Provably Correct, Asymptotically Efficient, Higher-Order Reverse-Mode Automatic Differentiation

FAUSTYNA KRAWIEC, University of Cambridge, United Kingdom
SIMON PEYTON-JONES, Microsoft Research, United Kingdom
NEEL KRISHNASWAMI, University of Cambridge, United Kingdom
TOM ELLIS, Microsoft Research, United Kingdom
RICHARD A. EISENBERG, Tweag, France
ANDREW FITZGIBBON, Microsoft Research, United Kingdom

In this paper, we give a simple and efficient implementation of reverse-mode automatic differentiation, which both extends easily to higher-order functions, and has run time and memory consumption linear in the run time of the original program. In addition to a formal description of the translation, we also describe an implementation of this algorithm, and prove its correctness by means of a logical relations argument.

CCS Concepts: • **Mathematics of computing** → **Automatic differentiation**; • **Theory of computation** → *Machine learning theory*; *Semantics and reasoning*; • **Software and its engineering** → Functional languages; *Correctness*.

Additional Key Words and Phrases: Reverse-Mode AD, Wengert List, Higher-Order Functions

ACM Reference Format:
Faustyna Krawiec, Simon Peyton-Jones, Neel Krishnaswami, Tom Ellis, Richard A. Eisenberg, and Andrew Fitzgibbon. 2022. Provably Correct, Asymptotically Efficient, Higher-Order Reverse-Mode Automatic Differentiation. *Proc. ACM Program. Lang.* 6, POPL, Article 48 (January 2022), 30 pages. https://doi.org/10.1145/3498710

1 INTRODUCTION

Automatic differentiation (AD) transforms a program into a new one that computes exact analytical derivatives of the original. Automatic differentiation has a rich literature, but it is hard to find an approach to AD that has all of the following properties:

(1) *Does reverse-mode AD.* Reverse-mode AD is of particular importance in machine learning, and other optimisation applications, but it is notoriously trickier than forward mode. Forward and reverse mode are defined in Section 3.2.
(2) *Higher order.* It is capable of differentiating a fully higher-order language with first-class anonymous functions, and with sum types as well as products and arrays.
(3) *Asymptotically efficient.* Reverse-mode AD runs with only a constant-factor slow-down relative to the original (or "primal") program.
(4) *Provably correct.* It comes with a proof that it does actually differentiate the program, despite complexities such as higher order functions and data structures.

Figure 17 compares a number of recent works on these four axes, as we discuss in Section 10, but none of them enjoys all four properties. Typically the systems that come with a proof of correctness do not offer a guarantee of asymptotic efficiency, while today's most sophisticated and efficient AD

2475-1421/2022/1-ART48
https://doi.org/10.1145/3498710

systems are often large, complex software artefacts whose internals are only truly understood by a few, and without detailed formal documentation of the AD machinery itself.

Our contribution is to describe an AD system that enjoys all of these properties simultaneously:

- We describe a small but expressive fully-higher-order purely-functional language, including sums, products, **let**, and first-class lambdas (Section 2). It is easy to add more similar features later (Section 8).
- We give a standard forward-mode AD translation for this language, using so-called *dual numbers* (Section 4). The transformation is extremely simple, and is fully compositional: **let** turns into **let**, lambda into lambda, application into application, and function definition into function definition. Moreover, it has no difficulty with higher-order functions, sum types, etc. The only action is in the primitive operations on floating-point numbers. None of this is new, but it establishes a firm baseline for our subsequent steps.
- Then we show how to use a variant of the exact same dual-number approach, including its extreme simplicity, to achieve reverse-mode AD (Section 5.1).
- The algorithm of Section 5.1 may be simple, but it is also utterly impractical because it embodies two gigantic (indeed asymptotic) inefficiencies. In Sections 5.2 and 5.3 we show how to overcome these inefficiencies, one at a time. The final, monadic translation and its supporting functions still fit in a couple of Figures (9 and 11). Our use of a monadic translation appears to be new; other work uses side effects to achieve efficiency (e.g. Kmett/Pearlmutter's ad library for Haskell [Kmett et al. 2021], or Wang et al. [2019]).
- We show that our final algorithm is asymptotically efficient (Section 5.3.4) and supports separate compilation (Section 8.1).
- We give a proof of correctness of our final algorithm in Section 7, based on logical relations.
- For most of the paper we concentrate on a "main expression" to be differentiated, of type $e : \mathbb{R}^a \to \mathbb{R}$. But everything we do has a natural generalisation to main expressions with arbitrary first-order types $e : S \to T$, where S and T can include integers, strings, and sum (tagged-union) types, as well as arbitrarily nested tuples. We give this generalisation in Section 9.

We sketch some other useful generalisations in Section 8 and discuss related work in Section 10.

No single aspect of our paper is new, but their *combination* is. Our account is remarkably straightforward, using only elementary concepts. Our final algorithm has the same features as many deployed systems, recording and replaying a kind of execution trace. However, our principled, step-by-step development makes it easier to understand, easier to prove correct, and (we believe) easier to use as a basis for exploring design variations, as indeed we do through the narrative of this paper.

It is also important to emphasize that when we talk about "reverse-mode AD", our only goal is to efficiently compute the *reverse derivative* as defined in Figure 2; not to follow the precise set of transformations that have become associated with current reverse-mode strategies.

2 THE LANGUAGE OF STUDY

We describe our approach as a source-to-source translation, going from the user's written source code of a function to the source code of its derivative. We thus begin our technical content by describing the programming language we work within.

The syntax of our source language is shown in Figure 1. It is an ordinary typed λ-calculus, augmented with **let** expressions and simple data structures.

$x, y, f, \delta x, \delta y$	::=	$\ldots$	Variables
n, a, b	::=	$0, 1, \ldots$	Natural numbers
r	::=	$\ldots, 0.0, \ldots$	Reals
$e, s, t, \delta s, \delta t$	::=	$x \mid \lambda x.\, e \mid e_1\ e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2$	λ-calculus forms
	|	$\textbf{fst}\ e \mid \textbf{snd}\ e \mid (e_1, e_2)$	Products (i.e. pairs)
	|	$\textbf{inl}\ e \mid \textbf{inr}\ e \mid \textbf{case}\ e\ \textbf{of inl}\ x_1 \to e_1,\ \textbf{inr}\ x_1 \to e_2$	Sums (i.e. unions)
	|	$k \mid r \mid op \mid polyop_A$	Constants
	|	$\textbf{pure}\ e \mid \textbf{do}\ \{\ stmts\ \}$	Monadic operations
k	::=	$\langle integer\ lits \rangle \mid \langle string\ lits \rangle \mid \ldots$	Discrete literals
op	::=	$\times_{\mathbb{R}} \mid +_{\mathbb{R}} \mid \ldots \mid +_{\mathbb{Z}} \mid \ldots$	Primitive functions
$polyop$	::=	*(see table below)*	Polytypic operations
$stmts$	::=	$e \mid x \leftarrow e;\ stmts$	Lists of statements
A, B, S, T	::=	$\mathbb{R} \mid K \mid A \times B \mid A + B \mid A \to B \mid M\ A$	Types
K	::=	$\mathbb{Z} \mid String \mid \ldots$	Discrete types
Γ	::=	$\bullet \mid \Gamma, x{:}A$	Type environments

Notation conventions:

- Expression $(e_1, e_2, \ldots, e_n)$ is shorthand for nested pairs.
- Symbolic operations, like $\times_{\mathbb{R}}$ or $\oplus_A$ are written infix.
- We sometimes omit the type on + and × when the type is apparent.
- We use $\lambda(x_1, x_2).\, e$ and $\textbf{let}\ (x_1, x_2) = e_1\ \textbf{in}\ e_2$ as a shorthand for usages of **fst** and **snd**.
- Type A^n where $n \geqslant 2$ is an n-product of As.
- Type $A \multimap B$ is a synonym for $A \to B$, where the former uses its argument exactly once.
- Type $\mathbb{N}$ is a synonym for $\mathbb{Z}$, used informally as documentation.
- Type δA is a synonym for A (Section 3.2 and Section 5.1).
- We use braces { } to denote a transformation on syntax trees, as in $\overrightarrow{\mathcal{D}}\{e\}$.

The following table provides notations and types for polytypic operations. Types are given when the operator is subscripted with type A.

Primitive op.	Type at A	Notes	
$\otimes$	$(\mathbb{R} \times \delta A) \to \delta A$	Section 2	Scale each $\mathbb{R}$ component of δA
$\oplus$	$(\delta A \times \delta A) \to \delta A$	Section 5.1	Add corresponding $\mathbb{R}$ components of δA
$\odot$	$(\delta A \times \delta A) \to \mathbb{R}$	Section 9.2	Dot product of $\mathbb{R}$ components of δA
primal	$\overrightarrow{\mathcal{D}}\{A\} \to A$	Section 4.1	Select primal component of all dual numbers

Fig. 1. Our language. Shaded constructs appear only in the output of AD and are explained in later sections.

Types. The language is statically typed. Types include real numbers[1] $\mathbb{R}$, so-called discrete types K, binary products and sums, and the function type. Discrete types are types that never contain real numbers, such as integers, strings, and possibly others. These types occur only at the leaves of structures. The typing judgement $\Gamma \vdash e : A$ is absolutely standard, and is given in the appendix.

Expressions. Expressions include the full lambda calculus (variables, applications, lambda), plus **let**. The latter could be encoded with lambda and application, but it is tiresome to do so.

[1] A typical implementation would use floating-point numbers to approximate reals. Our correctness results hold for true real numbers; our space and time complexity results apply to floating-point numbers.

Product or pair types. The language includes product types $A \times B$, introduced with (e_1, e_2), and eliminated with **fst** e/**snd** e. We informally permit ourselves to use pattern-matching on pairs in lambda and let, thus $\lambda(x_1, x_2).\ e$ and **let** $(x_1, y_2) = e_1$ **in** e_2, but merely as syntactic sugar for projections. We also allow ourselves to use n-ary products (e_1, e_2, e_3, e_4) rather than nested pairs, and types A^n meaning $A \times \cdots \times A$.

Sum or tagged-union types. The language also includes sum types $A+B$, introduced with **inl** e/**inr** e and eliminated with **case**. These sum types generalise booleans and **if**[2].

Literals and primitive functions. Constants include literals $r : \mathbb{R}$, as well as discrete literals (literals whose types do not mention any real numbers), and a range of built-in primitive functions *op*. The primitive functions include functions over reals ($+_{\mathbb{R}}$, $\times_{\mathbb{R}}$, etc) and other functions ($+_{\mathbb{Z}}$, etc). Discrete literals are distinguished from real-valued literals, because the former are entirely unaffected by differentiation, as we will see throughout the paper.

The language can easily be extended with other data types and their operations (notably including arrays) as we discuss in Section 8.

Polytypic primitives. Our language lacks polymorphism, but it is convenient to have some functions that are *polytypic*; that is, whose definition depends on the type at which they are used. They are listed in the table in Figure 1. Each should be understood as an infinite family of ordinary, lambda-definable functions, indexed by type. For example, $\otimes_A$ multiples all the real-valued fields of its δA-typed argument by the given scaling factor. We can specify how to generate all the (first order) mono-typed instances like this:

$$\begin{aligned}
\otimes_{\mathbb{R}} &= \lambda(s, x).\ s \times_{\mathbb{R}} x \\
\otimes_{K} &= \lambda(s, x).\ x \\
\otimes_{A\times B} &= \lambda(s, (x_1, x_2)).\ (s \otimes_A x_1, s \otimes_B x_2) \\
\otimes_{A+B} &= \lambda(s, x).\ \mathbf{case}\ x\ \mathbf{of}\ \mathbf{inl}\ x_1 \to \mathbf{inl}\ (s \otimes_A x_1),\ \mathbf{inr}\ x_2 \to \mathbf{inr}\ (s \otimes_B x_2)
\end{aligned}$$

We will introduce the other polytypic functions as we need them.

3 THE MAIN EXPRESSION AND DIFFERENTIATION

3.1 The Main Expression

Our task is to differentiate a top-level, closed (that is, lacking free variables) expression e of type $S \to T$; we call this the *main expression*.

We will use S and T exclusively to denote the input and output types, respectively, of the main expression $e : S \to T$. (Mnemonic: "Source" and "Target" types.) For the bulk of the paper we will focus on the special case of $S = \mathbb{R}^a$ and $T = \mathbb{R}$, so $e : \mathbb{R}^a \to \mathbb{R}$, leaving the generalisation to arbitrary (first-order) S and T to Section 9. However, we often use S rather than $\mathbb{R}^a$ when we want to stress that we are talking about "the argument type of the main expression"; and similarly T when talking about the result type.

Although we initially restrict ourselves to a main expression of type $\mathbb{R}^a \to \mathbb{R}$, from the outset *we make no restrictions on the types of its sub-expressions.* It can contain sub-expressions of any type A (see Figure 1), including functions, pairs, sum types, reals, integers, and so on. In particular, e will often start with a nested collection of let-bound auxiliary functions, thus:

$$\mathbf{let}\ f_1 = \lambda x.\ e_1\ \mathbf{in}\ \mathbf{let}\ f_2 = \lambda y.\ e_2\ \mathbf{in}\ e_3$$

Our translations are quite compatible with an alternative presentation in terms of multiple top-level bindings, but it is convenient to work with the single syntactic category of expressions. The approach is also compatible with separate compilation: no whole-program analysis is needed.

[2]Booleans would be most naturally encoded as $Unit + Unit$. In fact, however, we reduce clutter in our Figures by omitting $Unit$; booleans can still be encoded, if necessary, as $\mathbb{Z} + \mathbb{Z}$.

Given a closed expression $e : \mathbb{R}^a \to \mathbb{R}^b$, the *forward derivative* of e, written $\mathcal{F}\{e\}$, is a closed expression such that

- $\mathcal{F}\{e\} : (\mathbb{R}^a \times \delta\mathbb{R}^a) \to \delta\mathbb{R}^b$
- $\forall s : \mathbb{R}^a, \delta s : \delta\mathbb{R}^a,\ [\![\mathcal{F}\{e\}]\!](s, \delta s) = \mathcal{J}[\![e]\!](s) \bullet \delta s$

The *reverse derivative* of e, written $\mathcal{R}\{e\}$, is a closed expression such that

- $\mathcal{R}\{e\} : (\mathbb{R}^a \times \delta\mathbb{R}^b) \to \delta\mathbb{R}^a$
- $\forall s : \mathbb{R}^a, \delta t : \delta\mathbb{R}^b,\ [\![\mathcal{R}\{e\}]\!](s, \delta t) = \delta t^\top \bullet \mathcal{J}[\![e]\!](s)$

Notation:

- For any function $f : \mathbb{R}^a \to \mathbb{R}^b$, its Jacobian $\mathcal{J}f : \mathbb{R}^a \to \mathbb{R}^{b\times a}$ returns f's matrix of partial derivatives (Section 3.2).
- Given a closed $e : \mathbb{R}^a \to \mathbb{R}^b$, its denotational semantics, or just denotation, $[\![e]\!]$, describes e as a mathematical function in $\mathbb{R}^a \to \mathbb{R}^b$.
- The operation "$\bullet$" is ordinary matrix multiplication.
- The type $\delta\mathbb{R}$ is just a synonym for $\mathbb{R}$, but gives a hint to the reader that it denotes a small displacement.
- A vector $\delta s : \delta\mathbb{R}^a$ is a column vector, or $a \times 1$ matrix; its transpose $\delta s^\top$ is a row vector, or $1 \times a$ matrix.

