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

FAUSTYNA KRAWIEC, University of Cambridge, UK
NEEL KRISHNASWAMI, University of Cambridge, UK
SIMON PEYTON JONES, Microsoft Research, UK
TOM ELLIS, Microsoft Research, UK
ANDREW FITZGIBBON, Microsoft Research, UK
RICHARD EISENBERG, Tweag I/O, USA

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.

Draft, under review

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages.*

ACM Reference Format:
Faustyna Krawiec, Neel Krishnaswami, Simon Peyton Jones, Tom Ellis, Andrew Fitzgibbon, and Richard Eisenberg. 2018. Provably Correct, Asymptotically Efficient, Higher-Order Reverse-Mode Automatic Differentiation. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2018), 32 pages.

1 INTRODUCTION

Automatic differentiation (AD) is a way of transforming a program into a new one that computes exact analytical derivatives of the original. It allows an observation of how the results of a function vary as its inputs do and thus can guide an optimisation problem such as the training of a machine learning algorithm. Automatic differentiation has a rich and rapidly growing literature, as we survey in Section 12, but it is hard to find an approach to AD that has all these properties:

- *Does reverse mode AD* as well as forward. 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.
- *Fast.* It runs in linear time, with only a constant-factor slow-down relative to the original (or "primal") program.
- *Expressive.* It is capable of differentiating a *higher-order* language with first-class anonymous functions; with *sum types* as well as products and arrays; and with *functions and data types.*
- *Compositional.* If a program consists of many function definitions, we want to differentiate them separately, rather than (say) inlining all the calls and differentiating the result.

Authors' addresses: Faustyna Krawiec, Department of Computer Science, University of Cambridge, Street1 Address1, City1, State1, Post-Code1, UK, fmk31@cl.cam.ac.uk; Neel Krishnaswami, Department of Computer Science, University of Cambridge, Street1 Address1, City1, State1, Post-Code1, UK, nk480@cl.cam.ac.uk; Simon Peyton Jones, Microsoft Research, Street1 Address1, City1, State1, Post-Code1, UK, simonpj@microsoft.com; Tom Ellis, Microsoft Research, Street1 Address1, City1, State1, Post-Code1, UK, toellis@microsoft.com; Andrew Fitzgibbon, Microsoft Research, Street1 Address1, City1, State1, Post-Code1, UK, awf@microsoft.com; Richard Eisenberg, Tweag I/O, Street1 Address1, City1, State1, Post-Code1, USA, rae@richarde.dev.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

- *Parsimonious.* It can be understood without knowledge of sophisticated concepts such as tangent bundles, diffeological spaces, delimited continuations, cocartesian categories, etc..
- *Provably correct.* The relative simplicity of our system allows us to prove that it is correct. Today's most sophisticated AD systems are often presented as only as large, complex software artefacts whose internals are only truly understood by a few.

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

- 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 10).
- We give a standard forward-mode AD translation for this language, using so-called *dual numbers* (Section 4). The transformation is extremely simple: **let** turns into **let**, lambda into lambda, application into application. 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).
- The algorithm of Section 5 may be simple, but it is also utterly impractical because it embodies two gigantic (indeed asymptotic) inefficiencies. In Sections 6 and 7 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 (namely Figure 9 and 10). We show that it is asymptotically efficient (Section 7.4), and supports separate compilation (Section 7.5).
- We give a proof of correctness of our final algorithm in Section 9, based on logical relations.
- For most of the paper we concentrate on a "main expression", to be differentiated, 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 11.

We sketch some other useful generalisations in Section 10 and discuss related work in Section 12.

In a sense we have done nothing "new". Looked at from 100,000 feet, all AD algorithms end up looking pretty much the same; and 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 speculate) easier to use as a basis for exploring design variations (as indeed we do through the narrative of this paper).

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.

Types. The language is statically typed. Types include real numbers[1] $\mathbb{R}$, so-called leaf types L, binary products and sums, and the function type. Leaf 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 Appendix A.1.

[1]A typical implementation would use floating-point numbers to approximate reals. Our results hold for true real numbers; when real numbers are approximated by floating-point representations, our results are approximately true.

$x, y,$ dx, dy	$::=$	$\ldots$	Variables
n, a, b	$::=$	$0, 1, \ldots$	Natural numbers
r	$::=$	$\ldots, 0.0, \ldots$	Reals
$e, s, t,$ ds, dt	$::=$	$x \mid \lambda x.\, e \mid e_1\ e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$	λ-calculus forms
	$\mid$	$\mathbf{fst}\ e \mid \mathbf{snd}\ e \mid (e_1, e_2)$	Products (i.e. pairs)
	$\mid$	$\mathbf{inl}\ e \mid \mathbf{inr}\ e \mid \mathbf{case}\ e\ \mathbf{of}\ \mathbf{inl}\ x_1 \to e_1,\ \mathbf{inr}\ x_1 \to e_2$	Sums (i.e. unions)
	$\mid$	$k \mid r \mid op \mid$ $polyop_A$	Constants
	$\mid$	$\mathbf{pure}\ e \mid \mathbf{do}\ \{\ stmts\ \}$	Monadic operations
k	$::=$	$\langle integer\ lits\rangle \mid \langle string\ lits\rangle \mid \ldots$	Leaf 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	$::=$	$\mathbb{R} \mid L \mid A \times B \mid A + B \mid A \to B \mid$ $M\ A$	Types
L	$::=$	$\mathbb{Z} \mid String \mid \ldots$	Leaf 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 $\times$ when the type is apparent.
- We use $\lambda(x_1, x_2).\, e$ and $\mathbf{let}\ (x_1, x_2) = e_1\ \mathbf{in}\ e_2$ 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 11.2	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.

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

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 also informally permit ourselves to use pattern-matching on pairs in lambda and let, thus $\lambda(x_1, x_2).\, e$ and $\mathbf{let}\ (x_1, y_2) = e_1\ \mathbf{in}\ e_2$, but merely as syntactic sugar for projections. We also informally allow ourselves to use n-ary products (e_1, e_2, e_3, e_4) rather than nested pairs.

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 leaf 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). Leaf literals are distinguished from constants because leaves 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 10.

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:

$$\begin{aligned}
\otimes_\mathbb{R} &= \lambda(x,y).\, x \times_\mathbb{R} y \\
\otimes_\mathbb{Z} &= \lambda(x,y).\, y \\
\otimes_{A\times B} &= \lambda(x,(y_1,y_2)).\, (x \otimes_A y_1, x \otimes_B y_2) \\
\otimes_{A+B} &= \lambda(x,y).\, \mathbf{case}\ y\ \mathbf{of}\ \mathbf{inl}\ y_1 \longrightarrow \mathbf{inl}\ (x \otimes_A y_1),\ \mathbf{inr}\ y_2 \longrightarrow \mathbf{inr}\ (x \otimes_B y_2)
\end{aligned}$$

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 $S \longrightarrow T$; 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 $e : S \longrightarrow T$. (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 $e : \mathbb{R}^a \longrightarrow \mathbb{R}$, leaving the generalisation to arbitrary (first-order) S and T to Section 11. 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 $\mathbb{R}^a \longrightarrow \mathbb{R}$, 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.

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

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

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

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

Notation:

- For any function $f : \mathbb{R}^a \to \mathbb{R}^b$, its Jacobian $\mathcal{J}f \in \mathbb{R}^a \to \mathbb{R}^{b\times a}$ 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.

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 $f \in \mathbb{R}^a \to \mathbb{R}^b$, the Jacobian $\mathcal{J}f \in \mathbb{R}^a \to \mathbb{R}^{b\times a}$ is a function that, for each $x \in \mathbb{R}^a$, 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, $\delta\mathbb{R}^a \; = \; \mathbb{R}^a$, but we will still suggestively write $\delta\mathbb{R}$ where we wish to think about differences.
- Given a function $f{:}\mathbb{R}^a \to \mathbb{R}^b$, let $\mathcal{J}f \in \mathbb{R}^a \to \delta\mathbb{R}^{b\times a}$ 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 $e : \mathbb{R}^a \to \mathbb{R}^b$, $s \in S$, $ds \in \delta\mathbb{R}^a$, $dt \in \delta\mathbb{R}^b$, we have $dt \bullet [\![\mathcal{F}\{e\}]\!](s, ds) = [\![\mathcal{R}\{e\}]\!](s, dt) \bullet ds$*

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

[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 $\mathbb{Z} + \mathbb{Z}$.

