let rres = maybe (c 0.0) (mapLookup (tag res) r) in
            let n1Dep = or (eqName dx n1) (eqName dy n1) in
            let n2Dep = or (eqName dx n2) (eqName dy n2) in
            let a1 = if (eqName dx n1) {L} {R} in
```

```
155             let a2 = if (eqName dx n2) {L} {R} in
156             let same = eqName dx dy in
157             let dsame = p (der2 L b x y) (der2 R b x y) in
158             case (pair n1Dep n2Dep)
159                 { (pair true true) ->
160                     if same
161                     { h := mapUpdate (pair key
162                           (p h12
163                               (p (p (t dsame r2) (t dsame r1))
164                                  (p (t dsame (t dsame rres))
165                                     (t (dder2 true a1 a2 b x y) w))
166                               )
167                           )
168                       ) @h
169                     }
170                     { h := mapUpdate (pair key
171                           (p h12
172                               (p (p (t (der2 a1 b x y) r2)
173                                     (t (der2 a2 b x y) r1))
174                                  (p (t (der2 a1 b x y) (t (der2 a2 b x y) rres))
175                                     (t (dder2 false a1 a2 b x y) w))
176                               )
177                           )
178                       ) @h
179                     }
180                 | (pair true false) ->
181                     if same
182                     { h := mapUpdate (pair key (p h12 (t dsame r2))) @h}
183                     { h := mapUpdate
184                          (pair key (p h12 (t (der2 a1 b x y) r2))) @h
185                     }
186                 | (pair false true) ->
187                     if same
188                     { h := mapUpdate (pair key (p h12 (t dsame r1))) @h}
189                     { h := mapUpdate
190                          (pair key (p h12 (t (der2 a2 b x y) r1))) @h
191                     }
192                 | (pair false false) -> unit
193                 }
194         }
195
196 length : List X -> Int
197 length [] = 0
198 length (_ :: xs) = 1 + length xs
199
200 hessian : { List (Tagged X)
201             -> [RefState, Smooth X, Smooth (Tagged X)] (Tagged X)}
202        -> List X -> [RefState, Smooth X] List (List (Maybe X))
203 hessian f xs =
204     incrementName (1 - length xs) (
205         let z = map {x -> tagged x fresh!} xs in
206         let s = new (makeSet eqName []) in
207         let a = new (makeMap eqName []) in
```

```
208         let h = new (makeMap eqNamePair []) in
209         reverseHessian s a h (
210             let res = <Fresh> (f z) in
211             s := <Smooth> (setAdd (tag res) @s);
212             a := <Smooth> (mapUpdate (pair (tag res) (c 1.0)) @a);
213             h := <Smooth>
214                 (mapUpdate (pair (makeNamePair (tag res) (tag res)) (c 0.0)) @h)
215         );
216         map {r -> map {c -> mapLookup (makeNamePair (tag r) (tag c)) @h} z} z
217     )
```

Listing A.23: Hessian reverse mode (Eff)

```
1  type name = Name of int
2
3  let get_value (Name i) = i
4
5  type fresh = effect
6    operation fresh : unit -> name
7  end;;
8
9  let increment_name t init = handler
10   | t#fresh () k -> (fun i -> k (Name i) (i + 1))
11   | val x -> (fun _ -> x)
12   | finally f -> f init;;
13
14 type 'a tagged = Tag of 'a * name;;
15
16 let tag (Tag (_, t)) = t;;
17
18 let get_val (Tag (v, _)) = v;;
19
20 type name_pair = Name_Pair of name * name;;
21
22 let fst_name (Name_Pair (n, _)) = n;;
23
24 let snd_name (Name_Pair (_, m)) = m;;
25
26 let make_name_pair n m =
27   if get_value n < get_value m then Name_Pair (n, m) else Name_Pair (m, n);;
28
29 let concat xss = fold_right (@) xss [];;
30
31 let all_name_pairs s =
32   let xs = set_to_list s in
33   filter (fun (x, y) -> get_value x <= get_value y)
34     (concat (map (fun x -> map (fun y -> (x, y)) xs) xs));;
35
36 let rec concat_option ms =
37   match ms with
38   | [] -> []
39   | (None :: ns) -> concat_option ns
40   | (Some x :: ns) -> x :: concat_option ns;;
```

```
41
42  let option x m =
43    match m with
44    | None -> x
45    | Some y -> y;;
46
47  let fmap f m =
48    match m with
49    | None -> None
50    | Some x -> Some (f x);;
51
52  let rec foreach xs f = match xs with
53    | [] -> ()
54    | x :: ys -> ignore (f x); foreach ys f;;
55
56  let reverse_hessian i o f s a h =
57    let c x = o#ap0 (Const x) in
58    let ( ~- ) x = o#ap1 (Negate, x) in
59    let ( + ) x y = o#ap2 (Add, x, y) in
60    let ( * ) x y = o#ap2 (Multiply, x, y) in
61    let save_and_update res deps =
62      let w = option (c 0.0) (map_lookup (tag res) !a) in
63      a := map_update (tag res, c 0.0) !a;
64      let r = make_map (=) (
65        concat_option (map (fun v ->
66          let temp = map_lookup (make_name_pair v (tag res)) !h in
67          h := map_update (make_name_pair v (tag res), c 0.0) !h;
68          fmap (fun z -> (v, z)) temp
69        ) (set_to_list !s))
70      ) in
71      s := set_remove (tag res) !s;
72      foreach deps (fun dep -> s := set_add (tag dep) !s);
73      a := map_restrict (set_to_list !s) !a;
74      h := map_restrict (
75        let ss = set_to_list !s in
76        concat (map (fun n -> map (fun m -> make_name_pair n m) ss) ss)
77      ) !h;
78      (w, r)
79    in
80    handler
81      | i#ap0 n k ->
82        k (Tag (op0 o n, f#fresh ()))
83      | i#ap1 (u, Tag (x, dx)) k ->
84        let res = Tag (op1 o u x, f#fresh ()) in
85        k res;
86        let (w, r) = save_and_update res [Tag (x, dx)] in
87        let ax = option (c 0.0) (map_lookup dx !a) in
88        a := map_update (dx, ax + (der1 o u x * w)) !a;
89        foreach (all_name_pairs !s) (fun p ->
90          let (n1, n2) = p in
91          let key = make_name_pair n1 n2 in
92          let h12 = option (c 0.0) (map_lookup key !h) in
93          let r1 = option (c 0.0) (map_lookup n1 r) in
```

```
94          let r2 = option (c 0.0) (map_lookup n2 r) in
95          let rres = option (c 0.0) (map_lookup (tag res) r) in
96          let n1_dep = (dx = n1) in
97          let n2_dep = (dx = n2) in
98          match (n1_dep, n2_dep) with
99          | (true, true) ->
100           h := map_update (key,
101             h12 + ((c 2.0 * (der1 o u x * r1))
102                   + ((der1 o u x * (der1 o u x * rres))
103                     + (dder1 o u x * w)
104                     )
105                   )
106           ) !h
107         | (true, false) ->
108           h := map_update (key, h12 + (der1 o u x * r2)) !h
109         | (false, true) ->
110           h := map_update (key, h12 + (der1 o u x * r1)) !h
111         | (false, false) -> ()
112       )
113     | i#ap2 (b, Tag (x, dx), Tag (y, dy)) k ->
114       let res = Tag (op2 o b x y, f#fresh ()) in
115       k res;
116       let (w, r) = save_and_update res [Tag (x, dx); Tag (y, dy)] in
117       let ax = option (c 0.0) (map_lookup dx !a) in
118       a := map_update (dx, ax + (der2 o b L x y * w)) !a;
119       let ay = option (c 0.0) (map_lookup dy !a) in
120       a := map_update (dy, ay + (der2 o b R x y * w)) !a;
121       foreach (all_name_pairs !s) (fun p ->
122         let (n1, n2) = p in
123         let key = make_name_pair n1 n2 in
124         let h12 = option (c 0.0) (map_lookup key !h) in
125         let r1 = option (c 0.0) (map_lookup n1 r) in
126         let r2 = option (c 0.0) (map_lookup n2 r) in
127         let rres = option (c 0.0) (map_lookup (tag res) r) in
128         let n1_dep = (dx = n1) || (dy = n1) in
129         let n2_dep = (dx = n2) || (dy = n2) in
130         let a1 = if (dx = n1) then L else R in
131         let a2 = if (dx = n2) then L else R in
132         let same = (dx = dy) in
133         let dsame = der2 o b L x y + der2 o b R x y in
134         match (n1_dep, n2_dep) with
135         | (true, true) ->
136           if same then
137             h := map_update (key,
138               h12 + (
139                 ((dsame * r2) + (dsame * r1))
140                 +
141                 ((dsame * (dsame * rres)) + (dder2 o b true a1 a2 x y * w))
142               )
143             ) !h
144           else
145             h := map_update (key,
146               h12 + (
```

```
147                 ((der2 o b a1 x y * r2) + (der2 o b a2 x y * r1))
148                 +
149                 ( (der2 o b a1 x y * (der2 o b a2 x y * rres))
150                   +
151                   (dder2 o b false a1 a2 x y * w)
152                 )
153               )
154             ) !h
155         | (true, false) ->
156           if same then h := map_update (key, h12 + (dsame * r2)) !h
157           else h := map_update (key, h12 + (der2 o b a1 x y * r2)) !h
158         | (false, true) ->
159           if same then h := map_update (key, h12 + (dsame * r1)) !h
160           else h := map_update (key, h12 + (der2 o b a2 x y * r1)) !h
161         | (false, false) -> ()
162       )
163     | val x -> x;;
164
165 let hessian i o t f xs =
166   with increment_name t (1 - length xs) handle
167   let z = map (fun x -> Tag (x, t#fresh ())) xs in
168   let s = ref (make_set (=) []) in
169   let a = ref (make_map (=) []) in
170   let h = ref (make_map (=) []) in
171   let _ = with reverse_hessian i o t s a h handle (
172     let res = f i z in
173     s := set_add (tag res) !s;
174     a := map_update (tag res, op0 o (Const 1.0)) !a;
175     h := map_update (make_name_pair (tag res) (tag res), op0 o (Const 0.0)) !h
176   ) in
177   map (fun r ->
178     map (fun c ->
179       map_lookup (make_name_pair (tag r) (tag c)) !h
180     ) z
181   ) z;;
```

Listing A.24: Hessian reverse mode (Koka)

```
1  pub module hessian
2
3  import smooth
4  import map
5  import set
6
7  infixl 6 (+.)
8  infixl 7 (*.)
9
10 fun show(x : float64) : string {
11   show(x : float64, -17)
12 }
13
14 struct name{get-value : int}
15
```

```
16 fun show(n : name) : string {
17   "n(" ++ show(n.get-value) ++ ")"
18 }
19
20 effect fresh
21   fun fresh() : name
22
23 fun increment-name(init, action) {
24   var i := init
25   with handler
26     fun fresh() -> {val t = i; i := i + 1; Name(t)}
27   action()
28 }
29
30 struct name-pair{fst : name; snd : name}
31
32 fun show(n : name-pair) : string {
33   "(" ++ show(n.fst) ++ ", " ++ show(n.snd) ++ ")"
34 }
35
36 fun make-name-pair(n : name, m : name) : name-pair {
37   if n.get-value < m.get-value then Name-pair(n, m) else Name-pair(m, n)
38 }
39
40 fun all-name-pairs(s) {
41   with x <- s.to-list.flatmap
42   with y <- s.to-list.flatmap-maybe
43   if x.get-value > y.get-value then Nothing else Just((x,y))
44 }
45
46 fun eq-name-pair(p : name-pair, q : name-pair) {
47   eq-name(p.fst, q.fst) && eq-name(p.snd, q.snd)
48 }
49
50 fun eq-name(x : name, y : name) : bool {
51   x.get-value == y.get-value
52 }
53
54 type tagged<a>
55   Tag(v : a, tag : name)
56
57 fun reverse-hessian(
58     s : ref<h,set<name>>,
59     a : ref<h,map<name,float64>>,
60     h : ref<h,map<name-pair,float64>>,
61     action : (
62       () -> <
63         div,
64         st<h>,
65         fresh,
66         smooth<tagged<float64>>,
67         smooth<float64>
68       |e> ()
```

```
69      )
70    ) : <console,div,st<h>,fresh,smooth<float64>|e> () {
71    fun save-and-update(res, deps : list<tagged<float64>>) {
72        // Save needed values
73        val w = (!a).lookup(res.tag).maybe(c(0.0))
74        a := (!a).update(res.tag, c(0.0))
75        val r = make-map(
76          eq-name,
77          (!s).to-list.map(fn(v) {
78            val temp = (!h).lookup(make-name-pair(v,res.tag))
79            h := (!h).update(make-name-pair(v,res.tag), c(0.0))
80            temp.map(fn(z) {(v, z)})
81          }).concat-maybe
82        )
83        // Update live variables and restrict maps
84        s := (!s).remove(res.tag)
85        foreach(deps) fn(dep)
86          s := (!s).add(dep.tag)
87        a := (!a).restrict((!s).to-list)
88        h := (!h).restrict(({
89          with n <- (!s).to-list.map
90          with m <- (!s).to-list.map
91          make-name-pair(n,m)
92        })().concat)
93        (w, r)
94    }
95    with handler {
96      ctl ap0(n) -> resume(Tag(op0(n), fresh()))
97      ctl ap1(u,x) ->
98        val res = Tag(op1(u,x.v), fresh())
99        resume(res)
100       val (w, r) = save-and-update(res, [x])
101       // Update Adjoint
102       val ax = (!a).lookup(x.tag).maybe(c(0.0))
103       a := (!a).update(x.tag, ax +. (der1(u,x.v) *. w))
104       // Update Hessian
105       foreach(all-name-pairs(!s)) fn(p)
106         val (n1, n2) = p
107         val k = make-name-pair(n1,n2)
108         val h12 = (!h).lookup(k).maybe(c(0.0))
109         val r1 = r.lookup(n1).maybe(c(0.0))
110         val r2 = r.lookup(n2).maybe(c(0.0))
111         val rres = r.lookup(res.tag).maybe(c(0.0))
112         val n1-dep = eq-name(x.tag,n1)
113         val n2-dep = eq-name(x.tag,n2)
114         if n1-dep && n2-dep then
115           // So n1 == n2
116           h := (!h).update(k,
117             h12 +. (c(2.0) *. der1(u,x.v) *. r1)
118                 +. (der1(u,x.v) *. der1(u,x.v) *. rres)
119                 +. (dder1(u,x.v) *. w)
120           )
121         elif n1-dep then
```

```
122           h := (!h).update(k, h12 +. (der1(u,x.v) *. r2))
123         elif n2-dep then
124           h := (!h).update(k, h12 +. (der1(u,x.v) *. r1))
125         else
126           ()
127     ctl ap2(b,x,y) ->
128       val res = Tag(op2(b,x.v,y.v), fresh())
129       resume(res)
130       val (w, r) = save-and-update(res, [x,y])
131       // Update Adjoint
132       val ax = (!a).lookup(x.tag).maybe(c(0.0))
133       a := (!a).update(x.tag, ax +. (der2(b,L,x.v,y.v) *. w))
134       val ay = (!a).lookup(y.tag).maybe(c(0.0))
135       a := (!a).update(y.tag, ay +. (der2(b,R,x.v,y.v) *. w))
136       // Update Hessian
137       foreach(all-name-pairs(!s)) fn(p)
138         val (n1, n2) = p
139         val k = make-name-pair(n1,n2)
140         val h12 = (!h).lookup(k).maybe(c(0.0))
141         val r1 = r.lookup(n1).maybe(c(0.0))
142         val r2 = r.lookup(n2).maybe(c(0.0))
143         val rres = r.lookup(res.tag).maybe(c(0.0))
144         val n1-dep = eq-name(x.tag,n1) || eq-name(y.tag,n1)
145         val n2-dep = eq-name(x.tag,n2) || eq-name(y.tag,n2)
146         val a1 = if eq-name(x.tag,n1) then L else R
147         val a2 = if eq-name(x.tag,n2) then L else R
148         val same = eq-name(x.tag,y.tag)
149         val dsame = der2(b,L,x.v,y.v) +. der2(b,R,x.v,y.v)
150         if n1-dep && n2-dep then
151           if same then
152             h := (!h).update(k,
153               h12 +. (dsame *. r2)
154                   +. (dsame *. r1)
155                   +. (dsame *. dsame *. rres)
156                   +. (dder2(b,True,a1,a2,x.v,y.v) *. w)
157             )
158           else
159             h := (!h).update(k,
160               h12 +. (der2(b,a1,x.v,y.v) *. r2)
161                   +. (der2(b,a2,x.v,y.v) *. r1)
162                   +. (der2(b,a1,x.v,y.v) *. der2(b,a2,x.v,y.v) *. rres)
163                   +. (dder2(b,False,a1,a2,x.v,y.v) *. w)
164             )
165         elif n1-dep then
166           if same then h := (!h).update(k, h12 +. (dsame *. r2))
167           else h := (!h).update(k, h12 +. (der2(b,a1,x.v,y.v) *. r2))
168         elif n2-dep then
169           if same then h := (!h).update(k, h12 +. (dsame *. r1))
170           else h := (!h).update(k, h12 +. (der2(b,a2,x.v,y.v) *. r1))
171         else
172           ()
173   }
174   with mask<console>
```

```
  action()
}

fun hessian(
    f : (
      (list<tagged<float64>>) -> <
        st<h>,
        smooth<tagged<float64>>,
        smooth<float64>,
        div
      |e> tagged<float64>),
    xs : list<float64>
  ) : <st<h>,smooth<float64>,div,console|e> list<list<maybe<float64>>> {
  with increment-name(1 - xs.length)
  val z = map(xs, fn(x) {Tag(x, fresh())})
  val s = ref(make-set(eq-name,[]))
  val a = ref(make-map(eq-name,[]))
  val h = ref(make-map(eq-name-pair,[]))
  reverse-hessian(s,a,h) {
    with mask<fresh>
    val res = f(z)
    s := (!s).add(res.tag)
    a := (!a).update(res.tag, mask<smooth>{c(1.0)})
    h := (!h).update(make-name-pair(res.tag,res.tag), mask<smooth>{c(0.0)})
  }
  with r <- z.map
  with c <- z.map
  (!h).lookup(make-name-pair(r.tag,c.tag))
}
```

Listing A.25: Hessian reverse mode (OCaml)

```
open Effect.Deep
open Effect
open Smooth
open Option

type name = {get_value : int}

module type FRESH = sig
  type _ Effect.t += Fresh : unit -> name Effect.t

  val fresh : unit -> name
end

module Fresh : FRESH = struct
  type _ Effect.t += Fresh : unit -> name Effect.t

  let fresh () = perform (Fresh ())
end

type name_pair = Name_Pair of name * name

```

```
22 let make_name_pair n m =
23   if n.get_value < m.get_value then Name_Pair (n, m) else Name_Pair (m, n)
24
25 module Names =
26 struct
27   type t = name
28   let compare n m = Stdlib.compare n.get_value m.get_value
29 end
30
31 module Name_Set = Set.Make(Names)
32 module Name_Map = Map.Make(Names)
33
34 let all_name_pairs s =
35   let xs = Name_Set.to_seq s in
36   Seq.filter (fun (n, m) -> n.get_value <= m.get_value) (Seq.product xs xs)
37
38 module Name_Pairs =
39 struct
40   type t = name_pair
41   let compare (Name_Pair (n0, n1)) (Name_Pair (m0,m1)) =
42     match Stdlib.compare n0 m0 with
43         0 -> Stdlib.compare n1 m1
44       | c -> c
45 end
46
47 module Name_Pair_Map = Map.Make(Name_Pairs)
48
49 type 't tagged = {v : 't; tag : name}
50
51 module Hessian (T : SMOOTH) (F : FRESH) = struct
52   include Smooth (struct type t = T.t tagged end)
53   include Fresh
54
55   let increment_name init =
56     let i = ref init in {
57       retc = (fun x -> x);
58       exnc = raise;
59       effc = (fun (type a) (eff : a Effect.t) ->
60         match eff with
61         | Fresh () -> Some (fun (k : (a, _) continuation) ->
62             let t = !i in
63             i := !i + 1;
64             continue k {get_value = t}
65           )
66         | _ -> None
67       )
68   }
69
70   let reverse_hessian s a h =
71     let save_and_update res deps =
72       let open T in
73       let w = value (Name_Map.find_opt res.tag !a) ~default:(c 0.0) in
74       a := Name_Map.add res.tag (c 0.0) !a;
```

```
75        let r = Name_Map.of_seq (
76          Seq.concat_map (fun v ->
77            let temp = Name_Pair_Map.find_opt (make_name_pair v res.tag) !h in
78            h := Name_Pair_Map.add (make_name_pair v res.tag) (c 0.0) !h;
79            Seq.map (fun z -> (v, z)) (to_seq temp)
80          ) (Name_Set.to_seq !s)
81        ) in
82        s := Name_Set.remove res.tag !s;
83        List.iter (fun dep -> s := Name_Set.add dep.tag !s) deps;
84        a := Name_Map.filter (fun k _ -> Name_Set.mem k !s) !a;
85        let pairs =
86          let xs = Name_Set.to_seq !s in
87          Seq.map (fun (n, m) -> make_name_pair n m) (Seq.product xs xs)
88        in
89        h := Name_Pair_Map.filter
90          (fun k _ -> Seq.exists (fun k' -> k' = k) pairs) !h;
91        (w, r)
92      in {
93        retc = (fun x -> x);
94        exnc = raise;
95        effc = (fun (type b) (eff : b Effect.t) ->
96          match eff with
97          | Ap0 n -> Some (fun (k : (b, _) continuation) ->
98              let open T in
99              let open F in
100             continue k {v = op0 n; tag = fresh ()}
101           )
102         | Ap1 (u, x) -> Some (fun k ->
103             let open T in
104             let open F in
105             let res = {v = op1 u x.v; tag = fresh ()} in
106             continue k res;
107             let (w, r) = save_and_update res [x] in
108             let ax = value (Name_Map.find_opt x.tag !a) ~default:(c 0.0) in
109             a := Name_Map.add x.tag (ax +. (der1 u x.v *. w)) !a;
110             Seq.iter (fun (n1, n2) ->
111               let key = make_name_pair n1 n2 in
112               let h12 =
113                 value (Name_Pair_Map.find_opt key !h) ~default:(c 0.0) in
114               let r1 = value (Name_Map.find_opt n1 r) ~default:(c 0.0) in
115               let r2 = value (Name_Map.find_opt n2 r) ~default:(c 0.0) in
116               let rres = value (Name_Map.find_opt res.tag r) ~default:(c 0.0) in
117               let n1_dep = (x.tag = n1) in
118               let n2_dep = (x.tag = n2) in
119               match (n1_dep, n2_dep) with
120               | (true, true) ->
121                   h := Name_Pair_Map.add key (
122                     h12 +. (c 2.0 *. der1 u x.v *. r1)
123                         +. (der1 u x.v *. der1 u x.v *. rres)
124                         +. (dder1 u x.v *. w)
125                   ) !h
126               | (true, false) ->
127                   h := Name_Pair_Map.add key (h12 +. (der1 u x.v *. r2)) !h
```