Fig. 2. Correctness criteria for forward and reverse-mode derivatives

3.2 Forward and Reverse Derivatives, and the Jacobian

What exactly does automatic differentiation mean, and what does it mean for AD to be correct? To answer that we need some definitions. Given a function $f : \mathbb{R}^a \to \mathbb{R}^b$, the Jacobian $\mathcal{J}f : \mathbb{R}^a \to \mathbb{R}^{b\times a}$ is a function that, for each argument of type $\mathbb{R}^a$, gives a $b \times a$ matrix of partial derivatives[3]:

- For every type A, δA denotes the type of small changes to a value of type A. Because a change to a real number is described by a real number, $\delta\mathbb{R} = \mathbb{R}$, but we will still suggestively write $\delta\mathbb{R}$ and $\delta\mathbb{R}^n$ where we wish to think about differences.
- Given a function $f : \mathbb{R}^a \to \mathbb{R}^b$, let $\mathcal{J}f : \mathbb{R}^a \to \delta\mathbb{R}^{b\times a}$ be a function that produces f's *Jacobian* matrix, a $b \times a$ matrix of partial derivatives.
- $\mathcal{J}f(\vec{x})$ has a *column* for each of the a components of the input $\mathbb{R}^a$.
- $\mathcal{J}f(\vec{x})$ has a *row* for each of the b components of the output $\mathbb{R}^b$.
- The (i, j) element of $\mathcal{J}f(\vec{x})$ is $\frac{\partial f_i}{\partial x_j}(\vec{x})$, the partial derivative at $\vec{x}$ of f's i'th output with respect to its j'th input.

Now suppose $e : \mathbb{R}^a \to \mathbb{R}^b$, where e is (the syntax tree of) a closed expression. Then $\mathcal{F}\{e\}$ and $\mathcal{R}\{e\}$ are (the syntax trees of) the forward and reverse derivatives of e, with the correctness criteria given in Figure 2. This Figure says what it means for the source-to-source translations $\mathcal{F}\{e\}$ and $\mathcal{R}\{e\}$ to be *correct*, but it does not *define* them – doing that is the business of the rest of the paper.

The following lemma is a trivial consequence of the correctness criteria:

Lemma 1 (Relationship between forward and reverse mode). *For any closed* $e : \mathbb{R}^a \to \mathbb{R}^b$, $s : \mathbb{R}^a, \delta s : \delta\mathbb{R}^a, \delta t : \delta\mathbb{R}^b$, *we have* $\delta t^\top \bullet [\![\mathcal{F}\{e\}]\!](s, \delta s) = [\![\mathcal{R}\{e\}]\!](s, \delta t) \bullet \delta s$

Proof. Simply substitute for $[\![\mathcal{F}\{e\}]\!]$ and $[\![\mathcal{R}\{e\}]\!]$ using the correctness criteria in Figure 2. □

[3] A $b \times a$ matrix has b rows and a columns. The (i, j) element is in row i and column j.

Forward and reverse mode can both compute exactly the same derivatives, but they may differ radically in their efficiency, depending on the application. To see this, notice that for any $\vec{x} : \mathbb{R}^a$ we can reconstruct the Jacobian matrix $\mathcal{J}f(\vec{x})$ in either of these ways:

- Forward Plan: make a calls to $\mathcal{F}\{e\}$ $(\vec{x}, \mathit{onehot}_{\mathbb{R}^a}\ j)$, for $j \in 1..a$, each yielding a value in $\mathbb{R}^b$, namely the j'th column of the Jacobian matrix.
- Reverse Plan: make b calls to $\mathcal{R}\{e\}$ $(\vec{x}, \mathit{onehot}_{\mathbb{R}^b}\ i)$, for $i \in 1..b$, each yielding a value in $\mathbb{R}^a$, namely the i'th row of the Jacobian matrix.

Here $(\mathit{onehot}_{\mathbb{R}^a}\ j)$ is the value $(0, \ldots, 0, 1, 0, \ldots, 0) \in \mathbb{R}^a$, where the 1 is in the j'th position.

In the special case of machine learning, and other optimisation scenarios, the main expression is usually a "loss function", with a type like $\mathbb{R}^a \to \mathbb{R}$, where a is large, say 10^6. The a inputs are the model parameters, whose values we wish to learn. The output is the loss, the objective that we are trying to minimise. Learning proceeds by adjusting each of a inputs in proportion to their contribution to the loss, so we need a partial derivative of the function with respect to each of those a inputs.

If $a = 10^6$ and $b = 1$, the Reverse Plan computes all the required partial derivatives in one call, rather than 10^6 calls for Forward Plan. Since each call repeats all of the work of the original, or *primal*, program, the Reverse Plan is vastly more efficient — provided of course that $\mathcal{R}\{e\}$ is itself efficient.

However in other applications[4] (e.g. in which $b \gg a$) forward mode might be more suitable. Moreover, as we shall see, reverse mode trades space for time, accumulating a data structure that records the execution of the program, and then somehow running that record "backwards". So reverse mode can take a lot more space than forward, leading to work on checkpointing (which we do not discuss here).

4 FORWARD-MODE AUTOMATIC DIFFERENTIATION

To establish our notation, we start from a familiar forward-mode AD based on dual numbers, as presented in, for example, [Huot et al. 2020]. We begin by giving a fully compositional translation $\overrightarrow{\mathcal{D}}$ for terms and types (Section 4.1), after which we describe a *wrapper* that uses $\overrightarrow{\mathcal{D}}$ to build $\mathcal{F}\{e\}$, the function we really want (Section 4.2).

Nothing in this section is truly new, although the construction of the wrapper is seldom made as explicit as we do here. It will play an important part in our subsequent development.

4.1 The Forward-Mode AD Translation

Figure 3 gives the translation in three parts, overloading $\overrightarrow{\mathcal{D}}$ (where the arrow denotes "forward", not "vector") for all three purposes:

- A translation on types $\overrightarrow{\mathcal{D}}\{A\}$
- A translation on typing contexts $\overrightarrow{\mathcal{D}}\{\Gamma\}$
- A translation on terms $\overrightarrow{\mathcal{D}}\{e\}$. The key translation invariant is that translated terms have translated types, as shown in Figure 3.

Looking at the type translation in Figure 3, you can see that almost all types are translated homomorphically: products translate to products, functions translate to functions, and all discrete types K (including integers and strings) translate to themselves. The only exception is at the real-valued leaves: real numbers $\mathbb{R}$ translate to a pair of reals $\mathbb{R} \times \delta\mathbb{R}$.

[4] Corliss et al. [2002] is a good survey of applications of AD.

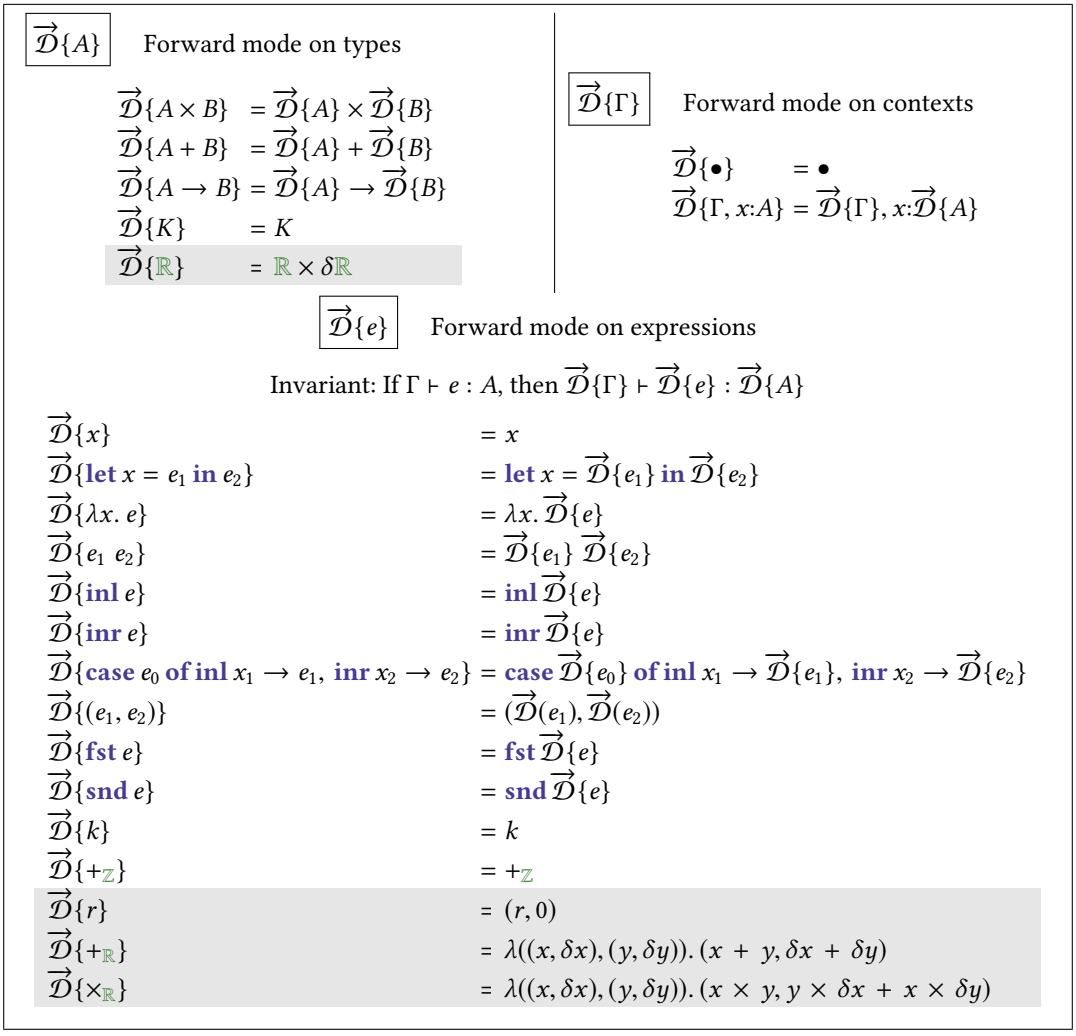

Fig. 3. Forward-mode translation. Note that everything is homomorphic except a few highlighted cases. Note also that the type of every binder $x{:}A$ changes (in **let**, **case**, and lambda), to $x : \overrightarrow{\mathcal{D}}\{A\}$.

The bottom line is this: *all that happens in the type translation is that each real is replaced with a pair of reals*—hence the term "dual number". Notice that *only real numbers are dualised*; integers (and booleans, strings, etc if we had them) are unaffected.

The translation on terms is, for this reason, laughably simple: pairs translate to pairs, **let** expressions to **let** expressions, and (crucially for the higher-order case) lambdas translate to lambdas and applications translate to applications. The only interesting part is the translation for literals and primitive functions over the reals. As you can see in Figure 3, real-number literals translate to a pair of that literal and zero; while each primitive function over reals has its own translation, one that expresses the mathematical knowledge of what the derivative of that function is. Discrete literals and their functions are entirely unaffected.

$$
\begin{array}{rl}
\text{Given} & e \;:\; \mathbb{R}^a \to \mathbb{R} \\
\text{and} & \overrightarrow{\mathcal{D}}\{e\} \;:\; (\mathbb{R} \times \delta\mathbb{R})^a \to (\mathbb{R} \times \delta\mathbb{R}) \\
\text{we produce} & \mathcal{F}\{e\} \;:\; (\mathbb{R}^a \times \delta\mathbb{R}^a) \to \delta\mathbb{R} \\
& \mathcal{F}\{e\} = \lambda((\ldots, s_i, \ldots), (\ldots, \delta s_i, \ldots)). \\
& \qquad\qquad \mathbf{snd}\,(\overrightarrow{\mathcal{D}}\{e\}\ (\ldots, (s_i, \delta s_i), \ldots))
\end{array}
$$

Fig. 4. The wrapper for forward differentiation

Although it looks simple, the transformation conceals one subtlety: the type of each binder changes, according to the type translation in Figure 3. For example:

$$\overrightarrow{\mathcal{D}}\{\lambda(x{:}\mathbb{R}).\ e\} \;=\; \lambda(x{:}\mathbb{R} \times \delta\mathbb{R}).\ \overrightarrow{\mathcal{D}}\{e\} \tag{1}$$

Notice the way the type of x changes between the original and transformed program.

The first component of each dual number is exactly what appeared in the original program, the so-called *primal* value. To be more precise, for any $e : S \to T$ and $s : \overrightarrow{\mathcal{D}}\{S\}$:

$$[\![\mathit{primal}_T\ (\overrightarrow{\mathcal{D}}\{e\}\ s)]\!] = [\![e\ (\mathit{primal}_S\ s)]\!]$$

where $\mathit{primal}_A : \overrightarrow{\mathcal{D}}\{A\} \to A$ is an operation that removes the second (delta) component of each dual number returning just the primal value, recurring homomorphically over products and sums. Keeping the original values alongside the tangent-space ones makes the transformation *compositional*; for example, the forward derivative of an application, $\overrightarrow{\mathcal{D}}\{e_1\ e_2\}$, is obtained by taking the derivatives of e_1 and e_2 separately, and combining them: $\overrightarrow{\mathcal{D}}\{e_1\}\ \overrightarrow{\mathcal{D}}\{e_2\}$ (see Figure 3).

Primal values are often used to compute delta values, for example in the translation for $\times_{\mathbb{R}}$:

$$\overrightarrow{\mathcal{D}}\{\times_{\mathbb{R}}\} = \lambda((x, \delta x), (y, \delta y)).\ (x \times y,\ y \times \delta x\ +\ x \times \delta y)$$

To compute the second component of the pair, we need the primal values of x and y.

4.2 The Wrapper

Figure 4 connects this recursively-defined $\overrightarrow{\mathcal{D}}\{e\}$ with the forward derivative $\mathcal{F}\{e\}$ that we established as our goal in Section 3. We call $\mathcal{F}\{e\}$ a *wrapper* around $\overrightarrow{\mathcal{D}}\{e\}$. As you can see, the bulk of the work is some tiresome shuffling, from a pair of a-tuples (passed into $\mathcal{F}\{e\}$) into an a-tuple of dual-number pairs (which is what $\overrightarrow{\mathcal{D}}\{e\}$ needs). Finally $\overrightarrow{\mathcal{D}}\{e\}$ returns a dual number, from which we extract the second component, discarding the primal result. To reduce clutter, we describe $\mathcal{F}\{e\}$ only for the special case of $S = \mathbb{R}^a$, $T = \mathbb{R}$. We generalise to arbitrary S and T in Section 9.

4.3 Correctness

Proposition 2 (Correctness of the forward-mode transformation).
If $e : \mathbb{R}^a \to \mathbb{R}$, $\vec{x} : \mathbb{R}^a$, *and* $\delta s : \delta\mathbb{R}^a$, *then* $[\![\mathcal{F}\{e\}]\!](\vec{x}, \delta s) = \mathcal{J}[\![e]\!](\vec{x}) \bullet \delta s$.

Proof. We refer to [Huot et al. 2020] for the proof of this statement. Because of the presence of function types, it is proved by means of a logical relations argument, which defines the correctness of the dual number representation at the real type, and at function types relates functions which (hereditarily) preserve correctness. □

The correctness argument for our reverse-mode algorithm, in Section 7, is directly inspired by their argument.