[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 $x : \mathbb{R}^a$ we can reconstruct the Jacobian matrix $\mathcal{J}f(\vec{x})$ in either of these ways:

- Forward Plan: make a calls to $\mathcal{F}\{e\}$ $(x, \mathit{onehot}_{\mathbb{R}^a}\ j)$, 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 $\mathcal{R}\{e\}$ $(x, \mathit{onehot}_{\mathbb{R}^b}\ i)$, 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 $(0, \dots, 0, 1, 0, \dots, 0) \in \mathbb{R}^a$, where the 1 is in the i'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 $\mathbb{R}^a \rightarrow \mathbb{R}$, where a is large, say 10^8. 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 $a = 10^8$ and $b = 1$, the Reverse Plan computes all the required partial derivatives in one call, rather than 10^8 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 well shall see, reverse mode trades space for time, accumulating a data structure that records the execution of 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 the Figure.

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 leaf literals 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[5] $\mathbb{R} \times \delta\mathbb{R}$.

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

[5] Recall that $\delta\mathbb{R}$ is a synonym for $\mathbb{R}$, Section 3.2.

$\boxed{\overrightarrow{\mathcal{D}}\{A\}}$ Forward-mode on types

$$
\begin{aligned}
\overrightarrow{\mathcal{D}}\{A \times B\} &= \overrightarrow{\mathcal{D}}\{A\} \times \overrightarrow{\mathcal{D}}\{B\} \\
\overrightarrow{\mathcal{D}}\{A + B\} &= \overrightarrow{\mathcal{D}}\{A\} + \overrightarrow{\mathcal{D}}\{B\} \\
\overrightarrow{\mathcal{D}}\{A \to B\} &= \overrightarrow{\mathcal{D}}\{A\} \to \overrightarrow{\mathcal{D}}\{B\} \\
\overrightarrow{\mathcal{D}}\{L\} &= L \\
\overrightarrow{\mathcal{D}}\{\mathbb{R}\} &= \mathbb{R} \times \delta\mathbb{R}
\end{aligned}
$$

$\boxed{\overrightarrow{\mathcal{D}}\{\Gamma\}}$ Forward-mode on contexts

$$
\begin{aligned}
\overrightarrow{\mathcal{D}}\{\bullet\} &= \bullet \\
\overrightarrow{\mathcal{D}}\{\Gamma, x{:}A\} &= \overrightarrow{\mathcal{D}}\{\Gamma\}, x{:}\overrightarrow{\mathcal{D}}\{A\}
\end{aligned}
$$

$\boxed{\overrightarrow{\mathcal{D}}\{e\}}$ Forward-mode on expressions

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

$$
\begin{aligned}
\overrightarrow{\mathcal{D}}\{x\} &= x \\
\overrightarrow{\mathcal{D}}\{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\} &= \mathbf{let}\ x = \overrightarrow{\mathcal{D}}\{e_1\}\ \mathbf{in}\ \overrightarrow{\mathcal{D}}\{e_2\} \\
\overrightarrow{\mathcal{D}}\{\lambda x.\, e\} &= \lambda x.\, \overrightarrow{\mathcal{D}}\{e\} \\
\overrightarrow{\mathcal{D}}\{e_1\ e_2\} &= \overrightarrow{\mathcal{D}}\{e_1\}\ \overrightarrow{\mathcal{D}}\{e_2\} \\
\overrightarrow{\mathcal{D}}\{\mathbf{inl}\ e\} &= \mathbf{inl}\ \overrightarrow{\mathcal{D}}\{e\} \\
\overrightarrow{\mathcal{D}}\{\mathbf{inr}\ e\} &= \mathbf{inr}\ \overrightarrow{\mathcal{D}}\{e\} \\
\overrightarrow{\mathcal{D}}\{\mathbf{case}\ e_0\ \mathbf{of}\ \mathbf{inl}\ x_1 \to e_1,\ \mathbf{inr}\ x_2 \to e_2\} &= \mathbf{case}\ \overrightarrow{\mathcal{D}}\{e_0\}\ \mathbf{of}\ \mathbf{inl}\ x_1 \to \overrightarrow{\mathcal{D}}\{e_1\},\ \mathbf{inr}\ x_2 \to \overrightarrow{\mathcal{D}}\{e_2\} \\
\overrightarrow{\mathcal{D}}\{(e_1, e_2)\} &= (\overrightarrow{\mathcal{D}}(e_1), \overrightarrow{\mathcal{D}}(e_2)) \\
\overrightarrow{\mathcal{D}}\{\mathbf{fst}\ e\} &= \mathbf{fst}\ \overrightarrow{\mathcal{D}}\{e\} \\
\overrightarrow{\mathcal{D}}\{\mathbf{snd}\ e\} &= \mathbf{snd}\ \overrightarrow{\mathcal{D}}\{e\} \\
\overrightarrow{\mathcal{D}}\{k\} &= k \\
\overrightarrow{\mathcal{D}}\{+_{\mathbb{Z}}\} &= +_{\mathbb{Z}} \\
\overrightarrow{\mathcal{D}}\{r\} &= (r, 0) \\
\overrightarrow{\mathcal{D}}\{+_{\mathbb{R}}\} &= \lambda((x, dx), (y, dy)).\, (x + y, dx + dy) \\
\overrightarrow{\mathcal{D}}\{\times_{\mathbb{R}}\} &= \lambda((x, dx), (y, dy)).\, (x \times y, y \times dx + x \times dy)
\end{aligned}
$$

Fig. 3. Forward mode translation

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. Leaf literals and their functions are entirely unaffected.

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

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

$$\overrightarrow{\mathcal{D}}\{\lambda(x{:}\mathbb{R}).\, e\} \quad = \quad \lambda(x{:}\mathbb{R} \times \delta\mathbb{R}).\, \overrightarrow{\mathcal{D}}\{e\}$$

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

$$primal_T\ (\overrightarrow{\mathcal{D}}\{e\}\ s) = e\ (primal_S\ s)$$

where $primal_A : \overrightarrow{\mathcal{D}}\{A\} \to A$ is an operation that removes the second (delta) component of each dual number, returning just the primal value. Keeping the original values alongside the tangent-space ones makes the transformation *compositional*; for example, the forward derivative of an expression $(e_1\ e_2)$ combines the forward derivatives of e_1, namely $\overrightarrow{\mathcal{D}}\{e_1\}$, and similarly e_2 (see Figure 3). To do that we need the *values* of e_1 and e_2. To see this very concretely, look at the translation for multiplication in Figure 3:

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

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, to 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.

4.3 Correctness

Proposition 2. *Correctness of the forward mode transformation If* $e : \mathbb{R}^a \to \mathbb{R}$ *and* $\vec{x}, \delta s \in \mathbb{R}^a$, *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 9, is directly inspired by their argument.

$\boxed{\overleftarrow{\mathcal{D}}^1_S\{A\}}$

Reverse-mode on types

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

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

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

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

Fig. 5. Reverse mode AD #1 (inefficient), 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 12. 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[6]. 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 $ds_1 \oplus_S ds_2$ (Figure 1) to combine two values of type δS; this operation (as others) assumes that the two values ds_1 and ds_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; $r \otimes_S ds$ multiplies each real in ds by r.

[6]The "1" superscript is there because this is our first version; subsequent versions improve upon this one.

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

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

$$\begin{array}{rcl}
\mathbf{data}\ \mathit{Delta} & = & \mathit{Zero} \\
 & | & \mathit{OneHot}\ \mathbb{N} \\
 & | & \mathit{Scale}\ \mathbb{R}\ \mathit{Delta} \\
 & | & \mathit{Add}\ \mathit{Delta}\ \mathit{Delta}
\end{array}$$

$\boxed{\overleftarrow{\mathcal{D}}^2\{A\}}$ Reverse-mode on types

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

$\boxed{\overleftarrow{\mathcal{D}}^2\{e\}}$ Reverse-mode on expressions

Invariant: If $\Gamma \vdash e : A$, then $\overleftarrow{\mathcal{D}}^2\{\Gamma\} \vdash \overleftarrow{\mathcal{D}}^2\{e\} : \overleftarrow{\mathcal{D}}^2\{A\}$

$$\begin{array}{rcl}
\overleftarrow{\mathcal{D}}^2\{r\} & = & (r, \mathit{Zero}) \\
\overleftarrow{\mathcal{D}}^2\{+_{\mathbb{R}}\} & = & \lambda((x, u_1), (y, u_2)).\, (x + y, \mathit{Add}\, u_1\, u_2) \\
\overleftarrow{\mathcal{D}}^2\{\times_{\mathbb{R}}\} & = & \lambda((x, u_1), (y, u_2)).(x \times y, \mathit{Add}\, (\mathit{Scale}\, y\, u_1) \\
 & & \qquad\qquad\qquad\qquad\qquad (\mathit{Scale}\, x\, u_2))
\end{array}$$

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

5.2 The wrapper

Next we build the wrapper $\mathcal{R}^1\{e\}$ (Figure 6) that connects $\overleftarrow{\mathcal{D}}^1_S\{e\}$ to external clients. While possible to do this in full generality, we instead write the wrapper somewhat informally and for the special case of $S = \mathbb{R}^a$, $T = \mathbb{R}$. Our final version of reverse-mode AD includes the fully general formulation.