```
128               | (false, true) ->
129                   h := Name_Pair_Map.add key (h12 +. (der1 u x.v *. r1)) !h
130               | (false, false) -> ()
131             ) (all_name_pairs !s)
132           )
133         | Ap2 (b, x, y) -> Some (fun k ->
134             let open T in
135             let open F in
136             let res = {v = op2 b x.v y.v; tag = fresh ()} in
137             continue k res;
138             let (w, r) = save_and_update res [x; y] in
139             let ax = value (Name_Map.find_opt x.tag !a) ~default:(c 0.0) in
140             a := Name_Map.add x.tag (ax +. (der2 b L x.v y.v *. w)) !a;
141             let ay = value (Name_Map.find_opt y.tag !a) ~default:(c 0.0) in
142             a := Name_Map.add y.tag (ay +. (der2 b R x.v y.v *. w)) !a;
143             Seq.iter (fun (n1, n2) ->
144               let key = make_name_pair n1 n2 in
145               let h12 =
146                 value (Name_Pair_Map.find_opt key !h) ~default:(c 0.0) in
147               let r1 = value (Name_Map.find_opt n1 r) ~default:(c 0.0) in
148               let r2 = value (Name_Map.find_opt n2 r) ~default:(c 0.0) in
149               let rres = value (Name_Map.find_opt res.tag r) ~default:(c 0.0) in
150               let n1_dep = (x.tag = n1) || (y.tag = n1) in
151               let n2_dep = (x.tag = n2) || (y.tag = n2) in
152               let a1 = if x.tag = n1 then L else R in
153               let a2 = if x.tag = n2 then L else R in
154               let same = (x.tag = y.tag) in
155               let dsame = der2 b L x.v y.v +. der2 b R x.v y.v in
156               match (n1_dep, n2_dep) with
157               | (true, true) ->
158                   if same then
159                     h := Name_Pair_Map.add key (
160                       h12 +. (dsame *. r2)
161                           +. (dsame *. r1)
162                           +. (dsame *. dsame *. rres)
163                           +. (dder2 b true a1 a2 x.v y.v *. w)
164                     ) !h
165                   else
166                     h := Name_Pair_Map.add key (
167                       h12 +. (der2 b a1 x.v y.v *. r2)
168                           +. (der2 b a2 x.v y.v *. r1)
169                           +. (der2 b a1 x.v y.v *. der2 b a2 x.v y.v *. rres)
170                           +. (dder2 b false a1 a2 x.v y.v *. w)
171                     ) !h
172               | (true, false) ->
173                   if same then
174                     h := Name_Pair_Map.add
175                       key (h12 +. (dsame *. r2)) !h
176                   else
177                     h := Name_Pair_Map.add
178                       key (h12 +. (der2 b a1 x.v y.v *. r2)) !h
179               | (false, true) ->
180                 if same then
```

```
                    h := Name_Pair_Map.add
                      key (h12 +. (dsame *. r1)) !h
                  else
                    h := Name_Pair_Map.add
                      key (h12 +. (der2 b a2 x.v y.v *. r1)) !h
              | (false, false) -> ()
            ) (all_name_pairs !s)
          )
        | _ -> None
      )
  }

  let hessian f xs =
    match_with (fun () ->
      let z = List.map (fun x -> {v = x; tag = F.fresh ()}) xs in
      let s = ref Name_Set.empty in
      let a = ref Name_Map.empty in
      let h = ref Name_Pair_Map.empty in
      let _ = match_with (fun () ->
        let res = f z in
        s := Name_Set.add res.tag !s;
        a := Name_Map.add res.tag (T.c 1.0) !a;
        h := Name_Pair_Map.add (make_name_pair res.tag res.tag) (T.c 0.0) !h
      ) () (reverse_hessian s a h) in
      List.map (fun r ->
        List.map (fun c ->
          Name_Pair_Map.find_opt (make_name_pair r.tag c.tag) !h
        ) z
      ) z
    ) () (increment_name (1 - List.length xs))
end
```

A.8 Checkpointing

Listing A.26: Checkpointed reverse mode (Eff)

```
type 'a prop = Prop of 'a * 'a ref

type 'a checkpoint = effect
  operation checkpoint : ('a prop smooth -> 'a prop) -> 'a prop
end

let rec evaluatet ch i o s =
  handler
    | i#ap0 n k ->
      let r = Prop (op0 o n, s) in k r
    | i#ap1 (u, Prop (x, _)) k ->
      let r = Prop (op1 o u x, s) in k r
    | i#ap2 (b, Prop (x, _), Prop (y, _)) k ->
      let r = Prop (op2 o b x y, s) in k r
    | ch#checkpoint p k ->
      let Prop (res, _) = with evaluatet ch i o s handle p i in
```

```
17        k (Prop (res, s))
18      | val x -> x;;
19
20  let rec reversec ch i o =
21    let c x = o#ap0 (Const x) in
22    let ( ~- ) x = o#ap1 (Negate, x) in
23    let ( + ) x y = o#ap2 (Add, x, y) in
24    let ( * ) x y = o#ap2 (Multiply, x, y) in
25    handler
26      | i#ap0 n k ->
27        let r = Prop (op0 o n, ref (c 0.0)) in
28        k r
29      | i#ap1 (u, Prop (x, dx)) k ->
30        let dr = ref (c 0.0) in
31        let r = Prop (op1 o u x, dr) in
32        ignore (k r);
33        dx := !dx + (der1 o u x * !dr)
34      | i#ap2 (b, Prop (x, dx), Prop (y, dy)) k ->
35        let dr = ref (c 0.0) in
36        let r = Prop (op2 o b x y, dr) in
37        ignore (k r);
38        dx := !dx + (der2 o b L x y * !dr);
39        dy := !dy + (der2 o b R x y * !dr)
40      | ch#checkpoint p k ->
41        let s = ref (c 0.0) in
42        let Prop (res, _) = with evaluatet ch i o s handle p i in
43        let Prop (r, dr) = Prop (res, ref (c 0.0)) in
44        k (Prop (r, dr));
45        with reversec ch i o handle (
46          let Prop (_, dres) = p i in
47          dres := !dr
48        )
49      | val x -> x;;
50
51  let grad ch i o f x =
52    let dz = ref (op0 o (Const 0.0)) in
53    let z = Prop (x, dz) in
54    ( with reversec ch i o handle
55        let Prop (r, dr) = f ch i z in
56        dr := op0 o (Const 1.0)
57    );
58    !dz;;
```

Listing A.27: Checkpointed reverse mode (Koka)

```
1  pub module checkpoint
2
3  import smooth
4  import reverse
5
6  rec effect checkpoint<h,a,e> {
7    ctl check(
8      prog : () -> <checkpoint<h,a,e>,smooth<prop<h,a>>,div|e> prop<h,a>
```

```
  ) : prop<h,a>
}

fun lift(
    action : () -> <smooth<prop<h,a>>|e> b
  ) : <smooth<prop<h,a>>, smooth<a>|e> b {
  (mask<smooth>{
    mask<smooth>{
      with handler {
        return(x) -> fn() {x}
        ctl ap0(n) -> fn() {
          (fn(z) {(mask<smooth>{mask<smooth>{resume(z)}})()}) (ap0(n))
        }
        ctl ap1(u,x) -> fn() {
          (fn(z) {(mask<smooth>{mask<smooth>{resume(z)}})()}) (ap1(u,x))
        }
        ctl ap2(b,x,y) -> fn() {
          (fn(z) {(mask<smooth>{mask<smooth>{resume(z)}})()}) (ap2(b,x,y))
        }
      }
      action()
  }})()
}

fun evaluatet(
    s : ref<h,a>,
    action : (() -> <checkpoint<h,a,e>,div,smooth<prop<h,a>>,smooth<a>|e> b)
  )
  : <div,smooth<a>|e> b {
  with handler {
    ctl check(p) -> {
      val r = evaluatet(s, {lift{p()}})
      resume(r)
    }
  }
  with handler {
    ctl ap0(n) -> resume(Prop(op0(n), s))
    ctl ap1(u,x) -> resume(Prop(op1(u,x.v), s))
    ctl ap2(b,x,y) -> resume(Prop(op2(b,x.v,y.v), s))
  }
  action()
}

fun reversec(
    action : (
      () -> <
        checkpoint<h,a,<st<h>|e>>,
        div,
        smooth<prop<h,a>>,
        smooth<a>,
        st<h>
      |e> ()
    )
```

```
  ) : <div,st<h>,smooth<a>|e> () {
  with handler {
    ctl check(p) -> {
      val s = ref(c(0.0))
      val res = evaluatet(s, {lift{p()}})
      val r = Prop(res.v, ref(c(0.0)))
      resume(r)
      reversec{set((lift{p()}).dv, !r.dv)}
    }
  }
  with reverse
  action()
}

fun gradc(f, x) {
    val z = Prop(x, ref(c(0.0)))
    reversec{set((f(z)).dv, mask<smooth>{c(1.0)})}
    !z.dv
}
```

Listing A.28: Checkpointed reverse mode (OCaml)

```
open Effect.Deep
open Effect
open Smooth

type 't prop = {v : 't; dv : 't ref}

module type CHECKPOINT = sig
  type t
  type _ Effect.t += Checkpoint : (unit -> t prop) -> t prop Effect.t

  val checkpoint : (unit -> t prop) -> t prop
end

module Checkpoint (T : sig type t end) : CHECKPOINT with type t = T.t = struct
  type t = T.t
  type _ Effect.t += Checkpoint : (unit -> t prop) -> t prop Effect.t

  let checkpoint p = perform (Checkpoint p)
end

module Reverse_checkpoint (T : SMOOTH) = struct
  include Smooth (struct type t = T.t prop end)
  include Checkpoint (struct type t = T.t end)

  let rec evaluatet s = {
    retc = (fun x -> x);
    exnc = raise;
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
      | Ap0 n -> Some (fun (k : (a, _) continuation) -> let open T in
          continue k {v = op0 n; dv = s}
```

```
32          )
33        | Ap1 (u, x) -> Some (fun k -> let open T in
34            continue k {v = op1 u x.v; dv = s}
35          )
36        | Ap2 (b, x, y) -> Some (fun k -> (let open T in
37            continue k {v = op2 b x.v y.v; dv = s}
38          ))
39        | Checkpoint p -> Some (fun k -> (
40            let {v = res; dv = _} = match_with p () (evaluatet s) in
41            continue k {v = res; dv = s}
42          ))
43        | _ -> None
44      )
45    }
46
47    let rec reversec () = {
48      retc = (fun x -> x);
49      exnc = raise;
50      effc = (fun (type a) (eff : a Effect.t) ->
51        match eff with
52        | Ap0 n -> Some (fun (k : (a, _) continuation) -> let open T in
53            continue k {v = op0 n; dv = ref (c 0.0)}
54          )
55        | Ap1 (u, x) -> Some (fun k -> let open T in
56            let r = {v = op1 u x.v; dv = ref (c 0.0)} in
57            continue k r;
58            x.dv := !(x.dv) +. (der1 u x.v *. !(r.dv))
59          )
60        | Ap2 (b, x, y) -> Some (fun k -> (let open T in
61            let r = {v = op2 b x.v y.v; dv = ref (c 0.0)} in
62            continue k r;
63            x.dv := !(x.dv) +. (der2 b L x.v y.v *. !(r.dv));
64            y.dv := !(y.dv) +. (der2 b R x.v y.v *. !(r.dv))
65          ))
66        | Checkpoint p -> Some (fun k -> (let open T in
67            let s = ref (c 0.0) in
68            let res = match_with p () (evaluatet s) in
69            let r = {v = res.v; dv = ref (c 0.0)} in
70            continue k r;
71            match_with (fun () ->
72              let {v = _; dv = dres} = p () in
73              dres := !(r.dv)
74            ) () (reversec ())
75          ))
76        | _ -> None
77      )
78    }
79
80    let grad f x =
81      let r = {v = x; dv = ref (T.c 0.0)} in
82      match_with (fun x -> (f x).dv := T.c 1.0) r (reversec ());
83      !(r.dv)
84  end
```

A.9 Higher-Order Functions

Listing A.29: Higher-order reverse mode (Eff)

```
type 'a prop = Prop of 'a * 'a ref

type 'a func = Func of 'a prop -> ('a prop * (unit -> unit))

type 'a abstraction = effect
  operation abs : ('a prop smooth -> 'a prop -> 'a prop) -> 'a func
  operation app : ('a func * 'a prop) -> 'a prop
end

let rec reverseh a i o =
  let c x = o#ap0 (Const x) in
  let ( ~-- ) x = o#ap1 (Negate, x) in
  let ( + ) x y = o#ap2 (Add, x, y) in
  let ( * ) x y = o#ap2 (Multiply, x, y) in
  handler
    | i#ap0 n k ->
      let r = Prop (op0 o n, ref (c 0.0)) in
      k r
    | i#ap1 (u, Prop (x, dx)) k ->
      let dr = ref (c 0.0) in
      let r = Prop (op1 o u x, dr) in
      let (res, bp) = k r in
      (res, (fun () ->
        bp ();
        dx := !dx + (der1 o u x * !dr)
      ))
    | i#ap2 (b, Prop (x, dx), Prop (y, dy)) k ->
      let dr = ref (c 0.0) in
      let r = Prop (op2 o b x y, dr) in
      let (res, bp) = k r in
      (res, (fun () ->
        bp ();
        dx := !dx + (der2 o b L x y * !dr);
        dy := !dy + (der2 o b R x y * !dr)
      ))
    | a#abs f k ->
      let g = (fun x -> with reverseh a i o handle f i x) in
      k (Func g)
    | a#app (Func f, x) k ->
      let (r, br) = f x in
      let (res, bp) = k r in
      (res, (fun () -> bp (); br ()))
    | val x -> (x, (fun () -> ()));;

let gradh a i o f x =
  let dz = ref (op0 o (Const 0.0)) in
  let z = Prop (x, dz) in
  let (_, bp) = (with reverseh a i o handle
    let Prop (r, dr) = f i z in
```

```
50      dr := op0 o (Const 1.0);
51      Prop (r, dr))
52    in
53    bp ();
54    !dz;;
```

Listing A.30: Higher-order reverse mode (Koka)

```
1  pub module reverse-higher-order
2
3  import smooth
4  import std/num/float64
5
6  type prop<h,a> {
7    Prop(v : a, dv : ref<h, a>)
8  }
9
10 rec type bp<h,a,e> {
11   Bp(do : () -> <st<h>,smooth<a>,div|e> ())
12 }
13
14 rec type func<h,a,e> {
15   Func(body : (prop<h,a> -> <st<h>,smooth<a>,div|e> (prop<h,a>, bp<h,a,e>)))
16 }
17
18 rec effect asmooth<h,a,e> {
19   ctl ap0_(n : nullary) : prop<h,a>
20   ctl ap1_(u : unary, arg : prop<h,a>) : prop<h,a>
21   ctl ap2_(b : binary, arg1 : prop<h,a>, arg2 : prop<h,a>) : prop<h,a>
22   ctl abs_(
23       f : prop<h,a> -> <st<h>,asmooth<h,a,e>,div|e> prop<h,a>
24     ) : func<h,a,e>
25   ctl app_(f : func<h,a,e>, x : prop<h,a>) : prop<h,a>
26 }
27
28 inline fun c__(i : float64) {
29     ap0_(Const(i))
30 }
31
32 inline fun (~..)(x) {
33     ap1_(Negate, x)
34 }
35
36 inline fun sin__(x) {
37     ap1_(Sin, x)
38 }
39
40 inline fun cos__(x) {
41     ap1_(Cos, x)
42 }
43
44 inline fun exp__(x) {
45     ap1_(Exp, x)
```

```
}

inline fun (+..)(x, y) {
    ap2_(Plus, x, y)
}

inline fun (-..)(x, y) {
    ap2_(Subtract, x, y)
}

inline fun (*..)(x, y) {
    ap2_(Times, x, y)
}

inline fun div__(x, y) {
    ap2_(Divide, x, y)
}

fun areverse(
    action : () -> <st<h>,asmooth<h,a,e>,div,smooth<a>|e> b
  ) : <st<h>,div,smooth<a>|e> (b, bp<h,a,e>) {
  with handler {
    return(x) -> (x, Bp(fn() {}))
    ctl ap0_(n) -> {
      val r = Prop(op0(n), ref(c(0.0)))
      resume(r)
    }
    ctl ap1_(u,x) -> {
      val r = Prop(op1(u,x.v), ref(c(0.0)))
      val (a, bp) = resume(r)
      (a, Bp(fn() {
        (bp.do)()
        set(x.dv, !x.dv +. (der1(u,x.v) *. !r.dv))
      }))
    }
    ctl ap2_(b,x,y) -> {
      val r = Prop(op2(b,x.v,y.v), ref(c(0.0)))
      val (a, bp) = resume(r)
      (a, Bp(fn() {
        (bp.do)()
        set(x.dv, !x.dv +. (der2(b,L,x.v,y.v) *. !r.dv))
        set(y.dv, !y.dv +. (der2(b,R,x.v,y.v) *. !r.dv))
      }))
    }
    ctl abs_(f) -> {
      val g = fn(x) {
        with areverse
        mask<smooth>{f(x)}
      }
      resume(Func(g))
    }
    ctl app_(f,x) -> {
      val (r, br) = (f.body)(x)
```

```
99        val (res, bres) = resume(r)
100       (res, Bp(fn() {(bres.do)(); (br.do)()}))
101     }
102   }
103   action()
104 }
105
106 fun grad(
107     f : prop<h,a> -> <st<h>,asmooth<h,a,e>,div,smooth<a>|e> prop<h,a>,
108     x : a
109   ) : <st<h>,div,smooth<a>|e> a {
110   val z = Prop(x, ref(c(0.0)))
111   val (_, bp) = areverse{set(f(z).dv, mask<asmooth>{c(1.0)})}
112   (bp.do)()
113   !z.dv
114 }
```

Listing A.31: Higher-order reverse mode (OCaml)

```
1  open Effect.Deep
2  open Effect
3  open Smooth
4
5  type 't prop = {v : 't; mutable dv : 't}
6
7  type 't func = Func of ('t prop -> ('t prop * (unit -> unit)))
8
9  module type LAMBDA = sig
10   type t
11   type _ Effect.t += Abs : (t prop -> t prop) -> t func Effect.t
12                    | App : (t func * t prop) -> t prop Effect.t
13
14   val abs : (t prop -> t prop) -> t func
15   val app : (t func * t prop) -> t prop
16 end
17
18 module Lambda (T : sig type t end) : LAMBDA with type t = T.t = struct
19   type t = T.t
20   type _ Effect.t += Abs : (t prop -> t prop) -> t func Effect.t
21                    | App : (t func * t prop) -> t prop Effect.t
22
23   let abs f = perform (Abs f)
24   let app fx = perform (App fx)
25 end
26
27 module Reverse_higher_order (T : SMOOTH) = struct
28   include Smooth (struct type t = T.t prop end)
29   include Lambda (struct type t = T.t end)
30
31   let rec reverse = {
32     retc = (fun x -> (x, (fun () -> ())));
33     exnc = raise;
34     effc = (fun (type a) (eff : a Effect.t) ->
```

```
      match eff with
      | Ap0 n -> Some (fun (k : (a, _) continuation) -> let open T in
          continue k {v = op0 n; dv = c 0.0}
        )
      | Ap1 (u, x) -> Some (fun k -> let open T in
          let r = {v = op1 u x.v; dv = c 0.0} in
          let (res, bp) = continue k r in
          (res, (fun () ->
            bp ();
            x.dv <- x.dv +. (der1 u x.v *. r.dv)
          ))
        )
      | Ap2 (b, x, y) -> Some (fun k -> (let open T in
          let r = {v = op2 b x.v y.v; dv = c 0.0} in
          let (res, bp) = continue k r in
          (res, (fun () ->
            bp ();
            x.dv <- x.dv +. (der2 b L x.v y.v *. r.dv);
            y.dv <- y.dv +. (der2 b R x.v y.v *. r.dv);
          ))
        ))
      | Abs f -> Some (fun k -> (
          let g = (fun x -> match_with f x reverse) in
          continue k (Func g)
        ))
      | App (Func f, x) -> Some (fun k -> (
          let (r, br) = f x in
          let (res, bp) = continue k r in
          (res, (fun () -> bp (); br ()))
        ))
      | _ -> None
    )
  }

  let grad f x =
    let r = {v = x; dv = T.c 0.0} in
    let (_, bp) = match_with (fun x ->
      let res = f x in
      res.dv <- T.c 1.0;
      res
    ) r reverse in
    bp ();
    r.dv
end
```

Listing A.32: Higher-order checkpointed reverse mode (Eff)

```
type 'a prop = Prop of 'a * 'a ref

type 'a func = Func of 'a prop -> ('a prop * (unit -> unit))

type 'a abstraction = effect
  operation abs : ('a prop smooth -> 'a prop -> 'a prop) -> 'a func
```