$\overleftarrow{\mathcal{D}}^1_S\{A\}$
Reverse mode on types

$\overleftarrow{\mathcal{D}}^1_S\{\mathbb{R}\} = \mathbb{R} \times \delta S$

$\overleftarrow{\mathcal{D}}^1_S\{e\}$ Reverse mode on expressions

Invariant: If $\Gamma \vdash e : A$, then $\overleftarrow{\mathcal{D}}^1_S\{\Gamma\} \vdash \overleftarrow{\mathcal{D}}^1_S\{e\} : \overleftarrow{\mathcal{D}}^1_S\{A\}$

$$\begin{aligned}
\overleftarrow{\mathcal{D}}^1_S\{r\} &= (r, \mathit{zero}_S) \\
\overleftarrow{\mathcal{D}}^1_S\{+_{\mathbb{R}}\} &= \lambda((x, \delta x), (y, \delta y)).\, (x + y, \delta x \oplus_S \delta y) \\
\overleftarrow{\mathcal{D}}^1_S\{\times_{\mathbb{R}}\} &= \lambda((x, \delta x), (y, \delta y)).\, (x \times y, (y \otimes_S \delta x) \oplus_S (x \otimes_S \delta y))
\end{aligned}$$

Fig. 5. Reverse-mode AD #1 (terribly inefficient, as every real carries a value of type δS), where omitted rules are as in Figure 3, with $\overleftarrow{\mathcal{D}}^1_S$ in place of $\overrightarrow{\mathcal{D}}$.

5 FINDING THE REVERSE DERIVATIVE

Having seen how to use dual numbers to get the *forward* derivative, we now replay the same script to get the *reverse* derivative. It is provably correct (always a good starting point), but it is tremendously inefficient. We will remedy that problem in subsequent sections.

5.1 One Pass instead of Many

Forward-mode AD is efficient if one only wants one partial derivative. However, to do gradient descent on a function $f : \mathbb{R}^a \to \mathbb{R}$, we need *all* of the partial derivatives. A naive approach for doing so is to simply run the forward-mode AD algorithm a times, the Forward Plan of Section 3.2.

But instead of running a forward passes, each with a different one-hot vector in its input, an obvious idea is to run *one* pass, computing a vector of all a results at once. More specifically, in our dual-number approach, instead of pairing each real with another real, thus $\mathbb{R} \times \delta\mathbb{R}$, we can pair it with vector of a reals, thus $\mathbb{R} \times \delta\mathbb{R}^a$. Now, we can compute all of the partial derivatives in a single run of the program. Although this is still a dual-number approach, it is extremely closely connected with the so-called *back-propagators* of Wang et al. [2019], as we explain in Section 10. However, there is no need to understand the mysteries of back-propagator functions to follow the rather simple idea of returning n results all at once, rather than calling a function n times.

To make this idea concrete, the new translation $\overleftarrow{\mathcal{D}}^1$ is given in Figure 5. The "1" superscript is there because this is our first version; subsequent versions improve upon this one. It follows precisely the same pattern as the dual-number forward-mode AD translation, comprising a translation on types, contexts, and terms. The translation on types is identical to that in Figure 3, except for the treatment of reals; on every other type, it acts homomorphically as before. Similarly, the translation on terms is identical to that in Figure 3 except for the translation on real numbers and their primitive functions. Indeed, in order to focus attention on the differences, Figure 5 does not even give the translations for constructs when these translations are identical to those for $\overrightarrow{\mathcal{D}}$.

In the type translation, we see $\overleftarrow{\mathcal{D}}^1_S\{\mathbb{R}\} = \mathbb{R} \times \delta S$, so we still have a dual-number approach, but now each real number is paired with a value of type δS, where S is, as always, the argument type of the main expression (see Section 3.1). Here δS is a synonym for S, but suggestive that its values range over small displacements (c.f. $\delta\mathbb{R}$ in Section 3.2).

In the translation for addition on reals, $+_{\mathbb{R}}$, we use $\delta s_1 \oplus_S \delta s_2$ (Figure 1) to combine two values of type δS; this operation (as others) assumes that the two values δs_1 and δs_2 have the same shape, and straightforwardly adds corresponding reals. Similarly, in the translation for multiplication, we use $\otimes$ to scale a δS value with a scalar real; $r \otimes_S \delta s$ multiplies each real in δs by r.

$$
\begin{array}{rl}
\text{Given} & e \;:\; \mathbb{R}^a \to \mathbb{R} \\
\text{and} & \overleftarrow{\mathcal{D}}^1_{\mathbb{R}^a}\{e\} \;:\; (\mathbb{R} \times \delta\mathbb{R}^a)^a \to (\mathbb{R} \times \delta\mathbb{R}^a) \quad \text{(as defined in Figure 5)} \\
\text{we produce} & \mathcal{R}^1\{e\} \;:\; (\mathbb{R}^a \times \delta\mathbb{R}) \to \delta\mathbb{R}^a \\
& \mathcal{R}^1\{e\} \;=\; \lambda((\ldots, s_i, \ldots), \delta t).\ \mathbf{snd}\,(\overleftarrow{\mathcal{D}}^1_{\mathbb{R}^a}\{e\}\ (\ldots, (s_i, \delta t \otimes_{\mathbb{R}^a} \mathit{onehot}_{\mathbb{R}^a}\ i), \ldots)) \\
\text{using primitive} & \mathit{onehot}_A \;:\; \mathbb{N} \to \delta A
\end{array}
$$

Fig. 6. Reverse-mode wrapper around $\overleftarrow{\mathcal{D}}^1$

5.1.1 *The Wrapper.*

Next we build the wrapper $\mathcal{R}^1\{e\}$ (Figure 6) that connects $\overleftarrow{\mathcal{D}}^1_S\{e\}$ to the API defined in Section 3.2. As before, the wrapper is for the special case of $S = \mathbb{R}^a$, $T = \mathbb{R}$; see Section 9 for the general case.

The wrapper applies $\overleftarrow{\mathcal{D}}^1_{\mathbb{R}^a}\{e\}$ to an n-vector of dual numbers, each of which is a pair of s_i with $\delta t \otimes_{\mathbb{R}^a} \mathit{onehot}_{\mathbb{R}^a}\ i$. Recall that the primitive $\mathit{onehot}_{\mathbb{R}^a}\ i \;:\; \mathbb{R}^a$ returns an a-vector that is zero everywhere except at position i.

We note in passing that the argument type to $\overleftarrow{\mathcal{D}}^1_{\mathbb{R}^a}\{e\}$, namely $(\mathbb{R} \times \delta\mathbb{R}^a)^a$ has size *quadratic in a*, a serious concern that we return to in Section 5.2.

5.1.2 *Summary.*

It is perhaps remarkable that such a simple translation supports compositional reverse-mode AD for a language with unrestricted higher order functions. It is far from efficient, as we will see, but it *is* simple. We defer a correctness proof until Section 7, when we have the final version in hand.

5.2 Towards Efficiency: Intensional Updates

The reverse-mode AD of Section 5.1 does only a constant-factor more *operations* than the original program. This must be so: just look at the transformation in Figure 5. The transformed program will take precisely the same execution path as the original, while the primitive operations are each modified to do a couple of extra operations alongside their original job. For example, we have

$$\overleftarrow{\mathcal{D}}^1_S\{\times_{\mathbb{R}}\} \;=\; \lambda((x, \delta s_1), (y, \delta s_2)).\ (x \times y, (y \otimes_S \delta s_1) \oplus_S (x \otimes_S \delta s_2))$$

The "extra operations" are the calls to $\oplus_S$ and $\otimes_S$. Alas, each of these extra operations is extremely expensive. In the case where $S = \mathbb{R}^a$, and supposing $a = 10^6$, that means that *each of those* $\oplus_S$ *or* $\otimes_S$ *operations consume and produce a million-element vector.* What was a single floating point instruction in the original program has blown up to a million instructions, to say nothing of the memory requirements. This will never work.

Another complication is concealed in the transformation

$$\overleftarrow{\mathcal{D}}^1_S\{r\} = (r, \mathit{zero}_S)$$

Here we are required to produce, out of thin air, a zero value of type δS. When $S = \mathbb{R}^a$, doing so might be expensive but is not difficult. But if $S = S_1 + S_2$ it is much trickier. Should we produce a zero of shape ($\mathbf{inl}\ s_1$) or ($\mathbf{inr}\ s_2$)? Well, it should be the same as in *the argument originally given to the main program.* So now this zero value depends not only on the *type* S but the *value* given to the original program.

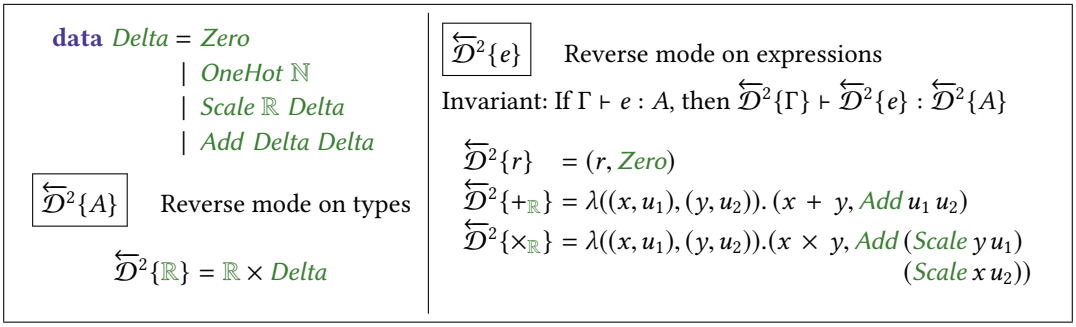

Fig. 7. Reverse-mode AD #2 (still inefficient, because of lost sharing), where omitted rules are as in Figure 3, with $\overleftarrow{\mathcal{D}}^2$ in place of $\overrightarrow{\mathcal{D}}$

5.2.1 *From Delta Values to Deltas of Values.*

One way to finesse both of these problems is to change the type translation, like this:

$$\overleftarrow{\mathcal{D}}^2\{\mathbb{R}\} = \mathbb{R} \times Delta$$

where we can think of *Delta* as specifying the *difference*, or "delta", between two values of type S. We might imagine choosing *Delta* to be a function of type $S \to S$, that transforms one S value into another nearby one. However, *we only need a limited vocabulary of such functions*, so it is much nicer to represent deltas by a simple algebraic data type[5], given in Figure 7. It turns out that four constructors suffice to describe all the deltas we need. This formulation allows, for example, inspection of a *Delta* and affords the chance for optimisation. This particular definition works only for $S = \mathbb{R}^a$ (the *OneHot* constructor does not work for more elaborate choices of S), but we will generalise our final version in Section 9.

Another way to regard *Delta* is as a sparse, or symbolic, representation for the values of type δS whose size troubled us in Figure 5.

We can interpret this data structure as a function, via the functions $eval_2$ and *addAt*, defined in Figure 8. Note that both of these functions produce *linear* transformations $\mathbb{R}^a \multimap \mathbb{R}^a$. Here, "linear" refers to the property maintained by linear type systems, where the input is guaranteed to be used exactly once. The implementation of *addAt* can thus be extremely efficient: *it can work by in-place mutation*; linearity assures us that the input to *addAt* will not be used again. So we totally eliminate the construction of big vectors of zeros and one-hot vectors. Instead, we have a single, mutable value of type $\mathbb{R}^a$ which we update in place, guided by the *Delta*.

5.2.2 *The Wrapper.*

The wrapper $\mathcal{R}^2\{e\}$ (for the special case of $S = \mathbb{R}^a$, $T = \mathbb{R}$) is described in Figure 8. We apply $\overleftarrow{\mathcal{D}}^2\{e\}$ to s_0, an initial value obtained by pairing each real number in the input with a *Delta* for that slot in the input. Then we run the $eval_2$ function on a zero starting vector z and the *Delta* u returned by $\overleftarrow{\mathcal{D}}^2\{e\}\ s_0$.

In retrospect, we can see that the *Delta* data structure plays a similar role to that of the "trace", or "tape", or "Wengert list" of other well-established approaches to reverse-mode AD [Griewank and Walther 2008]. Looked at from a sufficient distance, we have arrived a solution similar to these other approaches, but by a very different and perhaps more principled route.

[5] You can see this simply as representing the delta-function *intensionally*, rather than extensionally, using defunctionalisation [Reynolds 1998].

Given $e : \mathbb{R}^a \to \mathbb{R}$

and $\overleftarrow{\mathcal{D}^2}\{e\} : (\mathbb{R} \times \mathit{Delta})^a \to (\mathbb{R} \times \mathit{Delta})$ (as defined in Figure 7)

we produce $\mathcal{R}^2\{e\} : (\mathbb{R}^a \times \delta\mathbb{R}) \to \delta\mathbb{R}^a$

$$
\begin{aligned}
\mathcal{R}^2\{e\} = \lambda((\ldots, s_i, \ldots), \delta t).\ & \\
& \textbf{let } s_0 = (\ldots, (s_i, \mathit{OneHot}\ i), \ldots) \textbf{ in} \\
& \textbf{let } z = (\ldots, 0, \ldots) \textbf{ in} \\
& \textbf{let } (t, u) = \overleftarrow{\mathcal{D}^2}\{e\}\ s_0 \textbf{ in} \\
& \mathit{eval}_2\ \delta t\ u\ z
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{eval}_2 : \mathbb{R} \to \mathit{Delta} \to \mathbb{R}^a \multimap \mathbb{R}^a \\
&\mathit{eval}_2\ x\ \mathit{Zero}\ \vec{x} = \vec{x} \\
&\mathit{eval}_2\ x\ (\mathit{OneHot}\ i)\ \vec{x} = \mathit{addAt}\ i\ x\ \vec{x} \\
&\mathit{eval}_2\ x\ (\mathit{Scale}\ y\ u)\ \vec{x} = \mathit{eval}_2\ (x \times y)\ u\ \vec{x} \\
&\mathit{eval}_2\ x\ (\mathit{Add}\ u_1\ u_2)\ \vec{x} = \mathit{eval}_2\ x\ u_2\ (\mathit{eval}_2\ x\ u_1\ \vec{x})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{addAt} : \mathbb{N} \to \mathbb{R} \to \mathbb{R}^a \multimap \mathbb{R}^a \\
&\mathit{addAt}\ i\ x\ \vec{y} = \ldots \\
&\quad \text{-- Return } \vec{y} \text{ with its } i\text{'th component} \\
&\quad \text{-- increased by } x
\end{aligned}
$$

Fig. 8. Reverse-mode wrapper around $\overleftarrow{\mathcal{D}^2}$

5.3 Finally Efficient: A Monadic Translation

The translation of Section 5.2 eliminates one form of asymptotic inefficiency (blowing up values by a factor of a) but sadly introduces another that is just as bad. Consider f and its translation $\overleftarrow{\mathcal{D}^2}\{f\}$:

$$
\begin{aligned}
&f : \mathbb{R} \to \mathbb{R} \\
&f = \lambda x.\ \textbf{let } y{:}\mathbb{R} = e \textbf{ in } y + y
\end{aligned}
\qquad
\begin{aligned}
&\overleftarrow{\mathcal{D}^2}\{f\} : (\mathbb{R} \times \mathit{Delta}) \to (\mathbb{R} \times \mathit{Delta}) \\
&\overleftarrow{\mathcal{D}^2}\{f\} = \lambda x.\ \textbf{let } (y{:}\mathbb{R}, \vec{y}{:}\mathit{Delta}) = \overleftarrow{\mathcal{D}^2}\{e\} \textbf{ in } (y + y, \mathit{Add}\ \vec{y}\ \vec{y})
\end{aligned}
$$

Here the $\vec{y} : \mathit{Delta}$ is a perhaps-large data structure, arising from executing $\overleftarrow{\mathcal{D}^2}\{e\}$; and $\overleftarrow{\mathcal{D}^2}\{f\}$ returns a dual number whose second Delta component is $\mathit{Add}\ \vec{y}\ \vec{y}$. This latter structure is the problem, because the eval_2 function has no way to know that the two $\vec{y}$ are the same, and so will simply evaluate $\vec{y}$ twice. In effect, we totally lose sharing of the **let**.