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 $dt \otimes_{\mathbb{R}^a} \mathit{onehot}_{\mathbb{R}^a}\ i$. 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 6.

5.3 Summary

It is remarkable that such a simple translation supports fully 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 9, when we have the final version in hand.

6 TOWARDS EFFICIENCY: INTENSIONAL UPDATES

The reverse-mode AD of Section 5 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

$$\overleftarrow{\mathcal{D}}^1_S\{\times_{\mathbb{R}}\} = \lambda((x, ds_1), (y, ds_2)).\, (x \times y, (y \otimes_S ds_1) \oplus_S (x \otimes_S ds_2))$$

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 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.

6.1 From delta values to deltas of values

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

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

where we can think of *Delta* as specifying the *difference*, or "delta", between two values of type S. While 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 finite vocabulary of such functions*, so it is much nicer to represent deltas by a simple algebraic data type, given in Figure 7[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 11.

Another way to regard *Delta* is as a 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 *updateAt*, defined in Figure 7. 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 *updateAt* can thus be extremely efficient: *it can work by in-place mutation*; linearity assures us that the input to *updateAt* 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*.

6.2 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.

$$
\begin{array}{rl}
\text{Given} & e \;:\; \mathbb{R}^a \to \mathbb{R} \\
\text{and} & \overleftarrow{\mathcal{D}}^2\{e\} \;:\; (\mathbb{R} \times \mathit{Delta})^a \to (\mathbb{R} \times \mathit{Delta}) \quad \text{(as defined in Figure 7)} \\
\text{we produce} & \mathcal{R}^2\{e\} \;:\; (\mathbb{R}^a \times \delta\mathbb{R}) \to \delta\mathbb{R}^a \\
& \mathcal{R}^2\{e\} = \lambda((\ldots, s_i, \ldots), dt). \\
& \qquad \mathbf{let}\; s_0 = (\ldots, (s_i, \mathit{OneHot}\; i), \ldots)\; \mathbf{in} \\
& \qquad \mathbf{let}\; z = (\ldots, 0, \ldots)\; \mathbf{in} \\
& \qquad \mathbf{let}\; (t, u) = \overleftarrow{\mathcal{D}}^2\{e\}\; s_0\; \mathbf{in} \\
& \qquad \mathit{eval}_2\; dt\; u\; z
\end{array}
$$

$$
\begin{array}{l}
\mathit{eval}_2 : \mathbb{R} \to \mathit{Delta} \to \mathbb{R}^a \multimap \mathbb{R}^a \\
\mathit{eval}_2\; x\; \mathit{Zero} \qquad\qquad \vec{x} = \vec{x} \\
\mathit{eval}_2\; x\; (\mathit{OneHot}\; i) \quad \vec{x} = \mathit{updateAt}\; i\; x\; \vec{x} \\
\mathit{eval}_2\; x\; (\mathit{Scale}\; y\; u) \quad \vec{x} = \mathit{eval}_2\; (x \times y)\; u\; \vec{x} \\
\mathit{eval}_2\; x\; (\mathit{Add}\; u_1\; u_2)\; \vec{x} = \mathit{eval}_2\; x\; u_2\; (\mathit{eval}_2\; x\; u_1\; \vec{x})
\end{array}
$$

$$
\begin{array}{l}
\mathit{updateAt} : \mathbb{N} \to \mathbb{R} \to \mathbb{R}^a \multimap \mathbb{R}^a \\
\mathit{updateAt}\; i\; x\; \vec{y} = \ldots \\
\qquad \text{-- Return } \vec{y} \text{ with its } i\text{'th component} \\
\qquad \text{-- increased by } x
\end{array}
$$

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

7 FINALLY EFFICIENT: A MONADIC TRANSLATION

The translation of Section 6 eliminates one form of asymptotic inefficiency (blowing up values by a factor of n) but sadly introduces another that is just as bad. Consider f and its translation $\overleftarrow{\mathcal{D}}^2\{f\}$:

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

Here the $\vec{y} \;:\; \mathit{Delta}$ is a perhaps-large data structure, arising from executing $\overleftarrow{\mathcal{D}}^2\{e\}$; and 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:

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

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.

7.1 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[8] in Figure 9 (defining $\overleftarrow{\mathcal{D}}$, our final answer, with no version number), and its supporting functions in Figure 10.

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

[8]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.

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

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

$\boxed{\overleftarrow{\mathcal{D}}\{e\}}$ 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}\; \mathbf{inl}\; x_1 \to e_1, &= \mathbf{do}\; \{x \leftarrow \overleftarrow{\mathcal{D}}\{e_0\}; \\
\mathbf{inr}\; x_2 \to e_2\} &\qquad \mathbf{case}\; x\; \mathbf{of}\; \mathbf{inl}\; x_1 \to \overleftarrow{\mathcal{D}}\{e_1\}, \\
&\qquad\qquad \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)). \\
&\qquad \mathbf{do}\; \{u_3 \leftarrow \mathit{deltaLet}\; (\mathit{Add}\; u_1\; u_2); \\
&\qquad\qquad \mathbf{pure}\; (x + y, \mathit{Var}\; u_3)\} \\
\overleftarrow{\mathcal{D}}\{\times_{\mathbb{R}}\} &= \mathbf{pure}\; \lambda((x, u_1), (y, u_2)). \\
&\qquad \mathbf{do}\; \{u_3 \leftarrow \mathit{deltaLet}\; (\mathit{Add}\; (\mathit{Scale}\; y\; u_1)\; (\mathit{Scale}\; x\; u_2)); \\
&\qquad\qquad \mathbf{pure}\; (x \times y, \mathit{Var}\; u_3)\}
\end{aligned}
$$

Fig. 9. Monadic translation

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 10 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 6.2). We will see in Section 7.2 that we can use *Var* instead.

```
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 dR

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) → Delta
  -- Runs the computation, and wraps the result in
  -- the bindings produced by running the computation
runDelta delta_id m = foldl wrap delta binds
  where
    ((_, 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. 10. Supporting functions (rendered in Haskell-like syntax) for the monadic translation

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:

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

Fig. 11. Reverse-mode wrapper around $\overleftarrow{\mathcal{D}}$

```
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 the translations of $\overleftarrow{\mathcal{D}}\{+_{\mathbb{R}}\}$ 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.

7.2 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 11. 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 $M\ (\mathbb{R} \times \mathit{Delta})$
(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 10, 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* $dt\ u\ m$) extends the map m : *DeltaMap* with the extra contributions described by u to the free *DeltaId*s of u, scaled by dt.
(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 10 gives the rather simple definition of *eval*.