```
7     operation app : ('a func * 'a prop) -> 'a prop
8   end
9
10  let rec evaluatet a i o s =
11    handler
12      | i#ap0 n k ->
13        let r = Prop (op0 o n, s) in k r
14      | i#ap1 (u, Prop (x, _)) k ->
15        let r = Prop (op1 o u x, s) in k r
16      | i#ap2 (b, Prop (x, _), Prop (y, _)) k ->
17        let r = Prop (op2 o b x y, s) in k r
18      | a#abs f k ->
19        let g = (fun x -> with evaluatet a i o s handle f i x) in
20        k (Func g)
21      | a#app (Func f, x) k ->
22        let (r, _) = f x in k r
23      | val x -> (x, (fun () -> ()));;
24
25  let rec reversehc a i o =
26    let c x = o#ap0 (Const x) in
27    let ( ~- ) x = o#ap1 (Negate, x) in
28    let ( + ) x y = o#ap2 (Add, x, y) in
29    let ( * ) x y = o#ap2 (Multiply, x, y) in
30    handler
31      | i#ap0 n k ->
32        let r = Prop (op0 o n, ref (c 0.0)) in
33        k r
34      | i#ap1 (u, Prop (x, dx)) k ->
35        let dr = ref (c 0.0) in
36        let r = Prop (op1 o u x, dr) in
37        let (res, bp) = k r in
38        (res, (fun () ->
39          bp ();
40          dx := !dx + (der1 o u x * !dr)
41        ))
42      | i#ap2 (b, Prop (x, dx), Prop (y, dy)) k ->
43        let dr = ref (c 0.0) in
44        let r = Prop (op2 o b x y, dr) in
45        let (res, bp) = k r in
46        (res, (fun () ->
47          bp ();
48          dx := !dx + (der2 o b L x y * !dr);
49          dy := !dy + (der2 o b R x y * !dr)
50        ))
51      | a#abs f k ->
52        let g = (fun x ->
53          let dres = ref (c 0.0) in
54          let (Prop (res, _), _) = (with evaluatet a i o dres handle f i x) in
55          (Prop (res, dres), (fun () ->
56            let (_, bp) = (with reversehc a i o handle
57               let Prop (r, dr) = f i x in
58               dr := !dres;
59               Prop (r, dr)
```

```
60            ) in
61            bp ()
62          ))
63        ) in
64        k (Func g)
65      | a#app (Func f, x) k ->
66        let (r, br) = f x in
67        let (res, bp) = k r in
68        (res, (fun () -> bp (); br ()))
69      | val x -> (x, (fun () -> ()));;
70
71  let gradhc a i o f x =
72    let dz = ref (op0 o (Const 0.0)) in
73    let z = Prop (x, dz) in
74    let (_, bp) = (with reversehc a i o handle
75      let Prop (r, dr) = f i z in
76      dr := op0 o (Const 1.0);
77      Prop (r, dr))
78    in
79    bp ();
80    !dz;;
```

Listing A.33: Higher-order checkpointed reverse mode (Koka)

```
1  pub module reverse-higher-order-checkpoint
2
3  import smooth
4  import std/num/float64
5
6  type prop<h,a> {
7    Prop(v : a, dv : ref<h, a>)
8  }
9
10 rec type bp<h,a,e> {
11   Bp(do : () -> <st<h>,smooth<a>,div|e> ())
12 }
13
14 rec type func<h,a,e> {
15   Func(body : (prop<h,a> -> <st<h>,smooth<a>,div|e> (prop<h,a>, bp<h,a,e>)))
16 }
17
18 rec effect asmooth<h,a,e> {
19   ctl ap0_(n : nullary) : prop<h,a>
20   ctl ap1_(u : unary, arg : prop<h,a>) : prop<h,a>
21   ctl ap2_(b : binary, arg1 : prop<h,a>, arg2 : prop<h,a>) : prop<h,a>
22   ctl abs_(
23       f : prop<h,a> -> <st<h>,asmooth<h,a,e>,div|e> prop<h,a>
24     ) : func<h,a,e>
25   ctl app_(f : func<h,a,e>, x : prop<h,a>) : prop<h,a>
26 }
27
28 inline fun c__(i : float64) {
29     ap0_(Const(i))
```

```
30 }
31
32 inline fun (~..)(x) {
33     ap1_(Negate, x)
34 }
35
36 inline fun sin__(x) {
37     ap1_(Sin, x)
38 }
39
40 inline fun cos__(x) {
41     ap1_(Cos, x)
42 }
43
44 inline fun exp__(x) {
45     ap1_(Exp, x)
46 }
47
48 inline fun (+..)(x, y) {
49     ap2_(Plus, x, y)
50 }
51
52 inline fun (-..)(x, y) {
53     ap2_(Subtract, x, y)
54 }
55
56 inline fun (*..)(x, y) {
57     ap2_(Times, x, y)
58 }
59
60 inline fun div__(x, y) {
61     ap2_(Divide, x, y)
62 }
63
64 fun aevaluate(
65     s : ref<h,a>,
66     action : () -> <st<h>,asmooth<h,a,e>,div,smooth<a>|e> b
67   ) : <st<h>,div,smooth<a>|e> (b, bp<h,a,e>) {
68   with handler {
69     return(x) -> (x, Bp(fn() {}))
70     ctl ap0_(n) -> {
71       val r = Prop(op0(n), s)
72       resume(r)
73     }
74     ctl ap1_(u,x) -> {
75       val r = Prop(op1(u,x.v), s)
76       resume(r)
77     }
78     ctl ap2_(b,x,y) -> {
79       val r = Prop(op2(b,x.v,y.v), s)
80       resume(r)
81     }
82     ctl abs_(f) -> {
```

```
      val g = fn(x) {
        with aevaluate(s)
        mask<smooth>{f(x)}
      }
      resume(Func(g))
    }
    ctl app_(f,x) -> {
      val (r, _) = (f.body)(x)
      resume(r)
    }
  }
  action()
}

fun areverse(
    action : () -> <st<h>,asmooth<h,a,e>,div,smooth<a>|e> b
  ) : <st<h>,div,smooth<a>|e> (b, bp<h,a,e>) {
  with handler {
    return(x) -> (x, Bp(fn() {}))
    ctl ap0_(n) -> {
      val r = Prop(op0(n), ref(c(0.0)))
      resume(r)
    }
    ctl ap1_(u,x) -> {
      val r = Prop(op1(u,x.v), ref(c(0.0)))
      val (a, bp) = resume(r)
      (a, Bp(fn() {
        (bp.do)()
        set(x.dv, !x.dv +. (der1(u,x.v) *. !r.dv))
      }))
    }
    ctl ap2_(b,x,y) -> {
      val r = Prop(op2(b,x.v,y.v), ref(c(0.0)))
      val (a, bp) = resume(r)
      (a, Bp(fn() {
        (bp.do)()
        set(x.dv, !x.dv +. (der2(b,L,x.v,y.v) *. !r.dv))
        set(y.dv, !y.dv +. (der2(b,R,x.v,y.v) *. !r.dv))
      }))
    }
    ctl abs_(f) -> {
      val g = fn(x) {
        val s = ref(c(0.0))
        val (res, _) = aevaluate(s, {mask<smooth>{f(x)}})
        val r = Prop(res.v, ref(c(0.0)))
        (r, Bp(fn() {
          val (_, bp) = areverse{set(mask<smooth>{f(x).dv}, !r.dv)}
          (bp.do)()
        }))
      }
      resume(Func(g))
    }
    ctl app_(f,x) -> {
```

```
      val (r, br) = (f.body)(x)
      val (res, bres) = resume(r)
      (res, Bp(fn() {(bres.do)(); (br.do)()}))
    }
  }
  action()
}

fun grad(
    f : prop<h,a> -> <st<h>,asmooth<h,a,e>,div,smooth<a>|e> prop<h,a>,
    x : a
  ) : <st<h>,div,smooth<a>|e> a {
  val z = Prop(x, ref(c(0.0)))
  val (_, bp) = areverse{set(f(z).dv, mask<asmooth>{c(1.0)})}
  (bp.do)()
  !z.dv
}
```

Listing A.34: Higher-order checkpointed reverse mode (OCaml)

```
open Effect.Deep
open Effect
open Smooth

type 't prop = {v : 't; dv : 't ref}

type 't func = Func of ('t prop -> ('t prop * (unit -> unit)))

module type LAMBDA = sig
  type t
  type _ Effect.t += Abs : (t prop -> t prop) -> t func Effect.t
                   | App : (t func * t prop) -> t prop Effect.t

  val abs : (t prop -> t prop) -> t func
  val app : (t func * t prop) -> t prop
end

module Lambda (T : sig type t end) : LAMBDA with type t = T.t = struct
  type t = T.t
  type _ Effect.t += Abs : (t prop -> t prop) -> t func Effect.t
                   | App : (t func * t prop) -> t prop Effect.t

  let abs f = perform (Abs f)
  let app fx = perform (App fx)
end

module Reverse_higher_order_checkpoint (T : SMOOTH) = struct
  include Smooth (struct type t = T.t prop end)
  include Lambda (struct type t = T.t end)

  let rec evaluatet s = {
    retc = (fun x -> (x, (fun () -> ())));
    exnc = raise;
```

```
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
      | Ap0 n -> Some (fun (k : (a, _) continuation) -> let open T in
          continue k {v = op0 n; dv = s}
        )
      | Ap1 (u, x) -> Some (fun k -> let open T in
          continue k {v = op1 u x.v; dv = s}
        )
      | Ap2 (b, x, y) -> Some (fun k -> (let open T in
          continue k {v = op2 b x.v y.v; dv = s}
        ))
        | Abs f -> Some (fun k -> (
          let g = (fun x -> match_with f x (evaluatet s)) in
          continue k (Func g)
        ))
      | App (Func f, x) -> Some (fun k -> (
          let (r, _) = f x in
          continue k r
        ))
      | _ -> None
    )
  }
  let rec reverse = {
    retc = (fun x -> (x, (fun () -> ())));
    exnc = raise;
    effc = (fun (type a) (eff : a Effect.t) ->
      match eff with
      | Ap0 n -> Some (fun (k : (a, _) continuation) -> let open T in
          continue k {v = op0 n; dv = ref (c 0.0)}
        )
      | Ap1 (u, x) -> Some (fun k -> let open T in
          let r = {v = op1 u x.v; dv = ref (c 0.0)} in
          let (res, bp) = continue k r in
          (res, (fun () ->
            bp ();
            x.dv := !(x.dv) +. (der1 u x.v *. !(r.dv))
          ))
        )
      | Ap2 (b, x, y) -> Some (fun k -> (let open T in
          let r = {v = op2 b x.v y.v; dv = ref (c 0.0)} in
          let (res, bp) = continue k r in
          (res, (fun () ->
            bp ();
            x.dv := !(x.dv) +. (der2 b L x.v y.v *. !(r.dv));
            y.dv := !(y.dv) +. (der2 b R x.v y.v *. !(r.dv));
          ))
        ))
      | Abs f -> Some (fun k -> (let open T in
          let g = (fun x ->
            let dres = ref (c 0.0) in
            let ({v = res; dv = _}, _) = match_with f x (evaluatet dres) in
            ({v = res; dv = dres}, (fun () ->
              let (_, bp) = match_with (fun y ->
```

```
                let {v = r; dv = dr} = f y in
                dr := !dres;
                {v = r; dv = dr}
              ) x reverse in
              bp ();
            ))
          ) in
          continue k (Func g)
        ))
      | App (Func f, x) -> Some (fun k -> (
          let (r, br) = f x in
          let (res, bp) = continue k r in
          (res, (fun () -> bp (); br ()))
        ))
      | _ -> None
    )
  }

  let grad f x =
    let r = {v = x; dv = ref (T.c 0.0)} in
    let (_, bp) = match_with (fun x ->
      let res = f x in
      res.dv := T.c 1.0;
      res
    ) r reverse in
    bp ();
    !(r.dv)
end
```

A.10 Real World Benchmarks Code

Listing A.35: Smooth effect and helper functions, tensors (OCaml)

```
open Effect

type u_to_s = Const of float
type s_to_s = Negate | Log
type s's_to_s = Add | Subtract | Multiply | Divide

type u_to_t = Zeros of int array | Create of int array * float
type t_to_t
  = Squeeze of int array option
  | Reshape of int array
  | GetSlice of int list list
  | SliceLeft of int array
  | Transpose of int array option
  | Exp
  | Negate
  | PowerConst of float
  | SumReduce of int array option
  | LogSumExp of int option * bool option
  | Softmax of int option
```

```
type t't_to_t
  = Add
  | Subtract
  | Multiply
  | Divide
  | Einsum_ijk_mik_to_mij
  | Einsum_ijk_mij_to_mik
  | Einsum_mij_mik_to_ijk
  | SetSlice of int list list

type t_to_s = Get of int array | Sum
type s't_to_t = ScalarMultiply | SubtractScalar
type ta_to_t = Concatenate of int option | Stack of int option
type t_to_ta = Split of int option * int array

type arg = L | R

module type SMOOTH = sig
  type scalar
  type tensor
  type _ Effect.t +=
      Ap_u_to_s : u_to_s -> scalar Effect.t
    | Ap_s_to_s : s_to_s * scalar -> scalar Effect.t
    | Ap_s's_to_s : s's_to_s * scalar * scalar -> scalar Effect.t
    | Ap_u_to_t : u_to_t -> tensor Effect.t
    | Ap_t_to_t : t_to_t * tensor -> tensor Effect.t
    | Ap_t't_to_t : t't_to_t * tensor * tensor -> tensor Effect.t
    | Ap_t_to_s : t_to_s * tensor -> scalar Effect.t
    | Ap_s't_to_t : s't_to_t * scalar * tensor -> tensor Effect.t
    | Ap_ta_to_t : ta_to_t * tensor array -> tensor Effect.t
    | Ap_t_to_ta : t_to_ta * tensor -> tensor array Effect.t

  val c : float -> scalar
  val ( ~. ) : scalar -> scalar
  val log : scalar -> scalar
  val ( +. ) : scalar -> scalar -> scalar
  val ( -. ) : scalar -> scalar -> scalar
  val ( *. ) : scalar -> scalar -> scalar
  val ( /. ) : scalar -> scalar -> scalar

  (* Non-differentiable operations *)
  val shape : tensor -> int array
  val add_ : tensor -> tensor -> unit

  (* Creating constant tensors *)
  val zeros : int array -> tensor
  val create : int array -> float -> tensor

  (* Combining tensors *)
  val concatenate : ?axis:int -> tensor array -> tensor
  val stack : ?axis:int -> tensor array -> tensor

  (* Splitting tensors *)
```

```
73  val split : ?axis:int -> int array -> tensor -> tensor array
74
75  (* Changing tensor shape *)
76  val transpose : ?axis:int array -> tensor -> tensor
77  val reshape : tensor -> int array -> tensor
78
79  (* Shrinking and slicing tensors *)
80  val squeeze : ?axis:int array -> tensor -> tensor
81  val get_slice : int list list -> tensor -> tensor
82  val slice_left : tensor -> int array -> tensor
83  val get : tensor -> int array -> scalar
84  val set_slice : int list list -> tensor -> tensor -> tensor
85
86  (* Einsum operations *)
87  val einsum_ijk_mik_to_mij : tensor -> tensor -> tensor
88  val einsum_ijk_mij_to_mik : tensor -> tensor -> tensor
89  val einsum_mij_mik_to_ijk : tensor -> tensor -> tensor
90
91  (* Pointwise tensor operations *)
92  val exp : tensor -> tensor
93  val pow_const : tensor -> float -> tensor
94  val ( ~- ) : tensor -> tensor
95  val ( + ) : tensor -> tensor -> tensor
96  val ( - ) : tensor -> tensor -> tensor
97  val ( * ) : tensor -> tensor -> tensor
98  val ( / ) : tensor -> tensor -> tensor
99
100 (* Reduction operations *)
101 val sum : tensor -> scalar
102 val sum_reduce : ?axis:int array -> tensor -> tensor
103 val log_sum_exp : ?axis:int -> ?keep_dims:bool -> tensor -> tensor
104 val softmax : ?axis:int -> tensor -> tensor
105
106 (* Scalar-tensor operations *)
107 val scalar_mul : scalar -> tensor -> tensor
108 val sub_scalar : tensor -> scalar -> tensor
109
110 val op_u_to_s: u_to_s -> scalar
111 val op_s_to_s: s_to_s -> scalar -> scalar
112 val op_s's_to_s : s's_to_s -> scalar -> scalar -> scalar
113
114 val op_u_to_t : u_to_t -> tensor
115 val op_t_to_t : t_to_t -> tensor -> tensor
116 val op_t't_to_t : t't_to_t -> tensor -> tensor -> tensor
117
118 val op_t_to_s : t_to_s -> tensor -> scalar
119 val op_s't_to_t : s't_to_t -> scalar -> tensor -> tensor
120 val op_ta_to_t : ta_to_t -> tensor array -> tensor
121 val op_t_to_ta : t_to_ta -> tensor -> tensor array
122
123 val der_s_to_s : s_to_s -> scalar -> (scalar -> scalar)
124 val der_s's_to_s : s's_to_s -> scalar -> scalar -> (scalar -> scalar * scalar)
125
```

```
126   val der_t_to_t : t_to_t -> tensor -> (tensor -> tensor)
127   val der_t't_to_t : t't_to_t -> tensor -> tensor -> (tensor -> tensor * tensor)
128
129   val der_t_to_s : t_to_s -> tensor -> (scalar -> tensor)
130   val der_s't_to_t : s't_to_t -> scalar -> tensor -> (tensor -> scalar * tensor)
131   val der_ta_to_t : ta_to_t -> tensor array -> (tensor -> tensor array)
132   val der_t_to_ta : t_to_ta -> tensor -> (tensor array -> tensor)
133 end
134
135 module type SMOOTH_NON_DIFF = sig
136   type scalar
137   type tensor
138
139   val shape : tensor -> int array
140   val add_ : tensor -> tensor -> unit
141 end
142
143 module Smooth (T : SMOOTH_NON_DIFF) : SMOOTH
144   with type scalar = T.scalar
145   with type tensor = T.tensor
146 = struct
147   include T
148
149   type scalar = T.scalar
150   type tensor = T.tensor
151   type _ Effect.t +=
152       Ap_u_to_s : u_to_s -> scalar Effect.t
153     | Ap_s_to_s : s_to_s * scalar -> scalar Effect.t
154     | Ap_s's_to_s : s's_to_s * scalar * scalar -> scalar Effect.t
155     | Ap_u_to_t : u_to_t -> tensor Effect.t
156     | Ap_t_to_t : t_to_t * tensor -> tensor Effect.t
157     | Ap_t't_to_t : t't_to_t * tensor * tensor -> tensor Effect.t
158     | Ap_t_to_s : t_to_s * tensor -> scalar Effect.t
159     | Ap_s't_to_t : s't_to_t * scalar * tensor -> tensor Effect.t
160     | Ap_ta_to_t : ta_to_t * tensor array -> tensor Effect.t
161     | Ap_t_to_ta : t_to_ta * tensor -> tensor array Effect.t
162
163   let c s = perform (Ap_u_to_s (Const s))
164   let log s = perform (Ap_s_to_s (Log, s))
165   let ( ~. ) s = perform (Ap_s_to_s (Negate, s))
166   let ( +. ) s1 s2 = perform (Ap_s's_to_s (Add, s1, s2))
167   let ( -. ) s1 s2 = perform (Ap_s's_to_s (Subtract, s1, s2))
168   let ( *. ) s1 s2 = perform (Ap_s's_to_s (Multiply, s1, s2))
169   let ( /. ) s1 s2 = perform (Ap_s's_to_s (Divide, s1, s2))
170
171   let zeros ia = perform (Ap_u_to_t (Zeros ia))
172   let create ia s = perform (Ap_u_to_t (Create (ia, s)))
173   let concatenate ?axis ta = perform (Ap_ta_to_t (Concatenate axis, ta))
174   let stack ?axis ta = perform (Ap_ta_to_t (Stack axis, ta))
175   let split ?axis ia t = perform (Ap_t_to_ta (Split (axis, ia), t))
176   let transpose ?axis t = perform (Ap_t_to_t (Transpose axis, t))
177   let reshape t d = perform (Ap_t_to_t (Reshape d, t))
178   let squeeze ?axis t =  perform (Ap_t_to_t (Squeeze axis, t))
```

```
179   let get_slice ill t = perform (Ap_t_to_t (GetSlice ill, t))
180   let slice_left t ia = perform (Ap_t_to_t (SliceLeft ia, t))
181   let get t ia = perform (Ap_t_to_s (Get ia, t))
182   let set_slice ill t1 t2 = perform (Ap_t't_to_t (SetSlice ill, t1, t2))
183   let einsum_ijk_mik_to_mij a x =
184     perform (Ap_t't_to_t (Einsum_ijk_mik_to_mij, a, x))
185   let einsum_ijk_mij_to_mik a y =
186     perform (Ap_t't_to_t (Einsum_ijk_mij_to_mik, a, y))
187   let einsum_mij_mik_to_ijk y x =
188     perform (Ap_t't_to_t (Einsum_mij_mik_to_ijk, y, x))
189   let exp t = perform (Ap_t_to_t (Exp, t))
190   let ( ~- ) t = perform (Ap_t_to_t (Negate, t))
191   let pow_const t f = perform (Ap_t_to_t (PowerConst f,t))
192   let ( + ) t1 t2 = perform (Ap_t't_to_t (Add, t1, t2))
193   let ( - ) t1 t2 = perform (Ap_t't_to_t (Subtract, t1, t2))
194   let ( * ) t1 t2 = perform (Ap_t't_to_t (Multiply, t1, t2))
195   let ( / ) t1 t2 = perform (Ap_t't_to_t (Divide, t1, t2))
196   let sum t = perform (Ap_t_to_s (Sum, t))
197   let sum_reduce ?axis t = perform (Ap_t_to_t (SumReduce axis, t))
198   let log_sum_exp ?axis ?keep_dims t =
199     perform (Ap_t_to_t (LogSumExp (axis, keep_dims), t))
200   let softmax ?axis t = perform (Ap_t_to_t (Softmax axis, t))
201   let scalar_mul s t = perform (Ap_s't_to_t (ScalarMultiply, s, t))
202   let sub_scalar t s = perform (Ap_s't_to_t (SubtractScalar, s, t))
203
204   (* Simple expand operation. ia contains which axes to expand. *)
205   let _expand t shp ia =
206     let res = ref t in
207     for j = 0 to Stdlib.(Array.length ia - 1) do
208       res := concatenate ~axis:(ia.(j)) (Array.make shp.(ia.(j)) !res)
209     done;
210     !res
211
212   (* Inverse of a permutation *)
213   let _inv_perm p =
214     let l = Array.length p in
215     let q = Array.make l 0 in
216     for i = 0 to Stdlib.(l - 1) do
217       q.(p.(i)) <- i;
218     done;
219     q
220
221   let op_u_to_s (o : u_to_s) = match o with
222     | Const x -> c x
223   let op_s_to_s (o : s_to_s) s = match o with
224     | Negate -> ~. s
225     | Log -> log s
226   let op_s's_to_s (o : s's_to_s) s1 s2 = match o with
227     | Add -> s1 +. s2
228     | Subtract -> s1 -. s2
229     | Multiply -> s1 *. s2
230     | Divide -> s1 /. s2
231
```