This loss of sharing is unacceptable. Consider a program with nested bindings:

$$\textbf{let } x_1 = x + x \textbf{ in let } x_2 = x_1 + x_1 \textbf{ in let } x_3 = x_2 + x_2 \textbf{ in let } \ldots \textbf{ in } x_n + x_n$$

The primal program and transformed programs both execute in time linear in n, but the eval_2 function unravels the deeply-shared Delta structure into a tree of exponential size. This will not do.

5.3.1 *A Monadic Translation.*

The trick is, of course, to build a Delta structure with *explicit* rather than *implicit* sharing, and that is achieved by the translation[6] in Figure 9 (defining $\overleftarrow{\mathcal{D}}$, our final answer, with no version number), and its supporting functions in Figure 11. The translation itself is totally unsurprising: it is Moggi's call-by-value monadic translation [Moggi 1991].

[6]We adopt several conveniences inspired by Haskell:

- **do**-notation allows sequencing of monadic operations by using *statements*.
- In the statement $x \leftarrow e$, when $e : M\ A$, then $x : A$.
- The expression **pure** e has type $M\ A$ when $e : A$.
- The type of the **do**-expression is the same as the type of the expression in the last statement.

First, the *Delta* structure is extended with two new constructors: *Var DeltaId* and *Let DeltaId Delta Delta* which, as their names suggest, describe explicit sharing in a *Delta* structure. The type *DeltaId* is the *name*, or index, of a *Delta*. In fact, as Figure 11 shows, a *DeltaId* is just a natural number $\mathbb{N}$.

Perhaps surprisingly, in exchange, we can get rid of the *OneHot* constructor. Recall that it was only used in the wrapper, when constructing the initial argument value s_0 (Section 5.2.2). We will see in Section 5.3.2 that we can use *Var* instead.

Second, the reverse-mode translation $\overleftarrow{\mathcal{D}}\{e\}$ in Figure 9 transforms e with a standard monadic translation that makes evaluation order explicit. This monad *M* is a state monad whose state, *DeltaState*, comprises: (a) a unique supply to allocate fresh *DeltaIds*, and (b) an ordered list of *Delta* bindings:

```
type M a          = DeltaState → (a, DeltaState)
type DeltaState   = (DeltaId, DeltaBinds)
type DeltaBinds   = [(DeltaId, Delta)]
```

It is an invariant of the translation that the *Delta* paired with any dual number is of the form *Zero* or *Var id*, and hence can be freely copied without loss of sharing. This invariant comes under threat in the translations for $+_{\mathbb{R}}$ and $\times_{\mathbb{R}}$, when we construct new *Delta* values such as *Add* u_1 u_2. This is the whole point of the monad: it supports an operation *deltaLet* : *Delta* → *M DeltaId* which, given a *Delta* structure u, allocates a fresh *DeltaId*, say *uid*; extends the bindings with the pair (*uid*, u); and returns the *uid*. This function is used in $\overleftarrow{\mathcal{D}}\{+_{\mathbb{R}}\}$ and $\overleftarrow{\mathcal{D}}\{\times_{\mathbb{R}}\}$ to ensure that each new *Delta* value gets its own binding in the *DeltaBinds*, so we can return a *Var*, satisfying the invariant.

5.3.2 *The Wrapper.*

The new monadic translation needs a new wrapper, which as usual we give for the special case of $S = \mathbb{R}^a$, $T = \mathbb{R}$, in Figure 10. This wrapper repays careful attention:

(1) First we construct an initial value s_0, by pairing each real number in the input with a *DeltaId* that identifies that particular slot of the input.
(2) Then we apply $\overleftarrow{\mathcal{D}}\{e\}$ to s_0, to get a value of type *M* ($\mathbb{R} \times$ *Delta*)
(3) We use the function *runDelta* to execute the monadic computation, starting off the *DeltaId* supply at $a+1$, the first free *DeltaId*. This function, defined in Figure 11, runs the computation and then wraps the returned *Delta* in a nested series of *Let* bindings, This is the only place that *Let* is used.
(4) Now the *eval* function computes *finalMap* : *DeltaMap*. In general *eval* δt u m extends the map m : *DeltaMap* with the extra contributions described by u to the free *DeltaId*s of u, scaled by δt.
(5) So *finalMap* will have a binding for each *DeltaId* in $1..n$ (unless it is unused). These are the final partial derivatives, which we extract with *lookupOrZero*.

Figure 11 gives the rather simple definition of *eval*.

5.3.3 *Using an Array.*

The monad *M* is a state monad, and can therefore be implemented using mutable state: we simply need a mutable structure to which we can add new *DeltaBindings* as execution proceeds. For example, a dynamically-growable mutable array would work well.

The *eval* function takes and returns a *DeltaMap*, and uses it entirely linearly, as we have suggested by the linear arrow in its type. So we can readily use a mutable array, indexed by *DeltaId* (a natural number), rather than a *Map* data structure, thus

```
type DeltaMap = Array ℝ   -- Mutable
```

$\boxed{\overleftarrow{\mathcal{D}}\{A\}}$ Reverse mode on types

$$
\begin{aligned}
\overleftarrow{\mathcal{D}}\{A \times B\} &= \overleftarrow{\mathcal{D}}\{A\} \times \overleftarrow{\mathcal{D}}\{B\} \\
\overleftarrow{\mathcal{D}}\{A + B\} &= \overleftarrow{\mathcal{D}}\{A\} + \overleftarrow{\mathcal{D}}\{B\} \\
\overleftarrow{\mathcal{D}}\{A \to B\} &= \overleftarrow{\mathcal{D}}\{A\} \to M\ \overleftarrow{\mathcal{D}}\{B\} \\
\overleftarrow{\mathcal{D}}\{K\} &= K \\
\overleftarrow{\mathcal{D}}\{\mathbb{R}\} &= \mathbb{R} \times \mathit{Delta}
\end{aligned}
$$

$\boxed{\overleftarrow{\mathcal{D}}\{e\}}$ Reverse mode on expressions

Invariant: If $\Gamma \vdash e : A$, then $\overleftarrow{\mathcal{D}}\{\Gamma\} \vdash \overleftarrow{\mathcal{D}}\{e\} : M\ \overleftarrow{\mathcal{D}}\{A\}$

$$
\begin{aligned}
\overleftarrow{\mathcal{D}}\{x\} &= \mathbf{pure}\ x \\
\overleftarrow{\mathcal{D}}\{\lambda x.\ e\} &= \mathbf{pure}\ \lambda x.\ \overleftarrow{\mathcal{D}}\{e\} \\
\overleftarrow{\mathcal{D}}\{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\} &= \mathbf{do}\ \{x \leftarrow \overleftarrow{\mathcal{D}}\{e_1\};\ \overleftarrow{\mathcal{D}}\{e_2\}\} \\
\overleftarrow{\mathcal{D}}\{e_1\ e_2\} &= \mathbf{do}\ \{f \leftarrow \overleftarrow{\mathcal{D}}\{e_1\};\ x \leftarrow \overleftarrow{\mathcal{D}}\{e_2\};\ f\ x\} \\
\overleftarrow{\mathcal{D}}\{\mathbf{inl}\ e\} &= \mathbf{do}\ \{x \leftarrow \overleftarrow{\mathcal{D}}\{e\};\ \mathbf{pure}\ (\mathbf{inl}\ x)\} \\
\overleftarrow{\mathcal{D}}\{\mathbf{inr}\ e\} &= \mathbf{do}\ \{x \leftarrow \overleftarrow{\mathcal{D}}\{e\};\ \mathbf{pure}\ (\mathbf{inr}\ x)\} \\
\overleftarrow{\mathcal{D}}\{\mathbf{case}\ e_0\ \mathbf{of\ inl}\ x_1 \to e_1,\ \mathbf{inr}\ x_2 \to e_2\} &= \mathbf{do}\ \{x \leftarrow \overleftarrow{\mathcal{D}}\{e_0\};\ \mathbf{case}\ x\ \mathbf{of\ inl}\ x_1 \to \overleftarrow{\mathcal{D}}\{e_1\},\ \mathbf{inr}\ x_2 \to \overleftarrow{\mathcal{D}}\{e_2\}\} \\
\overleftarrow{\mathcal{D}}\{(e_1, e_2)\} &= \mathbf{do}\ \{x \leftarrow \overleftarrow{\mathcal{D}}\{e_1\};\ y \leftarrow \overleftarrow{\mathcal{D}}\{e_2\};\ \mathbf{pure}\ (x, y)\} \\
\overleftarrow{\mathcal{D}}\{\mathbf{fst}\ e\} &= \mathbf{do}\ \{x \leftarrow \overleftarrow{\mathcal{D}}\{e\};\ \mathbf{pure}\ (\mathbf{fst}\ x)\} \\
\overleftarrow{\mathcal{D}}\{\mathbf{snd}\ e\} &= \mathbf{do}\ \{x \leftarrow \overleftarrow{\mathcal{D}}\{e\};\ \mathbf{pure}\ (\mathbf{snd}\ x)\} \\
\overleftarrow{\mathcal{D}}\{k\} &= \mathbf{pure}\ k \\
\overleftarrow{\mathcal{D}}\{+_{\mathbb{Z}}\} &= \mathbf{pure}\ \lambda(x, y).\ \mathbf{pure}\ (x +_{\mathbb{Z}} y) \\
\overleftarrow{\mathcal{D}}\{r\} &= \mathbf{pure}\ (r, \mathit{Zero}) \\
\overleftarrow{\mathcal{D}}\{+_{\mathbb{R}}\} &= \mathbf{pure}\ \lambda((x, u_1), (y, u_2)).\ \mathbf{do}\ \{u_3 \leftarrow \mathit{deltaLet}\ (\mathit{Add}\ u_1\ u_2);\ \mathbf{pure}\ (x + y, \mathit{Var}\ u_3)\} \\
\overleftarrow{\mathcal{D}}\{\times_{\mathbb{R}}\} &= \mathbf{pure}\ \lambda((x, u_1), (y, u_2)).\ \mathbf{do}\ \{u_3 \leftarrow \mathit{deltaLet}\ (\mathit{Add}\ (\mathit{Scale}\ y\ u_1)\ (\mathit{Scale}\ x\ u_2));\ \mathbf{pure}\ (x \times y, \mathit{Var}\ u_3)\}
\end{aligned}
$$

Fig. 9. Monadic translation

$$
\begin{aligned}
\text{Given} \quad & e : \mathbb{R}^a \to \mathbb{R} \\
\text{and} \quad & \overleftarrow{\mathcal{D}}\{e\} : (\mathbb{R} \times \mathit{Delta})^a \to M\ (\mathbb{R} \times \mathit{Delta}) \quad \text{(as defined in Figure 9)} \\
\text{we produce} \quad & \mathcal{R}\{e\} : (\mathbb{R}^a \times \delta\mathbb{R}) \to \delta\mathbb{R}^a \\
& \mathcal{R}\{e\} = \lambda((\ldots, s_i, \ldots), \delta t). \\
& \qquad \mathbf{let}\ s_0 = (\ldots, (s_i, \mathit{Var}\ i), \ldots)\ \mathbf{in} \\
& \qquad \mathbf{let}\ (\_, u) = \mathit{runDelta}\ (a + 1)\ (\overleftarrow{\mathcal{D}}\{e\}\ s_0)\ \mathbf{in} \\
& \qquad \mathbf{let}\ \mathit{finalMap} = \mathit{eval}\ \delta t\ u\ \mathit{emptyMap}\ \mathbf{in} \\
& \qquad (\ldots, \mathit{lookupOrZero}\ i\ \mathit{finalMap}, \ldots)
\end{aligned}
$$

Fig. 10. Reverse-mode wrapper around $\overleftarrow{\mathcal{D}}$

Then *addDelta* is implemented by an in-place add to the specified slot in the array, while *lookup* indexes it. The initial array, initialised to zeros, is constructed by *emptyMap*, and its size can be obtained from *runDelta* as the final *DeltaId* allocated. (We omit the code for this little point.)

At this point we have essentially reconstructed Kmett/Pearlmutter/Siskind's automatic differentiation Haskell library ad [Kmett et al. 2021]. It is a very clever library, and now for the first time, we understand how it works and can prove it correct.

5.3.4 *Asymptotic Efficiency.*

Our monadic translation is asymptotically efficient. Our translation introduces a constant-factor overhead to the runtime, and space usage of a translated program has an overhead bounded by the runtime of the original program.

Lemma 3 (Translation Efficiency).

Suppose we have a program e, and its reverse-mode translation $\overleftarrow{\mathcal{D}}\{e\}$*, and further suppose that both of these programs are run using a call-by-value evaluation strategy. Then:*

(1) *The runtime of* $\overleftarrow{\mathcal{D}}\{e\}$ *is bounded by a constant-factor relative to the runtime of e.*
(2) *The memory usage of* $\overleftarrow{\mathcal{D}}\{e\}$ *is bounded by the memory usage of e, plus the runtime of e.*

Proof. The proof of each case is follows:

(1) To show the correctness of the runtime bound, observe that our translation takes a purely functional program and translates it into monadic form, and replaces all of the floating-point operations with new, instrumented versions. The monadic translation adds at most a constant factor overhead, since each reduction of the original program becomes a few steps of the instrumented program – since we use a state monad, each bind step involves an additional function call to pass the state.
The additional work in each of the primitive operations (like *plus* and *times*) involves doing a little more arithmetic, and passing some state to the Delta constructors. So assuming that memory allocation and arithmetic are all constant-time operations, then the translated primitives remain constant time. Therefore the cost of the arithmetic in the translation is a constant factor worse than the original program, and since the number of arithmetic operations is bounded by the runtime of the original program, this adds at most another constant-factor overhead to $\overleftarrow{\mathcal{D}}\{e\}$.
Since the sum of two constant factors is still constant, the runtime of the translated program is linear in the run time of the original program.
(2) The asymptotic memory usage of a monadic translation is the same as the original program. In practice, allocating and deallocating closures for monadic state-passing is expensive, but since the lifetime of all these intermediate values is short, the asymptotic allocation does not change.
However, each translated primitive allocates a constant additional amount of memory and adds it to the state. So the size of the state *DeltaBinds* being passed around will grow as the program executes, proportional to the number of primitive operations the original program performed. Since the number of primitive operations is bounded by the run time of the original program, this means that the additional memory allocated can be bounded by the run time of the original program.
Hence the additional asymptotic memory usage is at most linear in the runtime of the original program.

□

This is not quite enough to establish our time and space bounds, though. The $\mathcal{R}\{e\}$ function runs a translated program to produce a *Delta* representing the derivative, which needs to be evaluated in order to produce the actual derivative of interest. To do this, we will first show that *eval*, given in Figure 11, also runs in linear time.

Lemma 4. *Consider the expression eval x u um.*

(1) *The runtime of eval x u um is linear in the size of u.*
(2) *Evaluating eval x u um requires memory allocation at most linear in the size of u.*

Proof. The proof of each case is a straightforward induction on u.

(1) Since the *eval* function evaluates each sub-term exactly once, and does a constant amount of work in each case aside from the recursive calls, the run time of the *eval* function is linear in the size of u.
(2) The *eval* function can allocate memory in the variable case (when *addDelta* is invoked) if the variable is not already in the map *um*. Since the number of variables is bounded by the size of u, this means the total memory allocation is also at most linear in the size of u.