7.3 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
```

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`[9]. It is a very clever library, and now for the first time, we understand how it works and can prove it correct.

7.4 Asymptotic efficiency

Our monadic translation is asymptotically efficient. The run time is a constant factor worse than the untranslated program, and the memory usage of the translated program is linear in the run time of the program. Here is a sketch proof of this claim.

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. Assuming that the program originally executed in a call-by-value style, the monadic translation adds at most a constant factor overhead. The additional work in each of the primitive operations (like *plus* and *times*) also adds a constant-time overhead. So the time to run the translated program is linear in the run time of the original program.

Like all reverse-mode algorithms, the memory usage will be worse, however. Since each translated primitive allocates a constant additional amount of memory, this means that the additional memory allocated by the translated program is linear in 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 will be bounded by the run time of the original program.

This is not quite enough to establish our time and space bounds, though. Once we run a translated program, we have in our hands a *Delta* representing the derivative, which needs to be evaluated in order to produce the actual derivative of interest. To do this, we will show that *eval*, given in Figure 10, also runs in linear time. This function takes a *Delta* u, a scale factor x and a *DeltaMap* um mapping *DeltaId*s to their contribution to the overall derivative. It recursively traverses the *Delta*, updating each *DeltaId*'s contribution. The cases for *Scale* y u and *Add* u_1 u_2 are straightforward.

The two interesting cases are those for *Var* uid and *Let* uid u_1 u_2, which express sharing via variable binding. Unlike typical expression evaluators, our *eval* function does not carry around an environment. Instead, it records how much each variable has contributed to the derivative so far. When we call *eval* on the *Var* uid case it simply adds x to the *DeltaMap* entry for uid, indicating that we just saw a use of uid that has been scaled by a factor of x.

This work is used in the case of *Let* uid u_1 u_2. When we reach this case, we *do not* (as would be conventional) evaluate u_1 and bind the result to uid when evaluating u_2. Instead, we evaluate u_2 *first*. The returned *DeltaMap* will tell us the contribution (call it x) of uid to the derivative, and so we can now call *eval* on u_1 scaled by the factor x.

Thus, we evaluate u_2 and u_1 exactly once each. By scaling the usage of u_1 correctly, we get exactly the same result as we would if we re-evaluated it once for every occurrence of uid in the expression.

[9]https://github.com/ekmett/ad

Because each sub-term is evaluated exactly once, the run time of the *eval* function is linear in the size of the delta, which was itself linear in the run time of the original program. So running the *eval* function only slows the run time of our transformation by a constant factor relative to the original program.

The *eval* function needs to be given a constant amount memory for the array, plus it allocates a constant amount of memory *per variable* to store the usage information. However, the number of variables is linear in the size of the delta, and the size of the delta is linear in the run time of the original program, so the memory *eval* allocates is also linear in the original program.

Since the sum of two linear factors is itself linear, our reverse-mode AD program has a memory overhead which is linear in the run time of original program.

7.5 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, the type $\overleftarrow{\mathcal{D}}^1_S\{A\}$ *does* mention S, so that earlier translation does not support separate compilation.

8 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 a with type *Float* to do the primal computation, or with type *Dual* to do the dual-number computation, where

$$\textbf{data}\ \mathit{Dual} = \mathit{Dual}\ \mathit{Float}\ \mathit{Float}$$

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`[10] [11].

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*.

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.

[10] https://github.com/ekmett/ad

[11] 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 (D⃖{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.

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.

9 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 3 (Correctness of $\mathcal{R}\{e\}$).
If e is a closed term at $\mathbb{R}^a \to \mathbb{R}$ then for all $s \in \mathbb{R}^a$, and $\delta t \in \mathbb{R}$, $[\![\mathcal{R}\{e\}]\!]\ (s, \delta t) = dt \cdot \mathcal{J}[\![e]\!](s)$.

To prove this, we have to peek inside the $\mathcal{R}\{e\}$ wrapper function in Figure 11. 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 $\mathbb{R}^n$.

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

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.

9.1 Correctness of the translation

There are two features which makes 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].

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

Our relation relates the original type A to the translated type $\overleftarrow{\mathcal{D}}(A)$. However, our relation is not between syntactic programs – instead, it is a relation between $\mathbb{R}^a \to [\![A]\!]$ and $\mathbb{R}^a \to [\![\overleftarrow{\mathcal{D}}(A)]\!]$. Here, $[\![A]\!]$ is the set-theoretic interpretation of A, which means that instead of relating terms, we relate their *denotations*. This makes it easier to stipulate mathematical conditions in the relation. The second, more important change is that we do not relate individual elements of these sets. Instead, we index $\mathbb{R}^a$-indexed families of these sets. The reason for working with relations between families of values rather than relations between values is to make it easier to talk about derivatives.

We can see why when we look at the $\mathcal{V}^a_{\mathbb{R}}(C)$ clause of the logical relation in Figure 13. Here, a family f of real numbers is related to a function which returns a pair of the value of f at each point together with a *Delta* whose interpretation is the a partial derivatives of f – the ability to vary f is what lets us talk about the correct values of the vector of dual numbers. In addition to permitting the *Delta* to be a variable, we also permit it to be zero. This extra is not technically necessary, but is useful for handling constants. In order to connect terms to the parameterisation, in Figure 13, we fix a set of a variables $z_1 \ldots z_a$, and introduce a conventional substitution ϕ_a which associates each z_i to the i-th basis vector (i.e., $\mathit{onehot}_{\mathbb{R}}\ i$).