```
232   let op_u_to_t (o : u_to_t) = match o with
233     | Zeros ia -> zeros ia
234     | Create (ia, f) -> create ia f
235   let op_t_to_t (o : t_to_t) t = match o with
236     | Squeeze iao -> squeeze ?axis:iao t
237     | Reshape d -> reshape t d
238     | GetSlice ill -> get_slice ill t
239     | SliceLeft ia -> slice_left t ia
240     | Transpose iao -> transpose ?axis:iao t
241     | Exp -> exp t
242     | Negate -> ~- t
243     | PowerConst f -> pow_const t f
244     | SumReduce iao -> sum_reduce ?axis:iao t
245     | LogSumExp (io, bo) -> log_sum_exp ?axis:io ?keep_dims:bo t
246     | Softmax io -> softmax ?axis:io t
247   let op_t't_to_t (o : t't_to_t) t1 t2 = match o with
248     | Add -> t1 + t2
249     | Subtract -> t1 - t2
250     | Multiply -> t1 * t2
251     | Divide -> t1 / t2
252     | Einsum_ijk_mik_to_mij -> einsum_ijk_mik_to_mij t1 t2
253     | Einsum_ijk_mij_to_mik -> einsum_ijk_mij_to_mik t1 t2
254     | Einsum_mij_mik_to_ijk -> einsum_mij_mik_to_ijk t1 t2
255     | SetSlice ill -> set_slice ill t1 t2
256
257   let op_t_to_s (o : t_to_s) t = match o with
258     | Get ia -> get t ia
259     | Sum -> sum t
260   let op_s't_to_t (o : s't_to_t) s t = match o with
261     | ScalarMultiply -> scalar_mul s t
262     | SubtractScalar -> sub_scalar t s
263   let op_ta_to_t (o : ta_to_t) ta = match o with
264     | Concatenate io ->  concatenate ?axis:io ta
265     | Stack io -> stack ?axis:io ta
266   let op_t_to_ta (o : t_to_ta) t = match o with
267     | Split (io, ia) -> split ?axis:io ia t
268
269   let der_s_to_s (o : s_to_s) s = match o with
270     | Negate -> fun sd -> ~. sd
271     | Log -> fun sd -> sd /. s
272   let der_s's_to_s (o : s's_to_s) s1 s2 = match o with
273     | Add -> fun sd -> (sd, sd)
274     | Subtract -> fun sd -> (sd, ~. sd)
275     | Multiply -> fun sd -> (s2 *. sd, s1 *. sd)
276     | Divide -> fun sd -> (sd /. s2, (sd *. (~. s1)) /. (s2 *. s2))
277
278   let der_t_to_t (o : t_to_t) t = match o with
279     | Squeeze _ -> fun td -> reshape td (shape t)
280     | Reshape _ -> fun td -> reshape td (shape t)
281     | GetSlice ill -> fun td -> set_slice ill (zeros (shape t)) td
282     | SliceLeft ia -> fun td ->
283       let ill = Array.to_list (Array.map (fun i -> [i]) ia) in
284       let shp = Array.(append (make (length ia) 1) (shape td)) in
```

```
285       let tdr = reshape td shp in
286       set_slice ill (zeros (shape t)) tdr
287     | Transpose iao ->
288       let ia = match iao with
289         | None ->
290           let d = Array.length (shape t) in
291           Array.init d Stdlib.(fun i -> d - i - 1)
292         | Some ia -> ia
293       in
294       fun td -> transpose ~axis:(_inv_perm ia) td
295     | Exp -> fun td -> exp t * td
296     | Negate -> fun td -> ~- td
297     | PowerConst f -> fun td ->
298       scalar_mul (c f) (td * pow_const t Stdlib.(f -. 1.0))
299     | SumReduce iao ->
300       let ia = (match iao with
301         | None -> Array.init (Array.length (shape t)) (fun i -> i)
302         | Some ia -> ia
303       ) in
304       fun td -> _expand td (shape t) ia
305     | LogSumExp (io, bo) -> (
306       let (i, b) = match (io, bo) with
307         | (None, None) -> (0, true)
308         | (Some i, None) -> (i, true)
309         | (None, Some b) -> (0, b)
310         | (Some i, Some b) -> (i, b)
311       in
312       if b
313         then fun td -> td * softmax ~axis:i t
314         else fun td ->
315           let shp = shape t in
316           shp.(i) <- 1;
317           (reshape td shp) * (softmax ~axis:i t)
318       )
319     | Softmax _io -> raise (Invalid_argument "Softmax not implemented")
320   let der_t't_to_t (o : t't_to_t) t1 t2 = match o with
321     | Add -> fun td -> (td, td)
322     | Subtract -> fun td -> (td, ~- td)
323     | Multiply -> fun td -> (t2 * td, t1 * td)
324     | Divide -> fun td -> (td / t2, (td * (~- t1)) / (t2 * t2))
325     | Einsum_ijk_mik_to_mij -> fun td ->
326       (einsum_mij_mik_to_ijk td t2, einsum_ijk_mij_to_mik t1 td)
327     | Einsum_ijk_mij_to_mik -> fun td ->
328       (einsum_ijk_mik_to_mij t1 td, einsum_mij_mik_to_ijk t2 td)
329     | Einsum_mij_mik_to_ijk -> fun td ->
330       (einsum_ijk_mik_to_mij td t2, einsum_ijk_mij_to_mik td t1)
331     | SetSlice ill -> fun td ->
332       (set_slice ill td (zeros (shape t2)), get_slice ill td)
333
334   let der_t_to_s (o : t_to_s) t = match o with
335     | Get ia ->
336       let ill = Array.to_list (Array.map (fun i -> [i]) ia) in
337       (fun sd ->
```

```
338         let ones = Array.(make (length (shape t)) 1) in
339         set_slice ill (zeros (shape t)) (scalar_mul sd (create ones 1.0))
340       )
341     | Sum -> fun sd -> scalar_mul sd (create (shape t) 1.0)
342   let der_s't_to_t (o : s't_to_t) s t = match o with
343     | ScalarMultiply -> fun td -> (sum (t * td), scalar_mul s td)
344     | SubtractScalar -> fun td -> (~. (sum td), td)
345   let der_ta_to_t (o : ta_to_t) ta = match o with
346     | Concatenate io ->
347       let i = (match io with
348         | None -> 0
349         | Some i -> i
350       ) in
351       fun td -> split ~axis:i (Array.map (fun x -> (shape x).(i)) ta) td
352     | Stack io ->
353       let i = (match io with
354         | None -> 0
355         | Some i -> i
356       ) in
357       (fun td ->
358         let shp = shape td in
359         let ndim = Array.length shp in
360         let axis = Owl_utils.adjust_index i ndim in
361         let inp_shp = shape ta.(0) in
362         split ~axis:i (Array.make shp.(axis) 1) td
363           |> Array.map (fun x -> reshape x inp_shp)
364       )
365   let der_t_to_ta (o : t_to_ta) _ = match o with
366     | Split (io, _) ->
367       let i = (match io with
368         | None -> 0
369         | Some i -> i
370       ) in
371       fun tda -> concatenate ~axis:i tda
372 end
```

Listing A.36: Stateful reverse mode, tensors (OCaml)

```
1  open Effect.Deep
2  open Modules_effect_handlers_smooth_tensor
3
4  type 't prop = {v : 't; mutable dv : 't}
5
6  module Reverse_Non_Diff (T : SMOOTH_NON_DIFF) : SMOOTH_NON_DIFF
7    with type scalar = T.scalar prop
8    with type tensor = T.tensor prop
9  = struct
10   type scalar = T.scalar prop
11   type tensor = T.tensor prop
12
13   let shape t = T.shape t.v
14   let add_ x dx = T.add_ x.v dx.v; T.add_ x.dv dx.dv
15 end
```

```
16
17 module Reverse (T : SMOOTH) = struct
18   include Smooth (Reverse_Non_Diff (T : SMOOTH_NON_DIFF))
19
20   let reverse = {
21     retc = (fun x -> x);
22     exnc = raise;
23     effc = (fun (type a) (eff : a Effect.t) ->
24       match eff with
25       | Ap_u_to_s o -> Some (fun (k : (a, _) continuation) -> let open T in
26           continue k {v = op_u_to_s o; dv = c 0.0}
27         )
28       | Ap_s_to_s (o, s) -> Some (fun k -> let open T in
29           let r = {v = op_s_to_s o s.v; dv = c 0.0} in
30           continue k r;
31           s.dv <- s.dv +. (der_s_to_s o s.v r.dv)
32         )
33       | Ap_s's_to_s (o, s1, s2) -> Some (fun k -> let open T in
34           let r = {v = op_s's_to_s o s1.v s2.v; dv = c 0.0} in
35           continue k r;
36           let (dv1, dv2) = der_s's_to_s o s1.v s2.v r.dv in
37           s1.dv <- s1.dv +. dv1;
38           s2.dv <- s2.dv +. dv2
39         )
40       | Ap_u_to_t o -> Some (fun k -> let open T in
41           let v = op_u_to_t o in
42           continue k {v = v; dv = create (shape v) 0.0}
43         )
44       | Ap_t_to_t (o, t) -> Some (fun k -> let open T in
45           let v = op_t_to_t o t.v in
46           let r = {v = v; dv = create (shape v) 0.0} in
47           continue k r;
48           let dv = der_t_to_t o t.v r.dv in
49           if shape t.dv = shape dv then add_ t.dv dv else t.dv <- t.dv + dv
50         )
51       | Ap_t't_to_t (o, t1, t2) -> Some (fun k -> let open T in
52           let v = op_t't_to_t o t1.v t2.v in
53           let r = {v = v; dv = create (shape v) 0.0} in
54           continue k r;
55           let (dv1, dv2) = der_t't_to_t o t1.v t2.v r.dv in
56           if shape t1.dv = shape dv1
57             then add_ t1.dv dv1 else t1.dv <- t1.dv + dv1;
58           if shape t2.dv = shape dv2
59             then add_ t2.dv dv2 else t2.dv <- t2.dv + dv2
60         )
61       | Ap_t_to_s (o, t) -> Some (fun k -> let open T in
62           let r = {v = op_t_to_s o t.v; dv = c 0.0} in
63           continue k r;
64           let dv = der_t_to_s o t.v r.dv in
65           if shape t.dv = shape dv then add_ t.dv dv else t.dv <- t.dv + dv
66         )
67       | Ap_s't_to_t (o, s, t) -> Some (fun k -> let open T in
68           let v = op_s't_to_t o s.v t.v in
```

```
69            let r = {v = v; dv = create (shape v) 0.0} in
70            continue k r;
71            let (ds, dt) = der_s't_to_t o s.v t.v r.dv in
72            s.dv <- s.dv +. ds;
73            if shape t.dv = shape dt then add_ t.dv dt else t.dv <- t.dv + dt
74          )
75        | Ap_ta_to_t (o, ta) -> Some (fun k -> let open T in
76            let tva = Array.(map (fun t -> t.v) ta) in
77            let v = op_ta_to_t o tva in
78            let r = {v = v; dv = create (shape v) 0.0} in
79            continue k r;
80            let rdva = der_ta_to_t o tva r.dv in
81            ignore Array.(map2 (fun t rdv -> (
82              if shape t.dv = shape rdv then add_ t.dv rdv else t.dv <- t.dv + rdv
83            )) ta rdva)
84          )
85        | Ap_t_to_ta (o, t) -> Some (fun k -> let open T in
86            let va = op_t_to_ta o t.v in
87            let ra =
88              Array.(map (fun v -> {v = v; dv = create (shape v) 0.0}) va)
89            in
90            continue k ra;
91            let rdva = Array.(map (fun r -> r.dv) ra) in
92            let dv = der_t_to_ta o t.v rdva in
93            if shape t.dv = shape dv then add_ t.dv dv else t.dv <- t.dv + dv
94          )
95        | _ -> None
96      )
97    }
98
99    let grad f ta =
100     let ra = Array.map (fun t -> {v = t; dv = T.(create (shape t) 0.0)}) ta in
101     match_with (fun ta -> (f ta).dv <- T.c 1.0) ra reverse;
102     Array.map (fun r -> r.dv) ra
103 end
```

Listing A.37: GMM objective function (OCaml)

```
1  open Shared_gmm_data
2  open Shared_gmm_types
3
4  module type GMM_OBJECTIVE = sig
5    type tensor
6    type scalar
7
8    val gmm_objective : (scalar, tensor) gmm_input -> scalar
9  end
10
11 module Make
12   (S : GMM_SCALAR)
13   (T : GMM_TENSOR with type scalar = S.t) : GMM_OBJECTIVE
14   with type tensor = T.t
15   with type scalar = S.t
```

```
16  = struct
17    type tensor = T.t
18    type scalar = S.t
19    open S
20    open T
21
22    let ( + ) = add
23    let ( - ) = sub
24    let ( * ) = mul
25
26    let constructl d icfs =
27      let lparamidx = ref d in
28
29      let make_l_col i =
30        let nelems = Stdlib.(d - i - 1) in
31        (* Slicing in Owl requires inculsive indices, so will not create
32        * an empty tensor. Thus we have two cases.
33        *)
34        let max_lparamidx = (shape icfs).(0) in
35        let col =
36          if Stdlib.(!lparamidx >= max_lparamidx) then
37            zeros [|Stdlib.(i + 1)|]
38          else concatenate ~axis:0 [|
39            zeros [|Stdlib.(i + 1)|];
40            get_slice [[!lparamidx; Stdlib.(!lparamidx + nelems - 1)]] icfs;
41        |] in
42        lparamidx := Stdlib.(!lparamidx + nelems);
43        col
44      in
45
46      let columns = Array.init d make_l_col in
47      stack ~axis:1 columns
48
49    let qtimesx qdiag l x =
50      let y = einsum_ijk_mik_to_mij l x in
51      (qdiag * x) + y
52
53    let log_gamma_distrib a p =
54      Stdlib.(
55      let scalar = (0.25 *. (float_of_int (p * (p - 1))) *. log Float.pi) in
56      let summed = Array.fold_left (+.) 0.0
57        (Array.init p (fun i ->
58          Owl.Maths.loggamma (a +. 0.5 *. (float_of_int (1 - (i + 1))))
59        ))
60      in
61      scalar +. summed
62      )
63
64    let log_wishart_prior p wishart sum_qs qdiags icf =
65      let n = float_of_int (Stdlib.(p + wishart.m + 1)) in
66      let k = float_of_int ((shape icf).(0)) in
67
68      let out = sum_reduce (
```

```
69        (
70          scalar_mul (S.float 0.5 *. wishart.gamma *. wishart.gamma)
71            (squeeze (
72              sum_reduce ~axis:[|1|] (pow_const qdiags 2.0) +
73              sum_reduce ~axis:[|1|] (pow_const (get_slice [[]; [p;-1]] icf) 2.0)
74            ))
75        )
76        - (scalar_mul (S.float (float_of_int wishart.m)) sum_qs)
77      ) in
78
79      let c =
80        S.float n
81        *. S.float (float_of_int p)
82        *. (log (wishart.gamma /. S.float (Stdlib.sqrt 2.0)))
83      in
84      sub_scalar
85        out
86        (S.float k *. (c -. S.float (log_gamma_distrib Stdlib.(0.5 *. n) p)))
87
88    let gmm_objective param =
89      let xshape = shape param.x in
90      let n = xshape.(0) in
91      let d = xshape.(1) in
92      let k = (shape param.means).(0) in
93
94      let qdiags = exp (get_slice [[]; [0; Stdlib.(d - 1)]] param.icfs) in
95      let sqdiags = stack (Array.make n qdiags) in
96      let sum_qs = squeeze (
97        sum_reduce ~axis:[|1|] (get_slice [[]; [0; Stdlib.(d - 1)]] param.icfs)
98      ) in
99      (* Prevent implicit broadcasting *)
100     let ssum_qs = stack (Array.make n sum_qs) in
101
102     let icf_sz = (shape param.icfs).(0) in
103     let ls = stack (Array.init icf_sz (fun i ->
104       constructl d (slice_left param.icfs [|i|]))
105     ) in
106
107     let xcentered = squeeze (stack (Array.init n (fun i ->
108       let sx = slice_left param.x [|i|] in
109       (* Prevent implicit broadcasting *)
110       let ssx = stack (Array.make k sx) in
111       ssx - param.means
112     ))) in
113     let lxcentered = qtimesx sqdiags ls xcentered in
114     let sqsum_lxcentered = squeeze (
115       sum_reduce ~axis:[|2|] (pow_const lxcentered 2.0)
116     ) in
117     (* Prevent implicit broadcasting *)
118     let salphas = stack (Array.make n param.alphas) in
119     let inner_term =
120       salphas + ssum_qs - (scalar_mul (S.float 0.5) sqsum_lxcentered)
121     in
```

```
122     (* Uses the stable version as in the paper, i.e. max-shifted *)
123     let lse = squeeze (log_sum_exp ~axis:1 inner_term) in
124     let slse = sum_reduce lse in
125
126     let const = create [||] Stdlib.(
127       -. (float_of_int n) *. (float_of_int d) *. 0.5 *. log (2.0 *. Float.pi)
128     ) in
129
130     let wish = log_wishart_prior d param.wishart sum_qs qdiags param.icfs in
131     get (
132       const + slse
133             - scalar_mul (S.float (float_of_int n)) (squeeze (log_sum_exp param.alphas))
134             + wish
135     ) [|0|]
136 end
```

A.11 Frank Patch

Listing A.38: Patch for Frank codebase

```
1  diff --git a/.gitignore b/.gitignore
2  index ab16fa5..07fd4b5 100644
3  --- a/.gitignore
4  +++ b/.gitignore
5  @@ -4,3 +4,5 @@
6
7   # Ignore all the stuff generated by Stack
8   .stack-work/
9  +
10 +.vscode/
11 diff --git a/Compile.hs b/Compile.hs
12 index eb9e9d5..7269530 100644
13 --- a/Compile.hs
14 +++ b/Compile.hs
15 @@ -156,6 +156,7 @@ compileVPat ((StrPat s a) :: ValuePat Desugared) =
       ↪ compileVPat (compileStrPat s)
16     compileStrPat []     = DataPat "nil" [] a
17     compileStrPat (c:cs) = DataPat "cons" [CharPat c a, compileStrPat cs] a
18   compileVPat (CharPat c _) = return $ S.VPX [Left c]
19 +compileVPat (FloatPat f _) = return $ S.VPD f
20
21   compileTm :: Tm Desugared -> Compile S.Exp
22   compileTm (SC sc _) = compileSComp sc
23 @@ -165,6 +166,7 @@ compileTm (StrTm s a) = compileDataCon (f s) where
24     f [] = DataCon "nil" [] a
25     f (c:cs) = DataCon "cons" [CharTm c a, DCon (f cs) a] a
26   compileTm (IntTm n _) = return $ S.EI n
27 +compileTm (FloatTm f _) = return $ S.ED f
28   compileTm (CharTm c _) = return $ S.EX [Left c]
29   compileTm (TmSeq t1 t2 _) = (S.:!) <$> compileTm t1 <*> compileTm t2
30   compileTm (Use u _) = compileUse u
31 @@ -203,10 +205,24 @@ compileOp (CmdId id _) = return $ S.EA id
```

```
32
33   builtins :: M.Map String String
34   builtins = M.fromList [("+", "plus")
35  +                      ,("*", "times")
36                         ,("-", "minus")
37  +                      ,("*", "mult")
38                         ,("eqc" , "eqc")
39                         ,(">", "gt")
40  -                      ,("<", "lt")]
41  +                      ,("<", "lt")
42  +                      ,("==", "eqN")
43  +                      -- floats
44  +                      ,(">.", "gtF")
45  +                      ,("<.", "ltF")
46  +                      ,("+.", "plusF")
47  +                      ,("-.", "minusF")
48  +                      ,("*.", "multF")
49  +                      ,("/.", "divF")
50  +                      ,("==.", "eqF")
51  +                      ,("round", "roundF")
52  +                      ,("toFloat", "toFloat")
53  +                      ]
54
55   isBuiltin :: String -> Bool
56   isBuiltin x = M.member x builtins
57  diff --git a/Debug.hs b/Debug.hs
58  index 9a171e4..e6cf2c7 100644
59  --- a/Debug.hs
60  +++ b/Debug.hs
61  @@ -370,6 +370,7 @@ ppVType (FTVar x _) = if isDebugVerboseOn () then text x
        ↪ else text $ trimVar x
62   ppVType (StringTy _) = text "String"
63   ppVType (IntTy _) = text "Int"
64   ppVType (CharTy _) = text "Char"
65  +ppVType (FloatTy _) = text "Float"
66
67   ppTyArg :: (Show a, HasSource a) => TyArg a -> PP.Doc
68   ppTyArg (VArg t _) = ppParenVType t
69  diff --git a/DesugarSyntax.hs b/DesugarSyntax.hs
70  index a0c7391..051faa1 100644
71  --- a/DesugarSyntax.hs
72  +++ b/DesugarSyntax.hs
73  @@ -162,6 +162,7 @@ desugarVType (SCTy ty a) = do ty' <- desugarCType ty
74   desugarVType (StringTy a) = return $ desugaredStrTy (refToDesug a)
75   desugarVType (IntTy a) = return $ IntTy (refToDesug a)
76   desugarVType (CharTy a) = return $ CharTy (refToDesug a)
77  +desugarVType (FloatTy a) = return $ FloatTy (refToDesug a)
78
79   -- nothing happens on this level
80   desugarTyArg :: TyArg Refined -> Desugar (TyArg Desugared)
81  @@ -221,6 +222,7 @@ desugarTm (SC x a) = SC <$> desugarSComp x <*> pure (
        ↪ refToDesug a)
82   desugarTm (StrTm s a) = return $ StrTm s (refToDesug a)
```