□

We can put these two lemmas together to prove the efficiency of the overall algorithm:

Theorem 5 (Translation Efficiency). *Suppose we have a program e, and its reverse-mode translation $\mathcal{R}\{e\}$. Then:*

(1) *The runtime of $\mathcal{R}\{e\}$ is bounded by a constant-factor relative to the runtime of e.*
(2) *The memory usage of $\mathcal{R}\{e\}$ is bounded by the memory usage of e, plus the runtime of e.*

Proof. The $\mathcal{R}\{e\}$ function calls $\overleftarrow{\mathcal{D}}\{e\}$, and then builds a *Delta* named u from the *DeltaBinds* that $\overleftarrow{\mathcal{D}}\{e\}$ returns using the *runDelta* function, and then calls *eval* on that.

(1) We know that the size of the *DeltaBinds* returned from $\overleftarrow{\mathcal{D}}\{e\}$ is linear in the runtime of e, and since *runDelta* is just a fold that also returns a u linear in the size of the *DeltaBinds*. Calling *eval* on that u is also linear in the runtime of e, which means the overall runtime is linear in the runtime of e.
(2) Again, we know the size of the *DeltaBinds* returned from $\overleftarrow{\mathcal{D}}\{e\}$ is linear in the runtime of e. Constructing a u also takes memory at most linear in the runtime of e, and running *eval* on u takes memory at most linear in the runtime of e. Therefore the total memory overhead is at most linear in the runtime of e.

□

6 IMPLEMENTATION

One well-known advantage of the dual-number approach, which we use for both forward and reverse mode, is that in a language like OCaml or Haskell (and others) we do not need to do any source-to-source transformation at all. In Haskell, all arithmetic is overloaded, so rather than the top-level function having type $e : \mathbb{R}^a \to \mathbb{R}$, it has type

$$e : \forall t.\ \mathit{Real}\ t \Rightarrow t^a \to t$$

We can instantiate t with type *Float* to do the primal computation, or with type *Dual* to do the dual-number computation, where

```
data Dual = Dual Float Delta
```

```
data Delta = Zero | Scale ℝ Delta | Add Delta Delta | Var DeltaId | Let DeltaId Delta Delta
type DeltaBinds = [(DeltaId, Delta)]  -- In reverse dependency order
type DeltaState = (DeltaId, DeltaBinds)
type DeltaId    = ℕ
type DeltaMap   = Map DeltaId δℝ

eval : ℝ → Delta → DeltaMap ⊸ DeltaMap
eval x Zero           um = um
eval x (Scale y u)    um = eval (x × y) u um
eval x (Add u₁ u₂)    um = eval x u₂ (eval x u₁ um)
eval x (Var uid)      um = addDelta uid x um
eval x (Let uid u₁ u₂) um = let um₂ = eval x u₂ um in
                            case lookup uid um₂ of
                              Nothing → um₂
                              Just x  → eval x u₁ (delete uid um₂)

  -- API for the Map type
emptyMap    : DeltaMap
lookup      : DeltaId → DeltaMap → Maybe ℝ
lookupOrZero : DeltaId → DeltaMap → ℝ
delete      : DeltaId → DeltaMap ⊸ DeltaMap
addDelta    : DeltaId → ℝ → DeltaMap ⊸ DeltaMap
                -- Adds to an existing entry, or create an entry if one does not exist

  -- The monad M
type M a = DeltaState → (a, DeltaState)

runDelta : DeltaId → M (ℝ, Delta) → (ℝ, Delta)
  -- Runs the computation, and wraps the result in
  -- the bindings produced by running the computation
runDelta delta_id m = (res, foldl wrap delta binds)
  where
    ((res, delta), (_, binds)) = m (delta_id, [])
    wrap body (id, rhs)        = Let id rhs body

instance Monad M where
  return x = λs → (x, s)
  m >>= n  = λs → case m s of (r, s') → n r s'

deltaLet : Delta → M DeltaId
deltaLet delta = λ(delta_id, bs) → (delta_id, (delta_id + 1, (delta_id, delta) : bs))
```

Fig. 11. Supporting functions (rendered in Haskell-like syntax) for the monadic translation

The instance for *Real Dual* and *Num Dual* give the implementations for literals and the floating-point operations. This approach is well described by Karczmarczuk [1998], who generalises it

to a "lazy tower" of all the higher order derivatives as well, and taken up by Elliott [2009] and Kmett/Pearlmutter/Siskind's automatic differentiation Haskell library ad [7] [Kmett et al. 2021].

We can do much the same thing in OCaml by just parameterising over the implementation of reals. Then, programs can be written as terms that take a structure implementing the reals, and applying it to an implementation of the reals as dual numbers carrying a *Delta*.

[7]Since Haskell is a pure language one would expect that a Haskell implementation of reverse mode using dual numbers would require explicit source-to-source translation into monadic style. The ad library avoids the need for explicit translation by (safely) using internal compiler primitives (which are not type safe in general) to maintain observable sharing.

```
module Program (R : Real) = struct
  open R   -- access real from the module R
           -- along with arithmetic operations
  type vec3 = {x : real; y : real; z : real}
  type quaternion =
    {x : real; y : real; z : real; w : real}
  let q_to_vec (q : quaternion) : vec3 =
    {x = q.x; y = q.y; z = q.z}
  let dot (p : vec3) (q : vec3) : real =
    p.x × q.x + p.y × q.y + p.z × q.z
      -- Vector addition
  let (++) (p : vec3) (q : vec3) : vec3 =
    {x = p.x + q.x; y = p.y + q.y; z = p.z + q.z}
  let scale k (v : vec3) : vec3 =
    {x = k × v.x; y = k × v.y; z = k × v.z}
  let cross (a : vec3) (b : vec3) : vec3 =
    {x = a.y × b.z − a.z × b.y;
     y = a.z × b.x − a.x × b.z;
     z = a.x × b.y − a.y × b.x}
  let norm (x : vec3) : real = sqrt (dot x x)
  let rotate_vec_by_quat (v : vec3)
                         (q : quaternion) : vec3 =
    let u = q_to_vec q in
    let s = q.w in
    scale (from_float 2.0 × dot u v) u
    ++
    scale (s × s − dot u u) v
    ++
    scale (from_float 2.0 × s) (cross u v)
end
```

```
-- Result of runDelta 8 (D←{e} s0), where
-- e = λq v. (rotate_vec_by_quat v q).x
-- s0 = (q0; v0)
-- q0 = {(1.1; Var qx); (2.2; Var qy); (3.3; Var qz); (4.4; Var qw)}
-- v0 = {(5.5; Var vx); (6.6; Var vy); (7.7; Var vz)}

-- We informally use a Let/in notation for the constructor Let, and
-- use variable names instead of numbers, so xN stands for variable N+8
Let x1  = Add (Scale 5.5 (Var qy)) (Scale 2.2 (Var vx)) in
Let x2  = Add (Scale 6.6 (Var qx)) (Scale 1.1 (Var vy)) in
Let x3  = Add (Var x2) (Scale (−1.0) (Var x1)) in
Let x4  = Add (Scale 7.7 (Var qx)) (Scale 1.1 (Var vz)) in
Let x5  = Add (Scale 5.5 (Var qz)) (Scale 3.3 (Var vx)) in
Let x6  = Add (Var x5) (Scale (−1.0) (Var x4)) in
Let x7  = Add (Scale 6.6 (Var qz)) (Scale 3.3 (Var vy)) in
Let x8  = Add (Scale 7.7 (Var qy)) (Scale 2.2 (Var vz)) in
Let x9  = Add (Var x8) (Scale (−1.0) (Var x7)) in
Let x10 = Zero in
Let x11 = Add (Scale 4.4 (Var x10)) (Scale 2 (Var qw)) in
Let x12 = Add (Scale (−4.84) (Var x11)) (Scale 8.8 (Var x3)) in
Let x13 = Add (Scale 9.68 (Var x11)) (Scale 8.8 (Var x6)) in
Let x14 = Add (Scale (−4.84) (Var x11)) (Scale 8.8 (Var x9)) in
Let x15 = Add (Scale 3.3 (Var qz)) (Scale 3.3 (Var qz)) in
Let x16 = Add (Scale 2.2 (Var qy)) (Scale 2.2 (Var qy)) in
Let x17 = Add (Scale 1.1 (Var qx)) (Scale 1.1 (Var qx)) in
Let x18 = Add (Var x17) (Var x16) in
Let x19 = Add (Var x18) (Var x15) in
Let x20 = Add (Scale 4.4 (Var qw)) (Scale 4.4 (Var qw)) in
Let x21 = Add (Var x20) (Scale (−1.0) (Var x19)) in
Let x22 = Add (Scale 7.7 (Var x21)) (Scale 2.42 (Var vz)) in
Let x23 = Add (Scale 6.6 (Var x21)) (Scale 2.42 (Var vy)) in
Let x24 = Add (Scale 5.5 (Var x21)) (Scale 2.42 (Var vx)) in
Let x25 = Add (Scale 7.7 (Var qz)) (Scale 3.3 (Var vz)) in
Let x26 = Add (Scale 6.6 (Var qy)) (Scale 2.2 (Var vy)) in
Let x27 = Add (Scale 5.5 (Var qx)) (Scale 1.1 (Var vx)) in
Let x28 = Add (Var x27) (Var x26) in
Let x29 = Add (Var x28) (Var x25) in
Let x30 = Zero in
Let x31 = Add (Scale 45.98 (Var x30)) (Scale 2 (Var x29)) in
Let x32 = Add (Scale 3.3 (Var x31)) (Scale 91.96 (Var qz)) in
Let x33 = Add (Scale 2.2 (Var x31)) (Scale 91.96 (Var qy)) in
Let x34 = Add (Scale 1.1 (Var x31)) (Scale 91.96 (Var qx)) in
Let x35 = Add (Var x32) (Var x22) in
Let x36 = Add (Var x33) (Var x23) in
Let x37 = Add (Var x34) (Var x24) in
Let x38 = Add (Var x35) (Var x12) in
Let x39 = Add (Var x36) (Var x13) in
Let x40 = Add (Var x37) (Var x14) in
Var x40
```

Fig. 12. Rotating a vector by a quaternion. The function rotate_vec_by_quat defined on the left is applied at a given q and v, producing the Delta on the right, which is evaluated to compute the seven derivative entries.

Figure 12 shows a run from such an implementation, on a real-world example from computer vision. The key observation is that the delta structure which is evaluated in the "backward" pass is linear in runtime (operation count), and constructed entirely from the small set of *Delta* constructors.

7 CORRECTNESS

The property we would like to establish about our reverse-mode translation is given in Figure 2. Specializing it to the case of $\mathbb{R}^a \to \mathbb{R}$, we get the following statement:

Theorem 6 (Correctness of $\mathcal{R}\{e\}$).

If e is a closed term of type $\mathbb{R}^a \to \mathbb{R}$ then for all $s : \mathbb{R}^a$, and $\delta t : \delta\mathbb{R}$, $[\![\mathcal{R}\{e\}]\!]\,(s, \delta t) = \delta t^\top \bullet \mathcal{J}[\![e]\!](s)$.

To prove this, we have to peek inside the $\mathcal{R}\{e\}$ wrapper function in Figure 10. When we do so, we discover that it does two things. Given e, it first executes $\overleftarrow{\mathcal{D}}\{e\}$ to build a *Delta*. Then, the wrapper calls *eval* on the *Delta* to compute the actual result. We thus have two tasks. First, we have to prove that the translation of the source program yields a correct *Delta*. Second, we have to prove that our implementation of *eval* interprets our *Delta*s correctly.

Both of these tasks depend upon knowing what *Delta*s mean. We introduced *Delta*s in Figure 7 as a space-saving device, using them to represent the dual vectors in the original reverse-mode translation in Figure 5. We then augmented *Delta* with variables and *Let*-bindings to represent sharing. As a result, the language of *Delta*s looks very much like a simple language for symbolic vector expressions augmented with *Let*-bindings. Indeed, we can interpret these terms precisely so, working within a context mapping the bound *DeltaId*s to vectors $\mathbb{R}^a$. Our interpretation function $[\![\_]\!]_D$ (where the D stands for the *Delta*s this function interprets) is as follows:

$$
\begin{array}{lcl}
[\![\_]\!]_D\_ & : & \mathit{Delta} \to (\mathit{DeltaId} \to \mathbb{R}^a) \to \mathbb{R}^a \\
[\![\mathit{Zero}]\!]_D\gamma & = & \vec{0} \\
[\![\mathit{Add}\ u_1\ u_2]\!]_D\gamma & = & [\![u_1]\!]_D\gamma + [\![u_2]\!]_D\gamma \\
[\![\mathit{Var}\ \mathit{uid}]\!]_D\gamma & = & \gamma(\mathit{uid}) \\
[\![\mathit{Let}\ \mathit{uid}\ e_1\ e_2]\!]_D\gamma & = & [\![e_2]\!]_D(\gamma, [\![e_1]\!]_D\gamma/\mathit{uid}) \\
[\![\mathit{Scale}\ r\ e_1]\!]_D\gamma & = & r \cdot ([\![e_1]\!]_D\gamma)
\end{array}
$$

This is a perfectly conventional semantics for an expression language, and is the semantics we will use when defining the correctness of the reverse-mode translation. It is safe for us to do so, since the reverse-mode translation is the phase which builds the delta, which is then followed by a phase in which *eval* consumes the delta to produce the result.

7.1 Correctness of the Translation

Our goal is to prove that a certain class of programs – those of type $\mathbb{R}^a \to \mathbb{R}$ – are transformed into their reverse-mode derivatives by our translation $\overleftarrow{\mathcal{D}}$. There are two features which make proving the correctness of the translation difficult. One, our language includes first-class functions, and two, our translation is an imperative and monadic, with each arithmetic operation appending to a global tape. So we turn to the standard tool for dealing with higher-order stateful programs: we use a *binary Kripke logical relation* [Jung and Tiuryn 1993].

Kripke logical relations. A logical relation is a family of relations, one for every type in our programming language. Since we are relating each program to its translation, this relation must be binary. Since our translation produces monadic stateful programs, we have to index the relations by the possible (monotonically growing) states the program can execute in, which are the Kripke worlds, denoted with C. As is usual for a logical-relations approach, all the action is at the base

case – in our proof, for real numbers. The rest of the relation serves solely to show the behaviour on real numbers is preserved by the other type formers.

The Fundamental Property (Theorem 8) shows that every well-typed term is related to its translation. We can then use this property to show (Theorem 9) that, at the type $\mathbb{R}^a \to \mathbb{R}$, the translation of a program by $\overleftarrow{\mathcal{D}}$ computes the entries of its Jacobian matrix.

We define our logical relation in Figure 13. While the overall structure of the relation is standard, many of the individual pieces are unusual.

Before we get to the semantic novelties, though, there are a few modest simplifications in the formal proof relative to the implementation. In particular, we work with variables rather than de Bruijn indices, and we assume that we can freely invent fresh names rather than passing around a name supply. This means that the definition of the monad in Figure 13 is given as *T a = DeltaBinds → (a, DeltaBinds)* rather than *M a = DeltaState → (a, DeltaState)* (as in Figure 11).