Next, the Kripke worlds in this relation are $\mathbb{R}^a$-indexed lists of *Delta* bindings (that is, *DeltaBinds*), with the Kripke ordering being non-shadowing extension. Intuitively, our translation produces a monadic term which works on a set of bindings. So we want to define the meaning of types relative to the current set of bindings, and we want these meanings to remain stable as we extend the *DeltaBinds*. That is, adding variables irrelevant to us makes no difference, and execution of the program never deletes or changes any existing bindings.

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

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

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

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

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

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

$$\mathcal{E}^a_A(C) = \{(f, f') \mid \forall C' \sqsupseteq C.\ \exists C'' \sqsupseteq C', g'.\ (\lambda\vec{x}.\, f'\ \vec{x}\ (C'\ \vec{x})) = \langle g', C''\rangle \wedge (f, g') \in \mathcal{V}^a_A(C'')\}$$

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

$$C' \sqsupseteq C \iff C' \text{ is a non-shadowing extension of } C$$

$$\langle f, g\rangle\ x = (f\ x, g\ x)$$

$f; g$ is reverse function composition ("f is followed by g"), i.e. $\lambda x.\, g(f(x))$

$$\phi_a = \overrightarrow{\mathit{onehot}_{\mathbb{R}^b}\ i / z_i}^{i\in 1..a}$$

Fig. 13. Kripke logical relation

So the Kripke should "just" be the current *DeltaBinds*. However, since we are relating $\mathbb{R}^n$-indexed families of values, we also need to index the set of current bindings by $\mathbb{R}^n$, which is why the Kripke world is an $\mathbb{R}^n$-indexed list of *DeltaBinds*.

Even though we are using a denotational model, we still need an expression relation $\mathcal{E}^a_A(C)$. Again, this is because our translation sends a pure term to a monadic term, and so we need the expression relation to specify what this monadic computation does.

Theorem 4 (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, Appendix A.3. □

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

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 $\mathcal{V}^a_{\mathbb{R}}(C)$. Since $\phi_a(\mathrm{z}_i)$, the interpretation of z_i in the basis vector environment ϕ_a, is just the unit vector which is 1 at i, this means that each of the parameters of the parameterization become one of the arguments to the function. With each of the a parameters in the logical relation connecting to one of the a arguments to the function, we can get the desired result by unwinding definitions. □

9.2 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 6 (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*

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

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.

This last fact holds because our language of deltas has no nonlinear terms. It is also turns out to be essential for proving Theorem 3. 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.

10 GENERALISATIONS AND EXTENSIONS

10.1 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:

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

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

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

$$
\begin{array}{l}
\overrightarrow{\mathcal{D}}\{\mathtt{f}\} = \lambda((x_1, dx_1), \ldots, (x_n, dx_n)). \\
\quad \mathbf{let}\ \vec{x} = (x_1, \ldots, x_n)\ \mathbf{in} \\
\quad \mathbf{let}\ (y_1, \ldots, y_m) = \mathtt{f}\ \vec{x}\ \mathbf{in} \\
\quad ((y_1, \sum_{i=1}^{n} \mathtt{df}_{1i}\ \vec{x} \times dx_i), \ldots, \\
\quad\quad (y_m, \sum_{i=1}^{n} \mathtt{df}_{mi}\ \vec{x} \times dx_i))
\end{array}
$$

$$
\begin{array}{l}
\overleftarrow{\mathcal{D}}\{\mathtt{f}\} = \lambda((x_1, dx_1), \ldots, (x_n, dx_n)). \\
\quad \mathbf{let}\ \vec{x} = (x_1, \ldots, x_n)\ \mathbf{in\ let}\ (y_1, \ldots, y_m) = \mathtt{f}\ \vec{x}\ \mathbf{in} \\
\quad \mathbf{do}\ \{\ dy_1 \leftarrow \mathit{deltaLet}\ (\mathit{Add}_{i=1}^{n}\ (\mathit{Scale}\ (\mathtt{df}_{1i}\ \vec{x})\ dx_i)); \\
\quad\quad\quad \ldots; \\
\quad\quad\quad dy_m \leftarrow \mathit{deltaLet}\ (\mathit{Add}_{i=1}^{n}\ (\mathit{Scale}\ (\mathtt{df}_{mi}\ \vec{x})\ dx_i)); \\
\quad\quad\quad \mathbf{pure}\ ((y_1, dy_1), \ldots, (y_m, dy_m))\ \}
\end{array}
$$

Fig. 14. Translation of arbitrary differentiable function

whatsoever is kept of what operations were done in the forward execution. In the code for $\overleftarrow{\mathcal{D}}\{\mathit{sin}\}$ we allocate a data constructor $(\mathit{Scale}\ (\mathit{cos}\ x)\ u_1)$, but that $(\mathit{cos}\ x)$ is computed in the forward pass, with the resulting real number captured in the *Scale* data constructor. 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.

10.2 Arrays

Any serious AD system must support arrays and tensors well. Happily, it is very easy to do so. For example, suppose we add a type *Vector*, with operations *build* and *index* to construct arrays and take them apart[12]:

$$\mathit{index}_A : (\mathit{Vector}\ A \times \mathbb{N}) \rightarrow A \qquad \mathit{build}_A : (\mathbb{N} \times (\mathbb{N} \rightarrow A)) \rightarrow \mathit{Vector}\ A$$

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

Type translation

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

Term translation

$$
\begin{array}{l}
\overleftarrow{\mathcal{D}}\{\mathit{index}_A\} = \mathbf{pure}\ (\lambda x.\ \mathbf{pure}\ (\mathit{index}_{\overleftarrow{\mathcal{D}}\{A\}}\ x)) \\
\overleftarrow{\mathcal{D}}\{\mathit{build}_A\} = \mathbf{pure}\ (\lambda x.\ \mathit{sequence}_{\overleftarrow{\mathcal{D}}\{A\}}\ (\mathit{build}_{M\ \overleftarrow{\mathcal{D}}\{A\}}\ x))
\end{array}
$$

Here $\mathit{sequence}_A : \mathit{Vector}\ (M\ A) \rightarrow M\ (\mathit{Vector}\ A)$ 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 an $\mathit{Vector}\ \mathbb{R}$ translates to $\mathit{Vector}\ (\mathbb{R} \times \mathit{Delta})$, a vector of pairs. It might be more efficient to use a pair of vectors, using a translation like

Type translation

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

That would be entirely possible; but of course the translation of the primitives would also need the same special cases.

[12]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.

10.3 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.

10.4 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 possible.

A more promising approach is to 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 $\mathcal{R}\{\mathcal{R}\{e\}\}$.

11 GENERALISING TO ARBITRARY S AND T

Our compositional, recursive transformation $\overrightarrow{\mathcal{D}}$ 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[13].

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 10. The only changes needed are to the forward and reverse wrappers, and are shown in Figure 15.

11.1 Generalised forward mode

The forward-mode wrapper in Figure 15 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 15. 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 extract the second component of each dual number; it is the companion to *primal*.

[13]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):