```
 desugarTm (IntTm n a) = return $ IntTm n (refToDesug a)
 desugarTm (CharTm c a) = return $ CharTm c (refToDesug a)
+desugarTm (FloatTm f a) = return $ FloatTm f (refToDesug a)
 desugarTm (TmSeq tm1 tm2 a) = TmSeq <$> desugarTm tm1 <*> desugarTm tm2 <*>
    ↪ pure (refToDesug a)
 desugarTm (Use u a) = Use <$> desugarUse u <*> pure (refToDesug a)
 desugarTm (DCon d a) = DCon <$> desugarDCon d <*> pure (refToDesug a)
@@ -239,6 +241,7 @@ desugarVPat (DataPat x xs a) =
 desugarVPat (IntPat i a) = return $ IntPat i (refToDesug a)
 desugarVPat (CharPat c a) = return $ CharPat c (refToDesug a)
 desugarVPat (StrPat s a) = return $ StrPat s (refToDesug a)
+desugarVPat (FloatPat s a) = return $ FloatPat s (refToDesug a)

 desugarSComp :: SComp Refined -> Desugar (SComp Desugared)
 desugarSComp (SComp xs a) =
diff --git a/Parser.hs b/Parser.hs
index ed8b0cd..6f25a2f 100644
--- a/Parser.hs
+++ b/Parser.hs
@@ -278,6 +278,7 @@ vtype' :: MonadicParsing m => m (VType Raw)
 vtype' = parens vtype <|>
               (attachLoc $ SCTy <$> try ctype) <|>
               (attachLoc $ StringTy <$ reserved "String") <|>
+              (attachLoc $ FloatTy <$ reserved "Float") <|>
               (attachLoc $ IntTy <$ reserved "Int") <|>
               (attachLoc $ CharTy <$ reserved "Char") <|>
               -- could possibly also be a MkDTTy (determined during refinement)
@@ -351,12 +352,13 @@ usetm = (attachLoc $ Use <$> (try $ use nctm)) <|>    --
    ↪ use

 -- atomic term
 atm :: MonadicParsing m => m (Tm Raw)
-atm = (attachLoc $ SC <$> suspComp) <|>               -- { p_1 -> t_1 | ... }
-      (attachLoc $ StrTm <$> stringLiteral) <|>       -- "string"
+atm = (attachLoc $ SC <$> suspComp) <|>                       -- { p_1 -> t_1 |
    ↪  ... }
+      (attachLoc $ StrTm <$> stringLiteral) <|>               -- "string"
+      (attachLoc $ FloatTm <$> try double) <|>                    -- 3.14
       (attachLoc $ (IntTm . fromIntegral) <$> natural) <|>    -- 42
-      (attachLoc $ CharTm <$> charLiteral) <|>        -- 'c'
-      (attachLoc $ ListTm <$> listTm) <|>             -- [t_1, ..., t_n]
-      parens tm                                       -- (ltm ; ... ; ltm)
+      (attachLoc $ CharTm <$> charLiteral) <|>                -- 'c'
+      (attachLoc $ ListTm <$> listTm) <|>                     -- [t_1, ..., t_n
    ↪ ]
+      parens tm                                               -- (ltm ; ... ;
    ↪ ltm)

 letTm :: MonadicParsing m => m (Tm Raw) -> m (Tm Raw) -> m (Tm Raw)
 letTm p p' = attachLoc $ do reserved "let"
@@ -368,21 +370,33 @@ letTm p p' = attachLoc $ do reserved "let"
                             return $ Let x t t'
```

```
131  binOpLeft :: MonadicParsing m => m (Use Raw)
132 -binOpLeft = attachLoc $ do op <- choice $ map symbol ["+","-","*","/",">","<"]
133 +binOpLeft = attachLoc $ do op <- choice $ map symbol ((map (\x -> x ++ "."))
        ↪ arithOps ++ arithOps)
134                             return $ RawId op
135 +  where
136 +    arithOps = ["+","-","*","/",">","<", "=="]
137
138  binOpRight :: MonadicParsing m => m (Use Raw)
139 -binOpRight = attachLoc $ do op <- choice $ map symbol ["::"]
140 -                            let op' = if op == "::" then "cons" else op
141 +binOpRight = attachLoc $ do op <- choice $ map symbol ["::",":="]
142 +                            let op' = case op of {"::" -> "cons"; ":=" -> "
        ↪ write"; _ -> op}
143                             return $ RawId op'
144
145  -- unary operation
146  unOperation :: (MonadicParsing m) => m (Tm Raw)
147 -unOperation = provideLoc $ \a -> do
148 +unOperation = try negInt <|>
149 +              negFloat
150 +
151 +negInt :: (MonadicParsing m) => m (Tm Raw)
152 +negInt = provideLoc $ \a -> do
153                  symbol "-"
154                  t <- untm
155                  return $ Use (RawComb (RawId "-" a) [IntTm 0 a, t] a) a
156
157 +negFloat :: (MonadicParsing m) => m (Tm Raw)
158 +negFloat = provideLoc $ \a -> do
159 +                symbol "-."
160 +                t <- untm
161 +                return $ Use (RawComb (RawId "-." a) [FloatTm 0.0 a, t] a) a
162 +
163  -- use
164  use :: MonadicParsing m => m (Tm Raw) -> m (Use Raw)
165  use p = adapted (ncuse p) <|>                       -- <adp_1,...adp_n> ncuse
166 @@ -408,8 +422,13 @@ ncuse p = provideLoc $ \a -> do
167
168  -- atomic use
169  ause :: MonadicParsing m => m (Tm Raw) -> m (Use Raw)
170 -ause p = parens (use p) <|>                            -- (use)
171 -         idUse                                        -- x
172 +ause p = provideLoc $ \a -> do
173 +          deref <- optional (symbol "@")
174 +          t <- parens (use p) <|>                          -- (use)
175 +               idUse                                       -- x
176 +          return $ case deref of
177 +            Nothing -> t
178 +            Just _ -> RawComb (RawId "read" a) [Use t a] a
179
180  adapted :: MonadicParsing m => m (Use Raw) -> m (Use Raw)
181  adapted p = attachLoc $ do -- <adp_1,adp_2,...,adp_n> stm
```

```
@@ -494,6 +513,7 @@ valPat :: MonadicParsing m => m (ValuePat Raw)
 valPat = dataPat <|>
          (attachLoc $ do x <- identifier
                          return $ VarPat x) <|>
+         (attachLoc $ FloatPat <$> try double) <|>
          (attachLoc $ IntPat <$> try parseInt) <|> -- try block for unary minus
          (attachLoc $ CharPat <$> charLiteral) <|>
          (attachLoc $ StrPat <$> stringLiteral) <|>
diff --git a/ParserCommon.hs b/ParserCommon.hs
index 8edc032..68ad1cb 100644
--- a/ParserCommon.hs
+++ b/ParserCommon.hs
@@ -46,7 +46,7 @@ frankStyle = IdentifierStyle {
   , _styleLetter = satisfy (\c -> isAlphaNum c || c == '_' || c == '\'')
   , _styleReserved = HashSet.fromList [ "data", "interface"
                                       , "let", "in"
-                                      , "String", "Int", "Char"]
+                                      , "String", "Int", "Char", "Float"]
   , _styleHighlight = Hi.Identifier
   , _styleReservedHighlight = Hi.ReservedIdentifier }

diff --git a/RefineSyntax.hs b/RefineSyntax.hs
index a4f74a6..7757fa7 100644
--- a/RefineSyntax.hs
+++ b/RefineSyntax.hs
@@ -320,6 +320,7 @@ refineVType (TVar x a) =
 refineVType (StringTy a) = return $ StringTy (rawToRef a)
 refineVType (IntTy a) = return $ IntTy (rawToRef a)
 refineVType (CharTy a) = return $ CharTy (rawToRef a)
+refineVType (FloatTy a) = return $ FloatTy (rawToRef a)

 refineTyArg :: TyArg Raw -> Refine (TyArg Refined)
 refineTyArg (VArg t a) = VArg <$> refineVType t <*> pure (rawToRef a)
@@ -406,6 +407,7 @@ refineTm (SC x a) = do x' <- refineSComp x
 refineTm (StrTm x a) = return $ StrTm x (rawToRef a)
 refineTm (IntTm x a) = return $ IntTm x (rawToRef a)
 refineTm (CharTm x a) = return $ CharTm x (rawToRef a)
+refineTm (FloatTm x a) = return $ FloatTm x (rawToRef a)
 refineTm (ListTm ts a) =
   do ts' <- mapM refineTm ts
      return $
@@ -462,6 +464,7 @@ refineVPat (DataPat x xs a) =
 refineVPat (IntPat i a) = return $ IntPat i (rawToRef a)
 refineVPat (CharPat c a) = return $ CharPat c (rawToRef a)
 refineVPat (StrPat s a) = return $ StrPat s (rawToRef a)
+refineVPat (FloatPat f a) = return $ FloatPat f (rawToRef a)
 refineVPat (ConsPat x xs a) =
   do x' <- refineVPat x
      xs' <- refineVPat xs
@@ -488,16 +491,6 @@ initialiseRState dts itfs itfAls =
      putRCmds       cmds'
      putRCtrs       ctrs'

```

```
-makeIntBinOp :: Refined -> Char -> MHDef Refined
-makeIntBinOp a c = Def [c] (CType [Port [] (IntTy a) a
-                                   ,Port [] (IntTy a) a]
-                                   (Peg (Ab (AbVar "Âč" a) (ItfMap M.empty a) a)
    ↪  (IntTy a) a) a) [] a
-
-makeIntBinCmp :: Refined -> Char -> MHDef Refined
-makeIntBinCmp a c = Def [c] (CType [Port [] (IntTy a) a
-                                    ,Port [] (IntTy a) a]
-                              (Peg (Ab (AbVar "Âč" a) (ItfMap M.empty a) a)
-                               (DTTy "Bool" [] a) a) a) [] a

 {-- The initial state for the refinement pass. -}

@@ -528,10 +521,42 @@ builtinItfs = [Itf "Console" [] [Cmd "inch" [] [] (CharTy
    ↪ b) b
 builtinItfAliases :: [ItfAlias Raw]
 builtinItfAliases = []

-builtinMHDefs :: [MHDef Refined]
-builtinMHDefs = map (makeIntBinOp (Refined BuiltIn)) "+-" ++
-                map (makeIntBinCmp (Refined BuiltIn)) "><" ++
-                [caseDef, charEq, alphaNumPred]
+makeIntBinOp :: Refined -> Char -> MHDef Refined
+makeIntBinOp a c = Def [c] (CType [Port [] (IntTy a) a
+                                   ,Port [] (IntTy a) a]
+                                   (Peg (Ab (AbVar "Âč" a) (ItfMap M.empty a) a)
    ↪  (IntTy a) a) a) [] a
+
+makeIntBinCmp :: Refined -> Char -> MHDef Refined
+makeIntBinCmp a c = Def [c] (CType [Port [] (IntTy a) a
+                                    ,Port [] (IntTy a) a]
+                              (Peg (Ab (AbVar "Âč" a) (ItfMap M.empty a) a)
+                               (DTTy "Bool" [] a) a) a) [] a
+
+-- as above, but we now use Flaot instead.
+makeFloatBinOp :: Refined -> Char -> MHDef Refined
+makeFloatBinOp a c = Def (c : ".") (CType [Port [] (FloatTy a) a
+                                          ,Port [] (FloatTy a) a]
+                                          (Peg (Ab (AbVar "Âč" a) (ItfMap M.
    ↪ empty a) a) (FloatTy a) a) a) [] a
+
+makeFloatBinCmp :: Refined -> Char -> MHDef Refined
+makeFloatBinCmp a c = Def (c : ".") (CType [Port [] (FloatTy a) a
+                                           ,Port [] (FloatTy a) a]
+                                           (Peg (Ab (AbVar "Âč" a) (ItfMap M.
    ↪ empty a) a)
+                                                (DTTy "Bool" [] a) a) a) [] a
+
+intEq :: MHDef Refined
+intEq = Def "==" (CType [Port [] (IntTy a) a
+                         ,Port [] (IntTy a) a]
+                         (Peg (Ab (AbVar "Âč" a) (ItfMap M.empty a) a)
```

```
283 +                                (DTTy "Bool" [] a) a) a) [] a
284 +  where a = Refined BuiltIn
285 +
286 +floatEq :: MHDef Refined
287 +floatEq = Def "==." (CType [Port [] (FloatTy a) a
288 +                           ,Port [] (FloatTy a) a]
289 +                           (Peg (Ab (AbVar "Âč" a) (ItfMap M.empty a) a)
290 +                                (DTTy "Bool" [] a) a) a) [] a
291 +  where a = Refined BuiltIn
292
293  charEq :: MHDef Refined
294  charEq = Def "eqc" (CType [Port [] (CharTy a) a
295 @@ -540,6 +565,13 @@ charEq = Def "eqc" (CType [Port [] (CharTy a) a
296                                 (DTTy "Bool" [] a) a) a) [] a
297    where a = Refined BuiltIn
298
299 +refEq :: MHDef Refined
300 +refEq = Def "eqR" (CType [Port [] (DTTy "Ref" [VArg (TVar "X" a) a] a) a
301 +                           ,Port [] (DTTy "Ref" [VArg (TVar "X" a) a] a) a]
302 +                           (Peg (Ab (AbVar "Âč" a) (ItfMap M.empty a) a)
303 +                                (DTTy "Bool" [] a) a) a) [] a
304 +  where a = Refined BuiltIn
305 +
306  alphaNumPred :: MHDef Refined
307  alphaNumPred = Def "isAlphaNum"
308                  (CType [Port [] (CharTy a) a]
309 @@ -547,6 +579,23 @@ alphaNumPred = Def "isAlphaNum"
310                                 (DTTy "Bool" [] a) a) a) [] a
311    where a = Refined BuiltIn
312
313 +chrFunc :: MHDef Refined
314 +chrFunc = Def "chr"
315 +               (CType [Port [] (IntTy a) a]
316 +                           (Peg (Ab (AbVar "Âč" a) (ItfMap M.empty a) a)
317 +                                (CharTy a) a) a) [] a
318 +  where a = Refined BuiltIn
319 +
320 +roundMH :: MHDef Refined
321 +roundMH = Def "round" (CType [Port [] (FloatTy a) a]
322 +                            (Peg (Ab (AbVar "Âč" a) (ItfMap M.empty a) a) (IntTy
    ↪  a) a) a) [] a
323 +  where a = Refined BuiltIn
324 +
325 +toFloat :: MHDef Refined
326 +toFloat = Def "toFloat" (CType [Port [] (IntTy a) a]
327 +                            (Peg (Ab (AbVar "Âč" a) (ItfMap M.empty a) a) (
    ↪ FloatTy a) a) a) [] a
328 +  where a = Refined BuiltIn
329 +
330  caseDef :: MHDef Refined
331  caseDef = Def
332             "case"
333 @@ -562,6 +611,14 @@ caseDef = Def
```

```
                    [Use (Op (VarId "x" b) b) b] b) b) b] b
   where b = Refined BuiltIn

+
+builtinMHDefs :: [MHDef Refined]
+builtinMHDefs = map (makeIntBinOp (Refined BuiltIn)) "+-*" ++
+                map (makeIntBinCmp (Refined BuiltIn)) "><" ++
+                map (makeFloatBinOp (Refined BuiltIn)) "+-*/" ++
+                map (makeFloatBinCmp (Refined BuiltIn)) "><" ++
+                [caseDef, charEq, alphaNumPred, floatEq, intEq, roundMH,
    ↪ toFloat, refEq]
+
 builtinDTs :: DTMap
 builtinDTs = foldl add M.empty builtinDataTs
   where add m (DT id ps _ a) = M.insert id ps m
diff --git a/RefineSyntaxConcretiseEps.hs b/RefineSyntaxConcretiseEps.hs
index 877b449..448e8cd 100644
--- a/RefineSyntaxConcretiseEps.hs
+++ b/RefineSyntaxConcretiseEps.hs
@@ -66,7 +66,6 @@ concretiseEps dts itfs itfAls =
                       ([i], []) -> Just $ ItfNode i
                       ([], [i]) -> Just $ ItfAlNode i
                       _ -> Nothing
-
   -- Given graph (undecided-nodes, positive-nodes, negative-nodes), decide
   -- subgraphs as long as there are unvisited nodes. Finally (base case),
   -- return positive nodes.
@@ -156,6 +155,7 @@ concretiseEps dts itfs itfAls =
                                                  else hasEpsDTTy tvs x [] --
    ↪ ... or not (but data type)
   hasEpsVType tvs (StringTy _)  = hasNoEps
   hasEpsVType tvs (IntTy _)     = hasNoEps
+  hasEpsVType tvs (FloatTy _)   = hasNoEps
   hasEpsVType tvs (CharTy _)    = hasNoEps

   hasEpsDTTy :: [Id] -> Id -> [TyArg Raw] -> HasEps
diff --git a/Syntax.hs b/Syntax.hs
index cf0a72a..f0d4b2a 100644
--- a/Syntax.hs
+++ b/Syntax.hs
@@ -284,6 +284,7 @@ data TmF :: ((* -> *) -> (* -> *)) -> * -> * -> * where
   MkStr :: String -> TmF t r
   MkInt :: Int -> TmF t r
   MkChar :: Char -> TmF t r
+  MkFloat :: Double -> TmF t r           -- Frank floats are provisionally
   ↪ Haskell doubles.
   MkList :: [r] -> TmF (AnnotT Raw) r
   MkTmSeq :: r -> r -> TmF t r
   MkUse :: TFix t UseF -> TmF t r
@@ -302,6 +303,7 @@ pattern Let x tm1 tm2 a = Fx (AnnF (MkLet x tm1 tm2, a))
 pattern StrTm str a = Fx (AnnF (MkStr str, a))
 pattern IntTm n a = Fx (AnnF (MkInt n, a))
 pattern CharTm c a = Fx (AnnF (MkChar c, a))
```

```
+pattern FloatTm f a = Fx (AnnF (MkFloat f, a))
 pattern ListTm xs a = Fx (AnnF (MkList xs, a))
 pattern TmSeq tm1 tm2 a = Fx (AnnF (MkTmSeq tm1 tm2, a))
 pattern Use u a = Fx (AnnF (MkUse u, a))
@@ -403,6 +405,7 @@ data ValuePatF :: ((* -> *) -> (* -> *)) -> * -> * where
   MkDataPat :: Id -> [r] -> ValuePatF t r
   MkIntPat :: Int -> ValuePatF t r
   MkCharPat :: Char -> ValuePatF t r
+  MkFloatPat :: Double -> ValuePatF t r
   MkStrPat :: String -> ValuePatF t r
   MkConsPat :: r -> r -> ValuePatF (AnnotT Raw) r
   MkListPat :: [r] -> ValuePatF (AnnotT Raw) r
@@ -413,6 +416,7 @@ pattern VarPat x a = Fx (AnnF (MkVarPat x, a))
 pattern DataPat x vps a = Fx (AnnF (MkDataPat x vps, a))
 pattern IntPat n a = Fx (AnnF (MkIntPat n, a))
 pattern CharPat c a = Fx (AnnF (MkCharPat c, a))
+pattern FloatPat d a = Fx (AnnF (MkFloatPat d, a))
 pattern StrPat str a = Fx (AnnF (MkStrPat str, a))
 pattern ConsPat p1 p2 a = Fx (AnnF (MkConsPat p1 p2, a))
 pattern ListPat ps a = Fx (AnnF (MkListPat ps, a))
@@ -460,6 +464,7 @@ data VTypeF :: ((* -> *) -> (* -> *)) -> * -> * where
                  -- v
   MkStringTy :: NotDesugared (t Identity ()) => VTypeF t r                  --
    ↪  string type
   MkIntTy :: VTypeF t r                                                    --
    ↪  int type
   MkCharTy :: VTypeF t r                                                   --
    ↪  char type
+  MkFloatTy :: VTypeF t r                                                  --
    ↪  float type
 deriving instance (Show (TFix t TyArgF),
                    Show (TFix t CTypeF),
                    Show r, Show (TFix t VTypeF)) => Show (VTypeF t r)
@@ -475,6 +480,7 @@ pattern FTVar x a = Fx (AnnF (MkFTVar x, a))
 pattern StringTy a = Fx (AnnF (MkStringTy, a))
 pattern IntTy a = Fx (AnnF (MkIntTy, a))
 pattern CharTy a = Fx (AnnF (MkCharTy, a))
+pattern FloatTy a = Fx (AnnF (MkFloatTy, a))

 -- Interface-id -> list of bwd-list of ty arg's (each entry an instantiation)
 data ItfMapF :: ((* -> *) -> (* -> *)) -> * -> * where
@@ -567,8 +573,8 @@ getCtrs (DT _ _ xs _) = xs
 collectDTNames :: [DataT t] -> [Id]
 collectDTNames = map (\case (DT dt _ _ _) -> dt)

--- Convert ability to a list of interface names and effect variables
 {-
+-- Convert ability to a list of interface names and effect variables
 abToList :: Ab a -> [Id]
 abToList MkEmpAb = []
 abToList (MkAbVar id) = [id]
@@ -620,7 +626,7 @@ tyVar2rawTyVarArg (id, VT) = VArg (TVar id (Raw Generated))
   ↪ (Raw Generated)
```

```
 tyVar2rawTyVarArg (id, ET) = EArg (liftAbMod (AbVar id (Raw Generated)))
                                   (Raw Generated)