The relation at $\mathbb{R}$. Our relation $\mathcal{V}^a_A(C)$ relates the original type A to the translated type $\overleftarrow{\mathcal{D}}\{A\}$, where $C \in \mathbb{R}^a \to$ *DeltaBinds* is our Kripke world. However, our relation is not between individual programs – instead, it is a relation between $\mathbb{R}^a \to [\![A]\!]$ and $\mathbb{R}^a \to [\![\overleftarrow{\mathcal{D}}(A)]\!]$. The elements of the relation are (denotations) indexed by a vector in $\mathbb{R}^a$ (as is our Kripke world C).

The reason for working with families of values rather than values themselves is to make it possible to talk about derivatives. Consider the base case of the relation $\mathcal{V}^a_{\mathbb{R}}(C)$, stated near the top of Figure 13. Instead of relating *values* of the real number type to dual numbers, we relate *families of values* of real numbers to *families of dual numbers*. Suppose $f : \mathbb{R}^a \to \mathbb{R}$ is related to $g : \mathbb{R}^a \to (\mathbb{R} \times$ *Delta*$)$. Our relation requires f to be differentiable everywhere: note the appearance of $\mathcal{J}f$. Our relation further requires the first component of $g(\vec{x})$ to equal $f(\vec{x})$ (for all $\vec{x}$) and the second component of $g(\vec{x})$ to be a *Delta* whose interpretation is the appropriate entry in the Jacobian of f. Since Jacobians need to be evaluated at a specific point $\vec{x}$, we introduce the indexed-family structure to abstract over all the possible $\vec{x}$s.

Our translation $\overleftarrow{\mathcal{D}}$ produces a *Delta* intended to be consumed by *eval*, but our logical relation $\mathcal{V}^a_{\mathbb{R}}(C)$ interprets these *Delta*s using $[\![]\!]_D$. The *eval* function assumes its input *Delta* has a free variables – let us call them $\mathsf{z}_1 \ldots \mathsf{z}_a$ – corresponding to the a partial derivatives we seek. Accordingly, our interpretation $[\![]\!]_D$ requires each z_i to be mapped to a unit vector $onehot_{\mathbb{R}^a}\ i$; the ϕ_a environment is exactly this mapping: $\phi_a = \{\mathsf{z}_i \mapsto onehot_{\mathbb{R}^a}\ i | i \in 1..a\}$.

Most of the other clauses – for integers, functions, sums and products – are the standard for logical relations, with small adjustments being made to deal with the fact that we are relating families of values.

Monads and Kripke worlds. We must now formally relate our Kripke world C to the evolution of the *DeltaBinds* state in our monad T. Intuitively, the Kripke world represents the current *DeltaBinds* state of the monadic computation. As programs in our translation evaluate, they incrementally add bindings to this state. This context extension is the basis of our Kripke ordering: one state is later in the Kripke ordering if it is a (non-shadowing) extension of the other one.

But even though the actual program has a monotonically-growing *DeltaBinds*, our logical relation relates values indexed by $\mathbb{R}^a$. Therefore, our Kripke worlds need to be *DeltaBinds* which are *also* indexed by $\mathbb{R}^a$. So the actual Kripke ordering, given in Figure 13, says that $C' \sqsupseteq C$ when $C'(\vec{x})$ is an extension of $C(\vec{x})$, pointwise for every $\vec{x} \in \mathbb{R}^a$.

Since a monadic computation begins in a particular world, and ends in a bigger one, we also need to ensure that values introduced at one world C will continue to be valid at all later worlds. This is the "the Kripke property" of the logical relation.

$$\mathcal{V}^a_A \in (\mathbb{R}^a \to \mathit{DeltaBinds}) \to \mathcal{P}((\mathbb{R}^a \to [\![A]\!]) \times (\mathbb{R}^a \to [\![\overleftarrow{\mathcal{D}}(A)]\!]))$$

$$\mathcal{V}^a_{\mathbb{R}}(C) = \{(f, \lambda\vec{x}.\,(f\ \vec{x}, \mathit{Var}\ y)) \mid f{:}\mathbb{R}^a \to \mathbb{R} \wedge \forall C' \sqsupseteq C, \vec{x} \in \mathbb{R}^a.\ [\![\mathit{get}\ (C'\ \vec{x})\ (\mathit{Var}\ y)]\!]_D \phi_a = (\mathcal{J} f)(\vec{x})\}$$
$$\cup\ \{(f, \lambda\vec{x}.\,(f\ \vec{x}, \mathit{Zero})) \mid f{:}\mathbb{R}^a \to \mathbb{R} \text{ is constant}\}$$
$$\mathcal{V}^a_{\mathbb{Z}}(C) = \{(f, f) \mid f{:}\mathbb{R}^a \to \mathbb{Z} \text{ is constant}\}$$
$$\mathcal{V}^a_{A\times B}(C) = \{(\langle f, g\rangle, \langle f', g'\rangle) \mid (f, f') \in \mathcal{V}^a_A(C) \wedge (g, g') \in \mathcal{V}^a_B(C)\}$$
$$\mathcal{V}^a_{A+B}(C) = \{(f; \mathbf{inl}, g; \mathbf{inl}) \mid (f, g) \in \mathcal{V}^a_A(C)\} \cup \{(f; \mathbf{inr}\ , g; \mathbf{inr}\ ) \mid (f, g) \in \mathcal{V}^a_B(C)\}$$
$$\mathcal{V}^a_{A\to B}(C) = \{(f, f') \mid \forall C' \sqsupseteq C, (v, v') \in \mathcal{V}^a_A(C').\ ((\lambda\vec{x}.\ f\ \vec{x}\ (v\ \vec{x})), (\lambda\vec{x}.\ f'\ \vec{x}\ (v'\ \vec{x}))) \in \mathcal{E}^a_B(C')\}$$

$$\mathcal{V}^n_\Gamma \in (\mathbb{R}^a \to \mathit{DeltaBinds}) \to \mathcal{P}((\mathbb{R}^a \to [\![\Gamma]\!]) \times (\mathbb{R}^a \to [\![\overleftarrow{\mathcal{D}}(\Gamma)]\!]))$$

$$\mathcal{V}^a_\bullet(C) = \{(h, h) \mid h : \mathbb{R}^a \to 1 = \lambda\vec{x}.\ ()\}$$
$$\mathcal{V}^a_{\Gamma, x:A}(C) = \{(\langle\gamma, f\rangle, \langle\gamma', f'\rangle) \mid (\gamma, \gamma') \in \mathcal{V}^a_\Gamma(C) \wedge (f, f') \in \mathcal{V}^a_A(C)\}$$

$$\mathbf{type}\ T\ a = \mathit{DeltaBinds} \to (a, \mathit{DeltaBinds})$$

$$\mathcal{E}^n_A \in (\mathbb{R}^a \to \mathit{DeltaBinds}) \to \mathcal{P}((\mathbb{R}^a \to [\![A]\!]) \times (\mathbb{R}^a \to T\ ([\![\overleftarrow{\mathcal{D}}(A)]\!])))$$
$$\mathcal{E}^a_A(C) = \{(f, f') \mid \forall C' \sqsupseteq C.\ \exists C'' \sqsupseteq C', g'.\ (\lambda\vec{x}.\ f'\ \vec{x}\ (C'\ \vec{x})) = \langle g', C''\rangle \wedge (f, g') \in \mathcal{V}^a_A(C'')\}$$

$$\begin{aligned}
&\mathit{get} : \mathit{DeltaBinds} \to \mathit{Delta} \to \mathit{Delta}\\
&\mathit{get}\ [\,]\qquad\qquad\quad x = x\\
&\mathit{get}\ ((y, \mathit{delta}) : C)\ x = \mathit{get}\ C\ (\mathit{Let}\ y\ \mathit{delta}\ x)
\end{aligned}$$

$$C' \sqsupseteq C \text{ iff } \forall\vec{x} \in \mathbb{R}^a, C'(\vec{x}) \geq C(\vec{x}),\ \text{where}$$
$$C' \geq C \text{ iff } C' = C, C'' \text{ where } \mathit{dom}(C) \cap \mathit{dom}(C'') = \emptyset$$
$$\langle f, g\rangle\ x = (f\ x, g\ x)$$
$f; g$ is reverse function composition ("f is followed by g"), i.e. $\lambda x.\ g(f(x))$
$$\phi_a = \{\mathsf{z}_i \mapsto \mathit{onehot}_{\mathbb{R}^a}\ i \mid i \in 1..a\}$$

Fig. 13. Kripke logical relation

Lemma 7 (Kripke Monotonicity). *If $C' \sqsupseteq C$, then we have*

(1) $\mathcal{V}^a_A(a)(C') \supseteq \mathcal{V}^a_A(a)C$
(2) $\mathcal{E}^a_A(a)(C') \supseteq \mathcal{E}^a_A(a)C$

The statement of Kripke monotonicity talks about the value relation $\mathcal{V}^a_A(a)C$, but also talks about the expression relation $\mathcal{E}^a_A(a)C$. Since our translation goes into a monadic language, our logical relation also needs to have a clause relating purely functional computations to monadic computations. The definition in Figure 13 says the expected thing: a pure computation is related to a monadic one which computes a related value, but which possibly adds some additional elements to the *DeltaBinds* being threaded through the computation. That is, a monadic computation can begin in one world, and end in a later one.

The fundamental lemma is then proved as usual. We first lift the definition of the logical relation to contexts $\mathcal{V}^a_\Gamma(a)C$, yielding a relation between (families of) substitutions γ and γ', and then show that every well-typed term and its translation lies in the relation.

Theorem 8 (Fundamental Property).
For any $\Gamma, e, A, C, \gamma, \gamma'$, *if* $\Gamma \vdash e : A$ *and* $(\hat{\gamma}, \hat{\gamma}') \in \mathcal{V}^a_\Gamma(C)$, *then* $(\hat{\gamma}; [\![e]\!], \hat{\gamma}'; [\![\overleftarrow{\mathcal{D}}\{e\}]\!]) \in \mathcal{E}^a_A(C)$.

Proof. By rule induction, see appendix. □

As a consequence of the fundamental property, we can show that the reverse-mode translation is adequate – i.e., it computes the correct result for any term of type $\mathbb{R}^a \to \mathbb{R}$.

Theorem 9 (Adequacy of $\overleftarrow{\mathcal{D}}\{e\}$). *For all* $C \in$ *DeltaBinds*, $\vec{x} \in \mathbb{R}^a$, *if* e *is a closed term of type* $\mathbb{R}^a \to \mathbb{R}$ *and* $\underline{\vec{x}} = \overrightarrow{(\vec{x}_i, \mathit{Var}\, \mathrm{z}_i)^{i \in 1 \ldots a}}$ *then there exists a* C' *and* z *such that* $[\![\overleftarrow{\mathcal{D}}\{e\}]\!]\ ()\ \underline{\vec{x}}\ C = (C', ([\![e]\!]\ ()\ \vec{x}, z))$, *and* $[\![get\ C'\ z]\!]_D \phi_a = \mathcal{J}([\![e]\!]\ ())\ \vec{x}$.

Proof. The key observation to make is that $(\pi_i, \lambda\vec{x}.(\pi_i(\vec{x}), \mathit{Var}\, \mathrm{z}_i))$ is in the relation at $\mathcal{V}^a_{\mathbb{R}}(C)$. To see this, note that $\pi_i : \mathbb{R}^a \to \mathbb{R}$ simply picks out the i-th element of the vector of inputs. As a result, its Jacobian $\mathcal{J}\pi_i$ will be the unit vector whose i-th element is one and which has zeros in all other positions – the partial derivative $\frac{\partial \pi_i}{\partial x_j}$ is 1 if $i = j$ and 0 otherwise. Next, note that the interpretation of z_i in the basis vector environment ϕ_a, is just the unit vector which is 1 at i. As a result, the interpretation of $\mathit{Var}\, \mathrm{z}_i$ is precisely the Jacobian $\mathcal{J}\pi_i$.

Write I for $\lambda\vec{x}.(\pi_0(\vec{x})\, \mathit{Var}\, \mathrm{z}_0, \ldots, \pi_a(\vec{x})\, \mathit{Var}\, \mathrm{z}_a)$. We can see that $\langle \pi_i, \ldots, \pi_a \rangle$ is related to I in $\mathcal{V}^a_{\mathbb{R}^a}(a)$ for any Kripke world. Since $\langle \pi_i, \ldots, \pi_a \rangle = id_{\mathbb{R}^a}$, we can simplify a bit further and see that $id_{\mathbb{R}^a}$ is related to I. Then, the fundamental lemma tells us that $id_{\mathbb{R}^a}; [\![e]\!]$ is related to $I; [\![\overleftarrow{\mathcal{D}}\{e\}]\!]_D$ at $\mathcal{E}^a_{\mathbb{R}}$ for any Kripke world.

Finally, some algebraic simplification using the definitions of the logical relation and the monad gives the claimed property. Intuitively, we are choosing to pass each parameter of the $\mathbb{R}^a$ parameterisation directly as the corresponding argument to $[\![e]\!]$ and $[\![\overleftarrow{\mathcal{D}}\{e\}]\!]$, which is why the coefficient of $\mathit{Var}\, \mathrm{z}_i$ is the i-th partial derivative.

□

7.2 Correctness of *eval*

While the logical relation tells us that the reverse-mode translation gives us a *Delta* u which has the correct meaning (i.e., $[\![u]\!]_D$ is the right value), it is not enough to prove the correctness of $\mathcal{R}\{e\}$.

For efficiency's sake, $\mathcal{R}\{e\}$ uses the *eval* function to interpret the returned *Delta*. This is important because *eval* is guaranteed to examine each subterm of the expression exactly once, and never builds intermediate arrays – it can thus update a single mutable data structure.

We now prove the correctness of *eval*.

Lemma 10 (Correctness of eval). *If* u *is a* *Delta* *expression,* γ *is an environment mapping the free variables of* u *to vector values,* m *is a* *DeltaMap* *with bindings for every free variable of* u, *and* s *is a real number, then*

$$s \cdot [\![u]\!]\gamma = \sum_{a \in \mathrm{FV}(u)} [(\mathit{lookupOrZero}\ (\mathit{eval}\ s\ u\ m)\ a) - (\mathit{lookupOrZero}\ m\ a)] \cdot \gamma(a)$$

Proof. The proof is a mostly routine induction over the structure of u, with only a little slightly involved algebra in the *Let* case. □

More important is to understand what the lemma is telling us. It tells us that:

(1) the argument s is a scaling factor of the value of u;
(2) the result is encoded in the difference between the input and output maps; and
(3) most importantly, the value of u is a linear combination of the values of its free variables.