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

Wrapper for reverse derivative (generalised from Figure 11):

$$\begin{array}{rl}
\text{Given} & e \ :\ S \to T \\
\text{and} & \overleftarrow{\mathcal{D}}\{e\} \ :\ \overleftarrow{\mathcal{D}}\{S\} \to M\ \overleftarrow{\mathcal{D}}\{T\} \quad \text{(as defined in Figure 9)} \\
\text{we produce} & \mathcal{R}\{e\} \ :\ (S \times \delta T) \to \delta S \\
& \mathcal{R}\{e\} = \lambda(s, dt). \\
& \qquad \mathbf{let}\ (s_0, dv) = initVars_S\ (s, 0)\ \mathbf{in} \\
& \qquad \mathbf{let}\ t_0 = initZeros_T\ dt\ \mathbf{in} \\
& \qquad \mathbf{let}\ u = runDelta\ (dv + 1)\ (\overleftarrow{\mathcal{D}}\{\lambda(s, dt).\ e\ s \odot_T dt\}\ (s_0, dt_0))\ \mathbf{in} \\
& \qquad \mathbf{let}\ finalMap = eval\ 1\ u\ emptyMap\ \mathbf{in} \\
& \qquad lookup_S\ (finalMap, s_0)
\end{array}$$

New polytypic primitives:

$delta_A : \overrightarrow{\mathcal{D}}\{A\} \to \delta A$	Select delta component of all dual numbers
$zip_A : (A \times \delta A) \to \overrightarrow{\mathcal{D}}\{A\}$	Zip two A-structures together
$initVars_A : (A \times DeltaId) \to (\overleftarrow{\mathcal{D}}\{A\} \times DeltaId)$	Dualise each $\mathbb{R}$ with $Var\ uid$
$initZeros_A : A \to \overleftarrow{\mathcal{D}}\{A\}$	Dualise each $\mathbb{R}$ with $Zero$
$lookup_A : (DeltaMap \times \overleftarrow{\mathcal{D}}\{A\}) \to \delta A$	Look up each dual in the map

Fig. 15. Reverse-mode wrapper for general S and T

11.2 Generalised reverse mode

The reverse-mode wrapper in Figure 15 embodies the following changes, compared with Figure 11:

- The initial value $s_0{:}\overleftarrow{\mathcal{D}}\{S\}$ is obtained by replacing every real number s_i in s with a dual number $(s_i, Var\ uid)$, where uid is a distinct $DeltaId$. This is done by the polytypic function $initVars_S$, 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.
- We also define $dt_0{:}\overleftarrow{\mathcal{D}}\{\delta T\}$, by dualising dt 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 $\lambda(s, dt).\ e\ s \odot_T dt$, 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 dt, to get a $\mathbb{R}$.
- The transformed expression has type $\overleftarrow{\mathcal{D}}\{S\} \times \overleftarrow{\mathcal{D}}\{\delta T\} \to M\ (\mathbb{R} \times Delta)$, so we can apply it to (s_0, dt_0) to get a value of type $M\ (\mathbb{R} \times Delta)$ which is what $runDelta$ needs.
- Then, in the last line of the definition of $\mathcal{R}\{e\}$, walk over s_0 with the polytypic function $lookup_S$, replacing each dual-number leaf $(s_i, Var\ uid)$ 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 use **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.

12 RELATED WORK

There is a rich and rapidly growing literature on automatic differentiation. Here we focus primarily on work that tackles *reverse-mode* AD for *higher-order* languages with full first-class functions.

The original work on automatic differentiation was aimed at first-order imperative programming languages [Griewank and Walther 2008], but there has always been significant 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 real numbers means that literally any form of parameterisation over the representation over 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 wide use in the scientific computing world long before then.

Beginning with the pioneering work of Pearlmutter and Siskind in *"Lambda the ultimate backpropagator"* [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[14], it did offer a smoothly compositional and mathematically well-behaved treatment of AD, which proved influential in succeeding work.

[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, and 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

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

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 that 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] uses a version of the differential lambda calculus and give a direct operational semantics for their language.

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 6 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 Pashmina Cameron, Siddharth Krishna, Sarah Lewis, Tom Minka, and Dimitrios Vytiniotis for their helpful feedback.

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

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* (Edinburgh, Scotland) *(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.

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.

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

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

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

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

A APPENDIX

A.1 Typing

Definition 7 (Type system for source language). $\boxed{\Gamma \vdash e : A}$ *(Type system)*

$$\textsc{Typing-FloatConst}\quad \frac{}{\Gamma \vdash r : \mathbb{R}} \qquad \textsc{Typing-IntConst}\quad \frac{k : L}{\Gamma \vdash k : L} \qquad \textsc{Typing-Var}\quad \frac{\Gamma(x) = A}{\Gamma \vdash x : A} \qquad \textsc{Typing-Let}\quad \frac{\Gamma \vdash e_1 : A_1 \quad \Gamma, x{:}A_1 \vdash e_2 : A_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : A_2}$$

$$\textsc{Typing-Lam}\quad \frac{\Gamma, x{:}A \vdash e : B}{\Gamma \vdash \lambda x.\, e : A \to B} \qquad \textsc{Typing-App}\quad \frac{\Gamma \vdash e_1 : A \to B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B} \qquad \textsc{Typing-Pair}\quad \frac{\Gamma \vdash e_1 : A_1 \qquad \Gamma \vdash e_2 : A_2}{\Gamma \vdash (e_1, e_2) : A_1 \times A_2} \qquad \textsc{Typing-Fst}\quad \frac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \textbf{fst}\ e : A_1}$$

$$\textsc{Typing-Snd}\quad \frac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \textbf{snd}\ e : A_2} \qquad \textsc{Typing-Inl}\quad \frac{\Gamma \vdash e_1 : A_1}{\Gamma \vdash \textbf{inl}\ e_1 : A_1 + A_2} \qquad \textsc{Typing-Inr}\quad \frac{\Gamma \vdash e_1 : A_2}{\Gamma \vdash \textbf{inr}\ e_1 : A_1 + A_2}$$

Typing-CaseSm

$$\frac{\Gamma \vdash e_0 : A_1 + A_2 \quad \Gamma, x_1{:}A_1 \vdash e_1 : A \quad \Gamma, x_2{:}A_2 \vdash e_2 : A}{\Gamma \vdash \textbf{case}\ e_0\ \textbf{of}\ \textbf{inl}\ x_1 \longrightarrow e_1,\ \textbf{inr}\ x_2 \longrightarrow e_2 : A}$$

Typing-OpPlusR

$$\frac{}{\Gamma \vdash +_{\mathbb{R}} : (\mathbb{R} \times \mathbb{R}) \longrightarrow \mathbb{R}}$$

Typing-OpTimesR

$$\frac{}{\Gamma \vdash \times_{\mathbb{R}} : (\mathbb{R} \times \mathbb{R}) \longrightarrow \mathbb{R}}$$

A.2 Denotational semantics

Definition 8 (Denotational semantics for types in source language).

$$\begin{aligned}
[\![\mathbb{R}]\!] &= \mathbb{R} \\
[\![L]\!] &= L \\
[\![A_1 \times A_2]\!] &= [\![A_1]\!] \times [\![A_2]\!] = \{(x,\ y) \mid x \in [\![A_1]\!] \text{ and } y \in [\![A_2]\!]\} \\
[\![A_1 + A_2]\!] &= [\![A_1]\!] \uplus [\![A_2]\!] \\
&= \{(0, x) \mid x \in [\![A_1]\!]\} \cup \{(1, y) \mid y \in [\![A_2]\!]\} \\
[\![A_1 \longrightarrow A_2]\!] &= [\![A_1]\!] \Longrightarrow [\![A_2]\!]
\end{aligned}$$

Definition 9 (Denotational semantics for expressions in source language).

$$\begin{aligned}
[\![\Gamma \vdash x : A]\!]\gamma &= \Pi_x(\gamma) \\