--- transform type variable (+ its kind) to a rigid tye variable argument
+-- transform type variable (+ its kind) to a rigid type variable argument
 -- (prepare for later unification)
 tyVar2rigTyVarArg :: (Id, Kind) -> TyArg Desugared
 tyVar2rigTyVarArg (id, VT) = VArg (RTVar id (Desugared Generated))
diff --git a/TypeCheck.hs b/TypeCheck.hs
index ff5c17d..626e59b 100644
--- a/TypeCheck.hs
+++ b/TypeCheck.hs
@@ -252,10 +252,11 @@ inferUse adpd@(Adapted adps t a) =
 -- 2nd major TC function besides "check": Check that term (construction) has
 -- given type
 checkTm :: Tm Desugared -> VType Desugared -> Contextual (Tm Desugared)
-checkTm (SC sc a) ty = SC <$> (checkSComp sc ty) <*> (pure a)
+checkTm (SC sc a) ty = SC <$> checkSComp sc ty <*> pure a
 checkTm tm@(StrTm _ a) ty = unify (desugaredStrTy a) ty >> return tm
 checkTm tm@(IntTm _ a) ty = unify (IntTy a) ty >> return tm
 checkTm tm@(CharTm _ a) ty = unify (CharTy a) ty >> return tm
+checkTm tm@(FloatTm _ a) ty = unify (FloatTy a) ty >> return tm
 checkTm tm@(TmSeq tm1 tm2 a) ty =
   -- create dummy mvar s.t. any type of tm1 can be unified with it
   do ftvar <- freshMVar "seq"
@@ -410,6 +411,7 @@ checkVPat (DataPat k ps a) ty =
 checkVPat (CharPat _ a) ty = unify ty (CharTy a) >> return []
 checkVPat (StrPat _ a) ty = unify ty (desugaredStrTy a) >> return []
 checkVPat (IntPat _ a) ty = unify ty (IntTy a) >> return []
+checkVPat (FloatPat _ a) ty = unify ty (FloatTy a) >> return []
 -- checkVPat p ty = throwError $ "failed to match value pattern " ++
 --                  (show p) ++ " with type " ++ (show ty)

diff --git a/TypeCheckCommon.hs b/TypeCheckCommon.hs
index 6717270..c33e33a 100644
--- a/TypeCheckCommon.hs
+++ b/TypeCheckCommon.hs
@@ -78,6 +78,7 @@ fmv (FTVar x _) = S.singleton x
 fmv (RTVar x _) = S.empty
 fmv (StringTy _) = S.empty
 fmv (IntTy _) = S.empty
+fmv (FloatTy _) = S.empty
 fmv (CharTy _) = S.empty

 fmvAb :: Ab Desugared -> S.Set Id
diff --git a/Unification.hs b/Unification.hs
index 5b11bc3..eb36819 100644
--- a/Unification.hs
+++ b/Unification.hs
@@ -48,6 +48,7 @@ unify t0 t1 = do logBeginUnify t0 t1
   unify' (RTVar a _)     (RTVar b _) | a == b = return ()
   unify' (IntTy _)       (IntTy _)            = return ()
   unify' (CharTy _)      (CharTy _)           = return ()
```

```
+  unify' (FloatTy _)     (FloatTy _)          = return ()
   unify' fta@(FTVar a _) ftb@(FTVar b _)      = onTop $ \c d ->
     cmp (a == c) (b == c) d
     where cmp :: Bool -> Bool -> Decl -> Contextual Extension
diff --git a/examples/float-ops.fk b/examples/float-ops.fk
new file mode 100644
index 0000000..41b59e6
--- /dev/null
+++ b/examples/float-ops.fk
@@ -0,0 +1,2 @@
+main : {Bool}
+main! = ((toFloat 4) *. (-. 2.0)) <. (4.9 +. 5.1)
diff --git a/shonky/src/Shonky/Semantics.hs b/shonky/src/Shonky/Semantics.hs
index a6c294b..8817612 100644
--- a/shonky/src/Shonky/Semantics.hs
+++ b/shonky/src/Shonky/Semantics.hs
@@ -14,7 +14,7 @@ import qualified Data.Map.Strict as M
 import Shonky.Syntax
 import Shonky.Renaming

-import Debug.Trace
+-- import Debug.Trace (trace)
 import Debug

 -- There is no predefined Show instance
@@ -24,6 +24,7 @@ instance Show (IORef a) where
 data Val
   = VA String                                                    --
     ↪ atom
   | VI Int                                                       --
     ↪ int
+  | VD Double                                                    --
     ↪ float (double)
   | Val :&& Val                                                  --
     ↪ cons
   | VX String                                                    --
     ↪ string
   | VF Env [([Adap], [String])] [([Pat], Exp)]                   --
     ↪ function (anonymous), has environment, for each port: list of adaptors +
     ↪ list of commands to be captured, list of patterns + handling expressions
@@ -89,6 +90,8 @@ envToList g = envToList' g []
 ppVal :: Val -> Doc
 ppVal (VA s)  = text $ "'" ++ s    -- TODO: error message here?
 ppVal (VI n)  = int n
+ppVal (VD f)  = text $ show f      -- TODO: replace with something else; couldn'
    ↪ t
+                                   -- find a suitable builtin function
 ppVal v@(VA "cons" :&& (VX [_] :&& _)) = doubleQuotes (ppStringVal v)
 ppVal (VA "cons"   :&& (v :&& w))      = ppBrackets $ ppVal v <> ppListVal w
 ppVal (VA "nil"    :&& _)              = ppBrackets empty
@@ -159,61 +162,6 @@ sepBy s ds = vcat $ punctuate s ds
 bracketed :: Doc -> [Doc] -> Doc
 bracketed s ds = lbrack <+> (sepBy s ds <+> rbrack)
```

```
529
530 --- Given env and 2 operands (that are values), compute result
531 -plus :: Env -> [Comp] -> Val
532 -plus g [a1,a2] = VI (f a1 + f a2)
533 -  where f x = case x of
534 -          Ret (VI n) -> n
535 -          _ -> error "plus: argument not an int"
536 -plus g _ = error "plus: incorrect number of arguments, expected 2."
537 -
538 -minus :: Env -> [Comp] -> Val
539 -minus g [a1,a2] = VI (f a1 - f a2)
540 -  where f x = case x of
541 -          Ret (VI n) -> n
542 -          _ -> error "minus: argument not an int"
543 -minus g _ = error "minus: incorrect number of arguments, expected 2."
544 -
545 -builtinPred :: Bool -> Val
546 -builtinPred b = (if b then VA "true" else VA "false") :&& VA ""
547 -
548 -lt :: Env -> [Comp] -> Val
549 -lt g [a1,a2] = builtinPred ((f a1) < (f a2))
550 -  where f x = case x of
551 -          Ret (VI n) -> n
552 -          _ -> error "plus: argument not an int"
553 -lt g _ = error "plus: incorrect number of arguments, expected 2."
554 -
555 -gt :: Env -> [Comp] -> Val
556 -gt g [a1,a2] = builtinPred ((f a1) > (f a2))
557 -  where f x = case x of
558 -          Ret (VI n) -> n
559 -          _ -> error "plus: argument not an int"
560 -gt g _ = error "plus: incorrect number of arguments, expected 2."
561 -
562 -
563 -eqc :: Env -> [Comp] -> Val
564 -eqc g [a1,a2] = builtinPred ((f a1) == (f a2))
565 -  where f x = case x of
566 -          Ret (VX [c]) -> c
567 -          _ -> error "eqc: argument not a character"
568 -eqc g _ = error "eqc: incorrect number of arguments, expected 2."
569 -
570 -alphaNumPred :: Env -> [Comp] -> Val
571 -alphaNumPred g [a] =
572 -  (if isAlphaNum (f a) then VA "true" else VA "false") :&& VA ""
573 -  where f x = case x of
574 -          Ret (VX [c]) -> c
575 -          _ -> error "alphaNumPred: argument not a character"
576 -alphaNumPred g _ =
577 -  error "alphaNumPred: incorrect number of arguments, expected 2."
578 -
579 -
580 -builtins :: M.Map String (Env -> [Comp] -> Val)
581 -builtins = M.fromList [("plus", plus), ("minus", minus), ("eqc", eqc)
```

```
-                      ,("lt", lt), ("gt", gt)
-                      ,("isAlphaNum", alphaNumPred)]
-
 -- Look-up a definition
 fetch :: Env -> String -> Val
 fetch g y = go g where
@@ -241,6 +189,7 @@ compute :: Env -> Exp -> Agenda -> Comp
 compute g (EV x)       ls   = consume (fetch g x) ls                          --
    ↪ 1) look-up value
 compute g (EA a)       ls   = consume (VA a) ls                               --
    ↪ 1) feed atom
 compute g (EI n)       ls   = consume (VI n) ls                               --
    ↪ 1) feed int
+compute g (ED f)       ls   = consume (VD f) ls                               --
    ↪ 1) feed double
 compute g (a :& d)     ls   = compute g a (Car g d : ls)                      --
    ↪ 2) compute head. save tail for later.
 compute g (f :$ as)    ls   = compute g f (Fun g as : ls)                     --
    ↪ 2) Application. Compute function. Save args for later.
 compute g (e :! f)     ls   = compute g e (Seq g f : ls)                      --
    ↪ 2) Sequence.    Compute 1st exp.  Save 2nd for later.
@@ -477,13 +426,6 @@ txt (VA a)     = a
 txt (VX a)     = a
 txt (u :&& v)  = txt u ++ txt v

-envBuiltins :: Env
-envBuiltins = Empty :/ [DF "plus" [] []
-                       ,DF "minus" [] []
-                       ,DF "eqc"   [] []
-                       ,DF "gt"    [] []
-                       ,DF "lt"    [] []]
-
 prog :: Env -> [Def Exp] -> Env
 prog g ds = g' where
   g' = g :/ map ev ds
@@ -506,3 +448,161 @@ loadFile x = do
 try :: Env -> String -> Comp
 try g s = compute g e [] where
   Just (e, "") = parse pExp s
+
+--
+-- Builtins
+--
+
+-- Given env and 2 operands (that are values), compute result
+plus :: Env -> [Comp] -> Val
+plus g [a1,a2] = VI (f a1 + f a2)
+  where f x = case x of
+          Ret (VI n) -> n
+          _ -> error "plus: argument not an int"
+plus g _ = error "plus: incorrect number of arguments, expected 2."
+
+plusF :: Env -> [Comp] -> Val
```

```
628 +plusF g [a1, a2] = VD (f a1 + f a2)
629 +  where f x = case x of
630 +          Ret (VD n) -> n
631 +          _ -> error "plusF: argument not a float"
632 +plusF g _ = error "plusF: incorrect number of arguments, expected 2."
633 +
634 +minus :: Env -> [Comp] -> Val
635 +minus g [a1,a2] = VI (f a1 - f a2)
636 +  where f x = case x of
637 +          Ret (VI n) -> n
638 +          _ -> error "minus: argument not an int"
639 +minus g _ = error "minus: incorrect number of arguments, expected 2."
640 +
641 +minusF :: Env -> [Comp] -> Val
642 +minusF g [a1,a2] = VD (f a1 - f a2)
643 +  where f x = case x of
644 +          Ret (VD n) -> n
645 +          _ -> error "minusF: argument not an int"
646 +minusF g _ = error "minusF: incorrect number of arguments, expected 2."
647 +
648 +mult :: Env -> [Comp] -> Val
649 +mult g [a1,a2] = VI (f a1 * f a2)
650 +  where f x = case x of
651 +          Ret (VI n) -> n
652 +          _ -> error "mult: argument not an int"
653 +mult g _ = error "mult: incorrect number of arguments, expected 2."
654 +
655 +multF :: Env -> [Comp] -> Val
656 +multF g [a1,a2] = VD (f a1 * f a2)
657 +  where f x = case x of
658 +          Ret (VD n) -> n
659 +          _ -> error "multF: argument not an float"
660 +multF g _ = error "multF: incorrect number of arguments, expected 2."
661 +
662 +divF :: Env -> [Comp] -> Val
663 +divF g [a1,a2] = VD (f a1 / f a2)
664 +  where f x = case x of
665 +          Ret (VD n) -> n
666 +          _ -> error "multF: argument not an int"
667 +divF g _ = error "multF: incorrect number of arguments, expected 2."
668 +
669 +builtinPred :: Bool -> Val
670 +builtinPred b = (if b then VA "true" else VA "false") :&& VA ""
671 +
672 +lt :: Env -> [Comp] -> Val
673 +lt g [a1,a2] = builtinPred ((f a1) < (f a2))
674 +  where f x = case x of
675 +          Ret (VI n) -> n
676 +          _ -> error "lt: argument not an int"
677 +lt g _ = error "lt: incorrect number of arguments, expected 2."
678 +
679 +ltF :: Env -> [Comp] -> Val
680 +ltF g [a1,a2] = builtinPred ((f a1) < (f a2))
```

```
681 +  where f x = case x of
682 +          Ret (VD n) -> n
683 +          _ -> error "ltF: argument not an int"
684 +ltF g _ = error "ltF: incorrect number of arguments, expected 2."
685 +
686 +gt :: Env -> [Comp] -> Val
687 +gt g [a1,a2] = builtinPred ((f a1) > (f a2))
688 +  where f x = case x of
689 +          Ret (VI n) -> n
690 +          _ -> error "gt: argument not an int"
691 +gt g _ = error "gt: incorrect number of arguments, expected 2."
692 +
693 +gtF :: Env -> [Comp] -> Val
694 +gtF g [a1,a2] = builtinPred ((f a1) > (f a2))
695 +  where f x = case x of
696 +          Ret (VD n) -> n
697 +          _ -> error "gtF: argument not an int"
698 +gtF g _ = error "gtF: incorrect number of arguments, expected 2."
699 +
700 +eqc :: Env -> [Comp] -> Val
701 +eqc g [a1,a2] = builtinPred ((f a1) == (f a2))
702 +  where f x = case x of
703 +          Ret (VX [c]) -> c
704 +          _ -> error "eqc: argument not a character"
705 +eqc g _ = error "eqc: incorrect number of arguments, expected 2."
706 +
707 +eqN :: Env -> [Comp] -> Val
708 +eqN g [a1,a2] = builtinPred ((f a1) == (f a2))
709 +  where f x = case x of
710 +          Ret (VI n) -> n
711 +          _ -> error "eqN: argument not an int"
712 +eqN g _ = error "eqN: incorrect number of arguments, expected 2."
713 +
714 +eqF :: Env -> [Comp] -> Val
715 +eqF g [a1,a2] = builtinPred ((f a1) == (f a2))
716 +  where f x = case x of
717 +          Ret (VD n) -> n
718 +          _ -> error "eqF: argument not a float"
719 +eqF g _ = error "eqF: incorrect number of arguments, expected 2."
720 +
721 +eqR :: Env -> [Comp] -> Val
722 +eqR g [a1,a2] = builtinPred ((f a1) == (f a2))
723 +  where f x = case x of
724 +          Ret (VR r) -> r
725 +          _ -> error "eqR: argument not a ref"
726 +eqR g _ = error "eqR: incorrect number of arguments, expected 2."
727 +
728 +alphaNumPred :: Env -> [Comp] -> Val
729 +alphaNumPred g [a] =
730 +  (if isAlphaNum (f a) then VA "true" else VA "false") :&& VA ""
731 +  where f x = case x of
732 +          Ret (VX [c]) -> c
733 +          _ -> error "alphaNumPred: argument not a character"
```

```
+alphaNumPred g _ =
+  error "alphaNumPred: incorrect number of arguments, expected 2."
+
+chrFunc :: Env -> [Comp] -> Val
+chrFunc g [a] = VX [chr (f a)]
+  where f x = case x of
+          Ret (VI n) -> n
+          _ -> error "chr: argument not a int"
+chrFunc g _ =
+  error "chr: incorrect number of arguments, expected 1."
+
+roundF :: Env -> [Comp] -> Val
+roundF g [a] = VI (round (f a))
+  where f x = case x of
+          Ret (VD n) -> n
+          _ -> error "round: argument not a float"
+roundF g _ = error "round: incorrect number of arguments, expected 1."
+
+toFloat :: Env -> [Comp] -> Val
+toFloat g [a] = VD (fromIntegral (f a))
+  where f x = case x of
+          Ret (VI n) -> n
+          _ -> error "toFloat: argument not a float"
+toFloat g _ = error "toFloat: incorrect number of arguments, expected 1."
+
+builtins :: M.Map String (Env -> [Comp] -> Val)
+builtins = M.fromList [("plus", plus), ("minus", minus), ("mult", mult), ("eqc
    ↪ ", eqc)
+                      ,("lt", lt), ("gt", gt)
+                      ,("plusF", plusF), ("minusF", minusF), ("multF", multF),
    ↪ ("divF", divF)
+                      ,("ltF", ltF), ("gtF", gtF)
+                      ,("eqF", eqF), ("eqN", eqN), ("eqR", eqR)
+                      ,("isAlphaNum", alphaNumPred), ("chr", chrFunc)
+                      ,("roundF", roundF), ("toFloat", toFloat)
+                      ]
+
+-- TODO: Generate this from `builtins`.
+envBuiltins :: Env
+envBuiltins = Empty :/ map (\x -> DF x [] []) (M.keys builtins)
diff --git a/shonky/src/Shonky/Syntax.hs b/shonky/src/Shonky/Syntax.hs
index 927980a..83d57b0 100644
--- a/shonky/src/Shonky/Syntax.hs
+++ b/shonky/src/Shonky/Syntax.hs
@@ -15,6 +15,7 @@ import Shonky.Renaming
 data Exp
   = EV String                               -- variable
   | EI Int                                  -- int
+  | ED Double                               -- float (double)
   | EA String                               -- atom
   | Exp :& Exp                              -- cons
   | Exp :$ [Exp]                            -- n-ary application
@@ -58,6 +59,7 @@ data Pat
```

```
 data VPat
   = VPV String               -- LC: ?
   | VPI Int                  -- int value
+  | VPD Double               -- float value
   | VPA String               -- atom value
   | VPat :&: VPat            -- cons value
   | VPX [Either Char VPat]   -- LC: ?
diff --git a/stack.yaml.lock b/stack.yaml.lock
new file mode 100644
index 0000000..2d6e2d0
--- /dev/null
+++ b/stack.yaml.lock
@@ -0,0 +1,33 @@
+# This file was autogenerated by Stack.
+# You should not edit this file by hand.
+# For more information, please see the documentation at:
+#   https://docs.haskellstack.org/en/stable/lock_files
+
+packages:
+- completed:
+    hackage: wl-pprint-1.2.1@sha256:
    ↪ aea676cff4a062d7d912149d270e33f5bb0c01b68a9db46ff13b438141ff4b7c,734
+    pantry-tree:
+      size: 221
+      sha256: 750b375c6fc33400551f9e32e26e41844c372270a9bc3571e912fa36df7c6d4f
+  original:
+    hackage: wl-pprint-1.2.1
+- completed:
+    hackage: Unique-0.4.5@sha256:56
    ↪ d1a2db7b1e70e8e2b341af6ba4ed44f1e3f636a654840eeebccaab4e3f3d60,2338
+    pantry-tree:
+      size: 344
+      sha256: aef0170a489b8d56fba47d43e2b277f3f626fff87827de1a7d57274b94cf6e21
+  original:
+    hackage: Unique-0.4.5
+- completed:
+    hackage: indentation-trifecta-0.1.0@sha256:
    ↪ cead425151e4e98a98ae163186dab471fafeb9854bd520b9d5356652114ac18e,2460
+    pantry-tree:
+      size: 411
+      sha256: 143c87fbf98238027d45070a8ed21109a5ab89341eda335a3d9a40f28e71f05c
+  original:
+    hackage: indentation-trifecta-0.1.0
+snapshots:
+- completed:
+    size: 495203
+    url: https://raw.githubusercontent.com/commercialhaskell/stackage-snapshots
    ↪ /master/lts/13/8.yaml
+    sha256: 91139b0de6b320b13e27d5e6e6c368e239974b06fb9dc86d8d96da08697972ac
+  original: lts-13.8
diff --git a/tests/should-fail/floats/wrong-type.fk b/tests/should-fail/floats/
    ↪ wrong-type.fk
new file mode 100644
```