Suppose we have primitive function $\mathtt{f} : \mathbb{R}^n \to \mathbb{R}^m$, an implementation of mathematical function f, along with functions $\mathtt{df}_{ij}$ that compute $\frac{\partial f_i}{\partial x_j}$.
We can add f for the forward- and reverse-mode translations (Figures 3 and 9) as follows:

$$\begin{aligned}
&\overrightarrow{\mathcal{D}}\{\mathtt{f}\} = \lambda((x_1, \delta x_1), \ldots, (x_n, \delta x_n)).\\
&\quad \textbf{let}\ \vec{x} = (x_1, \ldots, x_n)\ \textbf{in}\\
&\quad \textbf{let}\ (y_1, \ldots, y_m) = \mathtt{f}\ \vec{x}\ \textbf{in}\\
&\quad ((y_1, \textstyle\sum_{i=1}^n \mathtt{df}_{1i}\ \vec{x} \times \delta x_i), \ldots,\\
&\qquad (y_m, \textstyle\sum_{i=1}^n \mathtt{df}_{mi}\ \vec{x} \times \delta x_i))
\end{aligned}$$

$$\begin{aligned}
&\overleftarrow{\mathcal{D}}\{\mathtt{f}\} = \lambda((x_1, \delta x_1), \ldots, (x_n, \delta x_n)).\\
&\quad \textbf{let}\ \vec{x} = (x_1, \ldots, x_n)\ \textbf{in}\ \textbf{let}\ (y_1, \ldots, y_m) = \mathtt{f}\ \vec{x}\ \textbf{in}\\
&\quad \textbf{do}\ \{\ \delta y_1 \leftarrow \mathit{deltaLet}\ (\mathit{Add}^n_{i=1}\ (\mathit{Scale}\ (\mathtt{df}_{1i}\ \vec{x})\ \delta x_i));\\
&\qquad \ldots;\\
&\qquad \delta y_m \leftarrow \mathit{deltaLet}\ (\mathit{Add}^n_{i=1}\ (\mathit{Scale}\ (\mathtt{df}_{mi}\ \vec{x})\ \delta x_i));\\
&\qquad \textbf{pure}\ ((y_1, \mathit{Var}\ \delta y_1), \ldots, (y_m, \mathit{Var}\ \delta y_m))\ \}
\end{aligned}$$

Fig. 14. Translation of arbitrary differentiable function

This last fact holds because our language of deltas has no nonlinear operators – we can only add two vectors or scale a vector by a constant. It also turns out to be essential for proving Theorem 6. Since the final expression has exactly the a free variables z_i, and the intended interpretation of each z_i is a unit vector orthogonal to all of the others, the scaling for each basis vector gives the size of the derivative in that coordinate. Consequently, our implementation can read off the coefficients from the update map, without ever needing to explicitly materialise the unit vectors in ϕ_a.

8 GENERALISATIONS AND EXTENSIONS

8.1 Separate Compilation

Although we have framed our formalism in terms of a "main expression" that includes **let**-bindings for all auxiliary functions, our approach is fully compatible with separate compilation of (the derivatives of) library functions. We transform each library function definition $f = e$ to its reverse derivative $f_{rev} = \overleftarrow{\mathcal{D}}\{e\}$, and subsequently transform $\overleftarrow{\mathcal{D}}\{f\}$ to f_{rev}.

Note carefully that the type of f_{rev} does not mention S, the argument type of the main expression; it mentions only *Delta*, which itself is also entirely independent of S. In contrast, in our earlier translation in Section 5.1, the type $\overleftarrow{\mathcal{D}}^1_S\{A\}$ *does* mention S, so that earlier translation does not support separate compilation (assuming C-style linkage without runtime polymorphism).

8.2 Adding More Primitive Operations

In our translations above the only primitive operations we gave translations for were $+_{\mathbb{R}}$ and $\times_{\mathbb{R}}$. However, it is easy to add a translation rule for any differentiable primitive operation over reals. For each one we must specify its (ordinary, forward) derivative in the translation. For example, to add *sin* and *cos* we simply add their translations to Figure 9 ($\mathit{negate}_{\mathbb{R}}$ is shorthand for $\lambda x.\ (-1.0) \times_{\mathbb{R}} x$):

$$\begin{aligned}
\overleftarrow{\mathcal{D}}\{\mathit{sin}\} &= \textbf{pure}\ (\lambda(x, u_1).\ \textbf{do}\ \{u_2 \leftarrow \mathit{deltaLet}\ (\mathit{Scale}\ (\mathit{cos}\ x)\ u_1);\ \textbf{pure}\ (\mathit{sin}\ x, \mathit{Var}\ u_2)\})\\
\overleftarrow{\mathcal{D}}\{\mathit{cos}\} &= \textbf{pure}\ (\lambda(x, u_1).\ \textbf{do}\ \{u_2 \leftarrow \mathit{deltaLet}\ (\mathit{Scale}\ (\mathit{negate}_{\mathbb{R}}\ (\mathit{sin}\ x))\ u_1);\ \textbf{pure}\ (\mathit{cos}\ x, \mathit{Var}\ u_2)\})
\end{aligned}$$

Because the difference between forward and reverse derivatives is not easily discerned for $\mathbb{R} \to \mathbb{R}$ functions, let us also look at the definition for a function from $\mathbb{R}^2 \to \mathbb{R}$, using *atan2* for concreteness:

$$\begin{aligned}
\overleftarrow{\mathcal{D}}\{\mathit{atan2}\} = \textbf{pure}\ &\lambda((y, u_2), (x, u_1)).\\
&\textbf{do}\ \{\ \textbf{let}\ t = 1\ /\ (x \times x + y \times y);\\
&\quad u_3 \leftarrow \mathit{deltaLet}\ (\mathit{Add}\ (\mathit{Scale}\ (\mathit{negate}_{\mathbb{R}}\ y \times t)\ u_1)\ (\mathit{Scale}\ (x \times t)\ u_2));\\
&\quad \textbf{pure}\ (\mathit{atan2}\ (y, x), \mathit{Var}\ u_3)\ \}
\end{aligned}$$

The general rules are given in Figure 14.

These examples also illustrate another interesting point. Most approaches to reverse-mode AD record some kind of *trace* of the forward execution, and then interpret that trace in reverse, executing the derivative operations in the process. But our *Delta* is a bit different to such a trace: it contains only five data constructors *Add*, *Scale*, *Zero*, *Var*, and *Let*, and real numbers. No record whatsoever is kept of what operations were done in the forward execution. In the code for $\overleftarrow{\mathcal{D}}\{sin\}$ we allocate a data constructor $(Scale\,(cos\;x)\,u_1)$, but that $(cos\;x)$ is computed in the forward pass, with the resulting real number captured in the *Scale* data constructor. Similarly in *atan2*, the local binding to t represents computation in the forward pass. This seems desirable — the work has to be done some time — but it is striking how simple and canonical the *Delta* type is. You can see a worked example in Figure 12, where only constructors and real numbers appear in the right hand column, despite the program on the left making calls to the *sqrt* function.

8.3 Arrays

Any serious AD system must support arrays and tensors well. Happily, it is easy to do so. For example, suppose we add a type *Vector*, with operations *build* and *index* to construct arrays and take them apart[8]:

$$index_A \;:\; (Vector\;A \times \mathbb{N}) \to A \qquad build_A \;:\; (\mathbb{N} \times (\mathbb{N} \to A)) \to Vector\;A$$

Then we can extend the translation of Figure 9 as follows:

Type translation

$$\overleftarrow{\mathcal{D}}\{Vector\;A\} = Vector\;\overleftarrow{\mathcal{D}}\{A\}$$

Term translation

$$\overleftarrow{\mathcal{D}}\{index_A\} = \mathbf{pure}\,(\lambda x.\,\mathbf{pure}\,(index_{\overleftarrow{\mathcal{D}}\{A\}}\;x))$$
$$\overleftarrow{\mathcal{D}}\{build_A\} = \mathbf{pure}\,(\lambda x.\,sequence_{\overleftarrow{\mathcal{D}}\{A\}}\,(build_{M\,\overleftarrow{\mathcal{D}}\{A\}}\;x))$$

Here $sequence_A \;:\; Vector\;(M\;A) \to M\;(Vector\;A)$ is a standard function in the monadic programmer's arsenal.

These translations may be *asymptotically* efficient, but they might not have a good constant factor; for example a $Vector\;\mathbb{R}$ translates to $Vector\;(\mathbb{R} \times Delta)$, a vector of pairs. It might be more efficient to use a pair of vectors, using a translation like

Type translation

$$\begin{array}{lcll} \overleftarrow{\mathcal{D}}\{Vector\;\mathbb{R}\} & = & Vector\;\mathbb{R} \times Vector\;Delta & \\ \overleftarrow{\mathcal{D}}\{Vector\;A\} & = & Vector\;\overleftarrow{\mathcal{D}}\{A\} & \text{when } A \neq \mathbb{R} \end{array}$$

That would be entirely possible; but of course the translation of the primitives would also need the same special cases. Indeed, many source languages have vectorised operations such as element-wise addition two tensors, or matrix multiplication, so another alternative might be to treat $Vector\;\mathbb{R}$ (and perhaps multi-dimensional versions) as new, primitive, differentiable data types alongside $\mathbb{R}$.

8.4 Floating Point Arithmetic

As mentioned above, our correctness results apply for true real numbers. This is valuable, and an important improvement over existing works as argued above. An additional extension, to perform a sensitivity analysis for floating point, would certainly be of interest, but is firmly future work.

[8]Our language lacks polymorphism, but we can allow these operations to be polymorphic by giving them their own typing rules, just as we do for (e_1, e_2) and **inl** e.

$$
\begin{array}{rl}
\text{Given} & e \;:\; \mathbb{R}^a \to \mathbb{R} \\
\text{and} & \overleftarrow{\mathcal{D}}\{e\} \;:\; (\mathbb{R} \times \mathit{Delta})^a \to M\,(\mathbb{R} \times \mathit{Delta}) \quad \text{(as defined in Figure 9)} \\
\text{we produce} & \overleftarrow{\mathcal{D}'}(e) \;:\; (\mathbb{R} \times \mathit{Delta})^a \to M\,(\mathbb{R} \times \mathit{Delta}) \\
& \overleftarrow{\mathcal{D}'}(e) = \lambda((x_1, \delta x_1), \ldots, (x_n, \delta x_n)). \\
& \qquad \textbf{let}\; x = ((x_1, \mathit{Var}\,1), \ldots, (x_n, \mathit{Var}\,n))\;\textbf{in} \\
& \qquad \textbf{let}\; (y, u) = \mathit{runDelta}\;(a\;+\;1)\;(\overleftarrow{\mathcal{D}}\{e\}\;x)\;\textbf{in} \\
& \qquad \textbf{let}\; \mathit{finalMap} = \mathit{eval}\;1\;u\;\mathit{emptyMap}\;\textbf{in} \\
& \qquad \textbf{do}\;\{\; \delta y \leftarrow \mathit{deltaLet}\;(\mathit{Add}^n_{i=1}\;(\mathit{Scale}\,(\mathit{lookupOrZero}\;i\;\mathit{finalMap})\;\delta x_i)); \\
& \qquad\quad \textbf{pure}\,(y, \mathit{Var}\;\delta y)\;\}
\end{array}
$$

Fig. 15. Fused reverse mode

8.5 Recursion and Recursive Types

Given that (a) we use a set-theoretic semantics for the language, and (b) the transformation is the identity everywhere except the real number type, it seems likely that adding inductive datatypes and folds over them will be unproblematic.

Adding full general recursion (and beyond that, mixed-variance recursive types) requires more involved changes to our correctness proof: we will need to extend our model to a domain-theoretic semantics or step-indexed logical relation, and we need a treatment (e.g., as offered in [Mazza and Pagani 2021]) of how to handle the nondifferentiable points arising from divergence.

8.6 Computing Higher-Order Derivatives

This paper is concerned with computing the first-order derivative of a function. What about higher-order derivatives? That is, we might want to know not only how a function f changes, but also how its derivative changes – f's second derivative. As we see in Figure 1, many forms of our language appear only in the *output* of differentiation and cannot be used in the *input*. Accordingly, simply running our transformation twice, by trying to compute $\mathcal{R}\{\mathcal{R}\{e\}\}$ is not straightforward.

A more promising approach is described by Karczmarczuk [1998], who generalises from the dual numbers to triple numbers, to compute first and second derivatives simultaneously, alongside the primal. That is, we currently transform a program over reals $\mathbb{R}$ to a program over pairs of reals $\mathbb{R} \times \delta\mathbb{R}$. To get second derivatives, we can produce a program over triples of reals $\mathbb{R} \times \delta\mathbb{R} \times \delta\delta\mathbb{R}$. To get higher derivatives we can use quadruples, etc, and indeed Karczmarczuk successfully generalises to a lazily-evaluated infinite tower of derivatives.

We have not worked out the details, but this seems to be a more promising approach than trying to compute $\mathcal{R}\{\mathcal{R}\{e\}\}$.

8.7 Selective Fusion of *Delta* Traces

Suppose our program has a (let-bound) function definition like this:

$$
\begin{array}{l}
f \;:\; \mathbb{R} \to \mathbb{R} \\
f \;=\; \lambda x.\, x \times_{\mathbb{R}} \mathit{sin}\; x
\end{array}
$$

Our translation will yield a definition of f:$(\mathbb{R} \times \mathit{Delta}) \to M\,(\mathbb{R} \times \mathit{Delta})$, and every call to f will build a bit of *Delta* data structure; and the bigger f is the more structure would be built. And yet, if we were to think of f as a primitive function, we could use the approach of Section 8.2 to define a function that built a *constant* amount of *Delta* structure.

The transformation is given in Figure 15, in the special case of expressions of type $\mathbb{R}^a \to \mathbb{R}$. We start from the existing transformation $\overleftarrow{\mathcal{D}}\{e\}$, and produce a modified transformation $\overleftarrow{\mathcal{D}'}(e)$ *with the same type signature*, so we can simply use $\overleftarrow{\mathcal{D}}\{e\}$ instead of $\overleftarrow{\mathcal{D}'}(e)$ wherever we please. The new transformation runs the wrapper code of Figure 10, and then wraps that in the impedance-matching code of Figure 14, using the partial derivatives returned by *runDelta*.

But why is this better? It is better because the *Delta* constructors produced by $\overleftarrow{\mathcal{D}}\{e\}$ can fuse (statically, at compile time) with the consumption of those constructors in *eval*. In our little example we would ultimately produce this code, which allocates one *Scale* constructor rather than many:

$$\begin{aligned} f \;&:\; (\mathbb{R} \times \mathit{Delta}) \to \mathit{M}\,(\mathbb{R} \times \mathit{Delta}) \\ f &= \lambda(x, u).\mathbf{do}\,\{\; y \leftarrow \mathit{deltaLet}\;(\mathit{Scale}\,(\mathit{sin}\;x +_{\mathbb{R}} (x \times_{\mathbb{R}} \mathit{cos}\;x))\,u); \\ &\qquad\qquad \mathbf{pure}\,(x +_{\mathbb{R}} \mathit{sin}\;x, \mathit{Var}\;y)\;\} \end{aligned}$$

We can apply the same trick to expressions of more general type, $S \to T$, using the techniques of Section 9. A harder question is this: exactly *when* should we use $\overleftarrow{\mathcal{D}'}(e)$ instead of $\overleftarrow{\mathcal{D}}\{e\}$? We do not yet have an answer to that question, and leave it for future work.

9 GENERALISING TO ARBITRARY S AND T

Our compositional, recursive transformations $\overrightarrow{\mathcal{D}}$ and $\overleftarrow{\mathcal{D}}$ work for expressions of *any* type, but we have thus far restricted the *main expression* to have type $\mathbb{R}^a \to \mathbb{R}$. Happily, everything we have done can be generalised to work for main expressions of type $S \to T$, for arbitrary first-order S and T, and we describe how to do so in this section. Generalising to arbitrary S and T is very useful in practice; for example, the main expression might take an integer parameter and two differently-shaped vectors, thus $e : \mathbb{R}^{a_1} \times \mathbb{Z} \times \mathbb{R}^{a_2} \to \mathbb{R}$.

We only deal with *first-order* types S and T; we do not attempt to differentiate main expressions whose argument or result types include functions[9].

To accomodate arbitrary first-order S and T, we need no changes to the translations in Figure 3 and 9; nor to the *Delta* data type; nor to the functions in Figure 11. The only changes needed are to the forward and reverse wrappers, and are shown in Figure 16.