[\![\Gamma \vdash \lambda x.\, e : A \longrightarrow B]\!]\gamma &= v \mapsto [\![\Gamma, x{:}A \vdash e : B]\!](\gamma, v/x) \\
[\![\Gamma \vdash e_1\ e_2 : A]\!]\gamma &= ([\![\Gamma \vdash e_1 : A \longrightarrow B]\!]\gamma)([\![\Gamma \vdash e_2 : A]\!]\gamma) \\
[\![\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : B]\!]\gamma &= [\![\Gamma \vdash e_2 : B]\!](\gamma, [\![\Gamma \vdash e_1 : A]\!]\gamma/x) \\
[\![\Gamma \vdash \textbf{inl}\ e : A_1 + A_2]\!]\gamma &= (0,\ [\![\Gamma \vdash e : A_1]\!]\gamma) = ([\![\Gamma \vdash e : A_1]\!]; \textbf{inl}\ )\gamma \\
[\![\Gamma \vdash \textbf{inr}\ e : A_1 + A_2]\!]\gamma &= (1,\ [\![\Gamma \vdash e : A_2]\!]\gamma) = ([\![\Gamma \vdash e : A_2]\!]; \textbf{inr}\ )\gamma \\
[\![\Gamma \vdash (e_1, e_2) : A_1 \times A_2]\!]\gamma &= ([\![\Gamma \vdash e_1 : A_1]\!]\gamma,\ [\![\Gamma \vdash e_2 : A_2]\!]\gamma) \\
[\![\Gamma \vdash \textbf{fst}\ e : A_1]\!]\gamma &= \Pi_1([\![\Gamma \vdash e : A]\!]\gamma) = \textbf{let}\ (x_1,\ x_2) = [\![\Gamma \vdash e : A]\!]\gamma\ \textbf{in}\ x_1 \\
[\![\Gamma \vdash \textbf{snd}\ e : A_2]\!]\gamma &= \Pi_2([\![\Gamma \vdash e : A]\!]\gamma) = \textbf{let}\ (x_1,\ x_2) = [\![\Gamma \vdash e : A]\!]\gamma\ \textbf{in}\ x_2 \\
[\![\Gamma \vdash \textbf{case}\ t_0\ \textbf{of}\ \textbf{inl}\ x_1 \longrightarrow t_1,\ \textbf{inr}\ x_2 \longrightarrow t_2 : B]\!]\gamma &= \textbf{let}\ (t,\ v) = [\![\Gamma \vdash e : A]\!]\gamma\ \textbf{in} \\
&\quad\ \textbf{if}\ t = 0\ \textbf{then}\ [\![\Gamma \vdash e_1 : A]\!](\gamma, v/x)\ \textbf{else}\ [\![\Gamma \vdash e_2 : A]\!](\gamma, v/y) \\
[\![\Gamma \vdash k : L]\!]\gamma &= k \\
[\![\Gamma \vdash r : \mathbb{R}]\!]\gamma &= r \\
[\![\Gamma \vdash +_{\mathbb{R}} : (\mathbb{R} \times \mathbb{R}) \longrightarrow \mathbb{R}]\!]\gamma &= \lambda(x_1, x_2).\, x_1 + x_2 \\
[\![\Gamma \vdash \times_{\mathbb{R}} : (\mathbb{R} \times \mathbb{R}) \longrightarrow \mathbb{R}]\!]\gamma &= \lambda(x_1, x_2).\, x_1 \times x_2
\end{aligned}$$

Lemma 10. *If $C' \sqsupseteq C$ then $\mathcal{V}^a_A(C') \supset \mathcal{V}^a_A(C)$.*

Proof. By induction on A. □

Lemma 11. *If $(f, f') \in \mathcal{V}^a_A(C)$ then $(f, \mathit{return}\ f') \in \mathcal{E}^a_A(C)$.*

Proof. By routine calculation. □

A.3 Proof of Theorem 4

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

Proof. By rule induction on $\Gamma \vdash e : A$ – selection of cases:

Typing-FloatConst

- $\overline{\Gamma \vdash r : \mathbb{R}}$

  We WTS:

$$\forall (\hat{\gamma}, \hat{\gamma}') \in \mathcal{V}^a_\Gamma(C).\, (\hat{\gamma}; \llbracket \Gamma \vdash r : \mathbb{R} \rrbracket, \hat{\gamma}'; \llbracket \overleftarrow{\mathcal{D}}\{\Gamma\} \vdash \overleftarrow{\mathcal{D}}\{r\} : \overleftarrow{\mathcal{D}}\{\mathbb{R}\} \rrbracket) \in \mathcal{E}^a_\mathbb{R}(C)$$

  From definition of $\overleftarrow{\mathcal{D}}$ (9) and Definition 9 this is equivalent to:

$$\forall (\hat{\gamma}, \hat{\gamma}') \in \mathcal{V}^a_\Gamma(C).\, (\hat{\gamma}; \lambda\gamma{:}\llbracket \Gamma \rrbracket.\, r, \hat{\gamma}'; \lambda\gamma{:}\llbracket \overleftarrow{\mathcal{D}}\{\Gamma\} \rrbracket.\, \llbracket \mathit{return}\ (r, \mathit{Zero}) \rrbracket \gamma) \in \mathcal{E}^a_\mathbb{R}(C)$$

  We know that:

$$\hat{\gamma}; \lambda\gamma.\, r = \lambda\vec{x}.\, (\lambda\gamma.\, r)(\hat{\gamma}\ \vec{x}) = \lambda\vec{x}.\, r$$

  and

$$\begin{aligned}\hat{\gamma}'; \lambda\gamma.\, &\llbracket \mathit{return}\ (r, \mathit{Zero}) \rrbracket \gamma \\ &= \lambda\vec{x}.\, (\lambda\gamma.\, \llbracket \mathit{return}\ (r, \mathit{Zero}) \rrbracket \gamma)(\hat{\gamma}'\ \vec{x}) \\ &= \lambda\vec{x}.\, \llbracket \mathit{return}\ (r, \mathit{Zero}) \rrbracket (\hat{\gamma}'\ \vec{x}) \\ &= \lambda\vec{x}.\, \lambda s.\, ((r, \vec{0}), s)\end{aligned}$$

  therefore we WTS that:

$$\forall (\hat{\gamma}, \hat{\gamma}') \in \mathcal{V}^a_\Gamma(C).\, (\lambda\vec{x}.\, r, \lambda\vec{x}.\, \lambda s.\, ((r, \vec{0}), s)) \in \mathcal{E}^a_\mathbb{R}(C)$$

  By definition of $\mathcal{E}^a_\mathbb{R}(C)$ (13) we equivalently WTS:

$$\begin{aligned}&\forall C' \sqsupseteq C.\\ &\exists C'' \sqsupseteq C', g'.\\ &\qquad \lambda\vec{x}.\, (\lambda\vec{x}.\, \lambda s.\, ((r, \vec{0}), s))\ \vec{x}\ (C'\ \vec{x}) = \langle g', C'' \rangle \\ &\qquad \wedge (\lambda\vec{x}.\, r, g') \in \mathcal{V}^a_\mathbb{R}(C'')\end{aligned} \tag{1}$$

  Simplifying the first condition yields:

$$\begin{aligned}\lambda\vec{x}.\, (\lambda\vec{x}.\, \lambda s.\, &((r, \vec{0}), s))\ \vec{x}\ (C'\ \vec{x}) \\ &= \lambda\vec{x}.\, ((r, \vec{0}), (C'\ \vec{x})) \\ &= \langle \lambda\vec{x}.\, (r, \vec{0}), C' \rangle\end{aligned} \tag{2}$$

  Choosing $g' = \lambda\vec{x}.\, (r, \vec{0})$ and $C'' = C'$:

$$\langle \lambda\vec{x}.\, (r, \vec{0}), C' \rangle = \langle g', C'' \rangle$$

  therefore from (2):

$$\lambda\vec{x}.\,(\lambda\vec{x}.\,\lambda s.\,((r,\vec{0}),s))\;\vec{x}\;(C'\;\vec{x}) = \langle g', C''\rangle \tag{3}$$

From the definition of $\mathcal{V}^a_{\mathbb{R}}(C)$ 13, we know that

$$\forall C, r.\,(\lambda\vec{x}.\,r, \lambda\vec{x}.\,(r,\vec{0})) \in \mathcal{V}^a_{\mathbb{R}}(C)$$

We assumed that $g' = \lambda\vec{x}.\,(r,\vec{0})$:

$$(\lambda\vec{x}.\,r, g') \in \mathcal{V}^a_{\mathbb{R}}(C'') \tag{4}$$

By (3) and (4)

$$(\hat{\gamma}; [\![\Gamma \vdash r : \mathbb{R}]\!], \hat{\gamma}'; [\![\overleftarrow{\mathcal{D}}\{\Gamma\} \vdash \overleftarrow{\mathcal{D}}\{r\} : \overleftarrow{\mathcal{D}}\{\mathbb{R}\}]\!]) \in \mathcal{E}^a_{\mathbb{R}}(C)$$

which is what we wanted to show in (1).

- Typing-Var
  $$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}$$
  We WTS:
  $$\forall(\hat{\gamma},\hat{\gamma}') \in \mathcal{V}^a_{\Gamma}(C).\,(\hat{\gamma}; [\![\Gamma \vdash x : A]\!], \hat{\gamma}'; [\![\overleftarrow{\mathcal{D}}\{\Gamma\} \vdash \overleftarrow{\mathcal{D}}\{x\} : \overleftarrow{\mathcal{D}}\{A\}]\!]) \in \mathcal{E}^a_A(C)$$
  From definition of $\overleftarrow{\mathcal{D}}$ (9) and Definition 9 this is equivalent to:
  $$\forall(\hat{\gamma},\hat{\gamma}') \in \mathcal{V}^a_{\Gamma}(C).\,(\hat{\gamma}; \lambda\gamma{:}[\![\Gamma]\!].\,\Pi_x(\gamma), \hat{\gamma}'; \lambda\gamma{:}[\![\overleftarrow{\mathcal{D}}\{\Gamma\}]\!].\,[\![\mathit{return}\;x]\!]\gamma) \in \mathcal{E}^a_A(C)$$
  We know that:
  $$\begin{aligned}\hat{\gamma}; \lambda\gamma.\,\Pi_x(\gamma) &= \lambda\vec{x}.\,(\lambda\gamma.\,\Pi_x(\gamma))(\hat{\gamma}\;\vec{x})\\ &= \lambda\vec{x}.\,\Pi_x(\hat{\gamma}\;\vec{x})\end{aligned}$$
  and
  $$\begin{aligned}\hat{\gamma}'; \lambda\gamma.\,[\![\mathit{return}\;x]\!]\gamma &= \lambda\vec{x}.\,(\lambda\gamma.\,[\![\mathit{return}\;x]\!]\gamma)(\hat{\gamma}'\;\vec{x})\\ &= \lambda\vec{x}.\,[\![\mathit{return}\;x]\!](\hat{\gamma}'\;\vec{x})\\ &= \lambda\vec{x}.\,[\![\lambda s \longrightarrow (x,s)]\!](\hat{\gamma}'\;\vec{x})\\ &= \lambda\vec{x}.\,(\lambda\gamma.\,\lambda s.\,([\![x]\!]\gamma, s))(\hat{\gamma}'\;\vec{x})\\ &= \lambda\vec{x}.\,\lambda s.\,([\![x]\!](\hat{\gamma}'\;\vec{x}), s)\\ &= \lambda\vec{x}.\,\lambda s.\,(\Pi_x(\hat{\gamma}'\;\vec{x}), s)\end{aligned}$$