```
index 0000000..e90e1f8
--- /dev/null
+++ b/tests/should-fail/floats/wrong-type.fk
@@ -0,0 +1,4 @@
+-- #desc Can't perform a float operation on a float and int
+-- #return failed to unify Int (line 4 , column 9) with Float (built-in)
+main : {Float}
+main! = 3 +. 3.0
diff --git a/tests/should-pass/floats/ops.fk b/tests/should-pass/floats/ops.fk
new file mode 100644
index 0000000..b571ef0
--- /dev/null
+++ b/tests/should-pass/floats/ops.fk
@@ -0,0 +1,5 @@
+-- #desc Perform some standard float operations.
+-- #return true
+
+main : {Bool}
+main! = ((2.0 +. 1.0) /. 3.0) <. ((-. 1.0) *. (3.0 -. 6.0))
```

Appendix B

Diffeology on subsets

In the main text, it seemed that the FFLR $p\colon$ **PredDiff** $\to$ **Diff** of example 8.1.13 had no diffeological content as it only talked about mere subsets. However, it happens that every subset A of a diffeological space X has an induced diffeology.

Definition B.0.1 (van der Schaaf, 2020, Definition 2.51)**.** Let X a diffeological space. The *subset diffeology*, defined on a subset $A \subseteq X$, is the collection $\mathcal{P}^{A\subseteq X}$ of plots in X that take values in A. We will usually just refer to this diffeology by $\mathcal{P}^A$ when X is clear from context.

Furthermore, the subset diffeology on a subset $A \subseteq X$ makes A a subspace of X in the following sense.

Definition B.0.2 (Baez and Hoffnung, 2009, Definition 33)**.** We say a smooth map $i\colon A \to X$ makes A a *subspace* of X if for any plot $\varphi \in \mathcal{P}^X_U$ with $\varphi(U) \subseteq i(A)$, there exists a unique plot $\psi \in \mathcal{P}^A_U$ with $i \cdot \psi = \varphi$.

Just like quotients, subspaces are an instance of a more general concept, namely *strong monomorphisms*.

Definition B.0.3 (Baez and Hoffnung, 2009, Definition 32)**.** In any category, a monomorphism $i\colon A \to X$ is *strong* if given any epimorphism $p\colon E \to B$ and morphisms f, g making the outer square here commute:

$$\begin{array}{ccc} E & \xrightarrow{f} & A \\ {\scriptstyle p}\downarrow & \overset{t}{\nearrow} & \downarrow{\scriptstyle i} \\ B & \xrightarrow[g]{} & X \end{array}$$

then there exists a unique $t\colon B \to A$ making the whole diagram commute.

Proposition B.0.4 (Baez and Hoffnung, 2009, Proposition 34)**.** *A smooth map $i\colon A \to X$ is a strong monomorphism if and only if i makes A a subspace of X.*

Furthermore, strong monomorphisms are *regular monomorphisms* in **Diff**.

Definition B.0.5. A *regular monomorphism* is a monomorphism which is the equalizer of some pair of arrows.

Proposition B.0.6. *In* **Diff***, every strong monomorphism is regular.*

Proof. Let $i\colon X \to A$ be a strong monomorphism. The set $2 \coloneqq \{0, 1\}$ with the indiscrete topology is the weak object classifier in **Diff**. As such, there exists a unique characteristic morphism $\chi_i\colon X \to 2$ making

$$\begin{array}{ccc} A & \xrightarrow{\ i\ } & X \\ \downarrow{\scriptstyle !_A} & \lrcorner & \downarrow{\scriptstyle \chi_i} \\ 1 & \xrightarrow{\ \top\ } & 2 \end{array}$$

a pullback. Define $\chi'\colon X \to 2$ by $\chi'(x) = 1$ if $x \in i(A)$ and 0 if $x \notin i(A)$. Then χ' is smooth because all maps into indiscrete spaces are and $(\chi' \cdot i)(a) = 1 = (\top \cdot !_A)(a)$. Thus by uniqueness $\chi_i = \chi'$ and so $i(A) = {\chi_i}^{-1}(\{1\})$.

We will show that i is an equalizer for $\chi_i, \top \cdot !_X\colon X \to 2$. Clearly $\top \cdot !_X \cdot i = \top \cdot !_A = \chi_i \cdot i$. Suppose we have a map $j\colon B \to X$ which equalizes χ_i and $\top \cdot !_X$. Then for any $b \in B$, $\chi_i(j(b)) = 1$ meaning $j(b) \in {\chi_i}^{-1}(\{1\}) = i(A)$. Define $u\colon B \to A$ by $u \coloneqq i^{-1}(j(b))$ which is well-defined because i is a monomorphism and thus injective on the underlying sets. We will show u is smooth.

Let $\varphi \in \mathcal{P}_U^B$ and consider $i.u.\varphi\colon U \to X$. Then $i \cdot u \cdot \varphi = j \cdot \varphi \in \mathcal{P}_U^X$ as j is smooth. Next, because $(i \cdot u \cdot \varphi)(U) = i(u \cdot \varphi(U)) \subseteq i(A)$ and i is strong, there exists a unique $\psi \in \mathcal{P}_U^A$ such that $i \cdot \psi = i \cdot u \cdot \varphi$. As i is a monomorphism, $u \cdot \varphi = \psi \in \mathcal{P}_U^A$ and so u is smooth.

Finally, if there is $v\colon B \to A$ such that $i \cdot v = j = i \cdot u$, then $v = u$ as i is a monomorphism. Therefore i is an equalizer and thus regular. □

We again take advantage of this equivalence to find a factorization system for logical relations for **Diff**.

Corollary B.0.7 (Cassidy, Hébert, and Kelly, 1985, Corollary 3.2)**.** *For a category $\mathcal{C}$, (epimorphisms, strong monomorphisms) is a factorization system if*

1. *$\mathcal{C}$ is finitely well-complete; or*

2. *$\mathcal{C}$ is finitely complete and admits all cointersections of epimorphisms; or*

3. *$\mathcal{C}$ is finitely complete, finitely cocomplete, and all strong monomorphisms are regular.*

Corollary B.0.8. Diff *has (epimorphisms, strong monomorphisms) as a factorization system.*

Theorem B.0.9. *The factorization system (epimorphisms, strong monomorphisms) on* **Diff** *is a factorization system for logical relations.*

Proof of theorem B.0.9. We must prove the five properties of definition 7.3.9.

Diff *has pullbacks of strong monomorphisms:* This is trivial as **Diff** is complete.

Every strong monomorphism is a monomorphism: Strong monomorphisms are monomorphisms by definition.

*For every $Y \in$ **Diff** the fibre $\mathbf{SMono}_Y$ has small products:* Fix an object $Y \in \mathbf{Diff}$. The objects in $\mathbf{SMono}_Y$ are the objects of **SMono** over Y via cod, i.e. strong monomorphisms $m \colon X \to Y$. For strong monomorphisms $m_1, m_2 \colon X_i \to Y$, a map $f \colon m_1 \to m_2$ in **SMono** is a morphism $f \colon X_1 \to X_2$ such that $m_1 = m_2 \cdot f$. Furthermore, in a full subcategory of an arrow category, which **SMono** is, the product of objects is given by their pullback if it belongs to the subcategory.

Let $\{m_i \colon X_i \to Y\}_{i \in I}$ be a set of objects of **SMono**. Then their pullback exists in **Diff** as it is complete. Denote this object by X and the induced projections by $\pi_i \colon X \to X_i$ so that $m = m_i \cdot \pi_i$ for all $i \in I$. Note that each π_i is a monomorphism our pullback legs consists of monomorphisms, and so m is a monomorphism as well. We must show that m is a strong monomorphisms. Fix an arbitrary $i \in I$. It suffices to show that π_i is a strong monomorphisms as they are closed under composition by virtue of being part of a factorization system.

We can construct the pullback X explicitly in **Diff** as follows:

$$X = \left\{ x \coloneqq (x_i)_{i \in I} \in \prod_{i \in I} X_i : \forall j, k \in I.\, m_j \cdot \pi_j(x) = m_k \cdot \pi_k(x) \right\}$$
$$\mathcal{P}_U^X = \left\{ \varphi \colon U \to X : \forall i \in I.\, \pi_i \cdot \varphi \in \mathcal{P}_U^{X_i} \right\}.$$

We must show that $\pi_i \colon X \to X_i$ makes X a subspace of X_i. Let $\varphi \in \mathcal{P}_U^{X_i}$ be such that $\varphi(U) \subseteq \pi_i(X)$. We must show that there exists a unique $\psi \in \mathcal{P}_U^X$ with $\pi_i \cdot \psi = \varphi$.

As $\varphi(U) \subseteq \pi_i(X)$, $(m_i \cdot \varphi)(U) \subseteq (m_i \cdot \pi_i)(X)$. Because m_i is smooth, $m_i \cdot \varphi \in \mathcal{P}_U^Y$. Next, $\pi_i(X) \subseteq X_i$ so $(m_i \cdot \varphi)(U) \subseteq m_i(X_i)$. Therefore, because m_i is a strong, there exists a unique $\theta_i \in \mathcal{P}_U^{X_i}$ such that $m_i \cdot \theta_i = m_i \cdot \varphi$. We deduce $\theta_i = \varphi$ as m_i is a monomorphism.

Next, as $(m_i \cdot \pi_i)(X) = (m_j \cdot \pi_j)(X)$ for all $j \in I \setminus \{i\}$ and $\pi_j(X) \subseteq X_j$, we get $(m_i \cdot \varphi)(U) \subseteq m_j(X_j)$. As m_j is a strong monomorphism, there exists a unique $\theta_j \in \mathcal{P}_U^{X_j}$ such that $m_j \cdot \theta_j = m_i \cdot \varphi$.

Define $\psi \coloneqq \langle \theta_i \rangle_{i \in I} \colon U \to \prod_{i \in I} X_i$. For any $k \in I$, $(m_k \cdot \pi_k)(\psi(u)) = m_k(\theta_k(u))$ $= m_i(\varphi(u))$ whether $k = i$ or $k \neq i$, and so $\psi \colon U \to X$. Furthermore, $\pi_k \cdot \psi = \theta_k \in \mathcal{P}_U^{X_k}$ and so we have $\psi \in \mathcal{P}_U^X$.

We now have $\psi \in \mathcal{P}_U^X$ such that $\pi_i \cdot \psi = \theta_i = \varphi$. Suppose there is $\psi' \in \mathcal{P}_U^X$ such that $\pi \cdot \psi' = \varphi$. Then $\pi_i \cdot \psi' = \pi_i \cdot \psi$ and so $\psi' = \psi$ because π_i is a monomorphism.

In conclusion, m_i and π_i are strong monomorphisms, and so $m = m_i \cdot \pi_i$ is a strong monomorphism and thus m is the product of the family $\{m_i\}_{i \in I}$.

SMono *is closed under binary coproducts:* Let $m_i \colon X_i \to Y_i$ for $i = 1, 2$ be strong monomorphisms. As objects in $\mathbf{Diff}^{\to}$, their coproduct is $m_1 + m_2 \colon X_1 + X_2 \to Y_1 + Y_2$ and the commutative square

$$\begin{array}{ccc} X_i & \xrightarrow{\iota_i} & X_1 + X_2 \\ {\scriptstyle m_i}\downarrow & & \downarrow{\scriptstyle m_1 + m_2} \\ Y_i & \xrightarrow[\iota_i]{} & Y_1 + Y_2 \end{array}$$

gives the coprojections $m_i \to m_1 + m_2$. Thus, we need to show that $m_1 + m_2$ is a strong monomorphism. The map $m_1 + m_2$ is smooth and is injective on the underlying sets, and thus is a monomorphism in **Diff**, so we need only show it is strong.

In **Diff**, the underlying set of $X_1 + X_2$ is the coproduct of the underlying sets, and the diffeology is given by

$$\mathcal{P}_U^{X_1+X_2} \coloneqq \left\{ \varphi \colon U \to X_1 + X_2 : V_i \coloneqq \varphi^{-1}(X_i) \subseteq_{\text{open}} U \text{ and } \varphi|_{V_i} \in \mathcal{P}_{V_i}^{X_i} \right\}.$$

and likewise for $Y_1 + Y_2$. Let $\varphi \in \mathcal{P}_U^{Y_1+Y_2}$ with $\varphi(U) \subseteq (m_1 + m_2)(X_1 + X_2) = m_1(X_1) + m_2(X_2)$. Then $V_i \coloneqq \varphi^{-1}(Y_i)$ is an open subset of U and $\varphi|_{V_i} \in \mathcal{P}_{V_i}^{Y_i}$. Thus, $\varphi|_{V_i} \subseteq m_i(X_i)$ as $\varphi|_{V_i} \subseteq Y_i \cap (m_1(X_1) + m_2(X_2))$. As m_i is strong, there exists a unique $\psi_i \in \mathcal{P}_{V_i}^{X_i}$ such that $m_i \cdot \psi_i = \varphi|_{V_i}$.

Note that $V_1 \cap V_2 = \emptyset$ as they are the preimage of disjoint sets $Y_1, Y_2 \subseteq Y_1 + Y_2$ and $V_1 \cup V_2 = U$, so that V_1, V_2 form an open partition of U. Furthermore,

$\iota_i \cdot \psi_i \in \mathcal{P}_{V_i}^{X_1+X_2}$, so $\iota_1 \cdot \psi_1$ and $\iota_2 \cdot \psi_2$ are compatible plots of $X_1 + X_2$ by disjointness V_1 and V_2 meaning there exists $\psi \in \mathcal{P}_U^{X_1+X_2}$ such that $\psi|_{V_i} = \iota_i \cdot \psi_i$.

Therefore, $((m_1 + m_2) \cdot \psi)|_{V_i} = (m_1 + m_2) \cdot \iota_i \cdot \psi_i = m_i \cdot \psi_i = \varphi|_{V_i}$ and so $(m_1 + m_2) \cdot \psi = \varphi$ as V_1, V_2 partition U. Furthermore, ψ is unique as $m_1 + m_2$ is a monomorphism. Thus, $m_1 + m_2$ is strong.

Epi *is closed under binary products:* Let $e_i \colon X_i \to Y_i$ be epimorphisms. As objects in $\mathbf{Diff}^{\to}$, their product is $e_1 \times e_2$ with projections analogously to the coproduct case. We must show $e_1 \times e_2$ is an epimorphism. We already know $e_1 \times e_2$ is smooth, and it is clearly surjective on the underlying sets, and so indeed it is an epimorphism. □

We can now combine lemma 7.3.10 and theorem B.0.9.

Corollary B.0.10. *Let* **SMono** *be the full subcategory of strong monomorphisms of the arrow category of* **Diff**. *The restricted codomain fibration* $\mathrm{cod} \colon \mathbf{SMono} \to \mathbf{Diff}$ *is a FFLR.*

Example 8.1.13 proved that $p \colon \mathbf{PredDiff} \to \mathbf{Diff}$ is an FFLR. **PredDiff** seemed not to contain diffeological information for the predicate as the objects are (A, X) where A is a mere subset of X and not a subobject. However, $\mathrm{cod} \colon \mathbf{SMono} \to \mathbf{Diff}$ is actually equivalent to $p \colon \mathbf{PredDiff} \to \mathbf{Diff}$ as fibrations via a fibered equivalence. Let $U \colon \mathbf{Diff} \to \mathbf{Set}$ be the forgetful functor. The equivalence is given by $\mathrm{im} \colon \mathbf{SMono} \to \mathbf{PredDiff}$ which on objects is $(i \colon A \to X) \mapsto (U(i(A)), X)$, and $\mathrm{em} \colon \mathbf{PredDiff} \to \mathbf{SMono}$ which on objects is $(A, X) \mapsto \Big((A, \mathcal{P}^{A \subseteq X}), X\Big)$. Finally, similarly to $\mathrm{cod} \colon \mathbf{Mono} \to \mathbf{Diff}$, $\mathrm{cod} \colon \mathbf{SMono} \to \mathbf{Diff}$ has fibres which are large preorders, but this does not matter as we used $p \colon \mathbf{PredDiff} \to \mathbf{Diff}$.

Bibliography

Abadi, Martín, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. URL: `https://www.tensorflow.org/`.

Abadi, Martín and Gordon Plotkin (Jan. 2020). "A simple differentiable programming language". en. In: *Proceedings of the ACM on Programming Languages* 4.POPL, pp. 1–28. ISSN: 2475-1421, 2475-1421. DOI: `10.1145/3371106`. URL: `https://dl.acm.org/doi/10.1145/3371106` (visited on 06/26/2020).

Adámek, Jiří, Stefan Milius, and Lawrence S Moss (Aug. 2021). *Initial Algebras, Terminal Coalgebras, and the Theory of Fixed Points of Functors*. en. Draft.

Alvarez-Picallo, Mario and C.-H. Luke Ong (Apr. 2019). "Change Actions: Models of Generalised Differentiation". en. In: *arXiv:1902.05465 [cs]*. arXiv: 1902.05465. URL: `http://arxiv.org/abs/1902.05465` (visited on 07/03/2020).

Ariola, Zena M., Hugo Herbelin, and Amr Sabry (Sept. 2009). "A type-theoretic foundation of delimited continuations". en. In: *Higher-Order and Symbolic Computation* 22.3, pp. 233–273. ISSN: 1573-0557. DOI: `10.1007/s10990-007-9006-0`. URL: `https://doi.org/10.1007/s10990-007-9006-0` (visited on 05/05/2024).

Awodey, Steve (June 2010). *Category Theory*. Second Edition. Oxford Logic Guides. Oxford, New York: Oxford University Press. ISBN: 978-0-19-923718-0.

Baez, John C. and Alexander E. Hoffnung (Oct. 2009). *Convenient Categories of Smooth Spaces*. arXiv:0807.1704 [math]. DOI: 10.48550/arXiv.0807.1704. URL: http://arxiv.org/abs/0807.1704 (visited on 09/20/2022).

Barr, Michael (1970). "Coequalizers and free triples". en. In: *Mathematische Zeitschrift* 116.4, pp. 307–322. ISSN: 0025-5874, 1432-1823. DOI: 10.1007/BF01111838. URL: http://link.springer.com/10.1007/BF01111838 (visited on 05/10/2023).

Barr, Michael and Charles Wells (2005). *TOPOSES, TRIPLES AND THEORIES*. en. Reprints in Theory and Applications of Categories 12. verson 1.1.

Bauer, Andrej (Mar. 2019). "What is algebraic about algebraic effects and handlers?" en. In: *arXiv:1807.05923 [cs]*. arXiv: 1807.05923. URL: http://arxiv.org/abs/1807.05923 (visited on 01/31/2020).

Bauer, Andrej and Matija Pretnar (Dec. 2014). "An Effect System for Algebraic Effects and Handlers". en. In: *Logical Methods in Computer Science* 10.4. Ed. by Stefan Milius, p. 9. ISSN: 18605974. DOI: 10.2168/LMCS-10(4:9)2014. URL: https://lmcs.episciences.org/1153 (visited on 05/04/2020).

— (Jan. 2015). "Programming with Algebraic Effects and Handlers". en. In: *Journal of Logical and Algebraic Methods in Programming* 84.1. arXiv: 1203.1539, pp. 108–123. ISSN: 23522208. DOI: 10.1016/j.jlamp.2014.02.001. URL: http://arxiv.org/abs/1203.1539 (visited on 03/05/2020).

Baydin, Atilim Gunes, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind (Feb. 2018). "Automatic differentiation in machine learning: a survey". en. In: *arXiv:1502.05767 [cs, stat]*. arXiv: 1502.05767. URL: http://arxiv.org/abs/1502.05767 (visited on 06/26/2020).

Berthomieu, Bernard and Camille le Moniès de Sagazan (1995). "A Calculus of Tagged Types, with applications to process languages". en. In: *Workshop on Types for Program Analysis*, pp. 1–15.

Betancourt, Michael (Dec. 2018). "A Geometric Theory of Higher-Order Automatic Differentiation". en. In: *arXiv:1812.11592 [stat]*. arXiv: 1812.11592. URL: http://arxiv.org/abs/1812.11592 (visited on 05/22/2019).

Biernacki, Dariusz, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski (Dec. 2017). "Handle with care: relational interpretation of algebraic effects and handlers". In: *Proceedings of the ACM on Programming Languages* 2.POPL, 8:1–8:30. DOI: 10.1145/3158096. URL: https://dl.acm.org/doi/10.1145/3158096 (visited on 07/24/2023).

Bischof, C. H., L. Roh, and A. J. Mauer-Oats (1997). "ADIC: an extensible automatic differentiation tool for ANSI-C". en. In: *Software: Practice and Experience* 27.12, pp. 1427–1456. ISSN: 1097-024X. DOI: 10.1002/(SICI)1097-024X(199712)27:12<1427::AID-SPE138>3.0.CO;2-Q. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291097-024X%199712%2927%3A12%3C1427%3A%3AAID-SPE138%3E3.0.CO%3B2-Q (visited on 07/02/2020).

Bischof, Christian, Peyvand Khademi, Andrew Mauer, and Alan Carle (Sept. 1996). "Adifor 2.0: Automatic Differentiation of Fortran 77 Programs". In: *IEEE Computational Science & Engineering* 3.3, pp. 18–32. ISSN: 1070-9924. DOI: 10.1109/99.537089. URL: https://doi.org/10.1109/99.537089 (visited on 07/02/2020).

Blute, R F, J R B Cockett, and R A G Seely (2009). "CARTESIAN DIFFERENTIAL CATEGORIES". en. In: p. 52.

— (Dec. 2006). "Differential categories". en. In: *Mathematical Structures in Computer Science* 16.06, p. 1049. ISSN: 0960-1295, 1469-8072. DOI: 10.1017/S0960129506005676. URL: http://www.journals.cambridge.org/abstract_S0960129506005676 (visited on 07/03/2020).

Borceux, Francis (1994a). *Handbook of Categorical Algebra: Volume 1: Basic Category Theory*. Vol. 1. Encyclopedia of Mathematics and its Applications. Cambridge: Cambridge University Press. ISBN: 978-0-521-44178-0. DOI: 10.1017/CBO9780511525858. URL: https://www.cambridge.org/core/books/handbook-of-categorical-algebra/A0B8285BBA900AFE85EED8C971E0DE14 (visited on 02/08/2023).

— (1994b). *Handbook of Categorical Algebra: Volume 2: Categories and Structures*. Vol. 2. Encyclopedia of Mathematics and its Applications. Cambridge: Cambridge University Press. ISBN: 978-0-521-44179-7. DOI: 10.1017/CBO9780511525865. URL: https://www.cambridge.org/core/books/handbook-of-categorical-algebra/EF86DCA2B6B7BBA10CE255ABC572601B (visited on 02/08/2023).

Brunel, Alois, Damiano Mazza, and Michele Pagani (Jan. 2020). "Backpropagation in the Simply Typed Lambda-calculus with Linear Negation". In: *Proceedings of the ACM on Programming Languages* 4.POPL. arXiv: 1909.13768, pp. 1–27. ISSN: 2475-1421, 2475-1421. DOI: 10.1145/3371132. URL: http://arxiv.org/abs/1909.13768 (visited on 02/19/2020).

Cassidy, C., M. Hébert, and G. M. Kelly (June 1985). "Reflective subcategories, localizations and factorization systems". en. In: *Journal of the Australian Mathematical Society* 38.3. Publisher: Cambridge University Press, pp. 287–329. ISSN: 0263-6115. DOI: 10.1017/S1446788700023624. URL: https://www.cambridge.org/core/journals/journal-of-the-australian-mathematical-society/article/reflective-subcategories-localizations-and-factorizationa-systems/0DDCB060FE9EB733D5B2EE94B0DAEF4C (visited on 04/28/2023).

Cockett, J. R. B. and G. S. H. Cruttwell (Apr. 2014). "Differential Structure, Tangent Structure, and SDG". en. In: *Applied Categorical Structures* 22.2, pp. 331–417. ISSN: 0927-2852, 1572-9095. DOI: 10.1007/s10485-013-9312-0. URL: http://link.springer.com/10.1007/s10485-013-9312-0 (visited on 07/03/2020).

Collobert, Ronan and Koray Kavukcuoglu (2011). "Torch7: A matlab-like environment for machine learning". In: *In BigLearn, NIPS Workshop*.

Convent, Lukas, Sam Lindley, Conor Mcbride, and Craig Mclaughlin (2020). "Doo bee doo bee doo". en. In: *Journal of Functional Programming* 30. Publisher: Cambridge University Press. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796820000039. URL: http://www.cambridge.org/core/journals/journal-of-functional-programming/article/doo-bee-doo-bee-doo/DEC5F8FDABF7DE3088270E07392320DD (visited on 10/09/2020).

Dalrymple, David (Jan. 2016). *2016 : WHAT DO YOU CONSIDER THE MOST INTERESTING RECENT [SCIENTIFIC] NEWS? WHAT MAKES IT IMPORTANT?: Differentiable Programming*. URL: https://www.edge.org/response-detail/26794 (visited on 06/29/2020).