9.1 Generalised Forward Mode

The forward-mode wrapper in Figure 16 is a straightforward generalisation of that in Figure 4 — indeed, it even looks a little simpler! It employs two new polytypic functions, whose types are given in Figure 16. First, zip_S does the re-assocation from a pair $(S \times \delta S)$ to a dualised S-structure with a dual number at each real-valued leaf. In the special case of Figure 4 this re-association took the form of transposing a pair of tuples into a tuple of pairs.

Finally delta_T takes the dual-number T structure returned by the function, and extracts the second component of each dual number; it is the companion to *primal*.

9.2 Generalised Reverse Mode

The reverse-mode wrapper in Figure 16 embodies the following changes, compared with Figure 10:

- The initial value $s_0{:}\overleftarrow{\mathcal{D}}\{S\}$ is obtained by replacing every real number s_i in s with a dual number $(s_i, \mathit{Var}\;\mathit{uid})$, where *uid* is a distinct *DeltaId*. This is done by the polytypic function $\mathit{initVars}_S$, which enumerates the real-valued slots of s left-to-right, giving each a distinct *DeltaId*. It returns the last used *DeltaId* as well as the dualised s.

[9]Perhaps we could, simply by treating such functions as constants, but it does not seem an important use-case to pursue.

Wrapper for forward derivative (generalised from Figure 4):

$$\begin{array}{rl} \text{Given} & e : S \to T \\ \text{and} & \overrightarrow{\mathcal{D}}\{e\} : \overrightarrow{\mathcal{D}}\{S\} \to \overrightarrow{\mathcal{D}}\{T\} \quad \text{(as defined in Figure 3)} \\ \text{we produce} & \mathcal{F}\{e\} : (S \times \delta S) \to \delta T \\ & \mathcal{F}\{e\} = \lambda x.\ \mathit{delta}_T\ (\overrightarrow{\mathcal{D}}\{e\}\ (\mathit{zip}_S\ x)) \end{array}$$

Wrapper for reverse derivative (generalised from Figure 10):

$$\begin{array}{rl} \text{Given} & e : S \to T \\ \text{and} & \overleftarrow{\mathcal{D}}\{e\} : \overleftarrow{\mathcal{D}}\{S\} \to M\ \overleftarrow{\mathcal{D}}\{T\} \quad \text{(as defined in Figure 9)} \\ \text{we produce} & \mathcal{R}\{e\} : (S \times \delta T) \to \delta S \\ & \mathcal{R}\{e\} = \lambda(s, \delta t). \\ & \quad \textbf{let}\ (s_0, \delta v) = \mathit{initVars}_S\ (s, 0)\ \textbf{in} \\ & \quad \textbf{let}\ \delta t_0 = \mathit{initZeros}_T\ \delta t\ \textbf{in} \\ & \quad \textbf{let}\ (\_, u) = \mathit{runDelta}\ (\delta v + 1)\ (\overleftarrow{\mathcal{D}}\{\lambda(s, \delta t).\ e\ s \odot_T \delta t\}\ (s_0, \delta t_0))\ \textbf{in} \\ & \quad \textbf{let}\ \mathit{finalMap} = \mathit{eval}\ 1\ u\ \mathit{emptyMap}\ \textbf{in} \\ & \quad \mathit{lookup}_S\ (\mathit{finalMap}, s_0) \end{array}$$

New polytypic primitives:

$\mathit{delta}_A : \overrightarrow{\mathcal{D}}\{A\} \to \delta A$	Select delta component of all dual numbers
$\mathit{zip}_A : (A \times \delta A) \to \overrightarrow{\mathcal{D}}\{A\}$	Zip two A-structures together
$\mathit{initVars}_A : (A \times \mathit{DeltaId}) \to (\overleftarrow{\mathcal{D}}\{A\} \times \mathit{DeltaId})$	Dualise each $\mathbb{R}$ with *Var uid*
$\mathit{initZeros}_A : A \to \overleftarrow{\mathcal{D}}\{A\}$	Dualise each $\mathbb{R}$ with *Zero*
$\mathit{lookup}_A : (\mathit{DeltaMap} \times \overleftarrow{\mathcal{D}}\{A\}) \to \delta A$	Look up each dual in the map

Fig. 16. Reverse-mode wrapper for general S and T

- We also define $\delta t_0{:}\overleftarrow{\mathcal{D}}\{\delta T\}$, by dualising δt in a similar way, but pairing each real value with *Zero* rather than *Var uid*. This is done by the polytypic function *initZeros*.
- We apply the $\overleftarrow{\mathcal{D}}$ transform not to e, but to the expression $\lambda(s, \delta t).\ e\ s \odot_T \delta t$, which has type $S \times \delta T \to \mathbb{R}$. That is, we take the (polytypic) dot-product (see Figure 1) of the result of the call $e\ s$ with δt, to get a $\mathbb{R}$.
- The transformed expression has type $\overleftarrow{\mathcal{D}}\{S\} \times \overleftarrow{\mathcal{D}}\{\delta T\} \to M\ (\mathbb{R} \times \mathit{Delta})$, so we can apply it to $(s_0, \delta t_0)$ to get a value of type $M\ (\mathbb{R} \times \mathit{Delta})$ which is what *runDelta* needs.
- Then, in the last line of the definition of $\mathcal{R}\{e\}$, we walk over s_0 with the polytypic function lookup_S, replacing each dual-number leaf $(s_i, \mathit{Var}\ \mathit{uid})$ with the result of looking up *uid* in *finalMap*, or zero if it is not in the map.

Notice that all this works smoothly for *sums* as well as *products*. If the input argument uses **inl** e at some point, so will the corresponding initial value s_0, and so will the returned value of type δS. The number of *DeltaId*s needed to build s_0 may be different for different input values s, even if they all have the same type S, but that is absolutely fine. The zip_S and $\odot_T$ operations used in Figure 16 require their two arguments to have the same "shape" – but that is already the case for vectors, where the sizes must match.

	(1) Reverse mode	(2) Higher order	(3) Asymptotically efficient	(4) Correctness proof
[Pearlmutter and Siskind 2008]	✓	✓	✓	✗
[Elliott 2018]	✓	✗	✗	✓
[Wang et al. 2019]	✓	✓	✓	✗
[Plotkin and Abadi 2020]	✓	✗	✓	✓
[Sherman et al. 2021]	✓	✓	✓	✗
[Huot et al. 2020]	✓	✓	✗	✗
[Mazza and Pagani 2021]	✓	✓	✗	✓
Ours	✓	✓	✓	✓

Fig. 17. Properties of various works on AD

10 RELATED WORK

There is a rich and rapidly growing literature on automatic differentiation, going back over 50 years [Wengert 1964]. Here we focus primarily on work that tackles *reverse-mode* AD for *higher-order* languages with full first-class functions. In Figure 17 we summarise some key works using the four properties described in the Introduction; we discuss each of these works below.

The original work on AD was aimed at first-order imperative programming languages [Griewank and Walther 2008], but there has always been interest in extending it to support richer programming languages. The case of forward mode is easy (indeed, almost trivial) to extend to higher-order; the dual number representation of the reals means that any form of parameterisation over the representation of the real number type (eg, Haskell type classes or ML functors) suffices to implement forward-mode AD. This observation is by no means original to functional languages. To our knowledge it was popularised by [Piponi 2004] in the context of C++ template meta-programming, and by [Karczmarczuk 1998] and [Elliott 2009] in the context of type-class overloading. However, the idea was in use in the scientific computing world long before then.

Beginning with the pioneering work of Pearlmutter and Siskind in *"Lambda the ultimate back-propagator"* [Pearlmutter and Siskind 2008], there has been significant effort to extend *reverse mode* to support higher-order programming languages. Their work uses an elaborate source-to-source transformation reminiscent of defunctionalisation, which made it possible to apply many of the techniques for first-order automatic differentiation to the higher-order case. Although it is full of insights that shaped following research, this work was implementation-focused, and offered neither an intended semantics nor a correctness proof.

The complexity of the ultimate-backpropagator approach prompted several responses. On the implementation side, [Wang et al. 2019] observed that much implementation complexity could be avoided through the strategic use of delimited control to get the same program to perform both the forward and reverse pass. Combined with the use of staging, this made it possible to write very efficient implementations of automatic differentiation.

In [Elliott 2018], Elliott argued that a useful lens for understanding automatic differentiation was to focus on designing transformations which operated in a compositional way. Though this work did not scale up to higher-order languages[10], it did offer a smoothly compositional and mathematically well-behaved treatment of AD, which proved influential in succeeding work. [Plotkin and Abadi 2020] also study a first-order programming language with a differentiation construct. They equip this language with both operational and denotational semantics, which they show coincide.

[10] But see http://conal.net/papers/higher-order-ad/ for Elliott's work in progress.

[Brunel et al. 2020] synthesised the ideas of [Elliott 2018] and [Wang et al. 2019], and showed that their continuation-based approach could be analysed in terms of a language with a linear negation operator. They do indeed give a formal correctness proof of their transformation, but the language they study does not include conditionals or looping.

To understand these issues better, [Mazza and Pagani 2021] studied reverse-mode AD in the context of PCF. To simplify their presentation, they gave up on the linear negation (and hence were not concerned with the efficiency of implementation), but were able to show that a language with higher-order functions, branching and recursion has an AD algorithm which was correct everywhere except a measure 0 set.

[Huot et al. 2020] proposed the approach we most directly base our work upon. They give a denotational semantics for a higher-order language with bounded iteration and conditionals using diffeological spaces, and use this semantics to show that the forward-mode AD algorithm is correct. Roughly, the structure of the category of diffeological spaces encodes the information of a logical relation, enabling them to model full function spaces. The semantics we give is more "low-tech", making it easier to work in things like state monads, but the core insights derive from this paper.

They also briefly sketch a reverse-mode translation phrased in terms of linear continuations of type $\mathbb{R} \multimap \mathbb{R}^a$; [Wang et al. 2019] use "back-propagator" for the same function. However, since these continuations are always linear, the type $\mathbb{R} \multimap \mathbb{R}^a$ is isomorphic to the type $\mathbb{R}^a$. By systematically applying this isomorphism to their reverse-mode translation, we can derive our first inefficient version of reverse mode in Figure 5, the baseline that we subsequently optimise.

The fact that simple, "geometry-ignorant" optimisations sufficed surprised us, since *a priori* one would expect the algebraic properties of derivatives to play a more significant role. In different ways, [Vákár 2021] and [Mak and Ong 2020] both take this approach. [Vákár 2021] uses a variant of the enriched effect calculus [Egger et al. 2012] and specifies a translation, while [Mak and Ong 2020] use a version of the differential lambda calculus and give a direct operational semantics.

Despite their differences, both of these approaches emphasise the significance of the linearity of the derivative in their respective calculi, by making it part of the syntax of the language. Our deltas, as Lemma 10 shows, never store nonlinear functions in their trace, and our use of lets to preserve sharing is mirrored in the A-normalising reduction rules of the pullback terms of [Mak and Ong 2020]. While we cannot yet make any precise claim, the connections are tantalising.

With the exception of [Mazza and Pagani 2021], all of the work above (including our own) focuses on smooth, differentiable functions. This would seem to prohibit using important activation functions like RelU, which is not smooth (RelU's derivative has a discontinuity at 0). This problem is attacked by [Sherman et al. 2021] (who claim inspiration from [Vákár 2021] and [Elliott 2018]).

This paper takes Elliott's non-higher-order semantics and considers presheaves over Elliott's category, enabling them to model sums and function types. Furthermore, they make use of the Clarke derivative (or subderivative) in order to give semantics to non-differentiable functions like ReLU. While this is definitely valid, one puzzling feature of this approach is that subderivatives were originally invented for convex optimisation, and it is unclear whether it works in principle for ML-style gradient descent problems. They do have an implementation of λ_S, but why it works remains somewhat mysterious – perhaps [Mazza and Pagani 2021] can shed light on this.

ACKNOWLEDGMENTS

We warmly thank Peter Braam, Pashmina Cameron, Edward Kmett, Siddharth Krishna, Sarah Lewis, Tom Minka, and Dimitrios Vytiniotis for their helpful feedback. We are also grateful for the helpful and detailed suggestions we received from our anonymous reviewers. This work was supported by Microsoft Research through its PhD Scholarship Programme.

REFERENCES

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

George Corliss, Christèle Faure, Andreas Griewank, and Uwe Naumann. 2002. *Automatic Differentiation of Algorithms: From Simulation to Optimization.* Springer. https://doi.org/10.1007/978-1-4613-0075-5

Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. 2012. The enriched effect calculus: syntax and semantics. *Journal of Logic and Computation* 24, 3 (06 2012), 615–654. https://doi.org/10.1093/logcom/exs025 arXiv:https://academic.oup.com/logcom/article-pdf/24/3/615/2785623/exs025.pdf

Conal Elliott. 2018. The simple essence of automatic differentiation. *Proc. ACM Program. Lang.* 2, ICFP (2018), 70:1–70:29. https://doi.org/10.1145/3236765

Conal M. Elliott. 2009. Beautiful Differentiation. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09).* Association for Computing Machinery, New York, NY, USA, 191–202. https://doi.org/10.1145/1596550.1596579

Andreas Griewank and Andrea Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* (second ed.). Society for Industrial and Applied Mathematics, USA. https://doi.org/10.1137/1.9780898717761

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

Achim Jung and Jerzy Tiuryn. 1993. A new characterization of lambda definability. In *Typed Lambda Calculi and Applications*, Marc Bezem and Jan Friso Groote (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 245–257. https://doi.org/10.1007/BFb0037110

Jerzy Karczmarczuk. 1998. Functional Differentiation of Computer Programs. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98).* Association for Computing Machinery, New York, NY, USA, 195–203. https://doi.org/10.1145/289423.289442

Edward Kmett, Barak Pearlmutter, and Jeffrey Mark Siskind. 2010–2021. ad: Automatic Differentiation. Haskell package at https://hackage.haskell.org/package/ad. (2010–2021). Accessed: 2021-11-10.

Carol Mak and Luke Ong. 2020. A Differential-form Pullback Programming Language for Higher-order Reverse-mode Automatic Differentiation. *CoRR* abs/2002.08241 (2020). arXiv:2002.08241 https://arxiv.org/abs/2002.08241

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

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

Barak A. Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Trans. Program. Lang. Syst.* 30, 2 (2008), 7:1–7:36. https://doi.org/10.1145/1330017.1330018

Dan Piponi. 2004. Automatic Differentiation, C++ Templates, and Photogrammetry. *J. Graphics, GPU, & Game Tools* 9, 4 (2004), 41–55. https://doi.org/10.1080/10867651.2004.10504901

Gordon Plotkin and Martin Abadi. 2020. A simple differentiable programming language. *Proc. ACM Program. Lang.* POPL (2020). https://doi.org/10.1145/3371106

John C. Reynolds. 1998. Definitional Interpreters for Higher-Order Programming Languages. *High. Order Symb. Comput.* 11, 4 (1998), 363–397. https://doi.org/10.1023/A:1010027404223

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

Matthijs Vákár. 2021. Reverse AD at Higher Types: Pure, Principled and Denotationally Correct. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science)*, Nobuko Yoshida (Ed.), Vol. 12648. Springer, 607–634. https://doi.org/10.1007/978-3-030-72019-3_22

Fei Wang, Daniel Zheng, James M. Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. 2019. Demystifying differentiable programming: shift/reset the penultimate backpropagator. *Proc. ACM Program. Lang.* 3, ICFP (2019), 96:1–96:31. https://doi.org/10.1145/3341700

RE Wengert. 1964. A simple automatic derivative evaluation program. *Communication of the ACM* 7 (Aug. 1964), 463–464. Issue 8. https://doi.org/10.1145/355586.364791