  therefore we WTS that:
  $$\forall(\hat{\gamma},\hat{\gamma}') \in \mathcal{V}^a_{\Gamma}(C).\,(\lambda\vec{x}.\,\Pi_x(\hat{\gamma}\;\vec{x}), \lambda\vec{x}.\,\lambda s.\,(\Pi_x(\hat{\gamma}'\;\vec{x}), s)) \in \mathcal{E}^a_A(C)$$
  By definition of $\mathcal{E}^a_A(C)$ (13) we equivalently WTS:
  $$\begin{aligned}&\forall C' \sqsupseteq C.\\ &\exists C'' \sqsupseteq C', g'.\\ &\quad \lambda\vec{x}.\,(\lambda\vec{y}.\,\lambda s.\,(\Pi_x(\hat{\gamma}'\;\vec{y}), s))\;\vec{x}\;(C'\;\vec{x}) = \langle g', C''\rangle\\ &\quad \wedge (\lambda\vec{x}.\,\Pi_x(\hat{\gamma}\;\vec{x}), g') \in \mathcal{V}^a_A(C'')\end{aligned} \tag{5}$$

Simplifying:

$$\begin{aligned} \lambda\vec{x}.\,(\lambda\vec{y}.\,\lambda s.\,(\Pi_x(\hat{\gamma}'\;\vec{y}), s))\;\vec{x}\;(C'\;\vec{x}) &= \lambda\vec{x}.\,(\Pi_x(\hat{\gamma}'\;\vec{x}), (C'\;\vec{x})) \\ &= \langle \lambda\vec{x}.\,\Pi_x(\hat{\gamma}'\;\vec{x}), C'\rangle \end{aligned}$$

Choosing $g' = \lambda\vec{x}.\,\Pi_x(\hat{\gamma}'\;\vec{x})$ and $C'' = C'$:

$$\lambda\vec{x}.\,(\lambda\vec{y}.\,\lambda s.\,(\Pi_x(\hat{\gamma}'\;\vec{y}), s))\;\vec{x}\;(C'\;\vec{x}) = \langle g', C''\rangle \tag{6}$$

and

$$\begin{aligned} (\lambda\vec{x}.\,\Pi_x(\hat{\gamma}\;\vec{x}), g') &= (\lambda\vec{x}.\,\Pi_x(\hat{\gamma}\;\vec{x}), \lambda\vec{x}.\,\Pi_x(\hat{\gamma}'\;\vec{x})) \\ &= \langle \hat{\gamma};\Pi_x, \hat{\gamma}';\Pi_x\rangle \end{aligned}$$

We assumed $(\hat{\gamma}, \hat{\gamma}') \in \mathcal{V}^a_\Gamma(C)$, so by definition of $\mathcal{V}^a_\Gamma(C'')$ and Lemma 10,

$$\langle \hat{\gamma};\Pi_x, \hat{\gamma}';\Pi_x\rangle \in \mathcal{V}^a_A(C'') \tag{7}$$

By (6) and (7) we have:

$$\begin{aligned} &\forall C' \sqsupseteq C. \\ &\exists C'' \sqsupseteq C', g'. \\ &\qquad \lambda\vec{x}.\,(\lambda\vec{y}.\,\lambda s.\,(\Pi_x(\hat{\gamma}'\;\vec{y}), s))\;\vec{x}\;(C'\;\vec{x}) = \langle g', C''\rangle \\ &\qquad \wedge (\lambda\vec{x}.\,\Pi_x(\hat{\gamma}\;\vec{x}), g') \in \mathcal{V}^a_A(C'') \end{aligned} \tag{8}$$

which is what we wanted to show in (5).

- Typing-App
  $$\frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\;e_2 : B}$$
  We WTS:
  $$\forall(\hat{\gamma}, \hat{\gamma}') \in \mathcal{V}^a_\Gamma(C).\,(\hat{\gamma}; [\![\Gamma \vdash e_1\;e_2 : B]\!], \hat{\gamma}'; [\![\overleftarrow{\mathcal{D}}\{\Gamma\} \vdash \overleftarrow{\mathcal{D}}\{e_1\;e_2\} : \overleftarrow{\mathcal{D}}\{B\}]\!]) \in \mathcal{E}^a_B(C)$$
  By applying the induction hypothesis to the premises we know:
  $$\forall C, (\hat{\gamma}, \hat{\gamma}') \in \mathcal{V}^a_\Gamma(C).\,(\hat{\gamma}; [\![e_1]\!], \hat{\gamma}'; [\![\overleftarrow{\mathcal{D}}\{e_1\}]\!]) \in \mathcal{E}^a_{A\to B}(C) \tag{9}$$
  $$\forall C, (\hat{\gamma}, \hat{\gamma}') \in \mathcal{V}^a_\Gamma(C).\,(\hat{\gamma}; [\![e_2]\!], \hat{\gamma}'; [\![\overleftarrow{\mathcal{D}}\{e_2\}]\!]) \in \mathcal{E}^a_A(C) \tag{10}$$
  Assume $C, (\hat{\gamma}, \hat{\gamma}') \in \mathcal{V}^a_\Gamma(C)$ – we WTS:
  $$(\hat{\gamma}; [\![e_1\;e_2]\!], \hat{\gamma}'; [\![\overleftarrow{\mathcal{D}}\{e_1\;e_2\}]\!]) \in \mathcal{E}^a_B(C) \tag{11}$$
  Assume $C' \sqsupseteq C$ and note that:
  $$\begin{aligned} &\overleftarrow{\mathcal{D}}\{e_1\;e_2\} \\ &= \textbf{do}\,\{f \leftarrow \overleftarrow{\mathcal{D}}\{e_1\};\; x \leftarrow \overleftarrow{\mathcal{D}}\{e_2\};\; f\;x\} \\ &\equiv \overleftarrow{\mathcal{D}}\{e_1\} \gg\!= \lambda f.\,\overleftarrow{\mathcal{D}}\{e_2\} \gg\!= \lambda x.\,f\;x \\ &\equiv \lambda s_1.\textbf{let}\,(t_1, s_1') = \overleftarrow{\mathcal{D}}\{e_1\}\;s_1\;\textbf{in}(\lambda f.\,\lambda s_2.\,\textbf{let}\,(t_2, s_2') = \overleftarrow{\mathcal{D}}\{e_2\}\;s_2\;\textbf{in}\,(\lambda x.\,f\;x)\;t_2\;s_2')\;t_1\;s_1' \end{aligned} \tag{12}$$

By IH (9) we know:

$$\exists C'' \sqsupseteq C', f'. \quad \lambda\vec{x}.\,(\hat{\gamma}'; [\![\overleftarrow{\mathcal{D}}\{e_1\}]\!])\ \vec{x}\ (C'\ \vec{x}) = \langle f', C''\rangle \wedge (\hat{\gamma}; [\![e_1]\!], f') \in \mathcal{V}^a_{A\to B}(C'') \tag{13}$$

Note $C'' \sqsupseteq C' \sqsupseteq C$, so $C'' \sqsupseteq C$.
By IH (10) on C'':

$$\exists C''' \sqsupseteq C'', x'. \quad \lambda\vec{x}.\,(\hat{\gamma}'; [\![\overleftarrow{\mathcal{D}}\{e_2\}]\!])\ \vec{x}\ (C''\ \vec{x}) = \langle x', C'''\rangle \wedge (\hat{\gamma}; [\![e_2]\!], x') \in \mathcal{V}^a_{A}(C''') \tag{14}$$

By Lemma 10 $(\hat{\gamma}; [\![e_1]\!], f') \in \mathcal{V}^a_{A\to B}(C''')$, and $(\hat{\gamma}; [\![e_2]\!], x') \in \mathcal{V}^a_{A}(C''')$.
By definition of $\mathcal{V}^a_{A\to B}(C)$:

$$(\lambda\vec{x}.\,(\hat{\gamma}; [\![e_1]\!])\ \vec{x}\ ((\hat{\gamma}; [\![e_2]\!])\ \vec{x}), \lambda\vec{x}.\, f'\ \vec{x}\ (x'\ \vec{x})) \in \mathcal{E}^a_B(C) \tag{15}$$

Simplifying the first element of the above pair we get:

$$\begin{aligned}
&\lambda\vec{x}.\,(\hat{\gamma}; [\![e_1]\!])\ \vec{x}\ ((\hat{\gamma}; [\![e_2]\!])\ \vec{x})\\
&= \lambda\vec{x}.\,(\lambda\vec{x}.\,[\![e_1]\!]\ (\hat{\gamma}\ \vec{x}))\ \vec{x}\ ((\hat{\gamma}; [\![e_2]\!])\ \vec{x})\\
&= \lambda\vec{x}.\,([\![e_1]\!]\ (\hat{\gamma}\ \vec{x}))\ ((\hat{\gamma}; [\![e_2]\!])\ \vec{x})\\
&= \lambda\vec{x}.\,([\![e_1]\!]\ (\hat{\gamma}\ \vec{x}))\ ((\lambda\vec{x}.\,[\![e_2]\!]\ (\hat{\gamma}\ \vec{x}))\ \vec{x})\\
&= \lambda\vec{x}.\,([\![e_1]\!]\ (\hat{\gamma}\ \vec{x}))\ ([\![e_2]\!]\ (\hat{\gamma}\ \vec{x}))\\
&= \lambda\vec{x}.\,([\![e_1]\!]\ (\hat{\gamma}\ \vec{x}))\ ([\![e_2]\!]\ (\hat{\gamma}\ \vec{x}))\\
&= \hat{\gamma}; \lambda\gamma.\,([\![e_1]\!]\ \gamma)\ ([\![e_2]\!]\ \gamma)\\
&= \hat{\gamma}; \lambda\gamma.\,[\![e_1\ e_2]\!]\ \gamma\\
&= \hat{\gamma}; [\![e_1\ e_2]\!]
\end{aligned} \tag{16}$$

Substituting (13) and (14) into the second element of the pair from (15):

$$\begin{aligned}
&\lambda\vec{x}.\, f'\ \vec{x}\ (x'\ \vec{x})\\
&= \lambda\vec{x}.\,\Pi_1(\lambda\vec{x}.\,(\hat{\gamma}'; [\![\overleftarrow{\mathcal{D}}\{e_1\}]\!])\ \vec{x}\ (C'\ \vec{x}))\ \vec{x}\ (\Pi_1(\lambda\vec{x}.\,(\hat{\gamma}'; [\![\overleftarrow{\mathcal{D}}\{e_2\}]\!])\ \vec{x}\ (C''\ \vec{x}))\ \vec{x})\\
&= \lambda\vec{x}.\,\Pi_1(\lambda\vec{x}.\,(\lambda\vec{x}.\,[\![\overleftarrow{\mathcal{D}}\{e_1\}]\!]\ (\hat{\gamma}'\ \vec{x}))\ \vec{x}\ (C'\ \vec{x}))\ \vec{x}\ (\Pi_1(\lambda\vec{x}.\,(\lambda\vec{x}.\,[\![\overleftarrow{\mathcal{D}}\{e_2\}]\!]\ (\hat{\gamma}'\ \vec{x}))\ \vec{x}\ (C''\ \vec{x}))\ \vec{x})\\
&= \lambda\vec{x}.\,\Pi_1(\lambda\vec{x}.\,[\![\overleftarrow{\mathcal{D}}\{e_1\}]\!]\ (\hat{\gamma}'\ \vec{x})\ (C'\ \vec{x}))\ \vec{x}\ (\Pi_1(\lambda\vec{x}.\,[\![\overleftarrow{\mathcal{D}}\{e_2\}]\!]\ (\hat{\gamma}'\ \vec{x})\ (C''\ \vec{x}))\ \vec{x})\\
&= \lambda\vec{x}.\,\Pi_1([\![\overleftarrow{\mathcal{D}}\{e_1\}]\!]\ (\hat{\gamma}'\ \vec{x})\ (C'\ \vec{x}))\ \Pi_1([\![\overleftarrow{\mathcal{D}}\{e_2\}]\!]\ (\hat{\gamma}'\ \vec{x})\ (C''\ \vec{x}))\\
&= \hat{\gamma}'; [\![\overleftarrow{\mathcal{D}}\{e_1\ e_2\}]\!] \qquad \text{by (12)}
\end{aligned} \tag{17}$$

Substituting (16) and (17) back into (15) we get:

$$(\hat{\gamma}; [\![e_1\ e_2]\!], \hat{\gamma}'; [\![\overleftarrow{\mathcal{D}}\{e_1\ e_2\}]\!]) \in \mathcal{E}^a_B(C)$$

which is what we wanted to show in (11).

□