de Vilhena, Paulo Emílio and Franccois Pottier (Aug. 2023). *Verifying an Effect-Handler-Based Define-By-Run Reverse-Mode AD Library*. arXiv:2112.07292 [cs]. DOI: 10.48550/arXiv.2112.07292. URL: http://arxiv.org/abs/2112.07292 (visited on 08/29/2023).

Doczkal, Christian (Sept. 2007). "Strong Normalization of Call-By-Push-Value". Bachelor's Thesis. URL: https://www.ps.uni-saarland.de/Publications/documents/doczkal_bachelor.pdf (visited on 05/25/2023).

Dolan, Stephen and Leo White (Sept. 2022). *Stack allocation for OCaml*. Ljubljana, Slovenia. URL: https://icfp22.sigplan.org/details/ocaml-2022-papers/9/Stack-allocation-for-OCaml (visited on 08/13/2023).

Ehrhard, Thomas and Laurent Regnier (Dec. 2003). "The Differential Lambda-calculus". In: *Theor. Comput. Sci.* 309.1. Number: 1, pp. 1–41. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(03)00392-X. URL: http://dx.doi.org/10.1016/S0304-3975(03)00392-X (visited on 05/26/2019).

Elliott, Conal (July 2018). "The Simple Essence of Automatic Differentiation". In: *Proc. ACM Program. Lang.* 2.ICFP. Number: ICFP, 70:1–70:29. ISSN: 2475-1421. DOI: 10.1145/3236765. URL: http://doi.acm.org/10.1145/3236765 (visited on 05/26/2019).

Felleisen, Matthias and Daniel P. Friedman (1987). "A reduction semantics for imperative higher-order languages". en. In: *PARLE Parallel Architectures and Languages Europe*. Ed. by J. W. de Bakker, A. J. Nijman, and P. C. Treleaven. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 206–223. ISBN: 978-3-540-47181-3. DOI: 10.1007/3-540-17945-3_12.

Fiore, Marcelo and Chung-Kil Hur (Oct. 2008). "Term Equational Systems and Logics". In: *Electronic Notes in Theoretical Computer Science (ENTCS)* 218, pp. 171–192. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2008.10.011. URL: https://doi.org/10.1016/j.entcs.2008.10.011 (visited on 05/04/2024).

Forster, Yannick, Ohad Kammar, Sam Lindley, and Matija Pretnar (2019). "On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control". en. In: *Journal of Functional Programming* 29. Publisher: Cambridge University Press. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796819000121. URL: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/on-the-expressive-power-of-userdefined-effects-effect-handlers-monadic-reflection-delimited-control/3FFAA9AD05B58A1467E411F80EE4E076 (visited on 04/17/2020).

Griewank, A. and A. Walther (Jan. 2008). *Evaluating Derivatives*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics. ISBN: 978-0-89871-659-7. DOI: 10.1137/1.9780898717761. URL: https://epubs.siam.org/doi/book/10.1137/1.9780898717761 (visited on 05/26/2019).

Hascoët, Laurent and Mauricio Araya-Polo (June 2006). "Enabling user-driven Checkpointing strategies in Reverse-mode Automatic Differentiation". In: *arXiv:cs/0606042*. arXiv: cs/0606042. URL: http://arxiv.org/abs/cs/0606042 (visited on 02/28/2020).

Hascoët, Laurent and Valérie Pascual (May 2013). "The Tapenade automatic differentiation tool: Principles, model, and specification". In: *ACM Transactions on Mathematical Software* 39.3, 20:1–20:43. ISSN: 0098-3500. DOI: 10.1145/2450153.2450158. URL: https://doi.org/10.1145/2450153.2450158 (visited on 07/02/2020).

Hillerström, Daniel and Sam Lindley (Oct. 2018). "Shallow Effect Handlers". English. In: *Proceedings of 16th Asian Symposium on Programming Languages and Systems (APLAS) 2018*. Springer, Cham, pp. 415–435. DOI: 10.1007/978-3-030-02768-1_22. URL: https://www.research.ed.ac.uk/en/publications/shallow-effect-handlers (visited on 07/28/2023).

Huot, Mathieu, Sam Staton, and Matthijs Vákár (Jan. 2020). "Correctness of Automatic Differentiation via Diffeologies and Categorical Gluing". In: *arXiv:2001.02209 [cs]*. arXiv: 2001.02209. URL: http://arxiv.org/abs/2001.02209 (visited on 02/19/2020).

Iglesias-Zemmour, Patrick (Apr. 2013). *Diffeology*. en. Vol. 185. Mathematical surveys and monographs. Providence, Rhode Island: American Mathematical Soc. ISBN: 978-0-8218-9131-5.

Jacobs, Bart (1999). *Categorical logic and type theory*. en. 1st ed. Studies in logic and the foundations of mathematics v. 141. Amsterdam ; New York: Elsevier Science. ISBN: 978-0-444-50170-7.

Jia, Yangqing, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell (June 2014). "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *arXiv:1408.5093 [cs]*. arXiv: 1408.5093. URL: http://arxiv.org/abs/1408.5093 (visited on 07/02/2020).

Johnstone, Peter T. (Sept. 2002a). *Sketches of an Elephant: A Topos Theory Compendium: Volume 1*. Oxford Logic Guides. Oxford, New York: Oxford University Press. ISBN: 978-0-19-853425-9.

— (Sept. 2002b). *Sketches of an Elephant: A Topos Theory Compendium: Volume 2*. Oxford Logic Guides. Oxford, New York: Oxford University Press. ISBN: 978-0-19-851598-2.

Kammar, Ohad (2014). "An Algebraic Theory of Type-and-Effect Systems". en. PhD thesis. University of Edingburgh.

Kammar, Ohad, Sam Lindley, and Nicolas Oury (2013). "Handlers in action". en. In: *Proceedings of the 18th ACM SIGPLAN international conference on Func-*

tional programming - ICFP '13. Boston, Massachusetts, USA: ACM Press, p. 145. ISBN: 978-1-4503-2326-0. DOI: 10.1145/2500365.2500590. URL: http://dl.acm.org/citation.cfm?doid=2500365.2500590 (visited on 05/04/2020).

Kammar, Ohad and Dylan McDermott (Apr. 2018). *Factorisation systems for logical relations and monadic lifting in type-and-effect system semantics*. arXiv:1804.03460 [cs]. DOI: 10.48550/arXiv.1804.03460. URL: http://arxiv.org/abs/1804.03460 (visited on 09/05/2022).

Katsumata, Shin-ya (Jan. 2013). "Relating computational effects by ⊤⊤-lifting". en. In: *Information and Computation*. 38th International Colloquium on Automata, Languages and Programming (ICALP 2011) 222, pp. 228–246. ISSN: 0890-5401. DOI: 10.1016/j.ic.2012.10.014. URL: https://www.sciencedirect.com/science/article/pii/S0890540112001551 (visited on 09/05/2022).

Kelly, G. M. (Aug. 1980). "A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves, and so on". en. In: *Bulletin of the Australian Mathematical Society* 22.1. Publisher: Cambridge University Press, pp. 1–83. ISSN: 1755-1633, 0004-9727. DOI: 10.1017/S0004972700006353. URL: http://www.cambridge.org/core/journals/bulletin-of-the-australian-mathematical-society/article/unified-treatment-of-transfinite-constructions-for-free-algebras-free-monoids-colimits-associated-sheaves-and-so-on/FE2E25E4959E4D8B4DE721718E7F55EE (visited on 10/27/2022).

Lambek, Joachim (1968). "A Fixpoint Theorem for complete Categories." und. In: *Mathematische Zeitschrift* 103, pp. 151–161. URL: https://eudml.org/doc/170906 (visited on 05/02/2024).

Launchbury, John and Simon L. Peyton Jones (June 1994). "Lazy functional state threads". In: *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. PLDI '94. New York, NY, USA: Association for Computing Machinery, pp. 24–35. ISBN: 978-0-89791-662-2. DOI: 10.1145/178243.178246. URL: https://dl.acm.org/doi/10.1145/178243.178246 (visited on 08/03/2023).

Lawson, Charles L. (Sept. 1971). *Computing Derivatives Using W-Arithmetic and U-Arithmetic*. Internal Computing Memorandum CM–286. Pasadena, Calif.: Jet Propulsion Laboratory.

LeCun, Yann (Jan. 2018). *OK, Deep Learning has outlived its usefulness as a buzz-phrase. Deep Learning est mort. Vive Differentiable Programming!* en. Library Catalog: www.facebook.com. URL: https://www.facebook.com/yann.lecun/posts/10155003011462143 (visited on 06/28/2020).

Leijen, Daan (Sept. 2005). "Extensible records with scoped labels". en-US. In: URL: https://www.microsoft.com/en-us/research/publication/extensible-records-with-scoped-labels/ (visited on 07/25/2023).

— (June 2014). "Koka: Programming with Row Polymorphic Effect Types". In: *Electronic Proceedings in Theoretical Computer Science* 153. arXiv:1406.2061 [cs], pp. 100–126. ISSN: 2075-2180. DOI: 10.4204/EPTCS.153.8. URL: http://arxiv.org/abs/1406.2061 (visited on 09/01/2022).

— (Jan. 2017). "Type directed compilation of row-typed algebraic effects". In: *ACM SIGPLAN Notices* 52.1, pp. 486–499. ISSN: 0362-1340. DOI: 10.1145/3093333.3009872. URL: https://dl.acm.org/doi/10.1145/3093333.3009872 (visited on 07/24/2023).

Leroy, Xavier, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, Jérôme Vouillon, and KC Sivaramakrishnan (Dec. 2022). "The OCaml system: Documentation and user's manual". In: *INRIA* 5, p. 991.

Levy, P. B. (2003). *Call-By-Push-Value: A Functional/Imperative Synthesis*. en. Semantics Structures in Computation. Springer Netherlands. ISBN: 978-1-4020-1730-8. DOI: 10.1007/978-94-007-0954-6. URL: https://www.springer.com/it/book/9781402017308 (visited on 04/30/2020).

Lindley, Sam, Conor McBride, and Craig McLaughlin (Jan. 2017). "Do be do be do". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: Association for Computing Machinery, pp. 500–514. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009897. URL: https://doi.org/10.1145/3009837.3009897 (visited on 02/22/2020).

Manzyuk, Oleksandr, Barak A. Pearlmutter, Alexey Andreyevich Radul, David R. Rush, and Jeffrey Mark Siskind (Nov. 2012). "Confusion of Tagged Perturbations in Forward Automatic Differentiation of Higher-Order Functions". In: *arXiv:1211.4892 [cs, math]*. arXiv: 1211.4892. URL: http://arxiv.org/abs/1211.4892 (visited on 05/26/2019).

McBride, Conor (Feb. 2023). *shonky*. original-date: 2015-07-28T18:04:33Z. URL: https://github.com/pigworker/shonky (visited on 07/28/2023).

Merriënboer, Bart van, Alexander B. Wiltschko, and Dan Moldovan (Nov. 2017). "Tangent: Automatic Differentiation Using Source Code Transformation in Python". In: *arXiv:1711.02712 [cs, stat]*. arXiv: 1711.02712. URL: http://arxiv.org/abs/1711.02712 (visited on 07/02/2020).

Olah, Christopher (Sept. 2015). *Neural Networks, Types, and Functional Programming*. URL: http://colah.github.io/posts/2015-09-NN-Types-FP/ (visited on 05/22/2019).

Pascual, Valérie and Laurent Hascoët (2008). "TAPENADE for C". en. In: *Advances in Automatic Differentiation*. Ed. by Christian H. Bischof, H. Martin Bücker, Paul Hovland, Uwe Naumann, and Jean Utke. Lecture Notes in Computational Science and Engineering. Berlin, Heidelberg: Springer, pp. 199–209. ISBN: 978-3-540-68942-3. DOI: 10.1007/978-3-540-68942-3_18.

Pearlmutter, Barak A and Jeffrey M Siskind (Jan. 2008). "Putting the Automatic Back into AD: Part II, Dynamic, Automatic, Nestable, and Fast (CVS: 1.1)". en. In: p. 11.

Pearlmutter, Barak A and Jeffrey Mark Siskind (n.d.). "Lazy Multivariate Higher-Order Forward-Mode AD". en. In: (), p. 6.

— (Mar. 2008). "Reverse-mode AD in a Functional Framework: Lambda the Ultimate Backpropagator". In: *ACM Trans. Program. Lang. Syst.* 30.2. Number: 2, 7:1–7:36. ISSN: 0164-0925. DOI: 10.1145/1330017.1330018. URL: http://doi.acm.org/10.1145/1330017.1330018 (visited on 05/26/2019).

Pedregosa, Fabian, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay (2011). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12.85, pp. 2825–2830. ISSN: 1533-7928. URL: http://jmlr.org/papers/v12/pedregosa11a.html (visited on 08/22/2023).

Peter, David (Mar. 2023). *hyperfine*. original-date: 2018-01-13T15:49:54Z. URL: https://github.com/sharkdp/hyperfine (visited on 08/22/2023).

Pfeiffer, F W (Jan. 1987). "Automatic differentiation in prose". In: *ACM SIGNUM Newsletter* 22.1, pp. 2–8. ISSN: 0163-5778. DOI: 10.1145/24680.24681. URL: https://doi.org/10.1145/24680.24681 (visited on 07/02/2020).

Phipps-Costin, Luna, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, K. C. Sivaramakrishnan, and Sam Lindley (Aug. 2023). *Continuing*

WebAssembly with Effect Handlers. arXiv:2308.08347 [cs]. DOI: 10.48550/arXiv.2308.08347. URL: http://arxiv.org/abs/2308.08347 (visited on 08/18/2023).

Plotkin, Gordon (Dec. 1977). "LCF considered as a programming language". In: *Theoretical Computer Science* 5.3, pp. 223–255. ISSN: 0304-3975. DOI: 10.1016/0304-3975(77)90044-5. URL: https://www.sciencedirect.com/science/article/pii/0304397577900445 (visited on 08/25/2023).

Plotkin, Gordon and John Power (2001a). "Adequacy for Algebraic Effects". en. In: *Foundations of Software Science and Computation Structures*. Ed. by Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Furio Honsell, and Marino Miculan. Vol. 2030. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–24. ISBN: 978-3-540-41864-1 978-3-540-45315-4. DOI: 10.1007/3-540-45315-6_1. URL: http://link.springer.com/10.1007/3-540-45315-6_1 (visited on 05/04/2020).

— (Nov. 2001b). "Semantics for Algebraic Operations". en. In: *Electronic Notes in Theoretical Computer Science*. MFPS 2001,Seventeenth Conference on the Mathematical Foundations of Programming Semantics 45, pp. 332–345. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(04)80970-8. URL: http://www.sciencedirect.com/science/article/pii/S1571066104809708 (visited on 06/26/2020).

— (Feb. 2003). "Algebraic Operations and Generic Effects". en. In: *Applied Categorical Structures* 11.1, pp. 69–94. ISSN: 1572-9095. DOI: 10.1023/A:1023064908962. URL: https://doi.org/10.1023/A:1023064908962 (visited on 07/30/2020).

Plotkin, Gordon and Matija Pretnar (2009). "Handlers of Algebraic Effects". en. In: *Programming Languages and Systems*. Ed. by Giuseppe Castagna. Vol. 5502. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 80–94. ISBN: 978-3-642-00589-3 978-3-642-00590-9. DOI: 10.1007/978-3-642-00590-9_7. URL: http://link.springer.com/10.1007/978-3-642-00590-9_7 (visited on 05/04/2020).

Pretnar, Matija (Dec. 2015). "An Introduction to Algebraic Effects and Handlers. Invited tutorial paper". en. In: *Electronic Notes in Theoretical Computer Science* 319, pp. 19–35. ISSN: 15710661. DOI: 10.1016/j.entcs.2015.12.003. URL: https://linkinghub.elsevier.com/retrieve/pii/S1571066115000705 (visited on 08/07/2023).

Rémy, Didier (Aug. 1994). "Type inference for records in natural extension of ML". In: *Theoretical aspects of object-oriented programming: types, semantics, and language design*. Ed. by Carl A. Gunter and John C. Mitchell. Cambridge, MA, USA: MIT Press, pp. 67–95. ISBN: 978-0-262-07155-0. (Visited on 07/17/2023).

Rosický, J (1984). "Abstract tangent functors". en. In: p. 12.

Seide, Frank and Amit Agarwal (Aug. 2016). "CNTK: Microsoft's Open-Source Deep-Learning Toolkit". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, p. 2135. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2945397. URL: https://doi.org/10.1145/2939672.2945397 (visited on 07/02/2020).

Siskind, Jeffrey Mark and Barak A Pearlmutter (2005). "Perturbation confusion and referential transparency: Correct functional implementation of forward-mode AD". en. In: *Implementation and Application of Functional Languages - 17th International Workshop, IFL'05*. Ed. by A. Butterfield. Dublin, Ireland, p. 9.

Siskind, Jeffrey Mark and Barak A. Pearlmutter (Sept. 2017). "Divide-and-Conquer Checkpointing for Arbitrary Programs with No User Annotation". In: URL: https://openreview.net/forum?id=BkYYXJ9i- (visited on 02/28/2020).

Sivaramakrishnan, K. C. (Feb. 2018). *Reverse-mode Algorithmic differentiation using effect handlers*. en. URL: https://github.com/ocaml-multicore/effects-examples/blob/master/algorithmic_differentiation.ml (visited on 08/29/2023).

Sivaramakrishnan, KC, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy (June 2021). "Retrofitting effect handlers onto OCaml". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. New York, NY, USA: Association for Computing Machinery, pp. 206–221. ISBN: 978-1-4503-8391-2. DOI: 10.1145/3453483.3454039. URL: https://dl.acm.org/doi/10.1145/3453483.3454039 (visited on 07/28/2023).

Speelpenning, Bert (1980). "Compiling fast partial derivatives of functions given by algorithms". AAI8017989. Ph.D. USA: University of Illinois at Urbana-Champaign.

Šrajer, Filip, Zuzana Kukelova, and Andrew Fitzgibbon (July 2018a). *A Benchmark of Selected Algorithmic Differentiation Tools on Some Problems in Computer Vision and Machine Learning*. arXiv:1807.10129 [cs]. DOI: 10.48550/arXiv.1807.10129. URL: http://arxiv.org/abs/1807.10129 (visited on 02/28/2023).

— (July 2018b). *A Benchmark of Selected Algorithmic Differentiation Tools on Some Problems in Computer Vision and Machine Learning*. arXiv:1807.10129 [cs]. DOI: 10.48550/arXiv.1807.10129. URL: http://arxiv.org/abs/1807.10129 (visited on 05/05/2024).

Thames, Joe M. (Aug. 1969). "SLANG a problem solving language for continuous-model simulation and optimization". In: *Proceedings of the 1969 24th national conference*. ACM '69. New York, NY, USA: Association for Computing Machinery, pp. 23–41. ISBN: 978-1-4503-7493-4. DOI: 10.1145/800195.805913. URL: https://doi.org/10.1145/800195.805913 (visited on 07/02/2020).

Theano Development Team (May 2016). "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv e-prints* abs/1605.02688. URL: http://arxiv.org/abs/1605.02688.

Vákár, Matthijs (July 2020). *Denotational Correctness of Foward-Mode Automatic Differentiation for Iteration and Recursion*. en. arXiv:2007.05282 [cs]. URL: http://arxiv.org/abs/2007.05282 (visited on 09/01/2022).

Vákár, Matthijs and Tom Smeding (June 2022). *CHAD: Combinatory Homomorphic Automatic Differentiation*. en. arXiv:2103.15776 [cs]. URL: http://arxiv.org/abs/2103.15776 (visited on 07/25/2022).

van der Schaaf, Nesta (June 2020). "Diffeology, Groupoids & Morita Equivalence". MA thesis.

Walther, Andrea (2009). "Getting Started with ADOL-C". In: *Combinatorial Scientific Computing*. Ed. by Uwe Naumann, Olaf Schenk, Horst D. Simon, and Sivan Toledo. Dagstuhl Seminar Proceedings. ISSN: 1862-4405 Issue: 09061. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. URL: http://drops.dagstuhl.de/opus/volltexte/2009/2084 (visited on 07/02/2020).

Wang, Fei and Tiark Rompf (June 2018). "A Language and Compiler View on Differentiable Programming". en. In: URL: https://openreview.net/forum?id=SJxJtYkPG (visited on 08/29/2023).

Wang, Fei, Xilun Wu, Gregory Essertel, James Decker, and Tiark Rompf (Mar. 2018). "Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator". In: *arXiv:1803.10228 [cs, stat]*. arXiv: 1803.10228. URL: http://arxiv.org/abs/1803.10228 (visited on 08/24/2018).

Wang, Fei, Daniel Zheng, James Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf (July 2019). "Demystifying differentiable programming: shift/reset the penultimate backpropagator". In: *Proceedings of the ACM on Programming Languages* 3.ICFP, 96:1–96:31. DOI: 10.1145/3341700. URL: https://doi.org/10.1145/3341700 (visited on 12/11/2020).

Wang, Liang, Jianxin Zhao, and Richard Mortier (2022). *OCaml Scientific Computing: Functional Programming in Data Science and Artificial Intelligence*. en. Undergraduate Topics in Computer Science. Cham: Springer International Publishing. ISBN: 978-3-030-97644-6 978-3-030-97645-3. DOI: 10.1007/978-3-030-97645-3. URL: https://link.springer.com/10.1007/978-3-030-97645-3 (visited on 02/28/2024).

Wang, Mu (2022). "High Order Reverse Mode of Automatic Differentiation". English. ISBN: 9780355256864. Ph.D. United States – Indiana: Purdue University. URL: https://www.proquest.com/docview/1975367062/abstract/C58EB4F969814D90PQ/1 (visited on 06/30/2022).

Wengert, R. E. (Aug. 1964). "A simple automatic derivative evaluation program". In: *Communications of the ACM* 7.8, pp. 463–464. ISSN: 0001-0782. DOI: 10.1145/355586.364791. URL: https://doi.org/10.1145/355586.364791 (visited on 06/29/2020).

Wu, Nicolas, Tom Schrijvers, and Ralf Hinze (Sept. 2014). "Effect handlers in scope". In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell*. Haskell '14. Gothenburg, Sweden: Association for Computing Machinery, pp. 1–12. ISBN: 978-1-4503-3041-1. DOI: 10.1145/2633357.2633358. URL: https://doi.org/10.1145/2633357.2633358 (visited on 06/20/2020).

Yang, Zhixuan, Marco Paviotti, Nicolas Wu, Birthe van den Berg, and Tom Schrijvers (2022). "Structured Handling of Scoped Effects". en. In: *Programming Languages and Systems*. Ed. by Ilya Sergey. Vol. 13240. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 462–491. ISBN: 978-3-030-99335-1 978-3-030-99336-8. DOI: 10.1007/978-3-030-99336-8_17. URL: https://link.springer.com/10.1007/978-3-030-99336-8_17 (visited on 07/25/2022).